The Chameleon System

Peter J. Stuckey *, Martin Sulzmann †and Jeremy Wazny ‡

April 22, 2004

Abstract

We give an overview of the major features of the Chameleon programming language, as well as use of the system, from a Haskell programmer's point of view. Chameleon supports type-class overloading a la Haskell. The novelty of Chameleon is to allow the programmer to specify almost arbitrary conditions on type classes in terms of Constraint Handling Rules (CHRs). In fact, Chameleon's entire type inference system and evidence translation of programs is phrased in terms of CHRs.

1 Introduction

Chameleon is a Haskell-style language to experiment with advanced type extensions. Chameleon covers most of the Haskell syntax [Has]. In contrast to Haskell, we only support singly-overloaded methods.

Example 1 *For instance, consider the following Haskell 98 type class and instance declarations.*

```
class Eq a where (==) :: a->a->Bool
instance Eq a => Eq [a] where
 (==) [] [] = True
 (==) (x:xs) (y:ys) = ((==) x y) && ((==) xs ys)
 (==) _ _ = False
```

The above example is written as follows in Chameleon.

Example 2

```
overload eqC

instance EqC (a->a->Bool) => EqC ([a]->[a]->Bool) where
 eqC [] [] = True
 eqC (x:xs) (y:ys) = (eqC x y) && (eqC xs ys)
 eqC _ _ = False

rule EqC a ==> a=t->t->Bool
```

*Dept. of Computer Science and Software Engineering, The University of Melbourne, Parkville 3052, Australia. pjs@cs.mu.oz.au

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

‡Dept. of Computer Science and Software Engineering, The University of Melbourne, Parkville 3052, Australia. jeremyrw@cs.mu.oz.au

In Chameleon each constraint `Method t` refers to an overloaded use of identifier `method` at type `t`. Hence, the statement `overload eqC` is similar to `class EqC a where eqC :: a` in Haskell. The novelty of Chameleon is to write almost arbitrary programs (on the level of types) in terms Constraint Handling Rules (CHRs) [Frü95]. The Chameleon programmer can introduce such CHRs via the `rule` keyword. For example, the above CHR `rule EqC a ==> a=t->t->Bool` (in combination with `overload eqC`) essentially mimics the Haskell class declaration `class Eq a where (==) :: a->a->Bool`. Note that in Chameleon we do not impose any conditions on type class constraints appearing in the instance context. E.g. Haskell 98 demands that all type class parameters must be variables.

In the following, we present an overview of the major features of the Chameleon programming language, as well as use of the system. In Section 2 we give an overview of the kind of CHRs found in Chameleon. In Section 3 we consider type inference whereas in Section 4 we consider translating Chameleon, i.e. replacing uses of overloaded methods by some specific instances. In Section 5 we consider some extended examples which make use of Chameleon's CHR-based type programming capabilities. In Section 6 we show how to make use of existing Haskell code within Chameleon. Finally, in Section 7 we show how to use the system to execute Chameleon programs.

The latest Chameleon distribution is available via

```
http://www.comp.nus.edu.sg/~sulzmann/chameleon
```

2 Constraint Handling Rules

In general, type inference works by *generating* and *solving* constraints. Here, constraint solving is performed w.r.t. the relations imposed on overloaded identifiers which are specified in terms of CHRs. For example, in Haskell the declaration `class Eq a where (==) :: a->a->Bool` states that any use of `(==)` must be of shape `t->t->Bool` where the constraint `Eq t` refers to a particular instance of the equality type class. In Chameleon, we can simply write

```
rule EqC a ==> a=t->t->Bool
```

to achieve the same behavior (keep in mind that constraints in Chameleon refer to exactly one method). Via the `rule` keyword the Chameleon programmer can introduce the following two forms of CHRs.

```
rule Method1 t1, ..., Methodn tn ==> s=t
```

and

```
rule Method1 t1, ..., Methodn tn ==> Method t
```

Both kinds are referred to as CHR *propagation* rules. The first kind allows us to impose stronger conditions on the shape of types. The second kind of CHRs is useful to impose dependencies among overloaded instances.

Example 3 *For instance, consider the following Chameleon program.*

```
overload ordC
rule OrdC a ==> EqC a
rule OrdC a ==> a=t->t->Bool
instance OrdC (a->a->Bool) => OrdC ([a]->[a]->Bool)
```

Note that if the instance body is omitted we assume the instance definition is undefined. The constraint symbol `OrdC` refers to an ordering relation. Hence, it is natural to require that each such instance implies an equality instance `EqC`. Indeed the above CHRs are pretty much equivalent to `class Eq a => Ord a where (<) :: a->a->Bool` in Haskell.

2.1 Confluence

In Chameleon, CHRs are checked for "consistency" before we proceed with type inference.

Example 4 *Consider the following Chameleon example where we have dropped* `OrdC (a->a->Bool)` *from the instance context.*

```
overload eqC
overload ordC

instance EqC (a->a->Bool) => EqC ([a]->[a]->Bool) where  -- (E1)
 eqC [] [] = True
 eqC (x:xs) (y:ys) = (eqC x y) && (eqC xs ys)
 eqC _ _ = False

rule EqC a ==> a=t->t->Bool

instance {-OrdC (a->a->Bool)=>-} OrdC ([a]->[a]->Bool)   -- (O1)

rule OrdC a ==> a=t->t->Bool

rule OrdC a ==> EqC a                                    -- (E2)
```

Chameleon complains that CHRs are "non-confluent". Note that each `instance C => U t` implies a (single-headed) *simplification* CHR of the form `rule U t <==> C` (the programmer does not have to specify such simplification rules explicitly, they are automatically generated from instances). We generally require that CHRs are *confluent*, i.e. the order of CHR applications does not matter. This is not the case for the above example. In case we encounter `Ord([a]->[a]->Bool)`, we can either apply rule (O1) which yields the empty constraint represented by `True`. Alternatively, we can apply rule (E2) which yields `Ord([a]->[a]->Bool),Eq ([a]->[a]->Bool)`. Applications of rules (E1) and (O1) finally leads to `Eq (a->a->Bool)` which is different from `True`. Hence, CHRs are non-confluent. In fact, a similar check is enforced in Haskell, however, the notion of confluence is more general. Note that if one may wish, the confluence check can be switched off via the "–no-conf" option.

2.2 Termination

For terminating CHRs, we can test for confluence by checking all "critical pairs". In fact, if the confluence check in Chameleon is invoked we assume that CHRs are terminating. However, we can easily write a "non-terminating" Chameleon program.

```
instance Erk [a] => Erk a
```

The CHR implied by the above instance is non-terminating. By default Chameleon performs a simple termination check. Note that CHRs are a Turing-complete language. Hence, any termination check must be necessarily incomplete. Via the "–no-term" option, the user can switch off the termination check.

2.3 Range-Restriction

Consider

```
instance C (Int->Int->Int)
instance C (a->b->c) => D ([a]->[b]) -- (D1)
```

Chameleon complains that the CHR resulting from (D1) is not "range-restricted". Note that if we ground variables `a` and `b` we still find the non-ground term `c` in the instance context. Range-restrictedness ensures that there are no unconstrained variables which potentially endangers completeness of type inference. In the upcoming Section 5.2 we consider an instance (Select) which is non-rangestricted but safe! Via the "–no-rr" option, the user may switch off the check for range-restriction.

2.4 CHR Solver

A description of the internal details of the CHR solver are beyond the scope of this document. Note that Chameleon comes with a fully-fledged CHR solver (on the level of types). That is, we also support *multi-headed* simplification rules.

Example 5 *In the following Chameleon program we model stack operations (on the level of types) such as push and pop. Their meaning is defined in terms of CHRs.*

```
interface Prelude -- will be explained in detail later
                  -- in essence, we teach Chameleon how to use the Haskell 98 Prelude

-- we model the stack as a singleton list
data Nil = Nil
data Cons x xs = Cons x xs

-- some stack elements
data V1 = V1
data V2 = V2
data V3 = V3

overload push
overload pop
overload stack

-- two multi-headed simplification rules
rule Push (x), Stack (s)      <==> Stack (Cons x s)
rule Pop (), Stack (Cons x s) <==> Stack (s)
rule Pop (), Stack (s)         ==> s=Cons x s'

-- we introduce a CHR to model initial constraint store
overload initStore
rule InitStore a <==> Stack (Nil), Push (V1), Push (V2)

run = initStore
```

Obviously, we cannot run (on the level of values) the above program. But we can execute the CHR program by invoking Chameleon's CHR-based type inference mechanism. For example, the call `chameleon --no-term -d stack.ch` invokes Chameleon in pure type inference mode (-d) where we have switched of our CHR termination check (–no-term). We reach a command-line environment with similar functionality compared to HUGS [HUG] or GHCi (besides executing program code) [GHC]. For example,

```
stack.ch> :t run
(Stack(Cons V2 (Cons V1 Nil))) => a
```

allows us to execute the above CHR program. The constraint component of the above type holds the result of running the above CHR program on the given constraint store.

3 Type Inference

Type inference in Chameleon proceeds by generating the appropriate constraints out of the program text and solving them w.r.t. the given set of CHRs. A program is type correct if there are no inconsistent constraints and all user-provided type annotations are correct. Additionally, we also check that expressions are unambiguous (though the user can switch off this check via the `--no-unambig` option). In case of a type error, incorrect type annotation or ambiguous expression Chameleon will issue an error message. The user can then invoke the Chameleon Type Debugger [SSW03b, SSW03a] to get some assistance in fixing such errors.

In contrast to other Haskell implementations such as Hugs and GHC we do not pretty-print types. Consider

Example 6

```
interface Prelude

overload eqC
overload ordC

rule EqC a ==> a=t->t->Bool
rule OrdC a ==> a=t->t->Bool
rule OrdC a ==> EqC a

f = ordC
```

If we invoke Chameleon's type inference/debugging environment (via `chameleon -d eq2C.ch`) and query the type of `f` we find the following

```
eq2C.ch> :t f
(EqC(a -> a -> Bool),OrdC(a -> a -> Bool)) => a -> a -> Bool
```

The constraint `EqC(a -> a -> Bool)` might be considered redundant (but harmless). Indeed, `OrdC(a -> a -> Bool) => a -> a -> Bool` is equivalent to the above type. We plan to incorporate pretty-printing of types in future versions of Chameleon.

In our current implementation type inference is more "lazy" compared to the HUGS or GHC.

Example 7 *Consider*

```
interface Prelude

f x = let h = (x::Int)
      in x
```

Chameleon reports that `f` has type `a->a` whereas e.g. Hugs reports `Int->Int`. Note that `h` only constraints the type of `x` without actually being used in `f`'s function body. Hence, Chameleon neglects the constraints associated to `h`. Obviously, we could easily enforce "strict" type inference but we have not done so in our current implementation.

Note also that Chameleon does not enforce the Haskell monomorphism restriction. Furthermore, there is no defaulting in Chameleon. For example, the following program is accepted by Chameleon, while Haskell would reject it.

```
p :: (Int, Float)
p = let n = 1
    in  (n,n)
```

The monomorphism restriction and defaulting often eliminate bothersome sources of ambiguity in Haskell programs. Consequently, there are ambiguous Chameleon programs whose Haskell counterparts are unambiguous. Consider the following:

```
one = if 1 == 1 then 1
                else undefined
```

Chameleon will report this as ambiguous, whereas Haskell will accept it.

It has been observed that all known Haskell implementations reject some seemingly well-typed programs.

Example 8 *Here is a recast of a Haskell program published on the Haskell-Cafe [JN00] mailing list in terms of Chameleon.*

```
interface Prelude

-- we model
-- class C t where op :: t->Bool
-- instance C [a]
hconstraint C
extern op :: C t => t->Bool
rule C [a] <==> True

p y = (let f :: c->Bool
           f x = op (y >> return x)
       in f,  y ++ [])

q y = (y ++ [],
       let f :: c->Bool
           f x = op (y >> return x)
       in f)
```

The above program is type correct. Note that Hugs and GHC either reject `p` or `q` whereas Chameleon accepts both functions!

In a recent paper [SSW04] we elaborate on this topic in detail.

Example 9 *Here is another interesting example from this paper*

```
interface Prelude

-- class HasEmpty a where isEmpty :: a->Bool
hconstraint HasEmpty
extern isEmpty :: HasEmpty a => a->Bool

hconstraint Erk

{-
instance HasEmpty [a]
instance HasEmpty (Maybe a)
```

```
instance Erk m => HasEmpty (m a)
instance Erk []
instance Erk Maybe
-}

rule HasEmpty [a] <==> True       -- (H1)
rule HasEmpty (Maybe a) <==> True -- (H2)
rule HasEmpty (m a) <==> Erk m    -- (E1)
rule Erk [] <==> True             -- (E2)
rule Erk Maybe <==> True          -- (E3)

test :: (Monad m, HasEmpty (m (m a))) => m a -> Bool
test y = let f :: d->Bool
             f x = isEmpty (y >> return x)
         in f y
```

We can invoke Chameleon via `chameleon --no-conf -d hasempty.ch` to check that the above program is type correct. We need to switch off the confluence check because the above instances are overlapping. For details we refer to the above mentioned paper.

4 Translating Chameleon

To resolve overloading on the value level we apply the well-known "evidence" translation scheme. Chameleon is translated to "plain" Haskell. However, there are some subtle differences among Chameleon-style and Haskell-style overloading.

Example 10 *Consider a variation of Example 4. Note that instance (O1) includes context* `OrdC (a->a->Bool)`. *Hence, CHRs are confluent.*

```
overload eqC
overload ordC

instance EqC (a->a->Bool) => EqC ([a]->[a]->Bool) where
 eqC [] [] = True
 eqC (x:xs) (y:ys) = (eqC x y) && (eqC xs ys)
 eqC _ _ = False

rule EqC a ==> a=t->t->Bool

instance OrdC (a->a->Bool) => OrdC ([a]->[a]->Bool)  -- (O1)

rule OrdC a ==> a=t->t->Bool

rule OrdC a ==> EqC a

f :: OrdC (a->a->Bool) => a->a->Bool           -- incorrect type annotation!
f = eqC
```

The above looks like a perfectly reasonable program. Surprisingly, Chameleon rejects the above program because `f`'s type annotation is incorrect. Note that in Haskell `Ord a` refers to a type class which contains a method `(<)` of type `a->a->Bool` and (due to sub-classing) also refers to an equality method (==) of type a->a->Bool. However, in Chameleon, each constraint Method t refers exactly to an overloaded identifier method of type t. Hence, the statement rule OrdC a ==> EqC a has only a meaning on the level of types, i.e. for each OrdC t there must also be a EqC t. In order to resolve overloading on the value level, we need to explicitly mention a constraint Method t for each use of method at type t.

Example 11 *Consider the following variation of Example 10. In the function body of* f *we find now* ordC *instead of* eqC*.*

```
interface Prelude -- will be explained in detail later
                  -- in essence, we teach Chameleon how to use the Haskell 98 Prelude

overload eqC
overload ordC

instance EqC (a->a->Bool) => EqC ([a]->[a]->Bool) where
 eqC [] [] = True
 eqC (x:xs) (y:ys) = (eqC x y) && (eqC xs ys)
 eqC _ _ = False

rule EqC a ==> a=t->t->Bool

instance OrdC (a->a->Bool)=> OrdC ([a]->[a]->Bool) where
 ordC = undefined
-- if the instance body is left empty we assume the "undefined" instance body by default

rule OrdC a ==> a=t->t->Bool

rule OrdC a ==> EqC a

f :: OrdC (a->a->Bool) => [a]->[a]->Bool
f = ordC
```

The translation yields the following result (call chameleon -o eqordC.hs eqordC.ch).

```
import Prelude

ec3 :: ((->) (((->) (a)) (((->) (a)) (Bool)))) (((->) ([(a)])) (((->) ([(a)])) (Bool)))
ec3 (pEqCIIARR_v18026I_IIARR_v18026I_BoolII) ([]) ([]) = let {pEqCIIARR_a18373I_IIARR_a18373I_BoolII
= pEqCIIARR_v18026I_IIARR_v18026I_BoolII} in  (True)
ec3 (pEqCIIARR_v18026I_IIARR_v18026I_BoolII) ((:) (x) (xs)) ((:) (y) (ys)) = let {pEqCIIARR_a18373I_II
ARR_a18373I_BoolII  = pEqCIIARR_v18026I_IIARR_v18026I_BoolII} in  ((&&) (((pEqCIIARR_a18373I_IIARR_a18
373I_BoolII) (x)) (y)) ((((ec3) (pEqCIIARR_a18373I_IIARR_a18373I_BoolII)) (xs)) (ys)))
ec3 (pEqCIIARR_v18026I_IIARR_v18026I_BoolII) (_) (_) = let {pEqCIIARR_a18373I_IIARR_a18373I_BoolII  =
pEqCIIARR_v18026I_IIARR_v18026I_BoolII} in  (False)

ec83 :: ((->) (((->) (a)) (((->) (a)) (Bool)))) (((->) ([(a)])) (((->) ([(a)])) (Bool)))
ec83 (pOrdCIIARR_v18174I_IIARR_v18174I_BoolII) = let {pOrdCIIARR_a18427I_IIARR_a18427I_BoolII  = pOrdC
IIARR_v18174I_IIARR_v18174I_BoolII} in  (undefined)

f :: ((->) (((->) (a)) (((->) (a)) (Bool)))) (((->) ([(a)])) (((->) ([(a)])) (Bool)))
f (pOrdCIIARR_v18236I_IIARR_v18236I_BoolII) = let {pOrdCIIARR_a18542I_IIARR_a18542I_BoolII  = pOrdCIIA
RR_v18236I_IIARR_v18236I_BoolII} in  ((ec83) (pOrdCIIARR_a18542I_IIARR_a18542I_BoolII))
```

which can be executed by any Haskell compiler. Briefly, we turn instance declarations into function definitions. E.g. `ec3` corresponds to `instance EqC (a->a->Bool) => EqC ([a]->[a]->Bool)`. Note the correspondence between `ec3`'s (pretty-printed) type annotation `(a->a->Bool)->[a]->[a]->Bool` and the instance declaration.

Obviously, the Chameleon user does not need to understand the the details of the implementation scheme behind Chameleon. The curious reader is referred to [SW] for more details. Note in our latest release we have incorporated the *weak* entailment check discussed on page 6 in [SW]. .

5 Examples

The real fun of programming in Chameleon starts when writing programs on the level of types. We briefly illustrate Chameleon's type level programming ability in connection with "functional dependencies". Functional dependencies [Jon00] are a popular type class extension where the programmer can take (to some extent) control over the type inference process. Consider the following Haskell program.

```
class C a b | a->b where c :: a->b
instance C Int Bool
```

The functional dependency `| a->b` states that in `C a b` the first component uniquely determines the second component. For example, if type inference encounters `C a b` and `C a c`, types are *improved* by adding the equality constraint `b=c`. In case of `C Int b` the functional dependency in combination with above the instance implies that `b=Bool`. In Chameleon, we can make such *improvement rules* explicit in terms of CHRs. Here is the Chameleon equivalent of the above program.

```
overload c
rule C x ==> x=a->b
-- FD rule
rule C (a->b), C (a->c) ==> b=c
instance C (Int->Bool)
-- instance improvement rule
rule C (Int->b) ==> b=Bool
```

In fact, CHRs not only give us a sound foundation for functional dependencies, CHRs even allow us to go beyond functional dependencies. The interested reader is referred to [GJDS04] for more details. Here, we briefly present two examples making use of more powerful functional dependencies expressible.

5.1 A Generic Family of Zip Functions

We start off with an encoding in Haskell, point out some limitations of functional dependencies and then present a solution in Chameleon.

```
-- Haskell encoding
  zip2 :: [a]->[b]->[(a,b)]
  zip2 (a:as) (b:bs) = (a,b) : (zip2 as bs)
  zip2 _      _      = []

  class Zip a b c | a c -> b, b c -> a where
    zip :: [a] -> [b] -> c
```

```
instance Zip a b [(a,b)] where
  zip = zip2
instance Zip (a,b) c e => Zip a b ([c]->e) where
  zip as bs cs = zip (zip2 as bs) cs

e2 = head (zip [True,False] ['a','b','c'])
e3 = head (zip [True,False] ['a','b','c'] [1::Int,2])
```

Note that the inferred types of e2 and e3 are

```
e2 :: Zip Bool Char [a] => a
e3 :: Zip (Bool,Char) Int [a] => a
```

rather than

```
e2 :: (Bool,Char)
e3 :: ((Bool,Char),Int)
```

Indeed, the relevant improvement rule here stated in terms of a CHRs is as follows

```
rule Zip a d [(a,b)]      ==> d=b
```

To achieve the desired typing for e2 and e3 we would like to state a slightly stronger condition such as

```
rule Zip a d [c] ==> c=(a,b), d=b
```

However, there is no mechanism in Haskell which allows us to do so.

```
------------
-- zip.ch --
------------
-- Chameleon encoding
-- we point out parts where Chameleon allows us to impose a stronger improvement rule
interface Prelude

overload zip

zip2 :: [a] -> [b] -> [(a,b)]
zip2 (x:xs) (y:ys) = (x,y) : (zip2 xs ys)
zip2 _ _ = []

-- enforce
-- class Zip ([a]->[b]->c) where zip :: [a]->[b]->c
rule Zip x ==> x=[a]->[b]->c

-- FD
rule Zip (a->b->c), Zip (a->d->c) ==> b=d
rule Zip (a->b->c), Zip (d->b->c) ==> a=d

-- base case
instance Zip ([a]->[b]->[(a,b)])  where
   zip= zip2
```

```
-- instance-improvement
-- rule Zip ([a]->c->[(a,b)]) ==> c=[b]   -- as specified by FD
rule Zip ([a]->d->[c]) ==> d=[b],c=(a,b)  -- though we employ a slightly stronger rule
                                          -- to achieve the desired typing

rule Zip (c->[b]->[(a,b)]) ==> c=[a]

-- general case
instance Zip ([(a,b)]->[cs]->r) => Zip ([a]->[b]->[cs]->r) where
         zip as bs cs = zip (zip2 as bs) cs

-- instance improvements for this case are trivial

-- annoations are not necessary here
--e1 :: [(Bool,Char)]
e1 = head (zip [True,False] ['a','b','c'])
--e2 :: [((Bool,Char),Int)]
e2 = head (zip [True,False] ['a','b','c'] [1::Int,2])

-- to run this program
-- you need to call "chameleon -o zip.hs zip.ch"
-- you can then run zip.hs
```

Obviously, we should not add arbitrary CHRs otherwise we might endanger soundness and decidability of type inference.

5.2 Extensible Records

As another application we present an encoding of extensible records in Chameleon. The basic idea is to encode field labels and records in terms of singleton types. Note that a similar encoding with functional dependencies would be too "weak".

```
---------------
-- record.ch --
---------------
interface Prelude

-- we use singleton lists to model records

data Nil = Nil
data Cons x xs = Cons x xs

-- we use singleton numbers to model record labels

data Zero   = Zero
data Succ n = Succ n

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

```
------ overloaded identifiers ------

overload eqR
overload select
overload select'
overload remove
overload remove'
overload extend
overload check
overload unique
overload unique'


------ equality among types ------

-- class declaration
rule EqR x ==> x=a->b->c

-- FD
rule EqR (a->b->c), EqR (a->b->d) ==> c=d

-- instances
instance EqR (Zero -> Zero -> T) where
         eqR Zero Zero = T
instance EqR (a->b->c) => EqR (Succ a -> Succ b -> c) where  -- (E2)
         eqR (Succ a) (Succ b) = eqR a b
instance EqR (Zero -> Succ a -> F) where
         eqR Zero (Succ a) = F
instance EqR (Succ a -> Zero -> F) where
         eqR (Succ a) Zero = F

-- instance improvement, note that for (E2) instance improvement is trivial
rule EqR (Zero->Zero->a)         ==> a=T
rule EqR (Succ a->Zero->b)       ==> b=F
rule EqR (Zero->Succ a->b)       ==> b=F


------ record selection ------

-- usage: select Record Label --> FieldEntry

-- class declaration
rule Select x ==> x=r->l->a

-- FD
rule Select (r->l->a), Select (r->l->b) ==>a=b
```

```
-- instance
instance (Select' (Cons (x1,v) e -> x2 -> b -> v'), EqR (x1->x2->b))
                => Select (Cons (x1,v) e -> x2 -> v') where
         select (Cons (x1,v) e) x2 = select' (Cons (x1,v) e) x2 (eqR x1 x2)
-- Chameleon complains that this instance is not range-restricted
-- However, we can argue that variable b (which is the trouble-maker) is
-- functionally defined by variables in the instance head

-- instance improvement is trivial here

-- class declaration
rule Select' x ==> x=e->c->a->b

-- FD
rule Select' (e->c->a->b),Select' (e->c->a->d) ==> b=d

instance Select' (Cons (x,v) e -> x -> T -> v) where
         select' (Cons (_,v) e) _ T = v
instance Select (e->x2->v') => Select' (Cons (x1,v) e -> x2 -> F -> v') where  -- (S'3)
         select' (Cons (x1,v) e) x2 F = select e x2

rule Select' (Cons (a, b) e -> c -> T -> d) ==> b=d

-- things we couldn't do with FDs
-- stronger improvement rules

rule Select (Nil->l->a) ==> False
rule Select' (Nil->l->t->a) ==> False
rule Select' ((Cons (x1,v1) e)->x2->T->v2) ==> x1=x2, v1=v2

------ record extension ------

-- we don't check whether a field already exists
-- usage: extend (Label,Entry) Record --> Record


-- class declaration

rule Extend a ==> a=(l,x)->r->r'

-- FD
rule Extend (a->b->r), Extend(a->b->r') ==> r=r'

-- instances
instance Extend ((l,x)->xs->Cons (l,x) xs) where
         extend (l,x) xs = Cons (l,x) xs

-- instance improvement
rule Extend ((l,x)->xs->ys) ==> ys=Cons (l,x) xs
```

```
------ check that all record labels are unique ------

-- usage: check Record --> Record
-- the identify operation if record is unique, otherwise type error

instance (Unique x) => Check (x->x) where
         check x = x

rule Check a ==> a=t->t

instance Unique Nil where
         unique = undefined

instance (Unique ys, Unique' (l,ys)) => Unique (Cons (l,x) ys) where
         unique = undefined

instance Unique' (l,Nil) where
         unique' = undefined

instance (EqR (l->l'->a), Fail a, Unique' (l',ys)) => Unique' (l', Cons (l,x) ys) where
         unique' = undefined

instance Fail F where
         fail = undefined

rule Fail T ==> False

------ field removal ------

-- we don't check whether a field doesn't exist
-- usage: remove Label Record --> Record

-- class declaration
rule Remove a ==> a=l->r->r'

-- FD
rule Remove (l->r->r'), Remove (l->r->r'') ==> r'=r''

-- instances
instance Remove (l -> Nil -> Nil) where
         remove l Nil = Nil

instance (EqR (l->l'->a), Remove' (a->l->(Cons (l',v) r)->r'))
                 => Remove (l -> (Cons (l',v) r) -> r') where
         remove l (Cons (l',v) r) = remove' (eqR l l') l (Cons (l',v) r)

-- instance improvements
```

```
rule Remove (l->Nil->r') ==> r'=Nil

-- class declaration
rule Remove' x ==> x=t->l->r->r'

-- FD
rule Remove' (t->l->r->r1), Remove' (t->l->r->r2) ==> r1=r2

-- instances
instance Remove' (T -> l -> (Cons (l',v) r) -> r) where
         remove' T l (Cons (l',v) r) = r

instance Remove (l->r->r') => Remove' (F -> l -> (Cons (l',v) r) -> (Cons (l',v) r')) where
         remove' F l (Cons (l',v) r) = Cons (l',v) (remove l r)

-- instance improvement
rule Remove' (T->a->(Cons (b,c) d)->e) ==> e=d
rule Remove' (F->a->(Cons (b,c) d)->e) ==> e=Cons (b,c) e'

------ examples ------

l0 = Zero
l1 = Succ l0
l2 = Succ l1

data MyString = A | B | C
data Phone  = P1 | P2


r1 = Nil
--r2 :: Cons (Zero,MyString) Nil
r2 = extend (l0,A) r1
r2' = check r2

r3 = extend (l1,P1) r2
r3' = check r3
-- r3 = extend (l0,P1) r2 yields type error,
--  check r3 verifies that all labels are unique!

x :: MyString
x = select r2 l0

y = remove l0 r3

z = extend (l1, P2) r3
-- note that (check z) yields an error

-- to run this program
-- call "chameleon --no-term --no-rr -o record.hs record.ch"
```

```
--  --no-term switches off the termination checker
--  --no-rr allows for non-range-restricted CHRs
```

6 Using Haskell from Chameleon

Programs written in Chameleon may make use of Haskell libraries directly, provided that an appropriate Chameleon interface file exists for each library. Library interfaces must be imported into a Chameleon program before the corresponding Haskell library can be used.

The keyword `interface Module`, allows you to access all of the functions and constructors defined in the Haskell module with the same name, provided that a corresponding interface file exists in the current directory. The interface file for Haskell module `Module` must be called `Module.ch`, and must be present in the current directory.

The following Chameleon program makes use of functions provided by the Haskell 98 Prelude.

```
interface Prelude
myReverse = foldl (flip (:)) [] <br>
palin xs = myReverse xs == xs <br>
```

In our translation of Chameleon programs, we simply replace `interface` by `import` statements. The Haskell 98 Standard Prelude interface file

```
http://www.comp.nus.edu.sg/~sulzmann/chameleon/interface/Prelude.ch
```

is distributed along with the Chameleon source code. Interfaces for other libraries remain to be created.

6.1 Interface Files

Interface files are nothing more than Chameleon source files. From the point-of-view of the compiler, an `interface` declaration causes the contents of the interface file to be treated as though they were part of the one program. Consequently, interface files should declare only the types of functions or constructors they provide, since their implementation is already part of an existing Haskell library. Note that we assume that inferface files must be in the same directory as the Chameleon program.

The `extern` keyword can be used to declare the types of both functions and data constructors without providing a definition. It may also be used to declare type constructors.

Its two forms are:

- `extern` *sig*: Declares a function/constructor, with the given type, e.g. `extern fst :: (a,b) -> a`, declares a function `fst` with the usual type; `extern Just :: a -> Maybe a` , declares the `Just` constructor of the standard Haskell `Maybe` type.
- `extern` *id*: Declares a new type constructor, *id*, e.g. `extern Maybe` declares the `Maybe` type constructor which could then be used to declare the type of `Just` as above.

Note that the Chameleon system currently does not perform any kind checking.

Consider the following Haskell library code (taken from the Haskell 98 Prelude Documentation):

```
data  Maybe a  =  Nothing | Just a
maybe              :: b -> (a -> b) -> Maybe a -> b
maybe n f Nothing  =  n
maybe n f (Just x) =  f x
```

To make use of this, Chameleon requires an interface file containing the following:

```
extern Maybe
extern Nothing :: Maybe a
extern Just :: a -> Maybe a
extern maybe :: b -> (a -> b) -> Maybe a -> b
```

Note that the correctness of the interface with respect to the original Haskell code must be ensured by the user.

Haskell class and instance declarations must be manually translated to CHRs. For details we refer to [SS02]. We plan to incorporate an automatic translation in the near future. All Haskell constraints (type class constraints) must also be declared using `hconstraint TC` declarations. E.g. the interface to a library declaring a type class `Eq`, must contain the declaration: `hconstraint Eq`. Note that the user must guarantee that the number of parameters for each type class are consistent throughout the program.

6.2 Basic Types and Literals in Chameleon

Chameleon has built-in support for:

- Character literals of form: `'`*char*`'`
- String literals, like `"A string."`
- Numeric literals, both integer and floating point.
- Boolean values, `True` and `False`

Any of these may appear as terms or as patterns.

The following types are also available by default: function (infix) `t1->t2`, list `[t]`, and tuples up to 7 arguments ; and basic types `Bool`, `Int`, `Char`. Note that the Prelude interface provides access to all of the standard Haskell types - although their corresponding values might not be easily expressed in Chameleon for now (e.g. Rationals.)

The following Haskell type class constraints are also known to Chameleon: `Num`, `Fractional`, `Enum` and `Monad`. No instances are declared by default - the standard Prelude interface provides all the type class instances you'd expect in Haskell 98.

6.3 Missing Haskell Features

Although Chameleon is closely related to Haskell, not all Haskell language features are supported in Chameleon. Notable omissions from Chameleon include:

- `class` and `instance` declarations
- the module system (Chameleon has only the simple notion of "interfaces" as mentioned above)
- record syntax for data types

Please note also the differences between Haskell and Chameleon type inference as outlined earlier.

Support for these (and other features) may be added in future.

6.4 Mixing Haskell and Chameleon Overloading

Though, we cannot define full Haskell classes and instances we can make use of Chameleon's overloading mechanism. Indeed, as long as all Haskell constraints are declared as such `hconstraint`, we can freely mix the two forms of overloading.

Consider the following Chameleon program.

```
interface Prelude
overload insert
rule Insert x ==> x=e->ce->ce
instance Ord a => Insert (a->[a]->[a]) where
   insert x [] = [x]
   insert x (y:ys) = if x < y then x:y:ys else y:(insert x ys)
```

Note that `Ord` is a Haskell type class whereas `Insert` refers to a overloaded Chameleon identifier. When translating such mixed code, we perform evidence translation of overloaded Chameleon identifiers. However, Haskell type classes are left untouched and will be dealt with when running the translated Chameleon program through a Haskell compiler.

In fact, by mixing Haskell and Chameleon overloading via interfaces allows us to directly express the zip and record Example in terms of a user-definable Haskell extension! Assume in `MZip.hs` we find the following Haskell code.

```
-------------
-- MZip.hs --
-------------
module MZip where
  mzip2 :: [a]->[b]->[(a,b)]
  mzip2 (a:as) (b:bs) = (a,b) : (mzip2 as bs)
  mzip2 _      _      = []

  class MZip a b c | a c -> b, b c -> a where
      mzip :: [a] -> [b] -> c
  instance MZip a b [(a,b)] where
      mzip = mzip2
  instance MZip (a,b) c e => MZip a b ([c]->e) where
      mzip as bs cs = mzip (mzip2 as bs) cs
```

Assume in `MZip.ch` we find the following interface specification which correctly describes the class and instance relations from above.

```
-------------
-- MZip.ch --
-------------
hconstraint MZip
extern mzip :: MZip a b c => [a] -> [b] -> c
rule MZip a b c, MZip a d c ==> b=d       -- (FD1)
rule MZip a b c, MZip d b c ==> a=d       -- (FD2)

rule MZip a b [(a,b)] <==> True             -- (Inst1)
rule MZip a b ([c]->e) <==> MZip (a,b) c e  -- (Inst2)

-- instance improvement linked to (Inst1)
```

```
rule MZip a d [(a,b)]      ==> d=b         -- (Impr1a)
rule MZip d b [(a,b)]      ==> d=a         -- (Impr1b)

-- note that instance improvement for (Inst2) is trivial
```

Then, we can define our own Haskell extension as follows.

```
--------------------
-- MZip-strong.ch --
--------------------
interface Prelude
interface MZip

-- we're strengthening rules (Impr1a) and (Impr1b)
rule MZip a d [c] ==> c=(a,b), d=b
rule MZip d b [c] ==> c=(a,b), d=a

e2 = head (mzip [True,False] ['a','b','c'])
e3 = head (mzip [True,False] ['a','b','c'] [1::Int,2])
```

Chameleon translates the above program as follows (via `chameleon -o MZip-strong.hs MZip-strong.ch`).

```
import Prelude
import MZip

e2 :: (Bool,Char)
e2 = head (mzip [True,False] ['a','b','c'])
e3 :: ((Bool,Char),Int)
e3 = head (mzip [True,False] ['a','b','c'] [1::Int,2])
```

That is, in our own Haskell extension we could sufficiently improve types such that a Haskell compiler can finally run the above program.

7 Running Chameleon Programs

Chameleon programs are translated into Haskell 98, which must be further compiled, before they can be run. The process is as follows.

- Run `chameleon` *`InputFile.ch`* `-o` *`OutputFile.hs`*
- Run your favorite Haskell 98 interpreter/compiler on the resulting output file.
  e.g. `hugs` *`Outfile.hs`*

For a brief description of available command line options, run `chameleon --help`.

References

[Frü95] T. Frühwirth. Constraint handling rules. In *Constraint Programming: Basics and Trends*, LNCS. Springer-Verlag, 1995.

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

[GJDS04] P. J. Stuckey G. J. Duck, S. Peyton-Jones and M. Sulzmann. Sound and decidable type inference for functional dependencies. In *Proc. of ESOP'04*, volume 2986 of *LNCS*, pages 49–63. Springer-Verlag, 2004.

[Has] Haskell 98 language report. http://research.microsoft.com/Users/simonpj/haskell98-revised/haskell98-report-html/.

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

[JN00] M. P. Jones and J. Nordlander, 2000. http://haskell.org/pipermail/haskell-cafe/2000-December/001379.html.

[Jon00] M. P. Jones. Type classes with functional dependencies. In *Proc. of ESOP'00*, volume 1782 of *LNCS*. Springer-Verlag, 2000.

[SS02] P. J. Stuckey and M. Sulzmann. A theory of overloading. In *Proc. of ICFP'02*, pages 167–178. ACM Press, 2002.

[SSW03a] P. J. Stuckey, M. Sulzmann, and J. Wazny. The Chameleon type debugger. In *Proc. of Fifth International Workshop on Automated Debugging (AADEBUG 2003)*, pages 247–258. Computer Research Repository (http://www.acm.org/corr/), 2003.

[SSW03b] P.J. Stuckey, M. Sulzmann, and J. Wazny. Interactive type debugging in Haskell. In *Proc. of Haskell Workshop'03*, pages 72–83. ACM Press, 2003.

[SSW04] P. J. Stuckey, M. Sulzmann, and J. Wazny. Type annotations in Haskell, 2004. submitted for publication.

[SW] M. Sulzmann and J. Wazny. Implementing overloading in Chameleon. `http://www.comp.nus.edu.sg/~sulzmann/chr/download/imp.ps.gz`.