ML Modules and Haskell Type Classes: A Constructive Comparison

Stefan Wehr[1] and Manuel M. T. Chakravarty[2]

[1] Institut für Informatik, Universität Freiburg, Georges-Köhler-Allee 079, 79110 Freiburg i. Br., Germany (e-mail: wehr@informatik.uni-freiburg.de)
[2] School of Computer Science and Engineering, The University of New South Wales, UNSW SYDNEY NSW 2052, Australia (e-mail: chak@cse.unsw.edu.au)

Abstract. Researchers repeatedly observed that the module system of ML and the type class mechanism of Haskell are related. So far, this relationship has received little formal investigation. The work at hand fills this gap: It introduces type-preserving translations from modules to type classes and vice versa, which enable a thorough comparison of the two concepts.

1 Introduction

On first glance, module systems and type classes appear to be unrelated programming-language concepts: Module systems allow large programs to be decomposed into smaller, relatively independent units, whereas type classes [1,2] provide a means for introducing ad-hoc polymorphism; that is, they give programmers the ability to define multiple functions or operators with the same name but different types. However, it has been repeatedly observed [3,4,5,6,7,8] that there is some overlap in functionality between the module system of the programming language ML [9], one of the most powerful module systems in widespread use, and the type class mechanism of the language Haskell [10], which constitutes a sophisticated approach to ad-hoc polymorphism.

It is natural to ask whether these observations rest on a solid foundation, or whether the overlap is only superficial. The standard approach to answer such a question is to devise two formal translations from modules to type classes and vice versa. The translations then pinpoint exactly the features that are easy, hard, or impossible to translate; thereby showing very clearly the differences and similarities between the two concepts.

Such a constructive comparison between ML modules and Haskell type classes is particularly interesting because the strength of one language is a weak point of the other: ML has only very limited support for ad-hoc polymorphism, so translating Haskell type classes to ML modules could give new insights on how to program with this kind of polymorphism in ML. Conversely, the Haskell module system is weak, so an encoding of ML's powerful module system with type classes could open up new possibilities for modular programming in Haskell.

Contributions. Following the path just described, we make four contributions:

– We devise two formal translations from ML modules to Haskell type classes and vice versa, prove that the translations preserve type correctness, and provide implementations for both.
– We use the insights obtained from the translations to compare ML modules with Haskell type classes thoroughly.
– We investigate if and how the techniques used to encode ML modules in terms of Haskell type classes and vice versa can be exploited for modular programming in Haskell and for programming with ad-hoc polymorphism in ML, respectively.
– We suggest a lightweight extension of Haskell's type class system that enables type abstraction.

Outline. We start with examples that motivate the key ideas behind the translations from ML modules to Haskell type classes (Sec. 2) and from Haskell type classes to ML modules (Sec. 3). We then sketch the formalization and implementation of the translations (Sec. 4). Next, we discuss similarities and differences between ML modules and Haskell type classes (Sec. 5). Finally, we compare with related work (Sec. 6) and conclude (Sec. 7).

2 From Modules to Classes

The idea of the translation from ML modules to Haskell type classes is the following: signatures are modeled as type class declarations, structures and functors are translated into instance declarations, and type and value components of signatures and structures are mapped to associated type synonyms [6] and type class methods, respectively. We now substantiate the idea by presenting example translations of signatures and structures (Sec. 2.1), of abstract types (Sec. 2.2), and of functors (Sec. 2.3). Next, we provide a summary (Sec. 2.4). Finally, we elaborate on alternative translation techniques (Sec. 2.5).[3]

2.1 Translating Signatures and Structures

Our first example is shown in Fig. 1. The ML code defines a structure `IntSet`, which implements sets of integers in terms of lists. The signature of `IntSet` is inferred implicitly in ML; however, we represent it explicitly as a type class `SetSig` in Haskell. The **`type`** declarations in this class introduce two associated type synonyms `Elem a` and `Set a`. The identities of such type synonyms depend on a particular instantiation of the class variable `a`. Hence, concrete definitions for `Elem` and `Set` are deferred to instance declarations of `SetSig`.

[3] We use Standard ML in this section; the Haskell code runs under GHC's [11] latest development version (after replacing **`abstype`** with **`type`**). Throughout the paper, we assume an ML function `any : ('a -> bool) -> 'a list -> bool` corresponding to Haskell's standard function `any :: (a -> bool) -> [a] -> Bool`. Moreover, we rely on functions `intEq`, `intLt`, and `stringEq` for comparing integers and strings.

ML

```
structure IntSet = struct type elem = int  type set = elem list
                          val empty = []   fun member i s = any (intEq i) s
                          fun insert i s = if member i s then s else (i::s)
                   end
```

Haskell

```
class SetSig a where
    type Elem  a;           type Set   a
    empty  ::  a -> Set a;  member ::  a -> Elem a -> Set a -> Bool
    insert ::  a -> Elem a -> Set a -> Set a
data IntSet = IntSet
instance SetSig IntSet where
    type Elem IntSet = Int;  type Set  IntSet = [Int]
    empty  _ = [];           member _ i s     = any (intEq i) s
    insert _ i s = if member IntSet i s then s else (i : s)
```

Figure 1. ML structure for integer sets and its translation to Haskell

ML

```
structure IntSet' = IntSet :> sig type elem = int  type set
                                  val empty : set
                                  val member : elem -> set -> bool
                                  val insert : elem -> set -> set   end
```

Haskell

```
data IntSet' = IntSet'
instance SetSig IntSet' where
    type Elem IntSet' = Elem IntSet;  abstype Set  IntSet' = Set IntSet
    empty _ = empty IntSet;           member _ = member IntSet
    insert _ = insert IntSet
```

Figure 2. Sealed ML structure for integer sets and its translation to Haskell

The data type `IntSet` corresponds to the name of the structure in ML. We translate the structure itself by defining an instance of `SetSig` for `IntSet`. The translation of the `insert` function shows that we encode access to the structure component `member` by indexing the method `member` with a value of type `IntSet`. We use the same technique to translate qualified access to structure components. For example, the ML expression `IntSet.insert 1 IntSet.empty` is written as `insert IntSet 1 (empty IntSet)` in Haskell.

2.2 Translating Abstract Types

The `IntSet` structure reveals to its clients that sets are implemented in terms of lists. This is not always desirable; often, the type `set` should be kept abstract outside of the structure. Our next example (Fig. 2) shows that we can achieve the desired effect in ML by sealing the `IntSet` structure with a signature that leaves the right-hand-side of `set` unspecified. Such signatures are called *translucent*, in contrast to *transparent* (all type components specified) and *opaque* (all type components unspecified) signatures.

Abstract types pose a problem to the translation because there is no obvious counterpart for them in Haskell's type class system. However, we can model

them easily by slightly generalizing associated type synonyms to also support *abstract associated type synonyms*. (We discuss other possibilities for representing abstract types in Haskell in Sec. 2.5). The idea behind abstract associated type synonyms is to limit the scope of the right-hand side of an associated type synonym definition to the instance defining the synonym: Inside the instance, the right–hand side is visible, but outside it is hidden; that is, the associated type synonym is equated with some fresh type constructor.[4] The first author's diploma thesis [13] includes a formalization of this extension.

The Haskell code in Fig. 2 demonstrates how our extension is used to model abstract types in Haskell. The new keyword **`abstype`** introduces an abstract associated type synonym `Set` in the instance declaration for `IntSet'`. The effect of using **`abstype`** is that the type equality `Set IntSet' = [Int]` is visible from within the instance declaration, but not from outside.

Note that there is no explicit Haskell translation for the signature of the structure `IntSet'`. Instead, we reuse the type class `SetSig` from Fig. 1. Such a reuse is possible because type abstraction in Haskell is performed inside instance (and not class) declarations, which means that the signatures of the ML structures `IntSet` and `IntSet'`—differing only in whether the type component `set` is abstract or not—would be translated into equivalent type classes.

2.3 Translating Functors

So far, we only considered sets of integers. ML allows the definition of generic sets through functors, which act as functions from structures to structures. Fig. 3 shows such a functor. (We removed the `elem` type component from the functor body to demonstrate a particular detail of the translation to Haskell.)

The Haskell version defines two type classes `EqSig` and `MkSetSig` as translations of the anonymous argument and result signatures, respectively. The class `MkSetSig` is a multi-parameter type class [14], a well-known generalization of Haskell 98's single-parameter type classes. The first parameter `b` represents a possible implementation of the functor body, whereas the second parameter `a` corresponds to the functor argument; the constraint `EqSig a` allows us to access the associated type synonym `T` of the `EqSig` class in the body of `MkSetSig`. (Now it should become clear why we removed the `elem` type component from the functor body: If `E.t` did not appear in a value specification of the functor body, the necessity for the class parameter `a` would not occur.) Note that we cannot reuse the names `Set`, `empty`, `member`, and `insert` of class `SetSig` because type synonyms and class methods share a global namespace in Haskell.

The instance of `MkSetSig` for the fresh data type `MkSet` and some type variable `a` is the translation of the functor body. The constraint `EqSig a` in the instance context is necessary because we use the associated type synonym `T` and the method `eq` in the instance body.

[4] Interestingly, this idea goes back to ML's **`abstype`** feature, which is nowadays essentially deprecated; the Haskell interpreter Hugs [12] implements a similar feature. In contrast to abstract associated type synonyms, these approaches require the programmer to specify the scope of the concrete identity of an abstract type explicitly.

ML

```
functor MkSet (E : sig type t  val eq : t -> t -> bool  end)
  =  struct type set = E.t list  val empty = []
            fun member x s = any (E.eq x) s
            fun insert x s = if member x s then s else (x :: s)       end
  :> sig type set  val empty : set  val member : E.t -> set -> bool
         val insert : E.t -> set -> set                               end
```

Haskell

```
class EqSig a where
    type T a;  eq :: a -> T a -> T a -> Bool
class EqSig a => MkSetSig b a where
    type Set' b a;  empty' :: b -> a -> Set' b a
    member' :: b -> a -> T a -> Set' b a -> Bool
    insert' :: b -> a -> T a -> Set' b a -> Set' b a
data MkSet = MkSet
instance EqSig a => MkSetSig MkSet a where
    abstype Set' MkSet a = [T a];  empty' _ _ = []
    member' _ a x s = any (eq a x) s
    insert' _ a x s = if member' MkSet a x s then s else (x : s)
```

Figure 3. ML functor for generic sets and its translation to Haskell

ML

```
structure StringSet = MkSet(struct type t = string  val  eq = stringEq end)
```

Haskell

```
data StringEq = StringEq
instance EqSig StringEq where
    type T StringEq = String;  eq _ = stringEq
```

Figure 4. Functor invocation in ML and its translation to Haskell

Fig. 4 shows how we use the `MkSet` functor to construct a set implementation for strings. To translate the functor invocation to Haskell, we define an appropriate `EqSig` instance for type `StringEq`. The combination of the two types `MkSetSig` and `StringEq` now correspond to the ML structure `StringSet`: accessing a component of `StringSet` is encoded in Haskell as an application (either on the type or the term level) with arguments `MkSet` and `StringEq`. For example, `StringSet.empty` translates to `empty' MkSet StringEq`.

To demonstrate that our Haskell implementation for sets of strings fits the general set framework, we provide an instance declaration for `SetSig` (Fig. 1):[5]

```
data StringSet = StringSet
instance SetSig StringSet where
    type Elem StringSet = String
    abstype Set StringSet = Set' MkSet StringEq
    empty _ = empty' MkSet StringEq;  member _ = member' MkSet StringEq
    insert _ = insert' MkSet StringEq
```

[5] The formal translation generates a class and an instance corresponding to the implicit signature of the ML structure `StringSet` and the structure itself, respectively.

ML	Haskell
structure signature	one-parameter type class
structure	instance of the corresponding type class
functor argument signature	single-parameter type class
functor result signature	two-parameter type class (subclass of the argument class)
functor	instance of the result class (argument class appears in the instance context)
structure/functor name	data type
type specification	associated type synonym declaration
type definition	associated type synonym definition
type occurrence	associated type synonym application
value specification	method signature
value definition	method implementation
value occurrence	method application

Table 1. Informal mapping from ML modules to Haskell type classes

2.4 Summary

Table 1 summarizes the (informal) translation from ML modules to Haskell type classes developed so far. We use the notion "type occurrence" ("value occurrence") to denote an occurrence of a type identifier (value identifier) of some structure in a type expression (in an expression).

2.5 Design Decisions Motivated

While developing our translation from ML modules to Haskell type classes, we have made (at least) two critical design decisions: associated type synonyms represent type components of signatures and structures, and abstract associated type synonyms encode abstract types. In this section, we discuss and evaluate other options for translating these two features.

To encode abstract types, we see two alternative approaches. Firstly, we could use Haskell's module system. It enables abstract types by wrapping them in a `newtype` constructor and placing them in a separate module that hides the constructor. This solution is unsatisfactory for two reasons: (i) Explicit conversion code is necessary to turn a value of the concrete type into a value of the abstract type and vice versa. (ii) We do not want Haskell's module system to interfere with our comparison of ML modules and Haskell type classes.

Secondly, we could use existentials [15,16] to encode abstract types. Fig. 5 shows the translation of the ML structure `IntSet'` from Fig. 2 for this approach. The type `IntSetAbs` hides the concrete identity of `Set a` by existentially quantifying over `a`. The constraint `Elem a ~ Int` ensures that the types `Elem a` and `Int` are equal [17]. In the following instance declaration, we use `IntSetAbs` to define the `Set` type and implement the methods of the instance by delegating the calls to the `SetSig` instance hidden inside `IntSetAbs`.

Haskell

```
data IntSetAbs = forall a. (SetSig a, Elem a ~ Int) => IntSetAbs a (Set a)
data IntSet'' = IntSet''
instance SetSig IntSet'' where
    type Elem IntSet'' = Int;  type Set IntSet'' = IntSetAbs
    empty _ = IntSetAbs IntSet (empty IntSet)
    member _ i (IntSetAbs a s) = member a i s
    insert _ i (IntSetAbs a s) = IntSetAbs a (insert a i s)
```

Figure 5. Alternative encoding of abstract types with Haskell's existential types

There is, however, a major problem with the second approach: It is unclear how to translate functions whose type signatures contain multiple occurrences of the same abstract type in argument position. For example, suppose the signature of structure `IntSet'` (Fig. 2) contained an additional function `union : set -> set -> set`. The translation in Fig. 5 then also had to provide a method `union` of type `IntSetAbs -> IntSetAbs -> IntSetAbs`. But there is no sensible way to implement this method because the first and the second occurrence of `IntSetAbs` may hide *different* set representation types.[6]

An obvious alternative to associated type synonyms for representing ML's type components are multi-parameter type classes [14] together with functional dependencies [18]. In this setting, every type component of an ML signature would be encoded as an extra parameter of the corresponding Haskell type class, such that the first parameter uniquely determined the extra parameters. Nevertheless, there are good reasons for using associated type synonyms instead of extra type class parameters: (i) The extra parameters are referred to by position; however, ML type components (and associated type synonyms) are referred to by name. (ii) Functional dependencies provide no direct support for abstract types, whereas a simple and lightweight generalization of associated type synonyms enables them. (Using existential types with functional dependencies has the same problem as discussed in the preceding paragraph.) Moreover, associated type synonyms are becoming increasingly popular and are already available in the development version of the most widely used Haskell compiler, GHC [11].

3 From Classes to Modules

The translation from Haskell type classes to ML modules encodes type classes as signatures and instances of type classes as functors that yield structures of these signatures. It makes use of two extensions to Standard ML, both of which are implemented in Moscow ML [19]: recursive functors [20,21] model recursive instance declarations, and first-class structures [22] serve as dictionaries providing runtime evidence for type-class constraints. We first explain how to use first-class structures as dictionaries (Sec. 3.1). Then we show ML encodings of type

[6] The situation is similar for Java-style interfaces: two occurrences of the same interface type may hide two different concrete class types.

Haskell

```
class Eq a          where eq :: a -> a -> Bool
class Eq a => Ord a where lt :: a -> a -> Bool
```

ML

```
signature Eq  = sig type t  val eq : t -> t -> bool     end
signature Ord = sig type t  val lt : t -> t -> bool
                    val superEq : [Eq where type t = t] end
```

Figure 6. Haskell type classes `Eq` and `Ord` and their translations to ML

class declarations (Sec. 3.2), of overloaded functions (Sec. 3.3), and of instance declarations (Sec. 3.4). Finally, we summarize our results (Sec. 3.5).[7]

3.1 First-class Structures as Dictionaries

Dictionary translation [2,23,24,25] is a technique frequently used to eliminate overloading introduced by type classes. Using this technique, type-class constraints are turned into extra parameters, so that evidence for these constraints can be passed explicitly at runtime. Evidence for a constraint comes as a dictionary that provides access to all methods of the constraint's type class.

The translation from Haskell type classes to ML modules is another application of dictionary translation. In our case, dictionaries are represented as first-class structures [22], an extension to Standard ML that allows structures to be manipulated on the term level. This article uses first-class structure as implemented in Moscow ML [19].

We need to explicitly convert a structure into a first-class structure and vice versa. Suppose `S` is a signature, and `s` is a structure of signature `S`. Then the construct `[structure s as S]` turns `s` into a first-class structure of type `[S]`. Such types are called *package types.* Conversely, the construct `let structure X as S = e1 in e2 end`, where the expression `e1` is expected to have type `[S]`, makes the structure contained in `e1` available in `e2` under the name `X`.

Clearly, there are alternative representations for dictionaries in ML; for example, we could use records with polymorphic fields, as featured by OCaml [26]. We are, however, interested in a comparison between Haskell-style type classes and ML's *module system*, so we do not pursue this approach any further.

3.2 Translating Type Class Declarations

Fig. 6 shows two Haskell type classes `Eq` and `Ord`, which provide overloaded functions `eq` and `lt`. We translate these classes into ML signatures of the same name. Thereby, the type variable `a` in the class head is mapped to an opaque type specification `t`, and the methods of the class are translated into value specifications. The signature `Ord` has an additional value specification `superEq` to account for the superclass `Eq` of `Ord`. Consequently, `superEq` has type `[Eq where type t = t]` which represents a dictionary for `Eq` at type `t`.

[7] We use Haskell 98 [10] in this section; the ML code runs under Moscow ML [19].

Haskell

```
elem :: Eq a => a -> [a] -> Bool
elem x l = any (eq x) l
```

ML

```
fun elem d (x:'a) l = let structure D as Eq where type t = 'a = d
                      in any (D.eq x) l end
```

Figure 7. Overloaded function in Haskell and its translation to ML

3.3 Translating Overloaded Functions

Fig. 7 shows the Haskell function `elem`, which uses the `eq` method of class `Eq`. Hence, the constraint `Eq a` needs to be added to the (optional) type annotation of `elem` to limit the types that can be substituted for `a` to instances of `Eq`.

As already noted in Sec. 3.1, such a constraint is represented in the ML version of `elem` as an additional parameter `d` which abstracts explicitly over the dictionary for the constraint `Eq a`. Hence, the type of `elem` in ML is `[Eq where type t = 'a] -> 'a -> 'a list -> bool`.

In the body of `elem`, we open the first-class structure `d` and bind the content to the structure variable `D`, so that we can access the equality comparison function as `D.eq`. Note that we cannot do without the type annotation `(x:'a)`: It introduces the lexically scoped type variable `'a` used in the signature required for opening `d`. (Lexically scoped type variables are part of Standard ML.)

3.4 Translating Instance Declarations

Finally, we turn to the translation of instance declarations. The Haskell code in Fig. 8 makes the type `Int` an instance of the type classes `Eq` and `Ord`. Furthermore, it specifies that lists can be compared for equality as long as the list elements can be compared for equality. This requirement is expressed by the constraint `Eq a` in the context of the instance declaration for `Eq [a]`. (The constraints to the left of the double arrow `=>` are called the *context*; the part to the right is called the *head*. The double arrow is omitted if the context is empty.)

The functors `EqInt` and `OrdInt` are translations of the instances `Eq Int` and `Ord Int`, respectively. These two functors do not take any arguments because the contexts of the corresponding instance declarations are empty. (We could use structures instead of functors in such cases; however, for reasons of consistency we decided to use functors even if the instance context is empty.) The definition of the `superEq` component in `OrdInt` demonstrates that dictionaries are created by coercing structures into first-class structures.

The translation of the instance declaration for `Eq [a]` is more interesting because the Haskell version is recursive (through the expression `eq xs ys` in the second equation of `eq`) and has a non-empty context. Consequently, the functor `EqList` for this instance has to be defined recursively and takes an argument of signature `Eq` corresponding to the constraint `Eq a` in the instance context.

To encode recursive functors, we use Moscow ML's recursive structures [20,21]. We first define an auxiliary structure `R` that contains a definition of the desired

Haskell

```
instance Eq Int where eq = intEq
instance Ord Int where lt = intLt
instance Eq a => Eq [a] where eq []     []     = True
                              eq (x:xs) (y:ys) = eq x y && eq xs ys
                              eq _      _      = False
```

ML

```
functor EqInt()  = struct type t = int  val eq = intEq end
functor OrdInt() = struct type t = int  val lt = intLt
                          val superEq = [structure EqInt()
                                          as Eq where type t = t] end
structure R = rec
  (R' : sig functor F : functor (X: Eq) -> Eq where type t = X.t list end)
struct functor F(X: Eq) =
         struct type t = X.t list
                fun eq []      []      = true
                  | eq (x::xs) (y::ys) =
                      let structure Y as Eq where type t = t
                          = [structure R'.F(X) as Eq where type t = t ]
                       in X.eq x y andalso Y.eq xs ys end
                  | eq _       _       = false
         end
end
functor EqList(X: Eq) = R.F(X)
```

Figure 8. Instance declarations in Haskell and their translations to ML

functor `F`. The keyword **`rec`** together with the forward declaration `(R' : ...)` makes the content of `R` available inside its own body. In the definition of `eq`, we use `R'.F` to invoke the functor recursively, pack the result as a first-class structure, immediately open this structure again, and bind the result to the variable `Y`. Now we can use `Y.eq` to compare `xs` and `ys`. The combination of pack and open operations is necessary to interleave computations on the term level with computations on the module level; it is not possible to invoke `R'.F(X).eq` directly. After the definition of `R`, we define `EqList` by invoking `R.F`.

It may seem awkward to use recursive functors in ML to encode recursive Haskell functions. Indeed, for the example just discussed, a recursive ML function would be sufficient. In general, however, it is possible to write polymorphic recursive functions [27] with Haskell type classes. For such cases, we definitely need to encode recursion in terms of recursive functors because polymorphic recursion is not available on the term level of Standard ML.[8]

3.5 Summary

We summarize the (informal) translation from Haskell type classes to ML modules in Table 2. Note that dictionaries are not part of Haskell's surface syntax;

[8] Extending Standard ML's term language with polymorphic recursion is an alternative option. For Haskell type classes, polymorphic recursion comes "for free" because class declarations provide explicit type information.

Haskell	ML
type class declaration	signature
class method	value component
superclass	superclass dictionary
dictionary	first-class structure
(recursive) instance declaration	(recursive) functor
constraint in instance context	argument to the corresponding instance functor
overloaded function	function with additional dictionary parameter(s)

Table 2. Informal mapping from Haskell type classes to ML modules

they only become manifest when evidence for constraints is made explicit by our translation technique.

4 Formalization and Implementation

So far, all we did was apply the translations between ML modules and Haskell type classes to some examples. How do we know that the translations work in general and not only for our examples? To answer this question, we have formalized the two translations, proved that they preserve types, and provided implementations for them. For space reasons, we only describe the source and target languages of the formalized translations. All the other details, all proofs, and the implementations are part of the first author's diploma thesis [13].

The source language of the formalized translation from modules to classes is a subset of Standard ML [9], featuring all important module language constructs except nested structures. The target language of the translation is Haskell 98 [10] extended with multi-parameter type classes [14], associated type synonyms [6], and abstract associated type synonyms (a contribution of the work at hand).

The translation from classes to modules uses a source language that supports type classes in the style of Haskell 98, but without constructor classes, class methods with constraints, and default definitions for methods. The target language of this translation is a subset of Standard ML extended with first-class structures [22] and recursive functors [20,21].

5 Discussion

Having developed translations from ML modules to Haskell type classes and vice versa, we now present a thorough comparison between the two concepts. Sec. 5.1 discusses how Haskell type classes perform as a replacement for ML modules. Sec. 5.2 changes the standpoint and evaluates how ML modules behave as an alternative to Haskell type classes.

5.1 Classes as Modules

Namespace management. ML modules provide proper namespace management, whereas Haskell type classes do not: It is not possible that two different type classes (in the same Haskell module) declare members of the same name.

Signature and structure components. Signatures and structures in ML may contain all sorts of language constructs, including substructures. Type classes and instances in Haskell 98 may contain only methods; extensions to Haskell 98 also allow type synonyms [6] and data types [5]. However, there exists no extension that allows nested type classes and instances.

Sequential vs. recursive definitions. Definitions in ML are type checked and evaluated sequentially, with special support for recursive data types and recursive functions. In particular, cyclic type abbreviations are disallowed. In Haskell, all top-level definitions are mutually recursive, so associated type synonyms must impose extra conditions to prevent the type checker from diverging while expanding their definitions. For our purpose, the original termination conditions [6] are too restrictive. Nevertheless, no program in the image of our translation from modules to type classes causes the type checker to diverge because the sequential nature of type abbreviations carries over to associated type synonym definitions.

Implicit vs. explicit signatures. In ML, signatures of structures are inferred implicitly. In Haskell, the type class to which an instance declaration belongs has to be stated explicitly. However, once recursive modules are introduced, ML also requires explicit signatures, so the difference between implicit and explicit signatures interplays with the preceding point of our comparison.

Anonymous vs. named signatures. Signatures in ML are essentially anonymous because named signatures can be removed from the language without losing expressiveness. Haskell type classes cannot be anonymous.

Structural vs. nominal signature matching. The difference between anonymous and named signatures becomes relevant when we compare signature matching in ML with its Haskell counterpart. In ML, matching a structure against a signature is performed by comparing the structure and the signature componentwise; the names of the structure and the signature—if present at all—do not matter. This sort of signature matching is often called *structural* matching. Our Haskell analog of signature matching is verifying whether the type representing a structure is an instance of the type class representing the signature. The name of a class is crucial for this decision. Therefore, we characterize our Haskell analog of signature matching as *nominal*.

Abstraction. In ML, abstraction is performed by sealing a structure with a translucent or opaque signature. In Haskell, we perform abstraction inside instance declarations through abstract associated type synonyms.

Unsealed and sealed view. A sealed structure in ML may look different depending on whether we view its body from inside or outside the signature seal: Inside, more values and types may be visible, some types may be concrete, and some values may have a more polymorphic type than outside. For our Haskell analog, the same set of types and values is visible and a value has the same type, regardless of whether we view the instance from inside or outside.

Translucent vs. opaque signatures Translucent signatures (signatures with both concrete and abstract type components) are a key feature of ML's module system. Signatures in Haskell (*i.e.*, type classes) may be classified as opaque because they do not provide definitions for type components (*i.e.*, associated type synonyms).[9]

First-class structures. First-class structures are a nontrivial extension to Standard ML [22]. In our representation of structures as data types and instance declarations, we get first-class structures for free, provided we only use top-level structures as first-class entities. This restriction is necessary because instance declarations in Haskell have to be top-level. All examples given by Russo [22,21] meet this restriction.

5.2 Modules as Classes

Implicit vs. explicit overloading resolution. Overloading in Haskell is resolved implicitly by the compiler. When type classes are simulated with ML modules, overloading has to be resolved explicitly by the programmer, which leads to awkward and verbose code.

Constructor classes. Our current translation scheme is unable to handle constructor classes because there is not direct counterpart of Haskell's higher-oder types in ML. We consider it as interesting future work to investigate whether an encoding of higher-order types as functors would enable a translation of constructor classes to ML modules.

Recursive classes. Type classes in Haskell may be recursive in the sense that a class can be used in a constraint for a method of the same class. We cannot translate such recursive classes to ML because signatures cannot be recursive.

Default definitions for methods. Haskell type classes may contain default definitions for methods. With our approach, such default definitions cannot be translated properly to ML because signatures specify only the types of value components and cannot contain implementations of value components.

Associated type synonyms. Type components in ML are similar to associated type synonyms in Haskell, but it is unclear whether they have the same expressivity as their Haskell counterpart. For example, consider Chakravarty and colleagues' use of associated type synonyms to implement a string formatting function. Their function `sprintf` has type `Format fmt => fmt -> Sprintf fmt`, where `Format` is a type class with an associated type synonym `Sprintf`. Given the translation presented in this article, we would use a first-class structure to encode the constraint `Format fmt` in ML. The translation of the result type `Sprintf fmt` would then require access to the type component of this structure that corresponds to the associated type synonym `Sprintf`. It is not clear how this can be realized.

[9] Default definitions for associated type synonyms do not help here because they may change in instance declarations.

6 Related Work

There is only little work on connecting modules with type classes. None of these works meet our goal of comparing ML modules with Haskell type classes based on formal translations.

The work closest to ours is Dreyer and colleagues' modular reconstruction of type classes [8]. This work, which strictly speaking came after our own [13], extends Harper & Stone's type-theoretic interpretation of modules [28] to include ad-hoc polymorphism in the style of Haskell type classes. Instead of adding an explicit notion of type classes to ML, certain forms of module signatures take the role of class declarations and matching modules may be nominated as being canonical for the purpose of overload resolution. The presented elaboration relation mirrors Haskell's notion of an evidence translation and is related to our translation of Haskell classes into ML modules. Dreyer and colleagues do not consider the converse direction of modeling modules by type classes.

Kahl and Scheffczyk [4] propose named instances for Haskell type classes. Named instances allow the definition of more than one instance for the same type; the instances are then distinguished by their name. Such named instances are not used automatically in resolving overloading; however, the programmer can customize overloading resolution by supplying them explicitly. Kahl and Scheffczyk motivate and explain their extension in terms of OCaml's module system [29,26]; they do not consider any kind of translation from ML modules to Haskell type classes or vice versa.

Shan [30] presents a formal translation from a sophisticated ML module calculus [31] into System F_ω [32]. The source ML module calculus is a unified formalism that covers a large part of the design space of ML modules. The target language System F_ω of Shan's translation can be encoded in Haskell extended with higher-rank types [33]; however, this encoding is orthogonal to the type class system. Kiselyov builds on Shan's work and translates a particular applicative functor into Haskell with type classes [34]. However, he does not give a formal translation, so it is unclear whether his approach works in general. Neither Shan nor Kiselyov consider translations from type classes to modules.

Schneider [3] adds Haskell-style type classes to ML. His solution is conservative in the sense that type classes and modules remain two separate concepts. In particular, he does not encode type classes as modules. Translations in the opposite direction are not addressed in his work.

Jones [35] suggests record types with polymorphic fields for modular programming. These record types do not support type components but explicit type parameterization. Jones then uses parametric polymorphism to express ML's sharing constraints and to abstract over concrete implementation types. His system supports first-class structures and higher-order modules.

Nicklisch and Peyton Jones [36] compare ML's with Haskell's module system. They report that the simple namespace mechanism offered by Haskell can compete with the module system offered by ML in many real–world applications. Moreover, they integrate Jones approach [35] into Haskell to find that the resulting system exceeds ML's module system in some cases.

7 Conclusion

This article demonstrates how to translate essential features of ML modules to Haskell type classes and vice versa. Both translations come with a formalization, a proof of type preservation, and an implementation. Based on the two translations, the article presents a thorough comparison between ML modules and Haskell type classes.

Acknowledgments. We thank the reviewers of FLOPS 2008 and APLAS 2008 for their detailed comments.

References

1. Kaes, S.: Parametric overloading in polymorphic programming languages. In Ganzinger, H., ed.: Proc. 2nd ESOP. Number 300 in LNCS, Springer (1988) 131–144
2. Wadler, P., Blott, S.: How to make ad-hoc polymorphism less ad-hoc. In: Proc. 16th ACM Symp. POPL, Austin, Texas, ACM Press (January 1989) 60–76
3. Schneider, G.: ML mit Typklassen. Master's thesis, Universität des Saarlandes (2000) `http://www.ps.uni-sb.de/Papers/abstracts/Schneider2000.html`.
4. Kahl, W., Scheffczyk, J.: Named instances for Haskell type classes. In Hinze, R., ed.: Proceedings of the 2001 Haskell Workshop. (2001)
5. Chakravarty, M., Keller, G., Peyton Jones, S., Marlow, S.: Associated types with class. In Abadi, M., ed.: Proc. 32nd ACM Symp. POPL, Long Beach, CA, USA, ACM Press (January 2005) 1–13
6. Chakravarty, M., Keller, G., Peyton Jones, S.: Associated type synonyms. In Pierce, B.C., ed.: Proc. ICFP 2005, Tallinn, Estonia, ACM Press, New York (September 2005) 241–253
7. Rossberg, A.: Post to the alice-users mailing list. `http://www.ps.uni-sb.de/pipermail/alice-users/2005/000466.html` (May 2005)
8. Dreyer, D., Harper, R., Chakravarty, M.: Modular type classes. In Felleisen, M., ed.: Proc. 34th ACM Symp. POPL, Nice, France, ACM Press (January 2007) 63–70
9. Milner, R., Tofte, M., Harper, R., MacQueen, D.: The Definition of Standard ML (Revised). MIT Press (1997)
10. Peyton Jones, S., ed.: Haskell 98 Language and Libraries, The Revised Report. Cambridge University Press (2003)
11. GHC: The Glasgow Haskell compiler. `http://www.haskell.org/ghc/` (2008)
12. Jones, M.P., Peterson, J.: The Hugs 98 user manual. `http://www.haskell.org/hugs/` (1999)
13. Wehr, S.: ML modules and Haskell type classes: A constructive comparison. Master's thesis, Albert-Ludwigs-Universität Freiburg (November 2005) `http://www.informatik.uni-freiburg.de/~wehr/publications/Wehr2005.html`.
14. Peyton Jones, S., Jones, M., Meijer, E.: Type classes: An exploration of the design space. In Launchbury, J., ed.: Proc. of the Haskell Workshop, Amsterdam, The Netherlands (June 1997)
15. Mitchell, J.C., Plotkin, G.D.: Abstract types have existential types. ACM Trans. Prog. Lang. and Systems **10**(3) (July 1988) 470–502
16. Läufer, K.: Type classes with existential types. J. Funct. Program. **6**(3) (1996) 485–517

17. Chakravarty, M.M.T., Keller, G., Peyton Jones, S.: Associated type synonyms. In Pierce, B.C., ed.: Proc. ICFP 2005, Tallinn, Estonia, ACM Press, New York (September 2005) 241–253
18. Jones, M.P.: Type classes with functional dependencies. In Smolka, G., ed.: Proc. 9th ESOP. Number 1782 in LNCS, Berlin, Germany, Springer (March 2000) 230–244
19. Romanenko, S., Russo, C., Kokholm, N., Larsen, K.F., Sestoft, P.: Moscow ML homepage. `http://www.dina.dk/~sestoft/mosml.html` (2007)
20. Crary, K., Harper, R., Puri, S.: What is a recursive module? In: Proc. 1999 PLDI, Atlanta, Georgia, USA (May 1999) 50–63 Volume 34(5) of SIGPLAN Notices.
21. Russo, C.V.: Recursive structures for Standard ML. In Leroy, X., ed.: Proc. 2001 ICFP, Florence, Italy, ACM Press, New York (September 2001) 50–61
22. Russo, C.V.: First-class structures for Standard ML. In Smolka, G., ed.: Proc. 9th ESOP. Number 1782 in LNCS, Berlin, Germany, Springer (March 2000) 336–350
23. Jones, M.P.: Qualified Types: Theory and Practice. Cambridge University Press, Cambridge, UK (1994)
24. Hall, C.V., Hammond, K., Peyton Jones, S.L., Wadler, P.L.: Type classes in Haskell. ACM Trans. Prog. Lang. and Systems **18**(2) (1996) 109–138
25. Faxén, K.F.: A static semantics for Haskell. J. Funct. Program. **12**(4&5) (July 2002) 295–357
26. OCaml: Objective Caml. `http://caml.inria.fr/ocaml/index.en.html` (2007)
27. Henglein, F.: Type inference with polymorphic recursion. ACM Trans. Prog. Lang. and Systems **15**(2) (April 1993) 253–289
28. Harper, R., Stone, C.: A type-theoretic interpretation of Standard ML. In Plotkin, G., Stirling, C., Tofte, M., eds.: Proof, Language, and Interaction: Essays in Honor of Robin Milner. MIT Press (2000)
29. Leroy, X.: Applicative functors and fully transparent higher-order modules. In: Proc. 1995 ACM Symp. POPL, San Francisco, CA, USA, ACM Press (January 1995) 142–153
30. Shan, C.: Higher-order modules in System F_ω and Haskell. `http://www.eecs.harvard.edu/~ccshan/xlate/` (July 2004)
31. Dreyer, D., Crary, K., Harper, R.: A type system for higher-order modules. In Morrisett, G., ed.: Proc. 30th ACM Symp. POPL, New Orleans, LA, USA, ACM Press (January 2003) 236–249 ACM SIGPLAN Notices (38)1.
32. Girard, J.Y.: Interpretation Fonctionnelle et Elimination des Coupures dans l'Arithmetique d'Ordre Superieur. PhD thesis, University of Paris VII (1972)
33. Peyton Jones, S., Vytiniotis, D., Weirich, S., Shields, M.: Practical type inference for arbitrary-rank types. J. Funct. Program. **17**(1) (2007) 1–82
34. Kiselyov, O.: Applicative translucent functors in Haskell. Post to the Haskell mailing list, `http://www.haskell.org/pipermail/haskell/2004-August/014463.html` (August 2004)
35. Jones, M.P.: Using parameterized signatures to express modular structure. In: Proc. 1996 ACM Symp. POPL, St. Petersburg, FL, USA, ACM Press (January 1996)
36. Nicklisch, J., Peyton Jones, S.: An exploration of modular programs. In: Proc. 1996 Glasgow Workshop on Functional Programming. (July 1996) `http://www.dcs.gla.ac.uk/fp/workshops/fpw96/Nicklisch.pdf`.