Beyond Type Classes

Andreas Rossberg
Programming Systems
Universität des Saarlandes, Germany
rossberg@ps.uni-sb.de

Martin Sulzmann
Department of Computer Science and Software Engineering
University of Melbourne, Vic. 3010, Australia
sulzmann@cs.mu.oz.au

ABSTRACT

We discuss type classes in the context of the Chameleon language, a Haskell-style language where overloading resolution is expressed in terms of the meta-language of Constraint Handling Rules (CHRs). In a first step, we show how to encode Haskell's single-parameter type classes into Chameleon. The encoding works by providing an approrpriate set of CHRs which mimic the Haskell conditions. We also consider constructor classes, multi-parameter type classes and functional dependencies. Chameleon provides a testbed to experiment with new overloading features. We show how some novel features such as universal quantification in context can naturally be expressed in Chameleon.

1. INTRODUCTION

Type classes [14, 23] are one of the most prominent features of Haskell [18]. They are also found in other languages such as Mercury [6, 9] , HAL [3] and Clean [19]. In particular Haskell has become the most popular playing field for type class acrobats. Advanced features such as constructor [11], multi-parameter [13] classes and functional dependencies [12] are found in most Haskell implementations.

The idea behind type classes is to allow the programmer to define relations over types. For single–parameter type classes, the type class relation simply states set membership. Consider the `Eq` type class, the declaration

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

states that every type `a` in type class `Eq` has an equality function `==`. Instance declarations "prove" that a type is in the class, by providing appropriate functions for the class methods. For example, `Int` is in `Eq`:

```
instance Eq Int where (==) = primIntEq
```

which states that the equality function for `Int`s is `primIntEq` where `primIntEq` is a built-in primitive function on integers. The `==` function can only be used on values with types that are in `Eq`. This is reflected by the function's *constrained* type:

```
(==) :: Eq a => a -> a -> Bool
```

which has a constraint component `Eq a` and a type component `a -> a -> Bool`.

Constructor classes [11] allow the programmer to define relations not just over types but also over type constructors. A typical example is the `Functor` class:

```
class Functor f where
   fmap :: (a -> b) -> (f a -> f b)
```

where the class parameter `f` ranges over type constructors such as `[]` and `Tree`.

Multi–parameter type classes [13] allow for multiple class parameters. For example,

```
class Collects ce e where
   empty  :: ce
   insert :: ce -> e -> ce
```

The `Collects` class defines a relation between element types `e` and the type `ce` of the collection itself. Unfortunately, this example has problems; we can't determine which instance declaration to use for `empty` because its type (`ce`) won't allow us to deduce an element type `e`. Therefore, `empty` is considered ambiguous. This can be resolved by adding another feature, functional dependencies [12] to retain unambiguity. The functional dependency `ce`$\leadsto$`e` states that for all instance declarations of `Collects` the element type can be uniquely determined from the collection type. In this case `empty` is unambiguous.

Further type class proposals can be found in [2, 8]. This is not an exhaustive list, we observe that there are many possibilities for additional features. For example, we might like to say that the `Integral` and `Fractional` type classes are disjoint (since we know a number can't be both). This can't be expressed in any current type class system and hence the function

```
f x y = x / y + x `div` y
```

has an inferred type of `f :: (Integral a, Fractional a) => a -> a -> a` rather than immediately causing a type error. As another example consider the following class declaration

```
class (forall a . (Eq a => Eq (s a))) => Sequence s
```

Our intention is to state that for any instance type `S` of the `Sequence` class there must be an instance for `Eq (S T)` whenever there is an instance `Eq T` for some type `T`. To our knowledge such a property can't be expressed by any type class system to date.

Stuckey and Sulzmann introduce in [21] a general overloading framework based on Constraint Handling Rules (CHRs). Essential algorithms such as type inference and unambiguity checks can be performed by running the underlying CHR evaluation engine. The system is user-programmable in the sense that the user is able to state additional properties via CHRs. That is, CHRs serve as a meta-language to impose conditions on the set of constraints allowed to appear.

In this paper, we make more explicit the connection between the form of overloading provided by the CHR-based system and type classes as found in Haskell. The ideas of the CHR-based overloading system have been incorporated into the Chameleon [22] language. The Chameleon syntax mostly follows Haskell syntax. Overloaded identifiers are defined using the `overload` keyword which is similar to `instance` in Haskell. However, no class declarations are necessary. The user can design her own system by providing CHRs via the `rule` keyword. For example, the functional dependency declaration `class Collects ce e | ce⇝e` can be specified as follows:

```
rule Collects ce e, Collects ce e' ==> e = e'
```

We can prevent types from being a member of the `Integral` and `Fractional` types class by the following CHR:

```
rule Integral a, Fractional a ==> False
```

The class declaration from before containing a universal quantifier can be expressed as follows:

```
rule Sequence s, Eq a ==> Eq (s a)
```

We continue in Section 2 by giving an overview of the Chameleon language. In Section 3 we show how to encode Haskell's type classes in Chameleon. The encoding is faithful, i.e. if the Haskell 98 program is typable then so will be the respective Chameleon program, with the same type. Section 4 considers more sophisticated type class features such as an alternative treatment of constructor classes and universal quantification in contexts. We conclude in Section 7 More details on Chameleon can be found under `http://www.cs.mu.oz.au/~sulzmann/chameleon`

2. CHAMELEON

A Chameleon program consists of a set of data types and (possibly overloaded) function definitions. Furthermore, we find user-definable CHR rules. Currently, we restrict all definitions to the top-level of the program, i.e. nested definitions are not permitted.

Example 1. *Here is an example Chameleon program:*

```
data Nat = Zero | Succ Nat

leqNat Zero Zero           = True
leqNat Zero (Succ n)       = False
leqNat (Succ n) Zero       = False
leqNat (Succ n1) (Succ n2) = leqNat n1 n2

overload leq :: Nat->Nat->Bool
         leq = leqNat

insList :: Leq (a->a->Bool) => [a]->a->[a]
insList [] y     = [y]
insList (x:xs) y = if leq x y then x:(insList xs y)
                                 else x:y:ys

overload insert :: Leq (a->a->Bool) => [a]->a->[a]
         insert = insList

rule Leq a               ==> a = b->b->Bool
rule Insert a            ==> a = ce->e->ce
rule Insert (ce->e->ce),
     Insert (ce->e'->ce)  ==> e = e'
rule Insert ([a]->b->[a]) ==> a = b
```

Roughly speaking, an overloaded definition in Chameleon corresponds to an instance for an implicit type class in Haskell. For each overloaded identifier we introduce a predicate symbol. For example, the overloaded identifier `insert` introduces the unary predicate symbol `Insert`. We can refer to such predicates in type signatures and `rule` definitions. Note that predicates in Chameleon are mostly of the form `P (t1->...->tn->t)`. Assuming that `P` corresponds to an overloaded identifier `p`, then `P (t1->...->tn->t)` indicates a possible invocation of `p` on argument types `t1,...,tn` and result type `t`. Each overloaded definitions must be annotated with a valid type signature. Type annotations are optional for regular function definitions.

For convenience, Chameleon also provides a constraint abbreviation mechanism similar to type synonyms in Haskell. For example, the statement

```
constraint Collects (ce,e) =
    Empty ce, Insert (ce->e->ce)
```

introduces `Collects (ce,e)` as an abbreviation for the predicate set on the right-hand side of `=`.[1] Constraint synonyms are expanded statically and can be referred to in type signatures and user-definable CHR rules. They may not be cyclic.

A user-definable CHR *propagation* rule is of the form

```
rule C ==> D
```

where `C` and `D` consist of a set of predicates. `D` may additionally consist of equality constraints. We often use the term "constraint" to refer either to a predicate, or an equality constraint, or a set thereof. Assume `C = c1, ...,cn` and `D = d1,...,dm`. Then, the logical meaning of such a CHR is as follows:

$$
\begin{array}{c}
[\![c_1,\ldots,c_n \quad \Longrightarrow \quad d_1,\ldots,d_m]\!] \\
= \\
\forall\bar{\alpha}.(c_1 \wedge \cdots \wedge c_n \rightarrow (\exists\bar{\beta}.d_1 \wedge \cdots \wedge d_m))
\end{array}
$$

where $\bar{\alpha} = fv(c_1 \wedge \cdots \wedge c_n)$ and $\bar{\beta} = fv(d_1 \wedge \cdots \wedge d_m) - \bar{\alpha}$. We assume *fv* is a function returning the free variables in a constraint. In Example 1, `rule Leq a ==> a=b->b->Bool` enforces that `leq` arguments must be of the same type and the result type is `Bool`. A similar condition is enforced by `rule Insert a ==> a=ce->e->ce`. The CHR

```
rule Insert (ce->e->ce), Insert (ce->e'->ce)
     ==> e = e'
```

essentially states a functionally dependency, i.e. the first input argument uniquely determines the second argument. The CHR

[1] Note that, technically, `Collects` is a unary predicate. We abuse tuples to simulate polyadic predicates.

```
rule Insert ([a]->b->[a]) ==> a = b
```

enforces the functional dependency for the particular overloaded definition

```
insert :: Leq (a->a->Bool) => [a]->a->[a]
```

We note that there is also another (implicit) kind of CHRs which arises from the set of overloaded definitions. The two overloaded definitions in Example 1 give rise to the following two CHR *simplification* rules:

```
rule Leq (Nat->Nat->Bool) <==> True
rule Insert ([a]->a->[a]) <==> Leq (a->a->Bool)
```

Note that this set of CHRs will be automatically generated from the set of overloaded definitions and cannot be written by the user. The logical reading of simplification rules is similar to the one for propagation rules, boolean implication is simply replaced by boolean equivalence. We commonly refer to the set of user-definable CHRs and the set of CHRs arising from overloaded definitions as the *program theory*. All essential properties of Chameleon programs can be defined in terms of the program theory.

Consider type inference for example. Out of the program text we generate the appropriate constraints which are then solved w.r.t. the program theory. The operational semantics of CHRs is straightforward. We assume that constraints are kept in a constraint store. CHRs define transitions from one set of constraints to an equivalent set. Whenever there is a matching copy of the left-hand side of a propagation (resp. simplification) rule in the store, we propagate (simplify), i.e. add (replace), the right-hand side. Type inference is decidable if the CHRs are terminating, i.e. any constraint set can be reduced in a finite number of steps such that no further CHRs are applicable. For a detailed exposition on type inference and related issues we refer to [21]. Here, we give a short overview of some of the important criterias which need to be satisfied by a Chameleon program.

2.1 Termination

The program theory must be terminating.

EXAMPLE 2. *Consider the following program fragment:*

```
overload eq :: Eq ([a]->[a]->Bool) => a->a->Bool
         eq x y = eq [x] [y]
```

The resulting CHR simplification rule would be

```
rule Eq (a->a->Bool) <==> Eq ([a]->[a]->Bool)
```

Immediately, we find that any program theory which contains the above rule is non-terminating. E.g. `Eq (t->t->Bool)` reduces to `Eq ([t]->[t]->Bool)` which in turn reduces to `Eq ([[t]]->[[t]]->Bool)` and so on.

EXAMPLE 3. *Assume we find the following user-defined propagation rule:*

```
rule Leq (a->a->Bool) ==> Leq ([a]->[a]->Bool)
```

In a non-terminating sequence, a constraint `Leq (t->t->Bool)` reduces to `Leq (t->t->Bool), Leq ([t]->[t]->Bool)`, and so on. Infinite application of a redundant propagation rule is prevented by applying a particular rule only once to a set of constraints.

EXAMPLE 4. *Consider*

```
rule Leq (a->a->Bool) ==> Eq (a->a->Bool)
```

For example, `Leq (t->t->Bool)` reduces to `Leq (t->t->Bool), Eq (t->t->Bool)` which is the final store. In essence, we prohibit adding, i.e. propagating, redundant constraints.

EXAMPLE 5. *Consider the following two propagation rules:*

```
rule X a ==> Y a
rule Y a ==> X a
```

There seems to be a cyclic dependency. However, `X t` reduces to `X t, Y t`, which is the final store. Application of the second rule would only add a redundant constraint.

We have incorporated various checks for termination of CHRs. See [22] for details. Clearly, each check is incomplete in the sense that there might be some CHRs which are terminating, but for which our termination check signals failure.

2.2 Confluence

Another important property of the program theory is confluence. Confluence states that, starting from a set of constraints, the CHRs can be applied in any arbitrary order and always leads to the same final constraint store.

Inconsistent Definitions

EXAMPLE 6. *Consider the following program fragment. For simplicity, we leave out the function bodies.*

```
overload leq :: Int->Int->Bool
         leq = ...

rule Leq (a->a->Bool) ==> Eq (a->a->Bool)
```

The above CHR propagation rule states that every definition of `leq` on type `a->a->Bool` implies that there must be a definition of `eq` on the same type. This is fairly similar to the super-class relationship found in Haskell. Note that there is a "missing instance": the propagation rule enforces that a definition of `eq` on type `Int->Int->Bool` must be present. However, no such definition is in scope. In Chameleon such inconsistencies among definitions can be detected by checking whether program theories are confluent. Clearly, the program theory in the above example is non-confluent.

EXAMPLE 7. *Consider*

```
overload eq  :: Eq (a->a->Bool) => Eq ([a]->[a]->Bool)
         eq  = ...
overload leq :: [a]->[a]->Bool
         leq = ...

rule Leq (a->a->Bool) ==> Eq (a->a->Bool)
```

In Haskell terminology, we would say that the "instance context" of `leq` on type `[a]->[a]->Bool` is "too general". Chameleon will complain about a non-confluent program theory:

```
rule Eq  ([a]->[a]->Bool) <==> Eq (a->a->Bool) -- 1
rule Leq ([a]->[a]->Bool) <==> True            -- 2
rule Leq (a->a->Bool)      ==> Eq (a->a->Bool) -- 3
```

For example, the constraint `Leq ([a]->[a]->Bool)` can be reduced to `True` via the second rule. However, there is another possible CHR derivation for `Leq ([a]->[a]->Bool)`. We first apply the third rule, i.e. we propagate the constraint `Eq ([a]->[a]->Bool)` which yields `Leq ([a]->[a]->Bool), Eq ([a]->[a]->Bool)`. Then, we apply the first rule which leads to `Leq ([a]->[a]->Bool), Eq (a->a->Bool)`. Finally, we apply the second rule which leads to `Eq (a->a->Bool)`.[2] Obviously, `Leq ([a]->[a]->Bool)` has two different CHR derivations. In the first case, the final constraint store consists of `True` whereas in the second case we find `Eq (a->a->Bool)`. This shows that the program theory consisting of the three rules above is not confluent. In Chameleon, a non-confluent program theory indicates problems among the constraint relations.Therefore, all program theories are checked for confluence. There is no Haskell equivalent to the confluence condition. In the Haskell 98 report (Section 4.3.2), the interplay between classes and instances is formulated via three ad-hoc conditions. We believe that confluence subsumes those conditions.

Overlapping Definitions

Confluence becomes a subtle issue in case of *overlapping* definitions.

EXAMPLE 8. *We add the following definition to the program in Example 1:*

```
overload insert :: [Nat]->Nat->[Nat]
         insert = ... special version ...
```

Although the program theory is confluent, we have a problem in case we require a definition of function `insert` on type `[Nat]->Nat->[Nat]`. We must take an indeterministic choice between two possibilities. In Chameleon, such problems are avoided by simply ruling out overlapping definitions. More details on how to deal with overlapping definitions are discussed in [21].

Completion of CHRs

There are also cases where the program theory is non-confluent but it is "safe" to add some CHRs to complete the program theory. Consider Example 1 again. If we would leave out `rule Insert ([a]->b->[a]) ==> a=b` the resulting program theory would not be confluent. The CHR `rule Insert (ce->e->ce), Insert (ce->e'->ce) ==> e=e'` states a general property which must hold for all definitions of `insert`. For each particular definition, we have to add in an additional propagation rule to complete the program theory. Chameleon provides the user with the convenience to add in such propagation rules automatically. The completion check builds "critical pairs" among all propagation and simplification rules. Confluence holds if all critical pairs are confluent. Note that pairs of propagation rules always satisfy the confluence condition. We also do not need to consider pairs of simplification rules because we require that overloaded definitions must always be non-overlapping. Now consider a pair of a propagation and a simplification rule, e.g.

```
rule U t, C1  ==> C2
rule U t'    <==> C3
```

where types `t` and `t'` are unifiable. We build the most general unifier ϕ of `t` and `t'`. It remains to check whether the critical pair consisting of the two constraints `U` ϕ`t,` ϕ`C1,` ϕ`C2` and ϕ`C1,` ϕ`C3` is confluent. Note that both constraints are derived by applying each rule to `U` ϕ`t,` ϕ`C1`. Assume that `U` ϕ`t,` ϕ`C1,` ϕ`C2` reduces to `D1` and ϕ`C1,` ϕ`C3` reduces to `D2` such that `D1` and `D2` are not equivalent, i.e. confluence is violated. In case none of the constraints entails the other we immediately report failure. Otherwise, assume `D1` entails `D2`. If the user-defined constraints `UD1` in `D1` are a subset of the user-defined constraints `UD2` in `D2` we add in a new propagation rule `rule UD2 ==> ED1` where `ED1` are all equality constraints in `D1`. Otherwise, we report failure.

EXAMPLE 9. *Consider*

```
rule Insert (ce->e->ce),
     Insert (ce->e'->ce)         ==> e = e'
rule Insert ([Nat]->Nat->[Nat]) <==> True
```

We build the critical pair consisting of the two constraint sets `Insert ([Nat]->Nat->[Nat]), Insert ([Nat]->e'->[Nat]), Nat=e'` and `Insert ([Nat]->e'->[Nat])`. The first constraint reduces to `Nat=e'` whereas `Insert ([Nat]->e'->[Nat])` already represents the final constraint. The two constraints are not equivalent but `Insert ([Nat]->e'->[Nat])` is entailed by `Nat=e'` w.r.t the above CHRs. The additional completion requirements are satisfied, we add in `rule Insert ([Nat]->e'->[Nat]) ==> Nat=e'`. This yields a confluent set of CHRs.

The described completion procedure is sound. We also conjecture that the procedure is complete. That is, in case of failure the program theory is not completable by a set of propagation rules. Note that completion fails for the CHRs in Example 7. Indeed, this set of CHRs can only be completed via an additional simplification rule. A detailed study of the completion problem will be the subject of a forthcoming paper.

2.3 Unambiguity

We require that programs must be unambiguous. Unambiguous programs lead to difficulties in providing a well-defined semantics (see e.g. [10]), and therefore, are ruled out. For Haskell 98, there is a simple syntactic check which ensures that programs are unambiguous. An expression e of type $\forall\bar{\alpha}.C \Rightarrow \tau$ is *unambiguous* iff for any $\alpha \in \bar{\alpha}$ such $\alpha \in fv(C)$ then $\alpha \in fv(\tau)$. In the presence of additional program properties specifiable via the `rule` keyword, the above check for unambiguity is too restrictive. Therefore, Chameleon implements an unambiguity check which subsumes the ones found for Haskell 98 and functional dependencies [12]. We refer to [21] for more details. The unambiguity condition (plus some other conditions) ensure that we can provide a well-defined semantics [20] for Chameleon programs.

2.4 Context

We refer to *context* as the set of constraints in type signatures and CHRs. There are no syntactic restrictions on contexts in Chameleon, in contrast to Haskell 98. For example, consider the invalid Haskell 98 program

```
x :: C [a] => a
x = undefined
```

In Haskell 98, all constraints in type signatures must be of the form `C a`. In Chameleon, the following would be valid.

[2]We silently omitted `True` in the final constraint store.

```
x :: C Bool => Bool
x = True
```

Clearly, to actually evaluate expression `x`, we require that `C Bool` must be reducable to the `True` constraint w.r.t. the current program theory.

3. ENCODING TYPE CLASSES

Chameleon does not require explicit declaration of type classes to achieve overloading. Ad-hoc overloading is thus simplified a lot. But type classes in Haskell not merely serve as the basic mechanism for providing overloading as such, they are also an important feature for structuring applications with respect to overloading polymorphism. A type class in Haskell is an abstraction that bundles related operations together into a higher-level entity, reminiscent of a module signature. By organizing overloaded functions within a class hierarchy, programs become more robust with respect to changes or additions/removals of individual operations.

In this section we show (with one minor caveat) that using propagation rules, we can faithfully encode type classes. That is, a Haskell typable program can be translated into Chameleon such that typing is preserved.

3.1 Single Classes with Monomorphic Methods

We start by considering single classes that do not contain polymorphic methods. Consider a class declaration

```
class TC a where
  m1 :: t1
  ...
  mk :: tk
```

Grouping several methods into a single class implies that whenever we define one of the methods, we also have to define all others[3]. Likewise, whenever we use one of the methods at some particular instance of `a`, a suitable implementation must not only exist for that particular method, but for all in the class. Obviously, expressing the same in Chameleon boils down to having appropriate propagation rules that enforce consistent presence of methods.

The encoding of monomorphic class and instance declarations is rather straight-forward[4]:

1. For each class declaration

   ```
   class TC a where
     m1 :: t1
     ...
     mk :: tk
   ```

   where `t1`,...,`tk` are types containing only `a` as free type variables, we introduce a constraint synonym

   ```
   constraint TC a = M1 t1, ..., Mk tk
   ```

   It allows interpreting class constraints `TC t` as an abbreviation for the set of corresponding method constraints. For each method `mi` we introduce a propagation rule

   ```
   rule Mi x ==> x = ti, TC a
   ```

   The equational predicate ensures that a definition of the corresponding method satisfies the type declaration as specified in the class. The second predicate (which expands according to the above constraint synonym declaration) mimics the method's membership in class `TC`.

2. Each instance declaration

   ```
   instance C => TC t where
     m1 = e1
     ...
     mk = ek
   ```

   where `e1`,...,`ek` are expressions, is translated as follows:

   ```
   overload m1 :: C => [t/a]t1
            m1 = e1
   ...
   overload mk :: C => [t/a]tk
            mk = ek
   ```

   We use the notation `[t/a]ti` to indicate substitution of the instance type for the class variable in each method type `ti`.

The encoding allows faithfully representing a legal Haskell program in Chameleon, and interpreting any Haskell expression as a Chameleon expression, with typing preserved.

EXAMPLE 10. *Consider (a subset of) the standard class* `Enum` *and a corresponding instance:*

```
class Enum a where
  pred, succ :: a -> a
  toEnum :: Int -> a

instance Enum Int where
  pred n = n - 1
  succ n = n + 1
  toEnum n = n
```

The translation yields:

```
constraint Enum a = Pred (a->a), Succ (a->a),
                    ToEnum (Int->a)
rule Pred x ==> x = a->a, Enum a
rule Succ x ==> x = a->a, Enum a
rule ToEnum x ==> x = Int->a, Enum a

overload pred :: Int -> Int
         pred n = n - 1
overload succ :: Int -> Int
         succ n = n + 1
overload toEnum :: Int -> Int
         toEnum n = n
```

Given an expression

```
f = map succ
```

the Haskell type system would assign the type

[3] We ignore Haskell's unfortunate defaulting to $\perp$.

[4] We do not consider modules and related name spacing issues.

```
f :: Enum a => [a] -> [a]
```

Under the Chameleon encoding, the same definition would be typed as

```
f :: Pred (a->a), Succ (a->a), ToEnum (Int->a) =>
     [a] -> [a]
```

Note that in the encoding, the constraint abbreviation `Enum a` expands to the same constraint as found in this context. Section 3.7 describes how for user presentation a type pretty printer could reverse-apply constraint abbreviations to actually display the shorter but equivalent type shown by a Haskell system.

Example 11. *Consider*

```
class TC a where
  f :: a -> Int
  g :: Bool -> a

instance TC Bool where
  f False = 0
  f True  = 1
```

Our translation yields:

```
constraint TC a = F (a->Int), G (Bool->a)

rule F x ==> x = a->Int, TC a
rule G x ==> x = Bool->a, TC a

overload f :: Bool->Int
         f True  = 1
         f False = 0
```

Note that the program theory consisting of

```
rule F x            ==> x = a->Int, TC a
rule G x            ==> x = Bool->a, TC a
rule F (Bool->Int) <==> True
```

is non-confluent. `F (Bool->Int)` can be either reduced to `True` or to `G (Bool->Bool)`. Clearly, there was a problem in the original Haskell program: we forgot to provide a definition for the member function `g`.

We conclude that the additonal propagation rules enforce that all uses of a method must conform to its declaration. Confluence ensures that all method definitions belonging to the same class must be provided.

3.2 Polymorphic Methods

We deliberately excluded polymorphic methods from the discussion in the previous section. In Chameleon, there is no problem in defining overloaded functions that are polymorphic in more than one variable. For example, we can straight-forwardly overload the `map` function as follows:

```
overload map :: (a->b) -> [a] -> [b]
         map = ...
```

However, to extend our type class encoding to methods polymorphic in more than the class variable alone, we have to apply a slight trick, because we do not want to lift these additional variables to class parameters.

Consider the following class:

```
class TC a where
    m1 :: a
    m2 :: a -> b -> b
```

Note that function `m2` is parametric in `b` but overloaded on `a`. We have to take this into account in our encoding of type classes. That is, the class membership relation states the following for `m1`:

$$\forall a.(M_1\ a \rightarrow \forall b.M_2\ (a \rightarrow b \rightarrow b)) \quad (3.1)$$

But the propagation rule

```
rule M1 a ==> M2 (a->b->b)
```

represents the logically weaker statement

$$\forall a.(M_1\ a \rightarrow \exists b.M_2\ (a \rightarrow b \rightarrow b)) \quad (3.2)$$

Fortunately, there exists a well-known technique to eliminate universal quantifiers. We simply introduce a new skolem constant for the universally quantified variable b. That is, statement 3.1 can be equivalently formulated as

$$\forall a.(M_1\ a \rightarrow M_2\ (a \rightarrow Erk \rightarrow Erk)) \quad (3.3)$$

where Erk is a new skolem constant. That formula corresponds to the following propagation rule[5]:

```
rule M1 a ==> M2 (a->Erk->Erk)
```

In summary, we generalise the encoding as follows. Consider a class of the form

```
class TC a where
  m1 :: t1
  ...
  mk :: tk
```

Let us assume that no type variable (apart from `a`) occurs in more than one of the type signatures — this can be achieved by trivial renaming. For each such type variable `bi`, we generate a fresh skolem type name `Bi`. Let φ be the corresponding substitution mapping each `bi` to `Bi`. To encode the class `TC`, we can then generate the abbreviation

```
constraint TC a = M1 t1', ..., Mk tk'
```

where $\texttt{ti}' = \varphi\,\texttt{ti}$. For each method `mi` we generate

```
rule Mi x ==> x = ti, TC a
```

Note that `ti` may not be skolemised in the latter rule. No change is required to the translation of instance declarations.

Example 12. *The class*

```
class IntColl c where
    empty :: c
    fold  :: (Int -> a -> a) -> a -> c -> a
```

is encoded as

```
constraint IntColl c = Empty c,
                       Fold ((Int->A->A)->A->c->A)
rule Empty x ==> x = c, IntColl c
rule Fold x  ==> x = (Int->a->a)->a->c->a, IntColl c
```

Under these declarations, the application `fold (+)` will be typed as `Int->c->Int` for some `c` and give rise to the constraint

[5]Note that rules may mention arbitrary type symbols.

```
Fold ((Int->Int->Int)->Int->c->Int),
  Empty c, Fold ((Int->A->A)->A->c->A)
```

Note that the second Fold constraint is essential to enforce that fold is not merely defined for Int, but is really polymorphic. For that reason, it is important that the Fold rule expands to

```
rule Fold x ==> x = (Int->a->a)->a->c->a,
                Empty c, Fold ((Int->A->A)->A->c->A)
```

That is, we have a Fold constraint re-appearing on the right-hand side. While this was redundant in the case of monomorphic methods, it is necessary now to capture the fact that fold is itself polymorphic.

Unfortunately, it is not possible to encode constrained method types easily. In a class like

```
class TC a where
    f :: Eq b => a -> b
```

the logical meaning of class membership of f is

$$F\ x \rightarrow x = s \wedge \forall b.(Eq\ b \rightarrow (a \rightarrow b))$$

Since there now appears an implication under the quantifier, the skolemisation trick no longer applies — we either needed a higher-order propagation rule or an ad-hoc check to implement that formula. See section 4.4 for a possible approach to express higher-order rules.

3.3 Superclasses

A superclass constraint can be encoded by adding a simple propagation rule. Consider:

```
class TC1 t1, ..., TCn tn => TC a where ...
```

A straight-forward encoding employs the rule

```
rule TC a ==> TC1 t1, ..., TCn tn
```

Note the use of constraint abbreviations on both sides of the rule here.

How do we enforce that overloaded declarations are consistent with the intended superclass hierarchy? A complete set of instances corresponds to a confluent set of CHRs. Mapping a Haskell program that lacks a super instance will result in a non-confluent program theory and thus yield a static error.

EXAMPLE 13. *Consider*

```
class A a where f :: a -> Int
class A a => B a where g :: a -> Int
instance B Int where g = ...
```

Our translation yields the following Chameleon program:

```
constraint A a = F (a->Int)
constraint B a = G (a->Int)

rule F x ==> x = a->Int, A a    -- 1
rule G x ==> x = a->Int, B a    -- 2
rule B a ==> A a                -- 3

overload g :: Int -> Int
         g = ...
```

The overloaded definition for g corresponds to an additional simplification rule

```
rule G (Int->Int) <==> True    -- 4
```

Any application of g to an integer will yield the constraint G (Int->Int) which can be either reduced directly to True via rule 4 or, in several steps using the third rule first, to F (Int->Int), with no further applicable rule. The program's theory is not confluent and the confluence checker will reject it. If, on the other hand, a definition for f :: Int->Int was included, then the second constraint could be reduced to True as well and confluence would be established.

3.4 Constructor Classes

The most prominent example of a constructor class is

```
class Functor f where
    fmap :: (a -> b) -> (f a -> f b)
```

According to this declaration, each instance t of the member function fmap must be of the form (t1 -> t2) -> (T t1 -> T t2) for some appropriate types t1 and t2 and type constructor T. Such side conditions can easily be expressed in Chameleon via appropriate CHRs:

```
constraint Functor f = Fmap ((A->B) -> f A -> f B)
rule Fmap x ==> x = (a->b)->(f a->f b), Functor f
```

In addition, we also need to ensure *kind* correctness. Chameleon follows the approach described in [11]. We omit the details.

3.5 Multi-Parameter Type Classes and Functional Dependencies

Classes, respectively methods, with multiple type parameters can naturally be expressed in Chameleon. The same holds for functional dependencies.

EXAMPLE 14. *Recall example 1 from section 2:*

```
class Collects ce e | ce -> e where
   empty  :: ce
   insert :: ce -> e -> ce
```

In Chameleon, we would express this as:

```
constraint Collects (ce,e) = Empty ce,
                             Insert (ce->e->ce)
rule Empty x  ==> x = ce, Collects (ce,e)
rule Insert x ==> x = ce->e->ce, Collects (ce,e)
rule Collects (ce,e), Collects (ce,e') ==> e = e'
```

We simply use tuple types to embed multiple parameters. The last rule encodes the functional dependency ce⇝e.

More generally, assume we want to express the functional dependency a1...am⇝b1...bn for a multi-parameter class TC a1...am b1...bn c1...ck (i.e. ci are additional class variables not mentioned in the particular functional dependency). A single propagation rule is always sufficient:

```
rule TC (a1,...,am,b1,...,bn,c1,...,ck),
     TC (a1,...,am,b1',...,bn',c1',...,ck')
 ==> b1 = b1', ..., bn = bn'
```

Issues related to completion of program theories in case we employ CHRs to model functional dependencies are discussed in Section 2.2.

3.6 Expressiveness

We have seen that Chameleon's overloading mechanism almost completely subsumes type classes in their current incarnations (with the exception of constrained methods). What about the inverse? That is, given an arbitrary Chameleon program, can we translate it into Haskell? The answer is no: there is no way to map arbitrary propagation rules, not even single-headed ones. And even overloaded definitions alone cannot be mapped in general, at least not to Haskell 98 — a direct encoding for overloaded declarations would be the following:

1. For every overloaded identifier `x` generate

   ```
   class X a where x :: a
   ```

2. For every overload declaration

   ```
   overload x :: C => t; x = e
   ```

   generate

   ```
   instance C => X t where x = e
   ```

In general, neither the head `X t` nor the context `C` of this instance will be valid Haskell 98, which restricts the syntactic form of these phrases severely, in order to guarantee decidability. A more general head is valid in GHC [16] with its extensions, a more general context with its undecidable instances switch only (however, decidability of type checking in Chameleon should guarantee that this never actually makes the GHC type checker loop for a translated program). Allowing undecidable instances is also necessary to deal with the extreme case of

```
overload x :: a
```

We conclude that overloading in Chameleon mostly subsumes type classes in Haskell. The essence of type classes are two restrictions on overloaded definitions:

1. Every definition must adhere to the type signature given in the class.
2. Definitions at a particular type must always be given for all methods of a class (and all its super-classes).

Propagation rules can be used to impose all necessary constraints to enforce (1) (with the exception constraints in method types). Confluence of these propagation rules and the simplification rules implied by overloaded definitions will enforce (2). Furthermore, constraint abbreviations allow to give names to concrete sets of constraints and thus provide the necessary potential for abstraction over concrete sets of methods, like with type classes. Consequently, type classes might mostly be regarded as syntactic sugar in our framework.

3.7 Simplification

For user presentation it is common to "simplify" constraints. An obvious simplification step is to remove equality constraints by building most general unifiers. Simplification becomes more tricky in case of superclass relationships specified via propagation rules.

Assume we find the following dependencies:

```
rule A x ==> B x
rule B x ==> C x
rule C x ==> D x
```

Then, constraints `A t, D t` and `A t, B t, C t, D t` are equivalent for any type `t`. Clearly, we would like to achieve the "best" representation of constraints when presenting the result to the user.

In Chameleon this could be achieved by turning the above propagation rules into multi-headed simplification rules:

```
rule A x, B x <==> A x
rule B x, C x <==> B x
rule C x, D x <==> C x
rule A x, C x <==> A x
rule A x, D x <==> A x
rule B x, D x <==> B x
```

Note that the last three rules are necessary to ensure confluence. These rules could be generated by an automatic method, similar to the method discussed in section 2.2. Likewise, we could turn constraint abbreviations into simplification rules. In general, we assume that these rules are only applied for user interaction. Such a mechanism has not been implemented yet.

4. BEYOND TYPE CLASSES

We present an alternative treatment of constructor classes. We also show that in Chameleon we can overcome some of the context restrictions found in Haskell.

4.1 Alternative Constructor Classes

Note that the propagation rule imposed on `fmap` in Section 3.4 is crucial.

EXAMPLE 15. *Consider*

```
cmap f g = (fmap g) . (fmap f)
```

Without `rule Fmap a ==> a = (b->c)->(f b->f c)` we find that

```
cmap :: Fmap (a'->a->b), Fmap (b'->b->c) =>
        a' -> b' -> a -> c
```

Note that the bound type variable `b` in `Fmap (a'->a->b)` does not appear in the type component. Therefore, `cmap`'s type is ambiguous. In contrast, the above CHR enforces

```
cmap :: Fmap ((a->b)->(f a->f b)),
        Fmap ((b->c)->(f b->f c))
        => (a->b)->((b->c)->(f a->f c))
```

Now, `cmap`'s type is unambiguous.

Maybe surprisingly, the sole purpose of the functor class seems to circumvent ambiguity problems. In fact, this is also the motivation provided in [11]. The concept of constructor classes simply allows us to enforce similar side conditions as the above CHR. Given the ability to formulate almost arbitrary side conditions in Chameleon, we present an alternative treatment of "constructor classes". We restrict our attention to the `Functor` class.

We postulate the following four conditions on `fmap`, each of which can be encoded via CHRs:

1. `fmap` should transform one function into another function.

   ```
   rule Fmap a ==> a = (b->c)->(fb->fc)
   ```

2. The input type `b` of the input function and the output type `fc` of the transformed function uniquely determine `c` (which is the output type of the input function) and `fb` (which is the input type of the transformed function).

   ```
   rule Fmap ((b->c)->(fb->fc)),
        Fmap ((b->c')->(fb'->fc))
        ==> b=b', fb=fb'
   ```

3. Similarly, `c` and `fb` uniquely determine `b` and `fc`.

   ```
   rule Fmap ((b->c)->(fb->fc)),
        Fmap ((b'->c)->(fb->fc'))
        ==> b=b', fc=fc'
   ```

4. The transformed function uniquely determines the input function.

   ```
   rule Fmap ((b->c)->(fb->fc)),
        Fmap ((b'->c')->(fb->fc))
        ==> b=b', c=c'
   ```

Note that the last three conditions essentially state some functional dependencies. Indeed, the above conditions could have been coded up by functional dependencies:

```
class Fmap a b fa fb | a fb -> b fa,
                       b fa -> a fb,
                       fa fb -> a b where
   fmap :: (a->b)->(fa -> fb)
```

The important insight is that the four CHRs specified above are sufficient to avoid the ambiguity problem in the function definition of `cmap`. Recall example 15. Now, we find that

```
cmap :: Fmap ((a2->a3)->(a->b)),
        Fmap ((b2->b3)->(b->c))
        => (a2->a3)->(b2->b3)->a->c
```

Note that type variable `b` is not mentioned in the type component. However, `b` is functionally defined by `b2` and `c`. Therefore, `cmap`'s type is unambiguous.

This alternative approach towards functors allows us to type some interesting programs. Assume that in addition to functors we would like to support cofunctors. We represent cofunctors in Chameleon via an overloaded function `comap`. Similarly to the four conditions imposed on `fmap`, we impose four conditions on `comap`.

Example 16. *Below follows the Chameleon code.*

```
overload fmap :: (b->c)->(a->b)->(a->c)
         fmap f g = f . g

overload comap :: (a->b)->(b->c)->(a->c)
         comap f g = g . f

rule Comap a ==> a = (b->c)->(fc->fb)
rule Comap ((b->c)->(fc->fb)),
     Comap ((b->c')->(fc->fb'))
     ==> b=b', fb=fb'
rule Comap ((b->c)->(fc->fb)),
     Comap ((b'->c)->(fc'->fb))
     ==> b=b', fc=fc'
rule Comap ((b->c)->(fc->fb)),
     Comap ((b'->c')->(fc->fb))
     ==> b=b', c=c'
```

Note that it is not straightforward anymore to switch back to the common presentation of constructor classes as found in Haskell. The overloaded definition of `fmap` on type `(b->c)->(a->b)->(` corresponds to an instance `Functor (->) a`. However, we encounter problems if we try to represent `comap` on type `(a->b)->(b->c)->(a->c)` using constructor classes. Assume we define cofunctors as follows:

```
class Cofunctor f where
  comap :: (a->b)->(f b->f a)
```

Then we would require for abstraction in the type language. The desired, but not valid, class instance would be `Cofunctor (\x -> (->) x c)`. It is certainly an interesting problem to investigate how to extend Haskell constructor classes while retaining decidable inference. Note that allowing for unrestricted abstraction in the type language immediately leads to undecidable type inference. In such a situation, the typing problem can be reduced to higher-order unification, which is undecidable. In our alternative formulation of constructor classes we simply avoid such problems altogether. We believe that more variations of "constructor classes" are possible. We leave this topic for future work.

4.2 Generalised Superclasses

When designing more complex class hierarchies one often reaches the limits of what is legal Haskell, even with respect to common extensions. Okasaki reports that in the design of the Edison library [17] he encountered examples like the following:

```
class (Eq k, Functor (m k)) => AssocX m k
class (UniqueHash a, CollX c Int)
      => CollX (HashColl c) a
```

Both these examples are not valid Haskell. Apart from the fact that they use multiple parameter type classes, the contexts appearing in these class declarations are not in "head form", i.e. the class predicates are applied to something else than type templates of the form `T a1...an`.

Such examples are no problem in the more general setting described here. They simply stand for the perfectly valid propagation rules

```
rule AssocX (m,k)          ==> Eq k, Functor (m k)
rule CollX (HashColl c,a) ==> UniqueHash a,
                              CollX (c,Int)
```

4.3 Superclasses with Universal Quantification

More involved are problems requiring universal quantification in superclass contexts. These arise when a class is "higher-kinded" than one of its superclasses. The following example is taken from Peyton Jones et.al. [13]. Consider

```
class (forall s. Monad (m s)) => StateMonad m
```

The idea is that the superclass context indicates that `m s` should be a monad for any type `s`. That is, the following should hold:

$$\forall m.(StateMonad\ m \rightarrow \forall s.Monad\ (m\ s))$$

Note that we have universal quantification on the right-hand side of the $\rightarrow$ symbol. Again, skolemization does the job — the above statement is equivalent to

$$\forall m.(StateMonad\ m \rightarrow Monad\ (m\ Erk))$$

where we have replaced the universal quantification over s by a new skolem constant Erk. The Chameleon formulation is

```
rule StateMonad m ==> Monad (m Erk)
```

Consider we would like to define a class for sequences like in Okasaki's paper, but want to require any instance to support equality. The class declaration had to look like this:

```
class (forall a . Eq (s a)) => Sequence s
```

Of course, this class constraint is overly restrictive, since we usually can only define equality on a container, if we have equality on its elements. We thus refine the constraint to:

```
class (forall a . (Eq a => Eq (s a))) => Sequence s
```

This expresses that any instance `S T` of a sequence type must support equality, as long as `T` does. In other words, there has to be an instance of the form:

```
instance Eq a => Eq (S a)
```

For any constructor `S` that is an instance of `Sequence`. A more general instance

```
instance Eq (S a)
```

would also be valid, of course.

We can express that directly in our framework using a simple propagation rule. Logically, the class declaration represents the implication

$$\forall s.Sequence\ s \rightarrow (\forall a.Eq\ a \rightarrow Eq\ (s\ a))$$

which is equivalent to

$$\forall s, a.Sequence\ s \rightarrow (Eq\ a \rightarrow Eq\ (s\ a))$$

which again is equivalent to

$$\forall s, a.Sequence\ s \wedge Eq\ a \rightarrow Eq\ (s\ a)$$

We can directly turn this last form into a propagation rule:

```
rule Sequence s, Eq a ==> Eq (s a)
```

Interestingly, the encoding of the latter example is actually simpler than that of the `StateMonad` class above, although it looks more complicated in Haskell. The reason is that we directly support multi-headed propagation rules, for which no equivalent exists in Haskell.

4.4 Instances with Universal Quantification

Hinze and Peyton Jones [7] give another example, where universal quantification appears in an instance context. They have a class similar to

```
class Binary a where bin :: t -> [Bool]
```

and a higher-order datatype representing generalised rose trees based on an arbitrary sequence type:

```
data GRose s a = GBranch a (s (GRose s a))
```

An instance declaration for `Binary` on this type requires universal quantification:

```
instance (Binary a,
          forall b . Binary b => Binary (f b))
         => Binary (GRose f a)
```

We cannot directly express this in our framework either, because it corresponds to a higher-order simplification rule:

```
rule Bin (GRose s a->[Bool]) <==>
     Bin a, (Bin (s b) <==> Bin b)
```

The intuition is that application of the above rule brings a new rule, `rule Bin (s b) <==> Bin b`, into scope.

Higher-order CHRs have not been investigated so far. However, it is possible to simulate such higher-order CHRs via some multi-headed simplification rules. Consider

```
rule Bin (GRose s a -> [Bool]) <==> Bin a, Tok s
rule Tok s, Bin (s b)          <==> Tok s, Bin b
```

These two rules achieve the same operational effect as the higher-order CHR above. Note that we had to invent a new constraint `Tok s`. To utilize that encoding we needed to extend the Chameleon typing rules to allow forgetting these special constraints, such that the inference algorithm was allowed to clean them up at appropriate points. We leave thorough investigation of these ideas to future research.

5. TYPE PROGRAMMING

The ability to specify arbitrary propagation rules allows the user to "customise" type inference to her needs. As a small example of the type-level programming possible in Chameleon, we look at operations polymorphic in the arity of tuples. In order to enable inductive definitions we assume that the base language is equipped with *extensible* tuples, i.e. an n-ary tuple type $(t_1, \cdots, t_n)$ is actually equivalent to the type $(t_1, \cdots (t_n, ()) \cdots)$ of nested pairs, terminated by the unit type $()$. The more general concept of extensible records has been proposed in various flavours [?, ?]. Interestingly enough, some approaches use constraints to enforce wellformedness [?], but we will not explore this here.

An interesting example is the generic `uncurry` function:

```
overload uncurry :: f->()->f
         uncurry f () = f
overload uncurry :: Uncurry (f'->t->r) =>
                    ((x->f')->(x,t)->r)
         uncurry f (x,t) = uncurry (f x) t
```

Given these instances, we can turn given functions into uncurried form and apply them to an appropriate argument tuple. Assume `h :: Int -> Int -> Bool -> String`. Then we can apply

```
uncurry h (2,3,True)
```

But this is not good enough. Consider we want to abstract over the concrete function:

```
fun f = uncurry f (2,3,True)
```

Then the type checker can only infer

```
fun :: Uncurry(f->(Int,Int,Bool)->a) => f -> a
```

although we would like to see

```
fun :: (Int->Int->Bool->a) -> a
```

The open world assumption underlying overloading does not allow the type checker to infer this type from the constraint given above. But we have propagation rules at our hands to force it to!

```
rule Uncurry x              ==> x = f->t->r
rule Uncurry (f->()->f')    ==> f = f'
rule Uncurry (f->(x,t)->r)  ==> f = x->g
```

With these additional rules the type checker is able to infer the desired type. The explicit CHRs and the ones induced by the definitions of `uncurry` are sufficient to derive the number and types of arguments for the function `f` from the type of the argument tuple. The type of the whole expression is then determined by `f`'s result type. Confluence will ensure that no later overloaded definition may conflict with the typing restrictions imposed by the propagation rules.

On the other hand, if we only pass an argument function to `uncurry`, without applying the result to a tuple, type inference can still not determine a concrete type for the application, even if the function's type is known already:

```
fun2 = uncurry h
```

The inferred type will just be

```
fun2 :: Uncurry ((Int->Int->Bool->String)->a) => a
```

And rightly so! There is no way of telling how many arguments shall be uncurried before seeing the actual tuple the function shall be applied to. The inferred type thus allows any choice that is consistent with the type of `h`. Because of polymorphism, it even allows different instantiations:

```
exp2 = fun2 (2,3)
exp3 = fun2 (2,3,True)
```

While `exp3` obviously is a string, `exp2` has the function type `Bool->String`. We might even apply `fun2` to `()` and get back the original `h`.[6]

The reverse operation, `curry`, is not much more difficult:

```
overload curry :: (()->r)->r
         curry f = f ()
overload curry :: (Curry ((t->r)->g)) =>
                  ((x,t)->r)->(x->g)
         curry f = \x -> curry (\t -> f (x,t))
```

Again we need additional propagation rules:

```
rule Curry x                ==> x = (t->r)->f
rule Curry ((()->r)->r')    ==> r = r'
rule Curry (((x,t)->r)->f)  ==> f = x->g
```

Alternatively, we could utilize the equivalence

`Curry ((t->r)->f)` $\Longleftrightarrow$ `Uncurry (f->(t->r))`

and replace the three rules by just

```
rule Curry ((t->r)->f) ==> Uncurry (f->(t->r))
```

for the same effect.

A similar definition of the `curry` and `uncurry` functions could have been given in Haskell, utilizing multi-parameter type classes with functional dependencies.

AR**Shortly mention other nice examples.**

[6] In the system of extensible tuples there is also syntax for 1-tuples, namely `(t,())`.

6. TYPE PROGRAMMING

The ability to specify arbitrary propagation rules allows the user to "customise" type inference to her needs. But CHRs as available in Chameleon are actually much more powerful than this: they allow quite sophisticated forms of type-level programming.

We are interested in specifying the syntax and semantics of a simple functional language. A standard approach would introduce the following algebraic data types:

```
data Type = TypeInt | TypePair Type Type
          | TypeFunc Type Type

data Exp  = ExpVar Int | ExpConst Int
          | ExpPair Exp Exp | ExpPi1 Exp | ExpPi2 Exp
          | ExpAbs Int Exp | ExpApp Exp Exp
```

Note that we use numbers to encode variables.

An interpreter is quickly written for our simple language:

```
data Value = ValConst Int | ValPair Value Value
           | ValFunc (Value->Value)
type Env   = [(Int,Value)]

lookup :: Env -> Int -> Value
lookup ((var,val):env) x
   | var == x  = val
   | otherwise = lookup env x

eval                 :: Env -> Exp -> Value
eval e (ExpVar v)    = lookup e v
eval e (ExpConst n)  = ValConst n
eval e (ExpPair x y) = ValPair (eval e x) (eval e y)
eval e (ExpPi1 x)    = case (eval e x) of
                         ValPair y z -> y
eval e (ExpPi2 x)    = case (eval e x) of
                         ValPair y z -> z
eval e (ExpAbs x y)  = ValFunc (\z -> eval ((x,z):e) y)
eval e (ExpApp x y)  = case (eval e x) of
                         ValFunc f -> f (eval e y)
```

Note that the above interpreter is written in *indirect* style. We use a universal data type `Value` to represent values of any type by a value of one (universal) type. The insertion of tags such as `ValConst` and `ValFunc` is necessary to make our interpreter type check.

It has been observed that not all tags are necessary at runtime to ensure the correct evaluation of interpreted object programs. The removal of unnecessary tags has recently attracted some attention [?, ?].

6.1 Direct-Style Interpreter

In Chameleon it is possible to provide a *direct* style formulation. We introduce singleton types to perform some compile-time manipulations of values.

```
-- language of object expressions
data ExpVar x      = ExpVar x
data ExpConst n    = ExpConst n
data ExpPair a1 a2 = ExpPair a1 a2
data ExpPi1 a      = ExpPi1 a
data ExpPi2 a      = ExpPi2 a
data ExpAbs x a    = ExpAbs x a
data ExpApp a1 a2  = ExpApp a1 a2

-- result of type-level computations
data T = T
data F = F

-- we use numbers to model variables
data Zero   = Zero
```

```
data Succ n = Succ n

overload eq :: Zero -> Zero -> T
         eq Zero Zero = T
overload eq :: Eq (a->b->c) => Succ a -> Succ b -> c
         eq (Succ a) (Succ b) = eq a b
overload eq :: Zero -> Succ a -> F
         eq Zero (Succ a) = F
overload eq :: Succ a -> Zero -> F
         eq (Succ a) Zero = F

-- we use lists for the environment
data Nil      = Nil
data Cons a b = Cons a b
```

We define a type-safe lookup function:

```
overload lookup ::
         (Lookup' (Cons (x1,v) e -> x2 -> b -> v'),
          Eq (x1->x2->b)) => Cons (x1,v) e -> x2 -> v'
         lookup (Cons (x1,v) e) x2 =
                lookup' (Cons (x1,v) e) x2 (eq x1 x2)

overload lookup' :: Cons (x1,v) e) -> x2 -> T -> v
         lookup' (Cons (_,v) e) _ T = v
overload lookup' :: Lookup (e->x2->v') =>
                    Cons (x1,v) e -> x2 -> F -> v'
         lookup' (Cons (x1,v) e) x2 F = lookup e x2
```

Note that we use the auxilliary function `lookup'` to mimic the behaviour of the guard clauses in function `lookup`.

We define our interpreter in direct style as follows:

```
overload eval :: (Lookup (e->x->v))
              => e -> ExpVar x -> v
         eval e (ExpVar x) = lookup e x
overload eval :: e -> ExpConst n -> n
         eval e (ExpConst n) = n
overload eval :: (Eval (e->a1->v1), Eval (e->a2->v2))
              => e -> ExpPair a1 a2 -> (v1,v2)
         eval e (ExpPair a1 a2) = (eval e a1, eval e a2)
overload eval :: (Eval (e->a->(v1,v2)))
              => e -> ExpPi1 a -> v1
         eval e (ExpPi1 a) = fst (eval e a)
overload eval :: (Eval (e->a->(v1,v2)))
              => e -> ExpPi2 a -> v2
         eval e (ExpPi2 a) = snd (eval e a)
overload eval :: (Eval ((Cons (x,v1) e)->a->v2))
              => e -> ExpAbs x a -> (v1->v2)
         eval e (ExpAbs x a) = \v1 -> eval (Cons (x,v1) e) a
overload eval :: (Eval (e->a1->(v2->v1)), Eval (e->a2->v2))
              => e -> ExpApp a1 a2 -> v1
         eval e (ExpApp a1 a2) = (eval e a1) (eval e a2)
```

To make this open set of overloaded definitions behave as if it were closed, we have to add propagation rules:

```
rule Eval (e->(ExpVar x)->v)
 ==> Lookup (e->x->v)
rule Eval (e->(ExpConst n)->v)
 ==> v = Int
rule Eval (e->(ExpPair a1 a2)->v)
 ==> v = (v1,v2), Eval (e->a1->v1), Eval (e->a2->v2)
rule Eval (e->(ExpPi1 a)->v1)
 ==> Eval (e->a->(v1,v2))
rule Eval (e->(ExpPi1 a)->v2)
 ==> Eval (e->a->(v1,v2))
rule Eval (e->(ExpAbs x a)->v)
 ==> v = v1->v2, Eval ((Cons (x,v1) e)->a->v2)
rule Eval (e->(ExpApp a1 a2)->v1)
 ==> Eval (e->a1->(v2->v1)), Eval (e->a2->v2)
```

Note that we have one rule per instance. This is not possible with functional dependencies, which can only be defined on a per-class basis. In particular, if we tried to write something similar in Haskell using a multi-parameter type class `Eval e a v` with the functional dependency `e a`$\leadsto$`v`, then the definition for `ExpAbs` would violate that dependency.

Now consider the expression

```
exp :: ExpApp (ExpConst Int) (ExpConst Int)
exp =  ExpApp (ExpConst 1) (ExpConst 2)

res = eval Nil exp
```

We can already statically determine that expression `res` cannot be evaluated. The required instance constraint `Eval (Nil -> ExpCons Int -> (a->b))` cannot be reduced to `True`.

6.2 Type Inference

As an additional excercise we write a type inference algorithm.

```
-- language of object types
data TypeInt       = TypeInt
data TypePair t1 t2 = TypePair t1 t2
data TypeFunc t1 t2 = TypeFunc t1 t2
```

We can write an inferencer as a set of CHRs. The constraint `Infer (env -> exp -> typ)` computes expression's `exp` type `typ` under the environment `env`.

```
overload infer :: (Lookup (e->x->t))
               => e -> ExpVar x -> t
         infer e (ExpVar x) = reify
overload infer :: e -> ExpConst Int -> TypeInt
         infer e (ExpConst n) = reify
overload infer :: (Infer (e->a1->t1), Infer (e->a2->t2))
               => e -> ExpPair a1 a2 -> TypePair t1 t2
         infer e (ExpPair a1 a2) = reify
overload infer :: (Infer (e->a->TypePair t1 t2))
               => e -> ExpPi1 a -> t1
         infer e (ExpPi1 a) = reify
overload infer :: (Infer (e->a->TypePair t1 t2))
               => e -> ExpPi2 a -> t2
         infer e (ExpPi2 a) = reify
overload infer :: (Infer ((Cons (x,t1) e)->a->t2))
               => e -> ExpAbs x a -> TypeFunc t1 t2
         infer e (ExpAbs x a) = reify
overload infer :: (Infer (e->a1->TypeFunc t2 t),
                   Infer (e->a2->t2))
               => e -> ExpApp a1 a2 -> t
         infer e (ExpApp a1 a2) = reify
```

Note the "fake" method implementations. They use the overloaded function

```
overload reify :: TypeInt
         reify =  TypeInt
overload reify :: (Reify t1, Reify t2) => TypePair t1 t2
         reify =  TypePair reify reify
overload reify :: (Reify t1, Reify t2) => TypeFunc t1 t2
         reify =  TypeFunc reify reify
```

All inference happens on the type level. It is not directly possible to implement it on the value level without going through the hassle of implementing unification (because we have to guess when typing abstractions). Using the type-level programming facilities, this is provided for free. We can than reify the singleton type inferred by the constraints to get back an actual value that might be printed, for example.

Again, we can use propagation rules to "close" the definition w.r.t. the expression types we use in order to actually trigger the static computation:

```
rule Infer (e->(ExpVar x)->t)
 ==> Lookup (e->x->t)
rule Infer (e->(ExpConst n)->t)
 ==> t = TypeInt
rule Infer (e->(ExpPair a1 a2)->t)
 ==> t = TypePair t1 t2,
     Infer (e->a1->t1), Infer (e->a2->t2)
rule Infer (e->(ExpPi1 a)->t1)
 ==> Infer (e->a->(TypePair t1 t2))
rule Infer (e->(ExpPi2 a)->t2)
 ==> Infer (e->a->(TypePair t1 t2))
rule Infer (e->(ExpAbs x a)->t)
 ==> t = t1->t2, Infer ((Cons (x,t1) e)->a->t2)
rule Infer (e->(ExpApp a1 a2)->t)
 ==> Infer (e->a1->(TypeFunc t2 t)), Infer (e->a2->t2)
```

Note again that we could not express this using functional dependencies — the `ExpApp` case is not functional.

We can combine evaluation and inference by the following definition:

```
exec :: (Eval (env->exp->val), Infer (tenv->exp->typ))
     => tenv -> env -> exp -> (typ, val)
exec tenv env exp = (infer tenv exp, eval env exp)
```

As a further refinement, we could also formalize that the value environment `env` and type environment `tenv` "agree".

AR**I wonder, isn't `Infer` completely redundant? We could as well define:**

```
overload reify :: Int -> TypeInt
         reify _ = TypeInt
overload reify :: (Reify (t1->t1'), Reify (t2->t2')) =>
                  (t1,t2) -> TypePair t1' t2'
         reify _ = TypePair (reify undefined) (reify undefined)
overload reify :: (Reify (t1->t1'), Reify (t2->t2')) =>
                  (t1->t2) -> TypeFunc t1' t2'
         reify _ = TypeFunc (reify undefined) (reify undefined)

-- and now:
exec :: Eval (env->exp->val) => env -> exp -> (typ, val)
exec env exp = (reify v, v) where v = eval env exp
```

AR**I.e., type inference is already built-in into evaluation. Or am I floating?**

7. CONCLUSION

One of the main exercises of the present paper is to establish a connection between the form of overloading found in Haskell and the CHR-based overloading in Chameleon. Chameleon allows us to go "beyond" type classes as found in Haskell. Many desired type class extensions can now simply be programmed via some set of CHRs. In Section 4, we discussed some novel features such as universal quantification in context. We also presented an alternative treatment of "constructor classes" which might be worthwhile to pursue further.

The ability of type programming in Haskell has already been recognized for a while, see e.g. [5]. In some recent work [4], Gasbichler, Neubauer, Sperber and Thiemann suggest to incorporate a functional-logic language on the type level. Indeed, as shown in [15], Haskell instance declaration with functional dependencies can most often be reformulated as functional programs. Certainly, it might be a matter of taste which kind of language is preferable on the level of types. The logic/constraint programming style provided by CHRs might not necessarily suit the taste of every functional programmer. However, one of the benefits of the CHR language is that some concise results are available which ensure decidable type inference [21] and a well-defined semantics [20] for Chameleon. Compare this to Augustsson's [1] dependently typed language Cayenne which comes with great expressiveness but gives no guarantees regarding decidable type inference. We even believe that a logic language such as CHRs seems to have some advantages when designing some complex superclass relations as shown in Section 4.3. We should also mention that a first-order functional language can always be easily translated into CHRs. Therefore, the programmer might not even get in touch with the underlying type level language. An interesting topic we leave for future work is to employ the power of an expressive type language to capture some program/data invariants [24].

8. REFERENCES

[1] L. Augustsson. Cayenne - a language with dependent types. In *Proc. of ICFP'98*, pages 239–250, 1998.

[2] K. Chen, P. Hudak, and M. Odersky. Parametric type classes. In *Proc. of ACM Conference on Lisp and Functional Programming*, pages 170–191, 1992.

[3] B. Demoen, M. García de la Banda, W. Harvey, K. Marriott, and P.J. Stuckey. An overview of HAL. In *Proc. of the Fourth International Conference on Principles and Practices of Constraint Programming*, pages 174–188, 1999.

[4] M. Gasbichler, M. Neubauer, M. Sperber, and P. Thiemann. Functional logic overloading. In *Proc. of POPL'02*, pages 233–244, 2002.

[5] Benedict R. Gaster and Mark P. Jones. A polymorphic type system for extensible records and variants. Technical report, University of Nottingham, November 1996.

[6] Thomas Hallgren. Fun with functional dependencies or (draft) types as values in static computations in haskell. Proc. of the Joint CS/CE Winter Meeting, January 2001.

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

[8] Ralf Hinze. A generic programming extension for Haskell. In *Proc. of the Third Haskell Workshop*, 1999.

[9] J. Hughes. Restricted data types in Haskell. In *Proc. of the Third Haskell Workshop*, 1999.

[10] D. Jeffery, F. Henderson, and Z. Somogyi. Type classes in Mercury. In *Proc. Twenty-Third Australasian Computer Science Conf.*, pages 128–135, 2000.

[11] M. P. Jones. *Qualified Types: Theory and Practice.* D.phil. thesis, Oxford University, 1992.

[12] M. P. Jones. A system of constructor classes: Overloading and implicit higher-order polymorphism. In *FPCA '93: Conference on Functional Programming Languages and Computer Architecture*, pages 52–61, 1993.

[13] M. P. Jones. Type classes with functional dependencies. In *Proc. of ESOP'00*, pages 230–244, 2000.

[14] S. Peyton Jones, M. P. Jones, and E. Meijer. Type classes: an exploration of the design space. In *Proc. of the Second Haskell Workshop*, 1997.

[15] S. Kaes. Parametric overloading in polymorphic programming languages. In *Proc. of ESOP'88*, pages 131–141, 1988.

[16] M. Gasbichler M. Neubauer, P. Thiemann and M. Sperber. A functional notation for functional dependencies. In *Proc. of the Fifth Haskell Workshop*, 2001.

[17] Simon Marlow, Simon Peyton Jones, and Others. *The Glasgow Haskell Compiler*. University of Glasgow, `http://www.haskell.org/ghc/`, 2002.

[18] Atsushi Ohori. A polymorphic record calculus and its compilation. In *ACM Transactions on Programmoing Languages and Systems*, volume 17, pages 844–895, 1995.

[19] C. Okasaki. An overview of Edison. *Electronic Notes in Theoretical Computer Science*, 41(1):35–55, 2001.

[20] S. Peyton Jones et al. Report on the programming language Haskell 98, February 1999. http://haskell.org.

[21] 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.

[22] Didier Rémy. Type inference for records in a natural extension of ML. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design*. MIT Press, 1993.

[23] A. Rossberg and M. Sulzmann. A theory of overloading part II: Semantics and coherence. Technical report, The University of Melbourne, 2002.

[24] P. J. Stuckey and M. Sulzmann. A theory of overloading. In *Proc. of ICFP'02*, 2002. to appear.

[25] M. Sulzmann et. al. The Chameleon Language Manual, May 2002. `http://www.cs.mu.oz.au/~sulzmann/chameleon`.

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

[27] Hongwei Xi. Dependently Typed Data Structures. In *Proc. WAAAPL'99*, pages 17–32, 1999.