CHR(Curry): Interpretation and Compilation of Constraint Handling Rules in Curry

Michael Hanus

Institut für Informatik, CAU Kiel, D-24098 Kiel, Germany.
mh@informatik.uni-kiel.de

Abstract. Constraint Handling Rules (CHR) is a rule-based language to specify application-oriented constraint solvers. CHR requires a host language that provides the basic constraints used in a CHR program. In this paper, we argue that an integrated functional logic language like Curry is an appropriate host language for CHR since it supports a natural formulation of constraint handling rules and a seamless integration into a typed environment. As a proof of concept, we describe CHR(Curry), an integration of CHR into Curry, together with two implementations. The first is an interpreter of CHR's refined operational semantics implemented in Curry, and the second compiles CHR rules into Prolog which can be directly used in Prolog-based Curry implementations, such as PAKCS.

1 Motivation

Functional logic languages [4,15] integrate the most important features of functional and logic languages in order to provide a variety of programming concepts. They support functional concepts like higher-order functions and lazy evaluation as well as logic programming concepts like non-deterministic search and computing with partial information. This combination allows better abstractions for application programming and has also led to new design patterns [1,5] as well as better abstractions in application programs such as implementing graphical user interfaces [12] or web frameworks [17]. The declarative multi-paradigm language Curry [11,18] is a modern functional logic language with advanced concepts for application programming [2,3].

An important application area of declarative, and in particular, logic programming languages is constraint programming [19,22]. Since logic programming is a subset of functional logic programming, there exist various attempts to extend functional logic languages with constraint solving facilities (see [24] for a survey). For instance, Lux [21] describes an implementation of a solver for real arithmetic constraints for Curry, and the inclusion of finite domain constraints in the functional logic language TOY [20] is described in [9].

An alternative to using a fixed set of constraint solvers are Constraint Handling Rules (CHR) [10]. CHR is a declarative language for specifying application-oriented constraint systems. They are useful for applications that require specific

constraints for which no standard solvers (like solvers for finite domain or real arithmetic constraints) exist. CHR defines the processing of multisets of constraints by the specification of multi-headed simplification or propagation rules. Thus, CHR is a high-level language to specify and implement constraint solvers for various application domains (see [10,27] for more detailed surveys).

Since CHR consists only of rewrite rules, CHR programs require a host language $\mathcal{H}$. On the one hand, the results of CHR computations are intended to be used in some application program, written in $\mathcal{H}$, that interacts with users, databases etc. On the other other hand, CHR is based on the existence of a set of basic constraints and data types that are used inside CHR rules. In order to make the reference to the host language $\mathcal{H}$ explicit, the notation CHR($\mathcal{H}$) is used. Most CHR systems implement CHR(Prolog) so that Prolog predicates can be used as basic constraints in CHR programs.

Example 1. The following CHR(Prolog) program [10] defines a generic less-than-or-equal relation `leq`.

```
reflexivity  @ leq(X,Y) <=> X=Y | true.
antisymmetry @ leq(X,Y), leq(Y,X) <=> X=Y.
transitivity @ leq(X,Y), leq(Y,Z) ==> leq(X,Z).
```

The first rule uses the Prolog predicate "`=`" to check the equality of the `leq` arguments, i.e., if both arguments are equal, then the CHR constraint `leq(X,Y)` can be omitted (or replaced by `true`). The second rule uses the same predicate as a constraint that unifies the arguments `X` and `Y` in order to enforce the antisymmetry property of `leq`. The detailed meaning of these rules will be explained in Section 3.

Most implementation and research efforts have been done for CHR(Prolog). Nevertheless, Prolog does not seem the most natural host language since non-Prolog features, like evaluable expressions, are sometimes used in example programs.

Example 2. The following simple CHR program, presented in [8], calculates the greatest common divisor (gcd) of two integers:

```
gcd1 @ gcd(0)           <=> true.
gcd2 @ gcd(N) \ gcd(M) <=> M >= N | gcd(M-N).
```

The intended use of this program is to put two CHR constraints `gcd(A)` and `gcd(B)` into the initial store. The second rule replaces the larger value by smaller ones (if `N` is positive) so that, after removing one CHR constraint by the first rule, the remaining CHR constraint contains the greatest common divisor.

Although the authors of [8] use the general notation of CHR(Prolog), they remark that the term `M-N` occurring in the second rule is not treated as in Prolog but it is "automatically evaluated" (as in functional programming). Since such functional notations occur also in many other examples (and they are translated in the actually implemented examples into non-declarative Prolog features), it seems that a functional logic language is a more appropriate host language than Prolog. In order to show that this idea is feasible, we propose in this paper CHR(Curry). Curry as a host language for CHR has the following advantages:

– The natural functional notation can be used in CHR rules.
– All functions defined in a Curry program as well as all predicates or constraints can be used in CHR rules.
– CHR constraints can be used in Curry programs. In particular, one can define application-oriented constraint solvers as high-level CHR rules and use them as any other predefined constraint.
– One can use high-level APIs developed in functional logic style to visualize the results of CHR computations, e.g., in graphical user interfaces [12], interactive web pages [13], or web frameworks [17].
– If CHR is embedded into a strongly typed host language, such as Curry, one gets type safety and (polymorphically) typed CHR constraints for free.

We develop CHR(Curry) as follows. In a first step, we show how CHR rules can be written in Curry without any language extension, i.e., we basically develop an eDSL (embedded domain specific language) for CHR in Curry. In a second step, we sketch two implementations of this eDSL: an interpreter oriented towards the refined operational semantics of CHR [8], and a compiler that translates CHR rules into an existing CHR(Prolog) implementation.

In the next section, we introduce some concepts of functional logic programming and the language Curry. Section 3 reviews the basic ideas of CHR. Section 4 contains our proposal to integrate CHR in Curry. Sections 5 and 6 sketches the implementations of this proposal before we conclude with a review of related work in Sections 7 and 8.

2 Basic Elements of Curry

We briefly review those elements of Curry which are necessary to understand the contents of this paper. More details can be found in recent surveys on functional logic programming [4,15] and in the language report [18].

Curry is a multi-paradigm declarative language that combines in a seamless way features from functional, logic, and concurrent programming and supports application-oriented programming (with types, modules, encapsulated search, monadic I/O [29]). The syntax of Curry is close to Haskell [23], i.e., type variables and names of defined operations usually start with lowercase letters and the names of type and data constructors start with an uppercase letter. Functional types are "curried," i.e., $\alpha \to \beta$ denotes the type of all functions mapping elements of type α into elements of type β, and the application of an operation f to an argument e is denoted by juxtaposition ("$f\ e$").

In addition to Haskell, Curry allows *free* (*logic*) *variables* in rules and initial expressions. Function calls with free variables are evaluated by a possibly nondeterministic instantiation of demanded arguments.

Example 3. The following Curry program defines the data type of polymorphic lists and operations to concatenate two lists and compute the last element of a list:

```
data List a  = [] | a : List a

(++) :: [a]  →  [a]  →  [a]
[]     ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)

last :: [a]  →  a
last xs | _ ++ [x] =:= xs
        = x
  where x free
```

The `data` type declaration defines `[]` (empty list) and `:` (non-empty list) as the constructors for polymorphic lists (`a` is a type variable ranging over all types and the type "`List a`" is written as `[a]` for conformity with Haskell). The (optional) type declaration ("`::`") of the operation "`++`" specifies that "`++`" takes two lists as input and produces an output list, where all list elements are of the same (unspecified) type. Since "`++`" can be called with free variables in arguments, the equation "`_ ++ [x] =:= xs`" in the condition of `last` is solved by instantiating the anonymous free variable `_` to the list `xs` without the last argument, i.e., the only solution to this equation satisfies that `x` is the last element of `xs`.

The (optional) condition of a program rule is a constraint, where a *constraint* is any expression of the built-in type `Success`. Each Curry system provides at least *equational constraints* of the form e_1 `=:=` e_2 which are satisfiable if both sides e_1 and e_2 are reducible to unifiable data terms. "c_1 `&` c_2" denotes the *concurrent conjunction* of the constraints c_1 and c_2, i.e., this expression is evaluated by proving both argument constraints concurrently. Some Curry systems also support more powerful constraint structures, like arithmetic constraints on real numbers or finite domain constraints, as in the PAKCS implementation [16]. The purpose of this paper is to provide a mechanism to specify application-oriented constraint solvers on the level of Curry programs.

3 Constraint Handling Rules

In this section we review the basic ideas of the language CHR. More details about the concept and implementation of CHR can be found in the surveys [10,27] and the CHR website[1].

A CHR program describes the processing of a multiset of user-defined constraints (also called the *constraint store*) by two kinds of rules. *Simplification rules* specify the replacement of several constraints by a multiset of constraints. *Propagation rules* specify the propagation of new constraints from several existing constraints, i.e., the new constraints are added to the constraint store. In order to restrict the applicability of rules, rules can contain guards that consist of predefined (built-in) primitive constraints. Such primitive constraints can also occur in the right-hand sides of simplification or propagation rules.

[1] `http://dtai.cs.kuleuven.be/CHR/`

For instance, the CHR program shown in Example 1 contains two simplification rules (reflexivity, antisymmetry) and one propagation rule (transitivity). Simplification and propagation rules are denoted by "`<=>`" and "`==>`", respectively. The primitive constraints to the left of the symbol "`|`" constitute the guard of a rule. Multiple constraints are separated by commas which are interpreted as logical conjunction. The rule

```
reflexivity  @ leq(X,Y) <=> X=Y | true.
```

specifies that an occurrence of a constraint `leq(X,Y)` can be eliminated provided that `X=Y` holds, i.e., both arguments are syntactically identical. The rule

```
antisymmetry @ leq(X,Y), leq(Y,X) <=> X=Y.
```

specifies that occurrences of both `leq(X,Y)` and `leq(Y,X)` in the constraint store can be replaced by `X=Y` that enforces the syntactic identity of `X` and `Y`. Note the different rôles of the primitive constraint `X=Y` in both rules. This constraint acts in rule `reflexivity` as a condition (test) to determine the applicability of the rule, whereas in rule `antisymmetry` it enforces the equality by manipulating the constraint store. In general, the applicability of a rule is tested without modifying the constraint store (in contrast to predicates in logic programming that are applied by instantiating the actual arguments), i.e., the left-hand side and the condition must be entailed by the constraint store before the constraints in the right-hand side are added to the store. The rule

```
transitivity @ leq(X,Y), leq(Y,Z) ==> leq(X,Z).
```

propagates a new constraint, i.e., `leq(X,Z)` is added to the constraint store if the store already contains the constraints `leq(X,Y)` and `leq(Y,Z)`. The redundancy in the constraint store caused by propagation is useful to enable the application of further simplification rules. For instance, if the constraint store contains

```
leq(X1,X2), leq(X3,X1), leq(X2,X3)
```

the application of rule `transitivity` adds the new constraint `leq(X1,X3)` so that the application of rule `antisymmetry` deletes the constraints `leq(X3,X1)` and `leq(X1,X3)` and enforces the syntactic equality between `X1` and `X3`. As a consequence, the remaining two constraints can be deleted by enforcing the equality between `X1` and `X2`.

Since the uncontrolled application of propagation rules might lead to nonterminating derivations, the operational semantics of CHR (see Section 5) defines conditions to restrict the application of such rules. Sometimes it is useful to combine a simplification and a propagation rule into one rule, called *simpagation rule*, where the left-hand side contains two parts separated by "\", as shown in Example 2:

```
gcd2 @ gcd(N) \ gcd(M) <=> M >= N | gcd(M-N).
```

The part to the left of "\" is kept like in a propagation rule and the right part is deleted like in a simplification rule. Actually, simpagation rules can also be seen as the most general form of CHR rules. This is further discussed in the following section where we present our syntactic embedding of CHR in Curry so that CHR rules become regular Curry expressions.

4 Constraint Handling Rules in Curry

As already mentioned in Section 1, instead of extending the syntax of Curry in order to deal with CHR, we want to embed CHR rules into Curry programs. For this purpose, we represent CHR rules as data objects in Curry. Remember that the most general form of a CHR rule is the simpagation rule

$$r \ @ \ \ H_1 \setminus H_2 \iff g \mid B$$

where r is a name of the rule, H_1 and H_2 are sequences of CHR (user-defined) constraints, the guard g is a sequence of built-in (primitive) constraints and B is a sequence of CHR and built-in constraints. Simplification and propagation rules are special cases of the simpagation rule with $H_1 = \varnothing$ and $H_2 = \varnothing$, respectively. Hence, it suffices to specifiy a data structure to represent simpagation rules.

In order to abstract from the set of CHR constraints used in actual programs, we assume that the type variable `chr` denotes the type of CHR constraints, which is usually an enumeration of the various CHR constraints occurring in a CHR program. Furthermore, the variables occurring in CHR rules have a distinct domain (e.g., `Int` in case of the gcd rules shown in Example 2) which we denote by the type variable `dom`. Using a single domain in CHR rules is not a restriction since this domain could also be a union type. Therefore, we can specify the structure of a CHR rule by the following data type:

```
data CHR dom chr =
  SimpaRule [chr] [chr] [PrimConstraint dom] (Goal dom chr)
```

The four arguments of `SimpaRule` correspond to the components H_1, H_2, g, and B of a simpagation rule. We do not include the name r of the rule since we will identify rules by program objects. The type `Goal` denotes sequences of user-defined and primitive constraints and is defined as follows:

```
data Goal dom chr = Goal [CHRconstr dom chr]

data CHRconstr dom chr = PrimCHR (PrimConstraint dom)
                       | UserCHR chr
```

Hence, `CHRconstr` is the union of primitive and user-defined constraints.

Finally, the type `PrimConstraint` contains the primitive (built-in) CHR constraints such as equality, disequality, etc. Moreover, one can also embed any constraint defined in a Curry program as a primitive constraint. For this purpose, we define this type as follows:

```
data PrimConstraint a =
    Eq a a                          -- equality
  | Neq a a                         -- disequality
  | Fail                            -- always unsatisfiable
  | Compare (a -> a -> Bool) a a    -- ordering constraint
  | Ground a                        -- ground value?
  | Nonvar a                        -- bound variable?
  | AnyPrim (() -> Success)         -- user-defined primitive
```

Although constraints like `Nonvar` and `Ground` have a non-declarative flavor, they are often used in CHR rules to control the application of rules. The argument

type of `AnyPrim` reflects the fact that any constraint abstraction available in Curry can be used as a primitive constraint.

Although these type definitions cover the essential structure of CHR rules, it would be tedious to use them for writing concrete rules. Therefore, we define a bunch of operations as syntactic sugar for writing CHR rules. Since some special characters (comma, vertical bar) belong to the syntax of Curry and are not allowed as operators, we can not provide the exact Prolog-oriented syntax of CHR. Nevertheless, we want to be very close to this syntax. For this purpose, we use a goal-oriented syntax to define CHR rules. For instance, to define simplification rules, we will define an operator of type

```
(<=>) :: Goal dom chr  →  Goal dom chr  →  CHR dom chr
```

where the left- and right-hand sides are goals. To construct goals in a readable manner, we define the operator "`/\`" for the conjunction of two goals:

```
(/\) :: Goal dom chr  →  Goal dom chr  →  Goal dom chr
(/\) (Goal c1) (Goal c2) = Goal (c1 ++ c2)
```

Similarly, we define `true` as the always satisfiable (empty) goal:

```
true :: Goal dom chr
true = Goal []
```

To support a nice notation for primitive constraints, we define a generic embedding of primitive constraints into goals by

```
primToGoal :: PrimConstraint dom  →  Goal dom chr
primToGoal pc = Goal [PrimCHR pc]
```

and introduce some operators[2] to denote the various primitive constraints:

```
fail     = primToGoal Fail
x .=.  y = primToGoal (Eq x y)
x ./=. y = primToGoal (Neq x y)
x .>=. y = primToGoal (Compare (>=) x y)
...
```

Finally, we introduce operators to write CHR rules in the usual way:

```
(<=>) :: Goal dom chr  →  Goal dom chr  →  CHR dom chr
g1 <=> g2 | null (primsOfGoal g1)
          = SimpaRule [] (uchrOfGoal g1) [] g2

(==>) :: Goal dom chr  →  Goal dom chr  →  CHR dom chr
g1 ==> g2 | null (primsOfGoal g1)
          = SimpaRule (uchrOfGoal g1) [] [] g2
```

Here we use operations `primsOfGoal` and `uchrOfGoal` that extract the list of primitive and user-defined CHR constraints from a goal. The condition expresses the fact that primitive constraints are not allowed in the left-hand sides of CHR rules.[3] To denote simpagation rules, we introduce the operator "`\\`":

[2] We omit in this paper the definition of the operator priorities since they should be clear from the context.

[3] Our actual implementation yields also a sensible error message if this condition is not satisfied.

```
(\\) :: Goal dom chr  → CHR dom chr  → CHR dom chr
g \\ (SimpaRule h1 h2 c b) | null (primsOfGoal g) && null h1
                           = SimpaRule (uchrOfGoal g) h2 c b
```

To attach a condition to a CHR rule, we define a guard operator "`|>`" (note that the right-hand side of the already existing rule becomes the condition of the new rule by the use of this operator):

```
(|>) :: CHR dom chr  → Goal dom chr  → CHR dom chr
(SimpaRule h1 h2 _ c) |> b | null (uchrOfGoal c)
                           = SimpaRule h1 h2 (primsOfGoal c) b
```

In order to exploit the strong type system of the host language in CHR programs, we introduce user-defined CHR constraints as a data type. For instance, the CHR program of Example 1 contains rules for a single CHR constraint `leq`. Since the arguments of `leq` are compared by equality in the reflexivity and antisymmetry rule, they can be arbitrary but have to be of the same type.[4] Thus, we define the following data type to represent this CHR constraint:

```
data LEQ a = Leq a a
```

Since user-defined CHR constraints should be embedded into CHR goals, our CHR implementation defines a generic embedding of binary constraints (actually, it defines a family of embeddings for various arities):

```
toGoal2 :: (a  → b  → chr)  → a  → b  → Goal dom chr
toGoal2 c x y = Goal [UserCHR (c x y)]
```

Hence, we define `leq` as a goal corresponding to the CHR constraint `Leq`:

```
leq = toGoal2 Leq
```

With this preparation and our CHR operators introduced above, we can write the rules of Example 1 as the following Curry program:

```
reflexivity  [x,y]   = leq x y <=> x .=. y  |> true
antisymmetry [x,y]   = leq x y /\ leq y x  <=> x .=. y
transitivity [x,y,z] = leq x y /\ leq y z  ==> leq x z
```

Apart from small syntactic differences, this is the "standard" notation for CHR rules. Note that all variables occurring in a CHR rule have to be introduced at some point. In Curry, they could be declared either as free variables or as parameters. In our eDSL for CHR, we decided to introduce these variables as parameters. The name of each CHR rule is represented by the name of the operation defining this rule. Thus, a CHR program consists of a list of operations (not a set, which is relevant for the refined operational semantics of CHR, see below) defining the various rules. As shown later, such a list is the input parameter to our implementations.

In a well typed CHR program, all rules have the same type, i.e., they operate over the same domain type and specify the semantics of user-defined constraints

[4] In Haskell, they should have the type class context `Eq`, but the current version of Curry does not support type classes so that equality is syntactically defined on any type.

of the same type. For instance, the reflexivity rule (as well as all other `leq` rules) has the type:

```
reflexivity :: [a]  →  CHR a (LEQ a)
```

It should be noted that the polymorphic type system of Curry automatically yields a polymorphic type system for CHR. This is in contrast to [6] where a separate (monomoprhic) type system and type checker for CHR has been developed. The soundness of our typing of CHR rules will be an immediate consequence of our well-typed interpreter (see below).

Example 4. As a final example of this section, we show the implementation of Example 2 in our framework. First, we define the type of gcd constraints

```
data GCD = GCD Int
```

and embed them into goals by

```
gcd = toGoal1 GCD
```

Then, we can easily write the two rules:

```
gcd1 []     = gcd 0 <=> true
gcd2 [m,n] = gcd n \\ gcd m <=> m .>=. n |> gcd (m-n)
```

Thanks to our embedding into Curry, we can actually use the functional notation `(m-n)` for the argument of `gcd` in rule `gcd1` without any further transformation, in contrast to CHR(Prolog).

5 Interpretation

In order to provide a first implementation of our embedded CHR language, we implement an interpreter for CHR in Curry. Since the interpreter is written in a strongly typed language, it also ensures the type correctness of CHR rules: since it manipulates a typed constraint store, the type system of Curry (which is a Hindley-Milner like polymorphic type system [7]) ensures that the constraint store always contains type-correct constraints.

The implementation of the interpreter is oriented towards the operational semantics of CHR. The original operational semantics of CHR [10] is defined as a transition system that describes the application of the different kinds of CHR rules. Since simpagation rules are the most general kind of CHR rules, it suffices to consider such kind of rules only. A state of the transition system is a triple $\langle G, S, B\rangle$ where the goal G and the store S are multi-sets of constraints and B consists of built-in constraints. The initial state has the form $\langle G, \varnothing, true\rangle$ and is reduced according to the following transition steps ($A \uplus B$ denotes the disjoint union of the multi-sets A and B):

1. Solve: $\langle \{c\} \uplus G, S, B\rangle \longmapsto \langle G, S, c \wedge B\rangle$ if c is a built-in constraint
2. Introduce: $\langle \{c\} \uplus G, S, B\rangle \longmapsto \langle G, \{c\} \uplus S, B\rangle$ if c is not a built-in constraint
3. Apply: $\langle G, H_1 \uplus H_2 \uplus S, B\rangle \longmapsto \langle C \uplus G, H_1 \uplus S, H_1' = H_1 \wedge H_2' = H_2 \wedge B\rangle$ where $r \ @\ H_1' \setminus H_2' \iff g \mid C$ is a renamed CHR rule and $B \rightarrow \exists \overline{x}(H_1' = H_1 \wedge H_2' = H_2 \wedge g)$ (i.e., the rule heads match and the condition is satisfied w.r.t. B)

Although these transition rules specify a superset of all possible evaluations, they are too weak to be used in practice. First of all, they do not include any mechanism to avoid trivial infinite propagations. This can be improved by adding a propagation history so that a rule is not applied again to the same literals [8]. Even with this improvement, the semantics is still a "theoretical only" semantics and not used in practice (i.e., not implemented by CHR systems). For instance, consider the gcd rules of Example 2 (which is a popular CHR example and one of the first appearing on the CHR website). With this theoretical semantics, the program is non-terminating since rule `gcd2` can always be applied to the constraints `gcd(0)` and `gcd(2)` so that the constraint `gcd(2)` is added to the goal in every application step. In practice, this is avoided by ordering rules and constraints and considering CHR constraints as procedure calls or active constraints that try to find matching partners constraints to apply a rule. For instance, the gcd solver immediately removes the constraint `gcd(0)` with the first rule `gcd1` so that the infinite loop is avoided.

A refined operational semantics covering these issues has been precisely defined in [8] by a refined set of transition rules. Due to lack of space, we do not recapitulate them here. In a declarative programming language, the transition rules can be implemented with reasonable effort. Hence, we have written a simple interpreter (approximately 50 lines of code) based on these transition rules in Curry. Since the standard evaluation mode of Curry is narrowing (i.e., unification + functional reduction), it cannot be directly used to implement CHR rules since the application of a rule requires the check for the applicability of a rule *without* instantiating free variables in a goal. Therefore, our implementation exploits the predefined operation `rewriteSome` of the library `Findall`[5] which evaluates an expression by term rewriting, i.e., without binding free variables.

The basic interface to our CHR interpreter has the following type:

```
runCHR :: [[dom] → CHR dom chr] → Goal dom chr → [chr]
```

Hence, it takes as input a list of CHR rules and a goal and returns, in case of a successful evaluation, the list of remaining user-defined constraints. For instance, the evaluation of the expression

```
runCHR [gcd1,gcd2] (gcd 16 /\ gcd 28)
```

yields the result `[GCD 4]`. In order to embed CHR constraints into Curry as predefined constraints, there is also an operation

```
solveCHR :: [[dom] → CHR dom chr] → Goal dom chr → Success
```

This solver succeeds in case of a successful evaluation and, in addition, it issues a warning if there are some remaining (suspended) constraints. Using `solveCHR`, we can use CHR constraints as any other constraint in Curry programs, e.g., we can write CHR constraints in conditions of defined operations in order to restrict their applicability.

As already mentioned, the type system of Curry ensures that well-typed CHR rules yield well-typed CHR computations, i.e., we obtain a polymorphic CHR

[5] `http://www.informatik.uni-kiel.de/~pakcs/lib/CDOC/Findall.html`

type system for free. In particular, we can also define CHR rules for polymorphic constraints.

Example 5. The union-find algorithm is an interesting example to demonstrate the power of CHR [26]. The algorithm maintains a collection of disjoint subsets with canonical elements (representatives) and operations `union` and `find`. Since the type of the elements is not important, the sets can be modeled as a polymorphic data type. Thus, the CHR(Prolog) program presented in [26] can be defined in a type-safe manner in CHR(Curry) as follows:

```
data UF a = Root a | Arrow a a | Make a
          | Union a a | Find a a | Link a a

root  = toGoal1 Root    (~>) = toGoal2 Arrow   make = toGoal1 Make
union = toGoal2 Union   find = toGoal2 Find    link = toGoal2 Link

makeI    [a]       = make a <=> root a
unionI   [a,b,x,y] = union a b <=> find a x /\ find b y /\ link x y
findNode [a,b,x]   = a ~> b \\ find a x <=> find b x
findRoot [a,x]     = root a \\ find a x <=> x .=. a
linkEq   [a]       = link a a <=> true
linkTo   [a,b]     = link a b /\ root a /\ root b <=> b ~> a /\ root a
```

Since the type `UF` is polymorphic, this union-find algorithm can be applied to sets of various types (e.g., sets containing integers, characters, or strings) and the type system ensures that sets of different types can not be mixed.

6 Compilation

Since our CHR interpreter is parameterized over the list of CHR rules, it is useful to develop and test CHR programs. For instance, one can evaluate CHR goals with different sets of rules or rules in various orders. However, due to the interpretive approach and purely declarative implementation without any side effects or global state, the implementation is quite inefficient compared to native CHR implementations. As an alternative, one can reuse existing CHR implementations to which we can compile our CHR(Curry) programs. For instance, there are good CHR(Prolog) implementations available for SICStus- or SWI-Prolog [25]. Since the Curry system PAKCS [16] compiles Curry programs into SICStus- or SWI-Prolog programs, it is reasonable to compile CHR(Curry) programs into CHR(Prolog) programs. For this purpose, our CHR library contains an operation

```
compileCHR :: String → [[dom] → CHR dom chr] → IO ()
```

The first argument is the name of the target Curry module into which the CHR rules, specified in the second argument, are compiled. Actually, the generated Curry module only contains an interface to access the compiled CHR constraints from Curry programs. The generated CHR(Prolog) constraints are accessed from this module by the usual foreign function interface provided by PAKCS.

As an example, consider the CHR(Curry) program to compute the greatest common divisor (Example 4). The call "`compileCHR "GCDC" [gcd1,gcd2]`" generates the following Curry module:

```
module GCDC where

import CHRcompiled

gcd :: Int → Goal GCD
gcd x1 = Goal (prim_gcd $!! x1)

prim_gcd external ⟨internal code to call the CHR Prolog code⟩
```

The imported module `CHRcompiled` contains some definitions that are required to handle (typed!) CHR goals also in combination with compiled CHR programs. For instance, there is the definition

```
data Goal chr = Goal Success
```

Hence, the argument of the data constructor `Goal` is a constraint, which is reasonable since it is a container for the compiled CHR(Prolog) constraints. However, the type is parameterized by a phantom type `chr` in order to avoid a mixture of CHR constraints with incompatible types. For instance, the conjunction of constraints is defined in the module `CHRcompiled` by

```
(/\) :: Goal chr → Goal chr → Goal chr
(/\) (Goal g1) (Goal g2) = Goal (g1 & g2)
```

so that only goals over the same domain can be combined. Hence, mixing union-find constraints (Example 5) over sets of integers and sets of characters in the same goal would be rejected by Curry's type system. In order to embed the CHR(Prolog) solver as a Curry constraint, `CHRcompiled` also defines the operation

```
solveCHR :: Goal chr → Success
```

which solves the CHR goal and issues a warning if there are some remaining (suspended) constraints.

It should be noted that the generated operation `gcd` evaluates its argument (by the strictly evaluating application operator "`$!!`") before putting the constraint into the constraint store. This is necessary to interface the functional features of Curry with CHR. Since the CHR semantics (see Section 5) does not evaluate arguments but consider them as free Herbrand terms as in logic programming, defined functions need to be evaluated before passing the CHR constraints to the CHR solver. Hence, we can write in the application program "`gcd (6+9*4)`" which is passed as the constraint `gcd(42)` to CHR(Prolog).

The actual CHR(Prolog) program is generated by a straightforward transformation of the CHR(Curry) rules. The only interesting aspect is the interfacing between the CHR(Prolog) solver and Curry, because CHR(Curry) rules can also contain calls to operations defined in Curry programs (e.g., calls to the greater-or-equal or subtraction operations in rule `gcd2`). Since CHR(Prolog) allows the use of any Prolog predicate inside rules and Curry operations are compiled into Prolog predicates by PAKCS, interfacing CHR(Prolog) and Curry is not difficult. For instance, rules `gcd1` and `gcd2` of Example 4 are translated into the following

CHR(Prolog) rules (the code is simplified since the actual code requires additional control information for PAKCS):

```
gcd(0) <=> true.
gcd(N) \ gcd(M) <=> eq('Prelude.>='(M,N),'Prelude.True')
                         | eq(X,'Prelude.-'(M,N)), gcd(X).
```

The Prolog predicate `eq` implements the strict unification operator "`=:=`", i.e., both arguments are evaluated to normal form and unified. Thus, the original argument `(m-n)` of `gcd` in rule `gcd2` is evaluated by applying the subtraction operation defined in the standard prelude of Curry (`Prelude.-`) and `X` is bound to the result before the constraint `gcd(X)` is activated. In this way any (type-correct) operation implemented in Curry can be used in CHR rules.

Compiled CHR constraints can be solved by `solveCHR` as any other Curry constraint, e.g., in initial goals or conditions of defined operations. In contrast to the interpreter "`runCHR`", remaining (suspended) CHR constraints are not returned but it is intended that all user-defined constraints should be removed at the end. This can usually be obtained by adding rules and specific constraints to access information contained in the constraint store. For instance, to retrieve the value of the greatest common divisor that would remain in the constraint store, we replace rule `gcd1` by the following new rule (we omit here the simple extension of the data type `GCD`):

```
gcda [n,x] = gcd 0 /\ gcd n /\ gcdanswer x <=> x .=. n
```

With this rule, the constraint `gcd 0` is not simply discarded but, at the same time, the argument of the constraint `gcdanswer` is unified with the remaining value and all three constraints are discarded. If we compile the rules `[gcda,gcd2]`, we yield for the Curry goal

```
solveCHR (gcdanswer x & gcd 16 & gcd 28)  where x free
```

the answer substitution `{x=4}`.

The concrete implementation of our compiler is rather technical so that we omit a more detailed description here. The complete implementation of CHR(Curry), i.e., the eDSL operations shown in Section 4, the interpreter and the compiler, is freely available as a Curry module (`CHR`) in recent distributions of PAKCS [16]. As shown by the examples above, operations defined in Curry can be used inside CHR(Curry) rules and CHR constraints can be used in Curry programs so that we obtained a thorough embedding of CHR in Curry. In addition to the examples presented in this paper, various constraint solvers have been implemented in CHR(Curry), like Boolean constraints, finite domain constraints, prime numbers, Gaussian elimination to solve linear equalities, or computing Fibonacci numbers (as shown in [8]). The latter example also demonstrates the improved efficiency of the compilation approach: our CHR interpreter needs 1.4/9.7 seconds to compute the 50./100. Fibonacci number, whereas the compiled CHR code computes these numbers in less than 10 milliseconds (with an Intel Core i7-4790/3.60Ghz processor).

7 Related Work

Since there are a lot of publications related to CHR ([10,27] provide good surveys on different stages of the CHR development), we compare our work to some closely related work only.

HaskellCHR[6] is an implementation of CHR in Haskell. It mainly emphasizes on the implementation of the operational semantics of CHR in Haskell but does not provide a deeper embedding of CHR rules in Haskell programs, e.g., neither a specific syntactical embedding nor a type system for CHR. It has been successfully used in the Chameleon system [28] to implement advanced type systems.

HCHR [6] is a deeper embedding of CHR into Haskell. Although HCHR implements a monadic interpreter for CHR in Haskell (including an implementation of logic variables and unification), HCHR is more restricted and less flexible than our approach. Since HCHR uses a specific syntactic extension to write CHR rules, it does not use Haskell's type system for CHR. Actually, it implements a monomorphic type system for CHR and transforms rules into Haskell operations so that the Haskell type checker is used to detect type errors.

The CHR(Prolog) implementation distributed with SICStus-/SWI-Prolog [25] also supports the declaration of type annotations to CHR constraints. Although one can introduce polymorphic data structures like lists, the type annotations to CHR constraints are restricted to monomorphic types.

An early predecessor of this work [14] contained a first proposal to integrate CHR into Curry. This implementation was much more restricted than the current approach. Only goals of a predefined set of types were supported, user-defined Curry operations were not allowed inside CHR rules, and the implementation was only a compiler into untyped CHR(Prolog) so that it was not clear that type correct CHR rules do not yield type errors at run time. All these restrictions are removed in our new framework.

8 Conclusion

In this paper we presented CHR(Curry), an embedding of CHR into the functional logic host language Curry. To avoid a CHR-specific language extension of Curry, we presented an eDSL to embed CHR rules into Curry programs with a notation closely related to "standard" CHR programs. This representation has the advantage that one can use functional notation in CHR rules, and Curry's type system can be exploited to check the well-typedness of CHR rules. Since we implemented the refined operational semantics of CHR in Curry, the strong type system of Curry ensures that well-typed CHR programs do not yield ill-typed constraints at run time. Since Curry's type system supports parametric polymorphism, one can also specify polymorphic constraints, as shown in the less-or-equal or union-find solvers. Due to the thorough embedding of CHR into Curry, one can use operations defined in Curry programs inside CHR rules and

[6] `http://www.comp.nus.edu.sg/~gregory/haskellchr/`

one can use CHR constraints in conditions of rules defining Curry operations. Hence, one can exploit the advantage of CHR to write application-specific constraint solvers.

The use of a functional logic host language instead of a purely logic host language for CHR has various advantages. For instance, the natural functional notation can be directly applied in CHR rules. This notation is often used in examples in papers about CHR but then manually translated into a flat relational notation in case of Prolog as a host language. Since our host language Curry comes with a polymorphic type system, we obtain a polymorphic type system for CHR for free.

We presented two implementations of CHR(Curry), an interpreter implemented in Curry and a compiler to CHR(Prolog). Whereas the interpreter is useful to develop and test various constraint solvers, the compiler is necessary to use CHR(Curry) in practice. For future work, it might be interesting to explore methods to improve the efficiency of the interpreter, e.g., advanced data structures, states, monadic computations, in order to get a more efficient implementation to quickly test also larger CHR systems.

References

1. S. Antoy and M. Hanus. Functional logic design patterns. In *Proc. of the 6th International Symposium on Functional and Logic Programming (FLOPS 2002)*, pages 67–87. Springer LNCS 2441, 2002.
2. S. Antoy and M. Hanus. Declarative programming with function patterns. In *Proceedings of the International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'05)*, pages 6–22. Springer LNCS 3901, 2005.
3. S. Antoy and M. Hanus. Set functions for functional logic programming. In *Proceedings of the 11th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'09)*, pages 73–82. ACM Press, 2009.
4. S. Antoy and M. Hanus. Functional logic programming. *Communications of the ACM*, 53(4):74–85, 2010.
5. S. Antoy and M. Hanus. New functional logic design patterns. In *Proc. of the 20th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2011)*, pages 19–34. Springer LNCS 6816, 2011.
6. W.-N. Chin, M. Sulzmann, and M. Wang. A type-safe embedding of constraint handling rules into Haskell. Honors thesis, School of Computing, Natnioanl University of Singapore, 2003.
7. L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proc. 9th Annual Symposium on Principles of Programming Languages*, pages 207–212, 1982.
8. G.J. Duck, M. Garcia de la Banda, P.J. Stuckey, and C. Holzbaur. The refined operational semantics of constraint handling rules. In *Proceedings of the 20th International Conference on Logic Programming (ICLP 2004)*, pages 90–104. Springer LNCS 3132, 2004.
9. A.J. Fernández, M.T. Hortalá-González, and F. Sáenz-Pérez. Solving combinatorial problems with a constraint functional logic language. In *Proc. of the 5th International Symposium on Practical Aspects of Declarative Languages (PADL 2003)*, pages 320–338. Springer LNCS 2562, 2003.

10. T. Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming*, 37(1-3):95–138, 1998.
11. M. Hanus. A unified computation model for functional and logic programming. In *Proc. of the 24th ACM Symposium on Principles of Programming Languages (Paris)*, pages 80–93, 1997.
12. M. Hanus. A functional logic programming approach to graphical user interfaces. In *International Workshop on Practical Aspects of Declarative Languages (PADL'00)*, pages 47–62. Springer LNCS 1753, 2000.
13. M. Hanus. High-level server side web scripting in Curry. In *Proc. of the Third International Symposium on Practical Aspects of Declarative Languages (PADL'01)*, pages 76–92. Springer LNCS 1990, 2001.
14. M. Hanus. Adding constraint handling rules to Curry. In *Proc. 20th Workshop on Logic Programming (WLP 2006)*, pages 81–90. INFSYS Research Report 1843-06-02 (TU Wien), 2006.
15. M. Hanus. Functional logic programming: From theory to Curry. In *Programming Logics - Essays in Memory of Harald Ganzinger*, pages 123–168. Springer LNCS 7797, 2013.
16. M. Hanus, S. Antoy, B. Braßel, M. Engelke, K. Höppner, J. Koj, P. Niederau, R. Sadre, and F. Steiner. PAKCS: The Portland Aachen Kiel Curry System. Available at `http://www.informatik.uni-kiel.de/~pakcs/`, 2015.
17. M. Hanus and S. Koschnicke. An ER-based framework for declarative web programming. *Theory and Practice of Logic Programming*, 14(3):269–291, 2014.
18. M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.8.3). Available at `http://www.curry-language.org`, 2012.
19. J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proc. of the 14th ACM Symposium on Principles of Programming Languages*, pages 111–119, Munich, 1987.
20. F. López-Fraguas and J. Sánchez-Hernández. TOY: A multiparadigm declarative system. In *Proc. of RTA'99*, pages 244–247. Springer LNCS 1631, 1999.
21. W. Lux. Adding linear constraints over real numbers to Curry. In *Proc. of the 5th International Symposium on Functional and Logic Programming (FLOPS 2001)*, pages 185–200. Springer LNCS 2024, 2001.
22. K. Marriott and P.J. Stuckey. *Programming with Constraints*. MIT Press, 1998.
23. S. Peyton Jones, editor. *Haskell 98 Language and Libraries—The Revised Report*. Cambridge University Press, 2003.
24. M. Rodríguez-Artalejo. Functional and constraint logic programming. In *Constraints in Computational Logics: Theory and Applications (CCL'99)*, pages 202–270. Springer LNCS 2002, 2001.
25. T. Schrijvers and B. Demoen. The K.U.Leuven CHR system: Implementation and applications. In *First Workshop on Constraint Handling Rules: Selected Contributions*, pages 8–12, 2004.
26. T. Schrijvers and T. Frühwirth. Optimal union-find in constraint handling rules. *Theory and Practice of Logic Programming*, 6(1-2):213–224, 2006.
27. J. Sneyers, P. Van Weert, T. Schrijvers, and L. De Koninck. As time goes by: Constraint handling rules. *Theory and Practice of Logic Programming*, 10(1):1–47, 2010.
28. P.J. Stuckey and M. Sulzmann. A theory of overloading. *ACM Transactions on Programming Languages and Systems*, 27(6):1216–1269, 2005.
29. P. Wadler. How to declare an imperative. *ACM Computing Surveys*, 29(3):240–263, 1997.