Modular Generic Programming with Extensible Superclasses

Martin Sulzmann
School of Computing, National University of Singapore
S16 Level 5, 3 Science Drive 2, Singapore 117543
sulzmann@comp.nus.edu.sg

Meng Wang
School of Computing, National University of Singapore
S16 Level 5, 3 Science Drive 2, Singapore 117543
wangmeng@comp.nus.edu.sg

Abstract

"Generics for the Masses" (GM) and "Scrap your Boilerplate" (SYB) are generic programming approaches based on some ingenious applications of Haskell type classes. To achieve modularity, the GM and SYB approach have been extended by using some experimental language extensions such as abstraction over type classes and recursive instances. Hence, the type class encodings behind the GM and SYB approach become less practical and harder to understand.

We show that none of these type class features are necessary if we use the single feature of extensible superclasses, the complement of subclass extension. We formalize type classes with extensible superclasses as the combination of a previously introduced type-passing translation scheme and a general type class framework. Our results shed some new light on the use of type classes to support generic programming.

1. Introduction

Generic programming is a style of programming where a single *generic* function definition is applicable to a wide range of data types. This is in contrast to a *ad-hoc polymorphic* function which provides a separate definition for each data type. There are a number of compelling approaches which exploit ad-hoc polymorphism to support generic programming.

Prominent examples are "Generics for the Masses" (GM) [9] by Hinze and "Scrap your Boilerplate" (SYB) [17] by Lämmel and Peyton Jones. These works employ Haskell type classes [22, 28], which are an elegant formulation to support ad-hoc polymorphism as an extension of Hindley/Milner. Besides Haskell, type classes can also be found in a number of other languages such as Clean [23], HAL [2] and Mercury [8, 14].

Unfortunately, the GM approach requires to update the class declarations with new method definitions for each ad-hoc type case. The consequence is that the GM approach is not modular. In some recent work [21] this problem has been addressed, by either relying on specific dispatcher functions for each generic function or using some non-standard type class extensions such as undecidable instances [4]. We believe that the proposed solutions are not entirely satisfactory and may affect the understandability of the GM approach.

The SYB approach has similar problems when it comes to the modular extension of generic functions. In [18], non-standard type class features such as type class abstraction [11] and recursive instances (a.k.a. recursive dictionaries, co-inductive type classes) [26] are employed to support modular, extensible generic functions.

In this paper, we take a fresh look at generic programming with type classes. Our contributions are as follows:

- We show that the GM and SYB approach can be made modular by employing *extensible superclasses*, a type class feature which supports the incremental extension of superclasses (Sections 5 and 4). In our opinion, extensible superclasses provide for a much more natural solution to the "modularity" problem.
- We formalize extensible superclasses using a combination of a previously proposed type-passing translation scheme by Thatte and our own Constraint Handling Rules based type class framework (Section 6).

In Section 7 we discuss further related work. We conclude in Section 8.

The GM and SYB approach make heavy use of type classes. Hence, we first give an introduction to Haskell type classes in the next section. Unless otherwise stated, we assume Haskell 98 type classes [22]. We implement a library which supplies evaluation and printing functions for a simple arithmetic expression language using type classes in the most straightforward way. In case we extend the functionality of our library, we wish to maintain close relations among the set of instances provided. But this requires to update existing class declarations which in turn forces us to recompile the entire program. Hence, modularity is broken. For very similar reasons, the GM and SYB approach struggle to achieve modularity.

In Section 3, we investigate why changing class declarations breaks modularity by taking a closer look at the dictionary-passing translation scheme [5, 26] underlying Haskell implementations. Haskell implementations support the modular extension of subclasses but *not* superclasses. If we switch to a type-passing translation scheme, we can incrementally introduce further superclasses without having to recompile existing code. This is the essence of our method to achieve modularity for the GM and SYB approach.

2. Haskell Type Classes

We start off with the definition and specific implementations of the evaluation function for a simple expression language.

```
-- expression language
data Lit = Lit Int
data Plus a b = Plus a b
```

```
-- evaluator
class Eval a where
  eval :: a->Int
instance Eval Lit where
  eval (Lit n) = n
instance (Eval a, Eval b) => Eval (Plus a b) where
  eval (Plus a b) = eval a + eval b
```

The class declaration states that class `Eval` has a function `eval` (also known as method) to evaluate values of type `t` to integers. A type class constraint such as `Eval t` expresses that type `t` is a member of class `Eval`. The instance declarations provide specific implementations of `eval` on data types `Lit` and `Plus a b`. Notice that the last instance declaration states that we can build an instance of `eval` on type `Plus a b` if we can provide definitions for the calls `eval a` and `eval b`. We commonly refer to `(Eval a, Eval b)` as the instance *context* and to `Eval (Prod a b)` as the instance *head*.

It is straightforward to define new cases by providing new instances.

```
data Minus a b = Minus a b
instance (Eval a, Eval b) => Eval (Minus a b) where
  eval (Minus a b) = eval a - eval b
```

We can also easily introduce new functions. For convenience, we omit the instance bodies in the program text below. They do not matter here.

```
class Print a where
  print :: a -> String
instance Print Lit
instance (Print a,Print b) => Print (Plus a b)
instance (Print a,Print b) => Print (Minus a b)
instance Print a => Print [a]                -- (L)
```

The addition of the new cases and functions does not require to recompile any existing code. Hence, type classes support the modular extension of libraries.

The trouble is that `eval` and `print`, respectively their type classes `Eval` and `Print`, are only loosely connected. We support printing on lists, see instance (L), but there is no such instance for `Eval`. This may result in some unexpected behavior for the user. For example, consider the following user program.

```
literals = [Lit 1, Lit 2]
evalAndprint = (eval literals, print literals)
```

We attempt to use `eval` and `print` on type `[Lit]` In case of `eval`, there is no definition that deals with this case. In Haskell speak, the above yields an unresolved instance error `Eval [Lit]`. We would much prefer to catch this error at the definition rather than use site. What we would like is to guarantee that `eval` and `print` are defined for the exact same set of instances.

In Haskell, we can give such guarantees via subclassing. We replace `Print`'s class declaration as follows.

```
class Eval a => Print a where print :: a -> String
```

This declaration defines `Print` to be a subclass of `Eval`. Or putting it the other way around, `Eval` is declared to be a superclass of `Print`. Then, any Haskell implementation such as GHC [4] or HUGS [12] will complain that there is an instance for `Print [a]`, see (L) above, but such an instance is missing for `Eval`.

Subclassing effectively guarantees that the set of instances of the superclass are a subset of the set of instances of the subclass. Hence, it is only natural to replace `Eval`'s class declaration by

```
class Print a => Eval a where eval :: a -> Int
```

Thus, we can guarantee that for every `print` instance there is a `eval` instance and vice versa. We have achieved our goal to catch the error at the definition site. But there is a serious problem.

Haskell currently prohibits recursive subclass relations. But this restriction can be safely lifted as we will argue later in Section 6.2. The real problem is that our use of subclassing breaks modularity. For example, at some later stage we may want to introduce

```
class Size a where size :: a -> Int
```

to compute the size of expressions. We leave out the instances for simplicity. We would like to ensure that `size` operates on the exact same set of types as `eval` and `print`. To impose this condition, we will need to alter `eval`'s and `print`'s class declarations.

```
class (Eval a, Size a)  => Print a
   where print :: a -> String
class (Print a, Size a) => Eval a
   where eval :: a -> Int
class (Eval a, Print a) => Size a
   where size :: a -> Int
```

But changing existing class declarations means we need to recompile existing code. Hence, modularity is broken. To better understand why this is the case and how to fix the problem, we take a look at possible ways to translate type classes.

3. Translating Type Classes

3.1 Dictionary-Passing Translation Scheme

In Haskell, we translate type classes by representing them via dictionaries. These dictionaries hold the actual method definitions. Each superclass dictionary is part of its (direct) subclass dictionary. For example, the declarations

```
class Eval a where eval :: a -> Int
class Eval a => Print a where              -- (PE)
  print :: a -> String
instance Eval Lit                          -- (E1)
instance (Eval a, Eval b) =>
         Eval (Plus a b)                   -- (E2)
instance Print Lit                         -- (P1)
instance (Print a,Print b) =>
         Print (Plus a b)                  -- (P2)
```

imply the dictionary declarations

```
data EvalDict a = E (a -> Int)
data PrintDict a = P (EvalDict a) (a -> String)
```

We can thus easily access the superclass dictionary via its subclass dictionary. The actual construction of dictionaries is described by instance declarations. For example, instance (E1) shows that a dictionary for `Eval Lit` exists. Instance (E2) shows how to build a dictionary for `Eval (Plus a b)` given dictionaries for `Eval a` and `Eval b`.

More formally, class and instance declarations specify a type class proof system $\Delta \vdash$ `d : TCDict t` where the environment Δ holds the set of given dictionaries and dictionary value `d` of type `TCDict t` can be concluded from Δ with respect to a given set of class and instance declarations.

In Figure 1, we give the specific type class proof system resulting from the above declarations. Rule (Var) allows us to lookup assumptions from the environment. Each of the next four rules correspond to one of the four instance declarations. We assume the following dictionary construction functions.

(Var) $$\frac{(\texttt{d} : \texttt{TCDict t}) \in \Delta}{\Delta \vdash \texttt{d} : \texttt{TCDict t}}$$

(E1) $$\Delta \vdash \texttt{instE1} : \texttt{EvalDict Lit}$$

(E2) $$\frac{\Delta \vdash \texttt{d1} : \texttt{EvalDict a} \quad \Delta \vdash \texttt{d2} : \texttt{EvalDict b}}{\Delta \vdash \texttt{instP2 (d1, d2)} : \texttt{EvalDict (Plus a b)}}$$

(P1) $$\Delta \vdash \texttt{instP1} : \texttt{PrintDict Lit}$$

(P2) $$\frac{\Delta \vdash \texttt{d1} : \texttt{PrintDict a} \quad \Delta \vdash \texttt{d2} : \texttt{PrintDict b}}{\Delta \vdash \texttt{instP2 (d1, d2)} : \texttt{PrintDict (Plus a b)}}$$

(PE) $$\frac{\Delta \vdash \texttt{P d' print} : \texttt{PrintDict a}}{\Delta \vdash \texttt{d'} : \texttt{EvalDict a}}$$

Figure 1. Dictionary-Based Type Class Proof Rules

```
instE1 :: EvalDict Lit
instE2 :: (EvalDict a,EvalDict b) ->
          EvalDict (Plus a b)
instP1 :: PrintDict Lit
instP2 :: (PrintDict a,PrintDict b) ->
          PrintDict (Plus a b)
```

They follow from the instance declarations by turning type classes in the instance context into argument dictionaries. We leave out their actual definitions for simplicity. Rule (PE) shows how to extract the `Eval` dictionary from `Print`. For convenience, we use pattern matching syntax instead of case expressions.

The translation of programs is now straightforward. For example, the source program

```
f1 :: Print a => a -> Int
f1 x = eval x
```

is translated by turning type classes into additional dictionary arguments. Methods are replaced with some appropriate dictionary values.

```
f1 :: PrintDict a -> a -> Int
f1 d = case d of
        P d' print -> case d' of
                        E eval -> eval x
```

In the above, the class context `Print a` is turned into an additional argument. The call `eval` demands a dictionary for `Eval a` which we can supply by extraction from the dictionary of `Print a`. In terms of the type class proof system, we can express this statement as follows:

$$\{\texttt{d} : \texttt{PrintDict a}\} \vdash \texttt{case d of P d' print} \rightarrow \texttt{d'} : \texttt{EvalDict a}$$

We use case expressions instead of the (short-hand) pattern matching syntax.

The next example makes use of dictionary constructors belonging to instances in the translation. The program

```
f2 :: (Eval a, Eval b) => a -> b -> Int
f2 x y = eval (Plus x y)
```

translates to

```
f2 :: (EvalDict a, EvalDict b) -> a->b->Int
f2 (d1, d2) x y = case (instP2 (d1,d2)) of
                    P _ eval -> eval (Plus x y)
```

The source program demands a dictionary for `Eval (Plus a b)` which we can supply by applying the dictionary constructor `instP2` to the two dictionaries supplied by the annotation.

Here comes the important observation. In a dictionary-passing translation scheme, we can introduce new instances and subclasses without having to recompile existing code. But if we include a new superclass, say `Size`, in `Print`'s class declaration, we need to change the definition of existing superclass extraction rules such as (PE). Hence, programs such as function `f1` need to be recompiled.

3.2 Type-Passing Translation Scheme

Let's see how to translate some of the above programs based on Thatte's type-passing translation scheme [27]. We use a System F style target language extended with type case as found in intentional type analysis [7]. The main idea is to pass around types instead of dictionaries. These types are made available anyway by the standard translation of Hindley/Milner to System F [6].

Then, the program from before

```
f1 :: Print a => a -> Int
f1 x = eval x
```

translates to

```
f1 = Λa. λx:a. eval a x
```

None of the type classes are turned into dictionaries. We simply erase them. We directly use the type `a` to lookup the specific method definition. In the translation, function `f1` receives a type argument, indicated by the "big" lambda Λ`a`, and passes `a` to the method lookup function `eval` to select the appropriate method. The details of `eval` are given below.

The important insight is that if at some later stage we introduce a further superclass of class `Print` or `Eval`, the target program of function `f1` remains unchanged and does not need to be recompiled. Hence, a type-passing translation scheme naturally supports the modular extension of superclasses. This is the essential feature we propose to simplify the "modular" type class encodings of the GM and SYB approach.

We yet need to discuss how to translate instances. In a dictionary-passing translation scheme each instance declaration is compiled into a separate dictionary constructing function. In a type-passing translation, we need to lump together all these instances. That is, for each method we need one central method lookup function to access the type-specific method definitions. Here is the method lookup function belonging to the above instance declarations of the `Eval` class.

```
eval = Λ a. typecase a of
               Lit -> ...
               Prod b c -> ...
```

Somebody may argue that this will break separate compilation in case we introduce new instances. But clearly we can compile the individual instances separately. Our assumption is that the linker collects the instance definitions (which are in pre-compiled form) and glues them together to build the method lookup function `eval`.

Another implementation detail is that the actual method definitions are only built at run-time. Hence, there may be a potential inefficiency if we use a type-passing translation scheme. For example, consider the situation where we need to built a definition for `Print [...[Lit]...]`. In Haskell's translation scheme, we may be able to built `Print [...[Lit]...]`'s dictionary based on given dictionaries more efficiently. Obviously, we can improve the translation by using dynamic programming techniques etc.

Besides supporting the modular extension of superclasses, the type-passing translation scheme has a further advantage For example, the program

```
g :: Eval (Plus a b) => a -> b -> Int
g x y = eval x + eval y
```

can be easily translated as follows

```
g = Λ a,b. λ x:a. λ y:b. eval a x + eval b y
```

The program text of g demands `Eval a` and `Eval b`. We can argue that these constraints follow (logically) from the constraint `Eval (Plus a b)` given by the annotation. The dictionary-passing translation scheme fails here because we must (constructively) build demanded dictionaries from given dictionaries. Rule (E2) in Figure 1 only tells us how to build `Eval (Plus a b)`'s dictionary given the dictionaries for `Eval a` and `Eval b`. But the other direction, necessary to translate function g, does not hold in general. The fact that under a type-passing translation scheme we can interpret instance declarations as "if-and-only-if" relations between instance context and instance head is interesting but not essential in our recast of the GM and SYB approach. Though, the ability to extend the set of superclasses is essential.

3.3 The Story So Far and The Next Steps

We summarize the main points. Under a dictionary-passing translation scheme, we cannot naturally support the extension of superclasses. Updating class declarations affects existing proofs, that is dictionaries, derived from the type class proof system. Hence, the program needs to be recompiled. Under a type-passing translation scheme, however, none of the existing proofs is affected if we introduce new superclasses. Hence, no recompilation is necessary.

In the upcoming sections, we revisit the GM and SYB approach and show that extensible superclasses is the crucial feature to support modularity. Existing Haskell implementations lack this feature. Therefore, the extensible version of GM [21] and SYB [18] rely on features such as abstraction over type classes and recursive instances. In essence, these features allow to mimic extensible superclasses under a dictionary-passing translation scheme. We will discuss this point in more detail for the SYB approach in Section 4. We argue that we can achieve modularity more directly by employing extensible superclasses. The type class encodings become more transparent and easier to maintain for the user.

The details of extensible superclasses are given in Section 6. We formalize them as an instance of our Constraint Handling Rules (CHRs) based type class framework married with Thatte's type-passing translation method. In such a system, we do not derive dictionaries from type class proofs. We only need to check that type class proofs are valid. Hence, we can give the user the flexibility to specify additional proof rules which are not necessarily connected to any class or instance declarations. We choose the CHR formalism to specify such proof rules. For example, the CHRs

```
rule Print a ==> Size a
rule Eval a ==>  Size a
```

declare `Size` to be a superclass of `Print` and `Eval`. Hence, the CHR notation `==>` can be read as "subclass of" which is somewhat contrary to the Haskell notation

```
class Size a => Eval a
class Size a => Print a
```

Though, subclassing corresponds logically to Boolean implication. Hence, `==>` is the more appropriate notation. Next, we take a look at the SYB and GM approach and make use of CHR proof rules to specify extensible superclasses.

4. Scrap Your Boilerplate

4.1 A Non-modular Encoding

The SYB approach allows to write generic functions via a combinator library. We take a look at an example taken from [18] to get a better understanding of the modular extension problem. We introduce a standard generic function to compute the size of a data structure.

```
gsize :: Data a => a -> Int
gsize t = 1 + sum (gmapQ gsize t)
```

The combinator `gmapQ` applies function `gsize` to each of the immediate children of `t`. The result is a list of these sizes which are then summed up and incremented by one to obtain the total size. In general, there are further combinators besides `gmapQ` to write generic functions other than queries. These combinators are methods of the `Data` class. For simplicity, we only consider the `gmapQ` combinator.

```
class Typable a => Data a where
  gmapQ :: (forall b. Data b => b -> r) -> a -> [r]
```

Notice that we give `gmapQ` a rank-2 type, an extension which is supported by GHC. Rank-2 types are necessary, for example see the upcoming instance (D) where we apply `f` on values of different type. The instances of the `Data` class can be derived automatically. Here are two of them.

```
instance Data Char where
  gmapQ f c = []
instance Data a => Data [a] where  -- (D)
  gmapQ f (x:xs) = [f x, f xs]
```

Each time we introduce a new generic function such as `gsize`, we define a new class `Size` with method `gsize`. Thus, we can easily specify type-specific behavior of `gsize` by providing a `Size` instance. Here is the straightforward representation in Haskell

```
class Size a where
  gsize :: a -> Int
-- specific instance
instance Size Name where ...
-- generic instance
instance Data t => Size t where         -- (S)
  gsize t = 1 + sum (gmapQ gsize t)
```

The last case is the default, generic case and defines the behavior on all types that do not match `Name`. This is an example of an overlapping instance, an extension supported by GHC. In the instance context we find `Data t` because of the call `gmapQ` in the instance body. Everything seems fine but the above program will not type check. The program text `gmapQ gsize` is the trouble maker. In this specific context, the combinator `gmapQ` expects (as its first argument) a function of type ∀`a.Data a=>a->Int` but the actual argument `gsize` has type ∀`a.Size a=>a->Int`. There is clearly a mismatch and therefore the program will not type check. As pointed out in [18], we can fix the problem by making `Size` a superclass of `Data`.

```
class (Typable a, Size a) => Data a where
  gmapQ :: (forall b. Data b => b -> r) -> a -> [r]
```

Then, type ∀`a.Size a=>a->Int` can be specialized to the type ∀`a.Data a=>a->Int` and therefore the above instance (S) type checks. The down-side is that we need to recompile all existing code that refers to type class `Data`. This will happen for each newly introduced generic function. The SYB authors have recognized this problem.

4.2 The SYB3 Solution

The key idea behind the solution proposed in [18] (also known as the SYB3 approach) is to employ *abstraction over type classes*, a feature that has been suggested by Hughes [11] in a different context. Here is the rewritten program using type class abstraction.

```
class (Typeable a, cxt a) => Data cxt a where
  gmapQ :: (forall b. Data cxt b => b -> r) ->
           a -> [r]
instance Data Size t => Size t where
  gsize t = 1 + sum (gmapQ gsize t)
```

Notice that variable `cxt` ranges over *type classes* instead of *types* (hence Hughes coined the term type class abstraction to refer to this feature). Then, the superclass described by `cxt` is not fixed when class `Data` is declared. Thus, we can instantiate `cxt` with `Size` later. See the instance declaration.

There are a number of further adjustments necessary. For example, abstraction over type classes introduces "ambiguous" types. Hence, the translation of programs may become ambiguous. Therefore, explicit type applications must be introduced. We cannot repeat all the details here and refer to the reader to [18].

The problem is that neither type class abstraction nor explicit type applications are features supported by any Haskell implementation. Although, it is possible to encode them by adding a auxiliary class `Sat` and type `Proxy`, the details are quite tricky and involve quite a bit of programmer effort in case we introduce new generic functions. As mentioned in [18], each generic function requires a record type, a `Sat` instance and a type proxy that needs to be inserted at the proper place. The encoding even makes it necessary to deal with another type class feature known as recursive instances. We briefly elaborate on this feature in Section 7.

4.3 Our Solution

In our proposed system of extensible superclasses, we can leave the declaration of class `Data` unchanged. We can introduce the new generic function `gsize` as presented in Section 4.1. All that is required is to impose the additional condition

```
rule Data a ==> Size a
```

which declares `Size` to be a superclass of `Data`.

The program is then accepted in our system. Here is the type-passing translation for the `gmapQ` and `gsize` instances.

```
gmapQ = Λ a r. λ f:(∀b.b -> r). λ x:a.
    typecase a of
      Char -> []
      [t] -> case x of (y:ys) -> [f t x, f [t] xs]
```

Each type case corresponds to one instance. The formal parameter `f` has a polymorphic type. Hence, in the translation we supply `f` with additional type arguments as in `[f t x, f [t] xs]`. The translation of `gsize` instances is similar.

```
gsize = Λ a. λ x:a. typecase a of
    Name -> ...
    t -> 1 + sum ((gmapQ a Int) gsize x)
```

In the generic case, the `gmapQ`-call is supplied with the type arguments `a` and `Int`.

A type-passing translation scheme allows us to keep type class relations flexible. We use dynamic type information to select the appropriate method definitions. Thus, our solution to obtain modular, extensible, generic functions is more light-weight (in terms of user programmer effort).

5. Generics for the Masses

Let's take a look at the GM approach where we will encounter very similar problems as in case of the SYB approach.

The main idea behind the GM approach is to provide a uniform representation of data types in terms of unit, sum and product types. Generic functions are defined in terms of this uniform rather than the concrete structural representation of a data type. The programmer only needs to maintain a type isomorphism between the uniform and concrete representation. Thus, there is no need to extend the (now generic) definition of functions in case we include new data types. The trouble is that we cannot override generic with specific (ad-hoc) behavior in a modular fashion. We will see shortly why.

Here is a (over-simplified) presentation of the GM approach applied to our example from Section 2. In the GM approach, each generic function is an instance of the class `Generic`.

```
data Lit = Lit Int
data Plus a b = Plus a b
data Iso a b = Iso {from :: a -> b,to :: b -> a}
class Generic g where
  lit :: g Lit
  plus :: g a -> g b -> g (Plus a b)
  view :: Iso a b -> g a -> g b
```

For simplicity, we assume that the concrete representations "literal" and "plus" are already part of the `Generic` class. They are structurally equivalent to the uniform representations for "unit" and "products" which are commonly found in the `Generic` class. Via the "view" function we can use the generic function on many Haskell data types given a type isomorphism between the data type and its structural representation. By including literal and plus from the start, we avoid defining some straightforward type isomorphisms which would make the whole presentation more noisy. We will see later in Section 5.1 an example of a type isomorphism. Notice that `g` in `Generic g` ranges over type constructors. This is an example of a constructor class [16] which is supported in Haskell.

Here is the generic definition of the evaluation function.

```
newtype Ev a = Ev{eval' :: a -> Int}
instance Generic Ev where
  lit = Ev (\x -> case x of Lit i -> i)
  plus a b =
    Ev (\p -> case p of
                (Plus x y) -> eval' a x + eval' b y)
  view iso a = Ev (\x -> eval' a (from iso x))
```

In order to use the evaluator on its familiar type, we need a "dispatcher" function to select the appropriate case of a generic function. The most straightforward approach is to use an ad-hoc polymorphic (therefore extensible) function.

```
class Rep a where
  rep :: Generic g => g a
instance Rep Lit where
  rep = lit
instance (Rep a,Rep b) => Rep (Plus a b) where
  rep = plus rep rep
eval :: Rep t => t -> Int
eval = eval' rep
```

The dispatcher function `rep` will select the appropriate generic case depending on the concrete type context. We can straightforwardly introduce new generic functions.

```
newtype Pr a = Pr{print' :: a -> String}
instance Generic Pr where
  lit = Pr (\x -> case x of Lit i -> show i)
  plus a b =
    Pr (\p -> case p of
                (Plus x y) -> print' a x ++
                              " + " ++ print' b y)
  view iso a = Pr (\x -> print' a (from iso x)
print :: Rep t => t -> Int
print = print' rep
```

The benefits of the GM approach compared to the Haskell type class approach become clear. In the GM approach, functions `eval` and `print` are represented by types which are instances of class `Generic`. Thus, both functions work on the same set of types. In the Haskell type class approach, we introduce a new class for each function and therefore we cannot give the same guarantees. On the other hand, we can easily introduce new (ad-hoc) cases by providing additional instances. This is a problem for the GM approach. We cannot specify ad-hoc cases without breaking modularity.

For example, data types `Plus a b` and `Minus a b` have the same uniform representation (as a product type). We obviously do not want to use the same generic definition for both types. To extend the `Generic` class with an ad-hoc "minus" case, we introduce a subclass.

```
class Generic g => GMinus g where
  minus :: g a -> g b -> g (Minus a b)
instance GMinus Ev where
  minus a b =
    Ev (\p -> case p of
                (Minus x y) -> eval' a x - eval' b y)
```

The problem is that we cannot access this new case, unless we update the type of the dispatcher function `rep`.

```
class Rep a where
  rep :: GMinus g => g a
-- original code: rep :: Generic g => g a
instance (Rep a,Rep b) => Rep (Minus a b) where
  rep = minus rep rep
eval :: Rep t => t -> Int
eval = eval' rep
```

The original code will not type check for similar reasons as encountered in the SYB approach.

Alternatively, we could leave the dispatcher untouched and make `GMinus` a superclass of `Generic`.

```
class GMinus g => Generic g where
  lit :: g Lit
  plus :: g a -> g b -> g (Plus a b)
  view :: Iso a b -> g a -> g b
```

We have not progressed much. Adding the superclass `GMinus` is again a non-local change and requires to recompile the entire program.

Each time we extend the generic function with a new ad-hoc case, we have to change the type declaration of the dispatcher. This is a non-local change and requires to recompile existing code. We conclude that the GM approach is not modular.

This problem has been addressed [21]. The solution is inspired by the modular extension of the SYB approach and requires some experimental type class extensions.

5.1 Our Solution

We first introduce the type class `GMinus` to represent the new ad-hoc case. We also provide a (default) generic instance definition for "minus" in terms of "plus" by means of the "view" case and and a type isomorphism between `Minus` and `Plus`.

```
rule GMinus g ==> Generic g
class GMinus g where
  minus :: g a -> g b -> g (Minus a b)
  minus a b = view isoMinus (plus a b)

isoMinus :: Iso (Minus a b) (Plus a b)
isoMinus = Iso fromMinus toMinus
fromMinus :: Minus a b -> Plus a b
fromMinus (Minus a b) = Plus a b
toMinus :: Plus a b -> Minus a b
toMinus (Plus a b) = Minus a b
```

The `rule` declaration guarantees that for each `GMinus` instance there is a `Generic` instance (as expected). This condition is required by the default instance definition. In Haskell, we would usually express such conditions via subclassing. But we want to make a point that under a type-passing translation scheme the introduction of sub-/superclasses is not necessarily connected to class declarations.

Next, we extend the dispatcher function by adding a new case for "minus".

```
rule Generic g ==> GMinus g
instance (Rep a,Rep b) => Rep (Minus a b) where
  rep = minus rep rep
```

Via the `rule` declaration we introduce `GMinus` as a superclass of `Generic`. This is essential, otherwise, the instance declaration for the "minus" case will not type check. Two issues arise here.

The `rule` declaration claims that for each `Generic` instance there is a `GMinus` instance. Above we have defined an instance of `Generic` on type `Pr a` but there is no such instance declaration for `GMinus`. Hence, there seems to be a problem. In terms of the terminology we develop later in Section 6.2, the type class proof system is "non-confluent". But wait! We forgot the generic, default definition for `GMinus`. The argument is that unless otherwise stated, for each `Generic` instance there is always a default `GMinus` instance. Hence, the type class proof system is confluent.

The second issue is that cyclic dependencies

```
rule GMinus g ==> Generic g
rule Generic g ==> GMinus g
```

among the `Generic` and `GMinus` class may threaten decidable type inference. Again, there is no problem here. Type inference remains decidable for such cases. We elaborate in Section 6.2.

We conclude that we can introduce new ad-hoc cases and easily inherit generic definitions without having to change any existing code. What we do next is override the default evaluator case with a new ad-hoc definition.

```
instance GMinus Ev where
  minus a b =
    Ev (\p -> case p of
                (Minus x y) -> eval' a x - eval' b y)
```

In summary, we have achieved a modular extension of the GM approach. To convince ourselves that our solution works correctly, we consider the type-passing translation of all declared instances.

Here is the translation of the Rep instances. As said earlier, we can translate the individual instances separately and link them together later. We show the final code produced by the linker.

```
rep = Λ a. typecase a of
  Lit -> Λ g. lit g
  Plus b c -> Λ g. plus g ((rep b) g) ((rep c) g)
  Minus b c -> Λ g. minus g ((rep b) g) ((rep c) g)
```

The translation is slightly more involved than outlined in Section 3. In the class declaration of Rep, the type of the method rep is locally constrained by Generic g. In case of the dictionary-passing translation scheme, we may therefore assume that the dictionary for Generic g will be locally supplied at the use site. Hence, the translation of the instance definitions takes an additional dictionary argument. The same principle applies in case of a type-passing translation scheme. For each type case branch, we introduce the local type abstraction Λ g. At a use site we supply the appropriate type argument, for example see (rep b) g. The presence of constructor classes also makes it necessary to switch to System F_ω as the target language (this applies to both translation schemes).

The translation of the remaining instances is straightforward.

```
lit = Λ a. typecase a of
    Ev -> Ev (λ x. case x of Lit i -> i)
    Pr -> ...

plus = Λ a. typecase a of
    Ev -> λ b c. Ev (λ p. case p of
            (Plus x y) -> eval' b x + eval' c y)
    Pr -> λ b c. Ev (λ p. case p of
            (Plus x y) -> print' b x ++
                          " + " ++ print' c y)

minus = Λ a. typecase a of
    Pr -> λ b c. to (plus Pr b c)
    Ev -> λ b c. Ev (λ p. case p of
            (Plus x y) -> eval' b x - eval' c y)
```

Notice that we (automatically) included the default instance for printing.

Function eval translates to

```
eval = Λ a. eval' ((rep a) Ev)
```

and the call

```
eval (Minus (Lit 2) (Lit 1))
```

translates to

```
eval (Minus Lit Lit) (Minus (Lit 2) (Lit 1))
```

It should be clear that this program text correctly evaluates to 1.

6. Extensible Superclasses

We formalize extensible superclasses using a combination of our own CHR-based type class framework [25] and Thatte's type-passing translation scheme. CHR stand for Constraint Handling Rule [3] and we use them to specify the type class proof system for extensible superclasses.

6.1 CHR Type Class Proof System

The syntax of class and instance declarations is as follows:

Types	t	::=	$a \mid t \to t \mid T\ \bar{t}$
Type Classes	tc	::=	$TC\ t$
Context	Ctx	::=	$(tc_1, ..., tc_n)$
Constraint	C	::=	$tc \mid C \wedge C$
Classes	cls	::=	class $Ctx \Rightarrow TC\ a$
Instances	$inst$	::=	instance $Ctx \Rightarrow TC\ t$

Notice that constraints and contexts effectively describe the same object, i.e. collections of type classes. Depending on the situation, we will use set, tuple or "$\wedge$" notation.

We make use of CHRs of the following two forms:

$$\texttt{rule}\ \ \texttt{TC t} \implies \texttt{TC}_1\ \texttt{t}'$$
$$\texttt{rule}\ \ \texttt{TC t} \iff \texttt{TC}_1\ \texttt{t}_1, ..., \texttt{TC}_n\ \texttt{t}_n$$

where we assume that TC refers to a type class and t refers to a type. The first CHR is referred to as a *propagation* rule and the second is referred to as *simplification* rule. Logically, the symbol $\implies$ denotes Boolean implication and $\iff$ denotes Boolean equivalence. CHRs have also a simple operational reading which we will ignore for the moment. In essence, CHRs model proof rules, similar to type class proof rules from Section 3.1.

Each declaration

$$\texttt{class}(\texttt{TC}_1\ \texttt{a}, ..., \texttt{TC}_n\ \texttt{a}) \Rightarrow \texttt{TC a}$$

translates to

```
rule TC a ==> TC1 a
...
rule TC a ==> TCn a
```

Subclassing states subset relations among instances which can logically be expressed in terms of Boolean implication. Notice that the Haskell subclass arrow => is (logically speaking) the wrong way around!

Each declaration

$$\texttt{instance}(\texttt{TC}_1\ \texttt{t}_1, ..., \texttt{TC}_n\ \texttt{t}_n) \Rightarrow \texttt{TC t}$$

translates to

```
rule TC t <==> TC1 t1,...,TCn tn
```

This is exactly the meaning required to describe type classes under a type-passing translation scheme. Recall that instances specify if-and-only-if relations.

Extensible superclasses can then be modeled straightforwardly by providing additional CHR propagation rules. For example, the following CHRs

```
rule Eval a ==> Print a                    -- (1)
rule Print a ==> Eval a                    -- (2)
```

state that Print is a superclass of Eval and vice versa. The introduction of cyclic CHRs such as (1) and (2) raises the concern whether we can maintain decidable type inference. We will address such issues further below.

We can also introduce "short-hand" notation via CHR simplification rules.

```
rule EvalAndPrint a <==> Eval a, Print a
```

The above introduces an *abstract* type class EvalAndPrint. We assume that there are no methods connected to this type class. In programs, that is in type annotations, we can then use EvalAndPrint a as a short-hand for Eval a, Print a.

Based on this understanding of type classes in terms of CHRs, we can express type class proofs directly in terms of some familiar first-order logic statements. Let P be the set of CHRs, derived from class and instance declarations and specified by the programmer. Let C be a conjunction of (given) type class constraints and TC t a (demanded) type class constraint. Then, we can write $P \models C \supset \texttt{TC t}$ to express that TC t is derivable from C under P. The symbol $\models$ denotes model-theoretic entailment. The statement $P \models C \supset \texttt{TC t}$ holds if TC t can be satisfied in any (first-order) model of P and C.

6.2 CHR Proof Checking

We briefly show how to verify that statements $P \models C \supset \texttt{TC t}$ hold. First, we perform some logical equivalence transformations. We have that $P \models C \supset \texttt{TC t}$ iff $P \models C \leftrightarrow C \wedge \texttt{TC t}$. CHRs have a straightforward operational reading in terms of rewritings among constraints, more formally written $\rightarrowtail^* \ P$. Thus, we can check $P \models C \leftrightarrow C \wedge \texttt{TC t}$ by executing $C \rightarrowtail_P^* C_1$ and $C \wedge \texttt{TC t} \rightarrowtail_P^* C_2$ testing whether C_1 and C_2 refer to the same canonical normal form. Decidability and completeness of this check depend on whether CHRs are terminating and confluent.

Let's take a look at the formal definition of the operational reading of CHRs. We assume that we are given a set C of constraints where $\texttt{TC t}' \in \texttt{C}$ and consider the (rewriting) effect the different forms of CHRs can have on $\texttt{TC t}'$.

Propagation step: We can apply

$$\texttt{rule TC t} \implies \texttt{TC}_1\ \texttt{t}'' \in P$$

by adding (propagating) $\texttt{TC}_1\ \phi(\texttt{t}'')$ to C, if we find a substitution ϕ such that `t'` and `phi(t)` are equal. Notice that we only perform matching (but not Prolog style unification). That is, we rewrite C to $C \cup \{\texttt{TC}_1\ \phi(\texttt{t}'')\}$, written $C \rightarrowtail_P C, \texttt{TC}_1\ \phi(\texttt{t}'')$. We assume that CHRs are renamed before rule application. Notice that we avoid infinite propagation by prohibiting to fire a rule on the same constraint twice. We refer to [1] for further details.

Simplification step: We can apply

$$\texttt{rule TC t} \iff \texttt{TC}_1\ \texttt{t}_1, ..., \texttt{TC}_\texttt{n}\ \texttt{t}_\texttt{n} \in P$$

if we find a substitution ϕ such that t' and $\phi(t)$ are equal. Then, we can rewrite C into $C - \{\texttt{TC t}'\} \cup \{\texttt{TC}_1\ \phi(\texttt{t}_1), ..., \texttt{TC}_\texttt{n}\ \phi(\texttt{t}_\texttt{n})\}$, i.e. the constraint resulting from C by replacing (simplifying) $\texttt{TC t}'$ with $\texttt{TC}_1\ \phi(\texttt{t}_1), ..., \texttt{TC}_\texttt{n}\ \phi(\texttt{t}_\texttt{n})$.

Each of the rewriting steps preserve the equivalence among constraints. Hence, the above checking method for $P \models \Delta \supset \texttt{TC t}$ is clearly correct.

We write $C \rightarrowtail_P^* C'$ to denote the exhaustive application of rewriting steps yielding the *final* constraint C'. We say a set of CHRs is *terminating* if for each constraint C we find a final constraint C' such that $C \rightarrowtail^* C'$. We say that a set of CHRs is *confluent* if different rewriting derivations starting from the same point can always be brought together again.

Let's see whether the (extensible superclass) examples we have seen so far satisfy termination and confluence of CHRs.

We consider the "cyclic" CHRs.

```
rule Eval a ==> Print a                    -- (1)
rule Print a ==> Eval a                    -- (2)
```

We find that

$$\begin{array}{rl} & \texttt{Eval a} \\ \rightarrowtail & \texttt{Eval a}, \texttt{Print a} \\ \rightarrowtail & \texttt{Eval a} \end{array}$$

In the first step, we apply CHR (1) on `Eval a` and add `Print a`. In the second step, apply CHR (2) on `Print a` which adds the constraint `Eval a`. But wait, this constraint is already present. We assume set semantics. Hence, no further constraint will be added. Recall that we prevent firing the same rule twice on the same constraint. We have already fired the propagation rule on `Eval a`. Hence, $\texttt{Eval a} \rightarrowtail^* \texttt{Eval a}, \texttt{Print a}$. Hence, the above CHRs are terminating.

Here is another "cyclic" set of CHRs which arises from the SYB example. The (simplified) declarations

```
class Size a => Data a       -- (Super)
instance Data a => Size a    -- (Inst)
```

yield

```
rule Data a ==> Size a
rule Size a <==> Data a
```

We find that

$$\begin{array}{rl} & \texttt{Size a} \\ \rightarrowtail & \texttt{Data a} \\ \rightarrowtail & \texttt{Data a}, \texttt{Size a} \\ \rightarrowtail & \texttt{Data a}, \texttt{Size a} \end{array}$$

In the first step, we apply the simplification rule. Then, the propagation rule adds `Size a`. In the final step, we apply again the simplification rule and simplify `Size a` by `Data a`. But this constraint is already present (recall set semantics). The propagation rule has already been fired on `Data a`. Hence, $\texttt{Size a} \rightarrowtail^* \texttt{Data a}, \texttt{Size a}$. Hence, the above CHRs are terminating.

The SYB authors claim that the above program is non-terminating. That is, the rewriting steps implied by class and instance declarations may not terminate. As we show above this is not the case. There is also no connection to recursive instances. The "recursion" goes through a superclass!

We conclude. Although extensible superclasses introduce "cyclic" relations, resulting CHRs are terminating for the examples we have seen so far. We leave it for future work to establish syntactic conditions (in style of the Haskell conditions [22] imposed on instances) that guarantee termination. Note that for terminating CHRs there is an easy check of confluence by testing whether all "critical pairs" are joinable.

For example, consider

```
rule Print a    ==> Eval a
rule Print Lit <==> True
```

We consider the critical pair `Print Lit`. We find two non-joinable derivations $\texttt{Print Lit} \rightarrowtail^* \texttt{True}$ and $\texttt{Print Lit} \rightarrowtail^* \texttt{Eval Lit}$. Hence, the above CHRs are non-confluent and hence we must reject the program. And rightly so, the first CHR claims that for each `Print t` there is a `Eval t` which is not the case because of the second CHR.

6.3 Type-Passing Translation Scheme

We highlight the main aspects. For simplicity, we only consider the translation of expressions and focus on the most interesting rules. The development is pretty much similar to the standard dictionary-passing translation method. The crucial difference is that we strictly erase type classes and use types to access specific method definitions.

We work with a simple expression language representing the core of a functional language with type classes. As our target language, we use a simplified version of System F. Function parameters are annotated with Hindley/Milner types which is sufficient here. There is also no need for a type case because we only consider the translation of expressions:

$$\begin{array}{lrcl} \text{Type Schemes} & \sigma & ::= & t \mid \forall a.\sigma \mid TC\ t \Rightarrow \sigma \\ \text{Expressions} & e & ::= & x \mid \lambda x.e \mid e\ e \mid \mathsf{let}\ g = e\ \mathsf{in}\ e \\ \text{Target} & E & ::= & x \mid \lambda x : \sigma.E \mid E\ E \mid \Lambda a.E \mid E\ t \end{array}$$

We follow the common path and employ a type-directed translation scheme formulated in terms of judgments $C, \Gamma \vdash e : \sigma \rightsquigarrow E$ where C is a constraint holding the set of type class constraints, Γ is a environment assigning type schemes to free variables, e is a expression with type σ and E is a target expression. The judgment

$C, \Gamma \vdash e : \sigma \rightsquigarrow E$ states that a well-typed expression e with type σ is translated to the target program E.

Here are the most interesting translation rules:

$$(\forall\text{Intro}) \qquad \frac{C, \Gamma \vdash e : \sigma \rightsquigarrow E \quad a \notin \mathit{fv}(\Gamma, C)}{C, \Gamma \vdash e : \forall a.\sigma \rightsquigarrow \Lambda a.E}$$

$$(\forall\text{Elim}) \qquad \frac{C, \Gamma \vdash e : \forall a.\sigma \rightsquigarrow E}{C, \Gamma \vdash e : [t/a]\sigma \rightsquigarrow E\ t}$$

$$(\Rightarrow\text{Intro}) \qquad \frac{C \wedge TC\ t, \Gamma \vdash e : t' \rightsquigarrow E}{C, \Gamma \vdash e : TC\ t \Rightarrow t' \rightsquigarrow E}$$

$$(\Rightarrow\text{Elim}) \qquad \frac{C, \Gamma \vdash e : TC\ t \Rightarrow \sigma \rightsquigarrow E \quad P \models C \supset TC\ t}{C, \Gamma \vdash e : \sigma \rightsquigarrow E}$$

The first two rules are familiar from translating Hindley/Milner to System F [6]. In rule (∀Intro), we assume that $\mathit{fv}(\Gamma, C)$ computes the set of free (type) variables of an environment and constraint. If a variable is not in this set we can universally quantify over this variable. We use a System F style target language, therefore, we make type abstraction (and later application) explicit via Λ, pronounced "big" lambda. In case of elimination (i.e. instantiation) of a universal quantifier we use type application in the target term. Up to know the same translation steps apply to the dictionary-translation scheme as well.

The crucial difference between the dictionary- and type-passing is manifested in the last two rules. Pushing a type class into a type scheme has no effect on the translated program under a type-passing scheme. See rule (⇒Intro). That is, type classes are simply erased during the translation. This is in contrast to the dictionary-passing scheme where we would introduce a lambda-abstraction because type classes are turned into dictionaries. In rule (⇒Elim), we eliminate a type class if we can prove that this type class follows from the given assumptions (represented by C) under the given set of proof rules (represented by P), i.e. $P \models C \supset TC\ t$ holds. Again, elimination has no effect on the translated program. In the dictionary-passing scheme, we would need to derive a dictionary our of the type class proof.

We leave it to future work to establish properties such as type soundness and coherence. We expect that these properties follow by straightforward application of methods and techniques found in [27, 25]. For example, Thatte has already proven type soundness whereas we have verified coherence if CHRs are confluent.

7. Related Work

7.1 Translating Type Classes

Thatte [27] is the first to propose a type-passing translation scheme for type classes. His main motivation is to provide an alternative semantics where type classes can be interpreted co-inductively. For example, consider the following program.

```
class Foo a where foo :: a->Int
instance Foo a => Foo a where foo = foo
```

Under the standard inductive interpretation of type classes, `Foo t` has no meaning for any type `t`. Simply because we cannot (inductively) rewrite `Foo t` to some simpler form. Under a co-inductive interpretation, however, we can give `Foo t` the "undefined" meaning.

In case we employ the standard dictionary-passing translation, the meaning of co-inductive type classes can be explained in terms of recursive dictionaries.

```
data Foo a = F (a->Int)
inst :: Foo a -> Foo a
inst (F foo) = F foo

d :: Foo t
d = inst d
```

Among others, such an extension is required in the SYB3 approach [18]. In our own work [26], we formalize recursive instances in the presence of a dictionary-passing translation scheme.

Thatte's work does not include superclasses. We show here that we can naturally support extensible superclasses in a combination of Thatte's type-passing translation scheme and our own CHR-based type class framework. The translation scheme we employ in the CHR-based type class framework (from here-on referred to as ATO) is in fact a hybrid. Similar to the dictionary-passing scheme, we assume dictionary constructing functions. Though, instance declarations describe if-and-only-if relations in ATO. For example, in ATO the statement `Print [a]` ⊢ `Print a` is correct. In Haskell, we cannot verify this statement because it is not obvious how to construct a proof (i.e. dictionary) for `Print a` out of the proof for `Print [a]`. In ATO this problem is solved by assuming that dictionary constructing functions are defined for all ground instances. Of course, to implement such a scheme Thatte's type-passing translation method is the natural choice.

Another alternative translation scheme for type classes, similar to Thatte's type-passing method, appears also in the work by Pottier and Gauthier [24]. They use GADTs (also known as guarded recursive data types [31]) instead of System F extended with type-case for the translation. The idea is to represent dictionaries via GADTs. Here is the dictionary representation of the `Print` and `Eval` class using the GADT notation as available in GHC.

```
data EvalDict a where
  EInstLit :: EvalDict List
  EInstProd :: EvalDict a ->
               EvalDict b -> EvalDict (Prod a b)
data PrintDict a where
  PInstLit  :: PrintDict Lit
  PInstProd :: PrintDict a ->
               PrintDict b -> PrintDict (Prod a b)
```

The above constructor definitions require GADTs because the (output) type changes. They directly correspond to the instance declarations.

As in case of the type-passing translation scheme, we need to lump together the instance definitions. Here is the translation of the `Eval` instances.

```
eval :: EvalDict a -> a-> Int
eval d = case d of
           EInstLit  -> ...
           EInstProd -> ...
```

Up to here this looks exactly like Thatte's type-passing translation. This is not surprising given that the idea of GADTs can be traced back to work on intentional type analysis.

However, the GADT-based translation scheme has a slight disadvantage when it comes to translating programs with sub-/superclasses. For example, the program we have seen earlier

```
f1 :: Print a => a -> Int
f1 x = eval x
```

translates to the GADT program

```
f1 :: PrintDict a -> a -> Int
f1 d = eval (super d) x
```

The function `super` to extract superclass from subclass dictionaries is defined as follows.

```
super :: PrintDict a -> EvalDict a
super PInstLit = EInstLit
super (PInstProd a b) = EInstProd (super a) (super b)
```

The disadvantage of the GADT translation scheme is that each time we introduce a new sub-/superclass we will need to adapt the definition of `super`. Though, this is still a local change. Hence, we do not need to recompile function `f1`. The point is that in a type-passing translation scheme there are literally no changes necessary. Hence, we can argue that for a practical implementation Thatte's type-passing translation method is the preferred choice over the encoding in terms of GADTs.

7.2 Generic Programming

Language extensions such as Generic Haskell [19] and PolyP [13] have direct support for generic functions but do not provide support for open extension. The Clean [23] supports both generic programming with the opportunity to override generic instances via specialization.

Wang, Chen and Khoo [29] apply aspect-oriented programming techniques to support openly extensible generic functions. Their idea is to use type classes to define the generic cases and *type-scoped* (also known as type-guarded) *aspects* for extensions. We yet have to work out the precise connections to our work. In [30], Washburn and Weirich propose a similar solution by modeling type class style overloading with aspects.

In [20], Löh and Hinze proposed a simple yet powerful solution to the problem of extensibility on both dimensions of functions and data types. Their source language allows definitions of function clauses and data constructors to be scattered in different modules; and merge them into one by preprocessing. It is obvious that many of the examples in this paper can be encoded in their language. The primary difference between our proposal and their's is the use of data constructors versus class instances. There are pros and cons for both approaches. For some applications such as extensible `eval`, a data type encoding of expression appears to be more straightforward. For the others such as `gsize`, ad-hoc polymorphism avoids clumsy embedding of types into data constructors. Another notable advantage of our approach is static safety. A function called with arguments which has no instances defined on will result in a static error; instead of a run-type pattern matching failure as in Löh's and Hinze's system.

Much of our work, concerns explaining how to make type class encodings of the GM and SYB approach "modular" in terms of the alternative concept of extensible superclasses. In this respect, our work can be seen orthogonal to work by Hinze, Löh and Oliveira [10] who explain the "spine-view" underlying the SYB approach. It is interesting to note that they use GADTs in their example programs. As argued above GADTs are nothing else than a convenient source-language notation to mimic a type-passing translation scheme.

8. Conclusion

We have given a new perspective how to achieve modularity for the GM and SYB generic programming approach via extensible superclasses. In our opinion, extensible superclasses provide for a much more natural solution compared to previous solutions which require type class abstraction and recursive instances. We have formalized the main aspects of extensible superclasses using a combination of Thatte's type-passing translation scheme and our own CHR-based framework. There is lots of future work ahead.

The translation scheme behind extensible superclasses demands significant changes to the dictionary-passing translation scheme currently employed in Haskell. We yet have to study the impact Thatte's translation scheme has on existing compiler optimizations. Realistically, we do not expect that any time soon systems such as GHC will be able to support extensible superclasses. However, the experimental Haskell compiler JHC [15] implements a type class translation scheme that is very close to Thatte's type-passing method. We consider this as evidence that a specialized Haskell compiler to support modular generic programming via extensible superclasses is feasible in the near future.

Thatte's type-passing translation scheme resembles method lookup as found in OO languages. Modularity, local modification and separate compilation have been studied in OO languages for years. We hope that we can take advantage of results in this area and plan to pursue this topic in future work.

Acknowledgments

We thank the reviewers and Bruno Oliveira for their helpful comments.

References

[1] S. Abdennadher. Operational semantics and confluence of constraint propagation rules. In *Proc. of CP'97*, LNCS, pages 252–266. Springer-Verlag, 1997.

[2] B. Demoen, M. García de la Banda, W. Harvey, K. Marriott, and P.J. Stuckey. An overview of HAL. In J. Jaffar, editor, *Proceedings of the Fourth International Conference on Principles and Practices of Constraint Programming*, LNCS, pages 174–188. Springer-Verlag, October 1999.

[3] T. Frühwirth. Constraint handling rules. In *Constraint Programming: Basics and Trends*, LNCS. Springer-Verlag, 1995.

[4] Glasgow haskell compiler home page. http://www.haskell.org/ghc/.

[5] C. V. Hall, K. Hammond, S. L. Peyton Jones, and P. L. Wadler. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems*, 18(2):109–138, 1996.

[6] R. Harper and J. C. Mitchell. On the type structure of standard ML. *ACM Transactions on Programming Languages and Systems*, 15(2):211–252, 1993.

[7] R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. In *Proc. of POPL'95*, pages 130–141. ACM Press, 1995.

[8] F. Henderson et al. The Mercury language reference manual, 2001. http://www.cs.mu.oz.au/research/mercury/.

[9] R. Hinze. Generics for the masses. In *Proc. of ICFP'04*, pages 236–243. ACM Press, 2004.

[10] R. Hinze, A. Löh, and B. Oliveira. "Scrap your boilerplate" reloaded. In *Proc. of FLOPS'06*, volume 3945 of *LNCS*, pages 13–29. Springer-Verlag, 2006.

[11] J. Hughes. Restricted datatypes in Haskell. In *Haskell Workshop*, September 1999.

[12] Hugs home page. haskell.cs.yale.edu/hugs/.

[13] P. Jansson and J. Jeuring. Polyp– a polytypic programming language extension. In *Proc. of POPL'97*, pages 470–482. ACM Press, 1997.

[14] D. Jeffery, F. Henderson, and Z. Somogyi. Type classes in Mercury. In J. Edwards, editor, *Proc. Twenty-Third Australasian Computer Science Conf.*, volume 22 of *Australian Computer Science Communications*, pages 128–135. IEEE Computer Society Press, January 2000.

[15] Jhc. http://repetae.net/john/computer/jhc/.

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

[17] R. Lämmel and S. Peyton Jones. Scrap your boilerplate: a practical approach to generic programming. In *Proc. of ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003)*, pages 26–37. ACM Press, 2003.

[18] R. Lämmel and S. Peyton Jones. Scrap your boilerplate with class: extensible generic functions. In *Proc. of ICFP'05*, pages 204–215. ACM Press, 2005.

[19] A. Löh, D. Clarke, and J. Jeuring. Dependency-style Generic Haskell. In *Proc. of ICFP'03*, pages 141–152. ACM Press, 2003.

[20] A. Löh and R. Hinze. Open data types and open functions. In *Proc. of PPDP'06*. ACM Press, 2006. To appear.

[21] B. Oliveira, R. Hinze, and A. Löh. Generics as a library. In Henrik Nilsson, editor, *Seventh Symposium on Trends in Functional Programming*, 2006.

[22] S. Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

[23] M.J. Plasmeijer and M.C.J.D. van Eekelen. Language report Concurrent Clean. Technical Report CSI-R9816, Computing Science Institute, University of Nijmegen, Nijmegen, The Netherlands, June 1998. ftp://ftp.cs.kun.nl/pub/Clean/Clean13/doc/refman13.ps.gz.

[24] F. Pottier and N. Gauthier. Polymorphic typed defunctionalization and concretization, 2005. To appear in *Higher-Order and Symbolic Computation*.

[25] P. J. Stuckey and M. Sulzmann. A theory of overloading. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(6):1–54, 2005.

[26] M. Sulzmann. Extracting programs from type class proofs. In *Proc. of PPDP'06*. ACM Press, 2006. To appear.

[27] S. R. Thatte. Semantics of type classes revisited. In *LFP '94: Proceedings of the 1994 ACM conference on LISP and functional programming*, pages 208–219. ACM Press, 1994.

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

[29] M. Wang, K. Chen, and S-C. Khoo. Coherent and type-directed weaving of aspect-oriented higher-order functional languages, 2006. Manuscript.

[30] G. Washburn and S. Weirich. Good advice for type-directed programming, 2006. In this volume.

[31] H. Xi, C. Chen, and G. Chen. Guarded recursive datatype constructors. In *Proc. of POPL'03*, pages 224–235. ACM Press, 2003.