Adding Constraint Handling Rules to Curry*

– Extended Abstract –

Michael Hanus

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

Abstract. This paper proposes an integration of Constraint Handling Rules (CHR), a rule-based language to specify application-oriented constraint solvers, into the declarative multi-paradigm language Curry. This integration provides a convenient way to specify and use flexible constraint systems in applications implemented in Curry. We propose to represent CHR as data objects in Curry programs so that the advantages of Curry (static typing, functional notation) can also be exploited to define CHR. In order to write CHR in a compact way, we define a set of abstractions that hide the concrete CHR data objects. We sketch an implementation of this concept in the Prolog-based Curry implementation PAKCS that compiles CHR data objects into CHR Prolog programs.

1 Motivation

Functional logic languages [7] integrate the most important features of functional and logic languages in order to provide a variety of programming concepts. For instance, the concepts of demand-driven evaluation and higher-order functions from functional programming are combined with logic programming features like computing with partial information (logic variables), unification, and nondeterministic search for solutions. This combination, supported by optimal evaluation strategies [1] and new design patterns [3], leads to better abstractions in application programs such as implementing graphical user interfaces [9] or programming dynamic web pages [10]. The declarative multi-paradigm language Curry [8, 15] is a functional logic language extended by concurrent programming concepts and has been used in various applications, like web programming [10, 14], embedded system programming [13], or database applications [11].

An important application area of declarative, and in particular, logic programming languages is constraint programming [17, 21]. Since logic programming is a subset of functional logic programming, there exist various attempts to extend functional logic languages with constraint solving facilities (see [23] for a survey). For instance, Lux [20] 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 [19] is described in [5]. If a functional logic language is compiled into Prolog (as in TOY [19] or PAKCS [12]), it is fairly easy to provide the functionality of any constraint solver supported by the underlying Prolog system to the source language via an appropriate interface. For instance, such a technique is described in [2] for the Curry programming environment PAKCS that compiles Curry programs into Prolog programs.

Constraint Handling Rules (CHR) [6] are 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 define 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 [6] for a more detailed survey).

Functional logic languages are intended to cover (constraint) logic programming and adds abstraction facilities of functional programming (e.g., higher-order functions, types, flexible (lazy) evaluation strategies). Thus, it is reasonable to integrate CHR also in a functional logic language

* This work was partially supported by the German Research Council (DFG) grant Ha 2457/5-1.

in order to provide the flexible construction of constraint solvers for various application domains. In this paper we discuss such an approach for the functional logic language Curry.

Due to the fact that functional logic programming extends logic programming, it is natural that there exist various approaches to implement functional logic languages by compilation into Prolog (e.g., [2, 5, 18, 19]). In such an implementation, a new CHR-based constraint solver could be made available as follows:

1. Define a CHR constraint solver in Prolog (note that implementations of CHR are included in various Prolog implementations, e.g., SICStus or SWI Prolog [16]).
2. Provide the functionality of the solver by defining an interface of the CHR constraints to the functional logic language (e.g., as described in [2]).
3. Use the CHR constraints via this interface in the application programs.

However, this solution is not satisfactory since the definition of CHR often depends on the application so that, for a specific application exploiting CHR, the implementor has to use *two* different programming languages: the main functional logic implementation language (like Curry) and the host language of CHR, e.g., Prolog. This could be a burden for the practical use of CHR in functional logic languages.[1] Therefore, we propose in this paper the integration of CHR into the source language which is, in our case, Curry. This integration reduces the practical difficulties when using CHR in applications implemented in Curry. Furthermore, it has the advantage that the features of Curry can be used to define CHR. For instance, the type system of Curry is useful to detect some inconsistencies of CHR at compile time, or one can generate CHR by user-defined functions.

In order to avoid a language extension to include CHR in Curry (language extensions are often major design steps with a lot of implementation efforts), we propose in this paper a simpler approach as a first step to integrate CHR into Curry. We represent CHR as data objects that can be manipulated and compiled into a CHR program implemented in Prolog. The compilation process also generates the interface to use CHR constraints in Curry programs so that the programmer can abstract from the underlying Prolog implementation. In order to write CHR in the usual compact way, we define a set of abstractions that hide the concrete CHR data objects.

In the next section, we review some concepts of functional logic programming and the language Curry in order to understand the rest of the paper. Section 3 reviews the basic ideas of CHR. Section 4 contains our proposal to integrate CHR in Curry. Section 5 sketches the implementation of this proposal before we conclude in Section 6.

2 Basic Elements of Curry

In this section we review those elements of Curry which are necessary to understand the contents of this paper. More details about Curry's computation model and a complete description of all language features can be found in [8, 15].

Curry is a multi-paradigm declarative language that combines in a seamless way features from functional, logic, and concurrent programming and supports programming-in-the-large with specific features (types, modules, encapsulated search). From a syntactic point of view, a Curry program is a functional program extended by the possibility to include free (logical) variables in conditions and right-hand sides of defining rules. Curry has a Haskell-like syntax [22], i.e., (type) variables and function names usually start with lowercase letters and the names of type and data constructors start with an uppercase letter. Function types are "curried," i.e., α`->`β denotes the type of all functions mapping elements of type α into elements of type β, and the application of a function f to an argument e is denoted by juxtaposition ("$f\ e$").

A Curry *program* consists of the definition of functions and data types on which the functions operate. Functions are evaluated lazily. To provide the full power of logic programming, functions can be called with partially instantiated arguments and defined by conditional equations with

[1] Note that many users of Curry are more familiar with functional languages, which are often taught in basic courses of a computer science curriculum, than with purely logic languages.

constraints in the conditions. Function calls with free variables are evaluated by a possibly non-deterministic instantiation of demanded arguments (i.e., arguments whose values are necessary to decide the applicability of a rule) to the required values in order to apply a rule (this evaluation mechanism is often called "*narrowing*"). In order to support concurrent programming (in a style that is also known as "*residuation*"), there is a primitive to define general "suspension" combinators for concurrent programming: the predefined operation `ensureNotFree` returns its argument evaluated to head normal form but suspends as long as the result is a free variable.

Example 1. The following Curry program defines the data types of Boolean values and polymorphic lists and functions for computing the concatenation of lists and the last element of a list:

```
infixr 5 ++

data Bool    = True    | False
data List a  = []      | a : List a

(++) :: [a] -> [a] -> [a]
[]     ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)

last :: [a] -> a
last xs | ys ++ [x] =:= xs   = x   where x,ys free
```

The `data` type declarations define `True` and `False` as the Boolean constants, and `[]` (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 *infix operator declaration* "`infixr 5 ++`" declares the symbol "`++`" as a right-associative infix operator with precedence `5` so that we can write function applications of this symbol with the convenient infix notation. The (optional) type declaration ("`::`") of the function "`++`" specifies that "`++`" takes two lists as input and produces an output list, where all list elements are of the same (unspecified) type. Since the function "`++`" can be called with free variables in arguments, the equation "`ys ++ [x] =:= xs`" is solved by instantiating the first argument `ys` 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`.

In general, functions are defined by (*conditional*) *rules* of the form

$f\, t_1 \ldots t_n$ `|` c `=` e `where` vs `free`

with f being a function, $t_1, \ldots, t_n$ *patterns* (i.e., expressions without defined functions) without multiple occurrences of a variable, the *condition* c is a constraint, e is a well-formed *expression* which may also contain function calls, lambda abstractions etc, and vs is the list of *free variables* that occur in c and e but not in $t_1, \ldots, t_n$.[2] The condition and the `where` parts can be omitted if c and vs are empty, respectively. The `where` part can also contain further local function definitions which are only visible in this rule. A conditional rule can be applied if its left-hand side matches the current call and its condition is satisfiable.

A *constraint* is any expression of the built-in type `Success`. For instance, the trivial constraint `success` is an expression of type `Success` that denotes the always satisfiable constraint. "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. 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 patterns. However, specific Curry systems also support more powerful constraint structures, like arithmetic constraints on real numbers or finite domain constraints, as in the PAKCS implementation [12]. The purpose of this paper is to provide a mechanism to specify application-oriented constraint solvers on the level of Curry programs.

[2] The explicit declaration of free variables is sometimes redundant (it is not redundant in case of nested scopes introduced by lambda abstractions or local definitions) but still useful to provide some consistency checks by the compiler.

The operational semantics of Curry, precisely described in [8, 15], is based on an optimal evaluation strategy [1] which is a conservative extension of lazy functional programming and (concurrent) logic programming. Due to its demand-driven behavior, it provides optimal evaluation (e.g., shortest derivation sequences, minimal solution sets) on well-defined classes of programs (see [1] for details). Curry also offers the standard features of functional languages, like higher-order functions or monadic I/O [24].

3 Constraint Handling Rules

In this section we review the basic ideas of Constraint Handling Rules (CHR). More details about the concept and implementation of CHR can be found in the survey [6] and the CHR web site[3].

CHR are rules that describe 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 express conditions for the applicability of rules, the rules can contain guards that consist of predefined (built-in) primitive constraints. Such primitive constraints can also be used in the right-hand sides of simplification or propagation rules.

Example 2. The following CHR [6] define the processing of user-defined constraints for a less-than-or-equal relation `leq` (in a Prolog-like syntax). `true` denotes the empty multiset of constraints and "`=`" denotes the primitive constraint of syntactic equality.

```
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).
```

Simplification and propagation rules are denoted by "`<=>`" and "`==>`", respectively. The primitive constraints to the left of the symbol "`|`" is the guard of a rule. Multiple constraints are separated by commas which can be interpreted as logical conjunction. The rule `reflexivity` 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` 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` 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 consists of the constraints {`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`.

Although simplification and propagation rules form the kernel of CHR (but note that even a propagation rule can be considered as an abbreviation of simplification rule where the left-hand side occurs also in the right-hand side), it is sometimes useful to combine a simplification and a

[3] `http://www.cs.kuleuven.ac.be/~dtai/projects/CHR/`

propagation rule into one rule, called *simpagation rule*, where the left-hand side contains two parts separated by "\": the left part is kept like in a propagation rule and the right part is deleted like in a simplification rule. For instance, the following rule specifies that multiple occurrences of the same constraint can be deleted:

```
idempotence  @ leq(X,Y) \ leq(X,Y) <=> true.
```

□

In the next section we show how CHR can be integrated in the language Curry by defining data types and operators so that rules of CHR become regular Curry expressions.

4 Constraint Handling Rules in Curry

As already mentioned in Section 1, we want to avoid to extend the language Curry in order to deal with CHR. As an alternative, we intend to express a CHR specification as elements of a Curry program. In order to support type checking of CHR specifications and a consistent integration of CHR into the statically typed language Curry, we will represent CHR as data objects in Curry. For doing so, we start with the following data type that represents the basic structure of a constraint handling rule:

```
data CHRule goal = SimpRule   goal goal
                 | PropRule   goal goal
                 | SimpagRule goal goal goal
```

The data type is parameterized (via the type variable `goal`) over the type of goals, i.e., conjunctions of primitive or CHR constraints. Simplification rules (`SimpRule`) and propagation rules (`PropRule`) have two goal arguments (the left- and right-hand side), whereas a simpagation rule (`SimpagRule`) has three goal arguments (the two goals of the left-hand side and the right-hand side goal).

Next we have to define the data type of goals. Since a goal can be a conjunction or a single primitive or CHR constraint which should be represented as data, we need data constructors to distinguish the different cases. Moreover, there are some primitive constraints that operate on arguments of an arbitrary type (e.g., "`=`") and others that operate on specific types (e.g., relations between numbers). Therefore, we need also different constructors for different types of constraints in order to enable a reasonable type checking of a CHR specification by the type system of Curry. Due to these considerations, we define the structure of goals by the following data type `Goal` which is parameterized over the type variable `a`:

```
data Goal a = C             String
            | C_a           String a
            | C_a_a         String a a
            ...
            | C_Int         String Int
            | C_Int_Int     String Int Int
            | C_Int_Int_Int String Int Int Int
            ...
            | Conjunction (Goal a) (Goal a)
            | GuardedGoal (Goal a) (Goal a)
```

The constructors `C`... represent (primitive or CHR) constraints of the corresponding type. The first string argument is always the name of the constraint in the corresponding target code (see below) and the remaining arguments are the arguments of the corresponding constraint. The constructors `Conjunction` and `GuardedGoal` represent a conjunction of two goals or a goal with a guard (only used in right-hand sides of the rules), respectively.

Using these data types, a rule like

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

can be represented by the following Curry expression:

```
SimpRule (C_a_a "leq" x y) (GuardedGoal (C_a_a "=" x y) (C "true"))
```

Since such expressions are not very readable and tedious to write, we introduce some useful abstractions for the data constructors by the following infix operators:[4]

```
infixr 4  /\
infix  3  |>
infix  2  <=>
infix  2  ==>
infix  1  \\

g1 <=> g2 = SimpRule g1 g2

g1 ==> g2 = PropRule g1 g2

g1 \\ (SimpRule g2 g3) = SimpagRule g1 g2 g3

guard |> goal = GuardedGoal guard goal

c1 /\ c2 = Conjunction c1 c2
```

Furthermore, we define some standard primitive constraints as functions,[5] e.g.,

```
infix 5 .=.
x .=. y = C_a_a "=" x y      -- syntactic equality

true    = C "true"           -- trivial goal

fail    = C "fail"           -- unsatisfiable goal
```

Finally, we use a similar definition to denote the CHR constraint in a convenient way:

```
leq x y = C_a_a "leq" x y
```

By exploiting these definitions, we can write the above reflexivity rule as follows:

```
leq x y <=> x .=. y |> true
```

Thus, we obtain a CHR notation in Curry with almost the same syntax (but in a Curry-oriented style) as introduced in Section 3. The complete Curry module containing the rules for `leq` of Example 2 is as follows (remember that, in contrast to Prolog, all free variables must be declared in Curry in order to provide some consistency checks by the compiler):[6]

```
import CHR

leq x y = C_a_a "leq" x y

reflexivity  = leq x y <=> x .=. y |> true        where x,y free
```

[4] Note that "`|`", "`,`", and "`\`" cannot be used as infix operators in Curry since these symbols are part of the syntax of Curry. Therefore, we define the infix operators "`|>`" to represent guards, "`/\`" to represent conjunctions, and "`\\`" to separate goals in simpagation rules in CHR.

[5] Note that "`=`" cannot be defined as an infix operator since this is a symbol for equational definitions in Curry. Therefore, we define the infix operator "`.=.`" to represent syntactic equality in CHR.

[6] The imported module `CHR` contains the definitions for representing CHR in Curry as above and the CHR compiler described in the next section.

```
antisymmetry = leq x y /\ leq y x <=> x .=. y    where x,y free
transitivity = leq x y /\ leq y z ==> leq x z    where x,y,z free
idempotence  = leq x y \\ leq x y <=> true       where x,y free
```

Note that this CHR specification is a valid Curry program. Thus, we can load this module in a Curry programming environment like PAKCS in order to check the type-correctness of the specification. Furthermore, we can compile the CHR specification into executable code by evaluating the call

```
compileCHR "Leq" [reflexivity,antisymmetry,transitivity,idempotence]
```

where `compileCHR` is the CHR compiler defined in module `CHR`. This generates a module `Leq` containing the definition of a constraint

```
leq :: a -> a -> Success
```

whose semantics is defined by the given CHR rules interpreted in the underlying CHR language (which is, actually, the refined operational semantics of CHR [4]). In our current implementation, we provide only the functionality of "basic" CHR programs without pragmas and further compiler options. This could be added in future work. In the next section, we sketch the compilation process that generates the executable CHR program from our Curry-based definition.

5 Implementation

After having fixed the data structures to represent CHR in Curry, the implementation becomes straightforward if some CHR implementation in the target language is available. For instance, our Curry programming environment PAKCS compiles Curry programs into SICStus Prolog programs. Since SICStus Prolog also contains a CHR implementation, we can use this implementation to implement Curry CHR specifications. This is done by the operation "`compileCHR` *mod rules*", where *mod* is the name of the target Curry module containing the definition of the constraints defined by the list of rules *rules*. The operation `compileCHR` performs the following steps:

1. Evaluate the second argument *rules* to normal form and translate the resulting rules into an equivalent CHR Prolog program.
2. Generate an interface to provide access to the CHR constraints (implemented by the CHR Prolog program) from Curry programs. This is done by the same mechanism as for any other primitive operation provided by Curry.
3. Generate a Curry module containing definitions of constraints that call the corresponding Prolog code of the interface. For instance, the Curry module generated by the call to `compileCHR` shown at the end of Section 4 is as follows:

   ```
   module Leq where

   leq :: a -> a -> Success
   leq x1 x2 =  ...internal code to call the CHR Prolog code...
   ```

   Thus, any other application program implemented in Curry can import this module and use the CHR constraint `leq`.

The types of the CHR constraints in the generated Curry module depend on the structure of the rules and they are inferred by our CHR compiler. For instance, consider the following rule defining the behavior of the CHR constraint `leq` in the case that both arguments are integer numbers (the predefined constraint "`.<=.`" is the comparison on integer numbers):

```
leqint  =  leq x y <=> x .<=. y |> true       where x,y free
```

Note that this rule is of type `CHRule (Goal Int)`. If we add this rule to the compilation of the above `leq` rules, the resulting type of the CHR constraint `leq` is restricted to integer arguments since an application of the rule `leqint` to arguments that are not integers would lead to ill-typed run-time calls. Thus, our compiler generates the type definition

```
leq :: Int -> Int -> Success
```

in the Curry module `Leq` when we also compile the rule `leqint`. This ensures that application programs use the CHR constraint `leq` only with integer arguments.

Since the concrete implementation of our compiler based on this concept is rather technical, we omit a more detailed description here. The complete implementation is freely available from the author.

Note that the final structure of the rules is evaluated in the first step of our CHR compiler by the standard evaluation process of Curry. Thus, it is not important in which concrete syntactic way the rules are defined. For instance, in Section 4 we have defined each rule by a separate constant operation so that the list of all these operations is passed to `compileCHR`. This method is useful to test CHR specifications with different set of rules. As an alternative, we can also directly write all rules for each CHR constraint as a single list. For instance, consider a CHR specification of a solver for Boolean constraints. Then, the rules that define a CHR constraint `and` (Boolean conjunction) can be specified as follows:

```
andRules = [and x y z <=> x .=. 0 |> z .=. 0,
            and x y z <=> y .=. 0 |> z .=. 0,
            and x y z <=> x .=. 1 |> y .=. z,
            and x y z <=> y .=. 1 |> x .=. z,
            and x y z <=> z .=. 1 |> x .=. 1 /\ y .=. 1,
            and x y z <=> x .=. y |> y .=. z]
 where x,y,z free
```

If we do this for every Boolean CHR constraint (e.g., `and`, `or`, `neg`, `xor`, `imp`), we can finally compile the complete solver by

```
compileCHR "Bool" (andRules++orRules++negRules++xorRules++impRules)
```

If the rules have a regular structure that can be generated from some input data, we can also define functions that generate the rules from these inputs.

6 Conclusion

In this paper we presented the integration of CHR into the declarative multi-paradigm language Curry. In order to avoid a CHR-specific language extension of Curry which might be difficult to implement in different Curry systems, we proposed a representation of CHR as data objects in Curry programs that supports a notation of CHR rules in Curry similarly to the standard CHR notation. This representation has the advantage that CHR specifications are checked for well-typedness by the standard Curry type checker and CHR constraints can be used as any other constraint in source programs in a type-safe way. Furthermore, the rules can be defined in various ways using the functional notation provided by Curry. We also sketched an implementation of this concept in the Curry programming environment PAKCS that compiles Curry programs and our CHR specifications into SICStus Prolog. All examples presented in this paper have been tested with this implementation.

For future work, it might be interesting to investigate a direct implementation of CHR in Curry. This might have the advantage that, similarly to the usual Prolog implementations of CHR, we can use arbitrary user-defined Curry operations in the rules instead of a fixed set of primitive constraints as in our current approach.

Acknowledgements

The author is grateful to the anonymous referees for helpful comments and suggestions to improve this paper.

References

1. S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. *Journal of the ACM*, Vol. 47, No. 4, pp. 776–822, 2000.
2. S. Antoy and M. Hanus. Compiling Multi-Paradigm Declarative Programs into Prolog. In *Proc. International Workshop on Frontiers of Combining Systems (FroCoS'2000)*, pp. 171–185. Springer LNCS 1794, 2000.
3. S. Antoy and M. Hanus. Functional Logic Design Patterns. In *Proc. of the 6th International Symposium on Functional and Logic Programming (FLOPS 2002)*, pp. 67–87. Springer LNCS 2441, 2002.
4. 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)*, pp. 90–104. Springer LNCS 3132, 2004.
5. 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)*, pp. 320–338. Springer LNCS 2562, 2003.
6. T. Frühwirth. Theory and Practice of Constraint Handling Rules. *Journal of Logic Programming*, Vol. 37, No. 1-3, pp. 95–138, 1998.
7. M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, Vol. 19&20, pp. 583–628, 1994.
8. M. Hanus. A Unified Computation Model for Functional and Logic Programming. In *Proc. of the 24th ACM Symposium on Principles of Programming Languages (Paris)*, pp. 80–93, 1997.
9. M. Hanus. A Functional Logic Programming Approach to Graphical User Interfaces. In *International Workshop on Practical Aspects of Declarative Languages (PADL'00)*, pp. 47–62. Springer LNCS 1753, 2000.
10. 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)*, pp. 76–92. Springer LNCS 1990, 2001.
11. M. Hanus. Dynamic Predicates in Functional Logic Programs. *Journal of Functional and Logic Programming*, Vol. 2004, No. 5, 2004.
12. M. Hanus, S. Antoy, 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/`, 2005.
13. M. Hanus, K. Höppner, and F. Huch. Towards Translating Embedded Curry to C. *Electronic Notes in Theoretical Computer Science*, Vol. 86, No. 3, 2003.
14. M. Hanus and F. Huch. An Open System to Support Web-based Learning. In *Proc. 12th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2003)*, pp. 269–282. Technical Report DSIC-II/13/03, Universidad Politécnica de Valencia, 2003.
15. M. Hanus (ed.). Curry: An Integrated Functional Logic Language (Vers. 0.8). Available at `http://www.informatik.uni-kiel.de/~curry`, 2003.
16. C. Holzbaur and T. Frühwirth. Compiling Constraint Handling Rules into Prolog with Attributed Variables. In *Proc. of the International Conference on Principles and Practice of Declarative Programming (PPDP'99)*, pp. 117–133. Springer LNCS 1702, 1999.
17. J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In *Proc. of the 14th ACM Symposium on Principles of Programming Languages*, pp. 111–119, Munich, 1987.
18. R. Loogen, F. Lopez Fraguas, and M. Rodríguez Artalejo. A Demand Driven Computation Strategy for Lazy Narrowing. In *Proc. of the 5th International Symposium on Programming Language Implementation and Logic Programming*, pp. 184–200. Springer LNCS 714, 1993.
19. F. López-Fraguas and J. Sánchez-Hernández. TOY: A Multiparadigm Declarative System. In *Proc. of RTA'99*, pp. 244–247. Springer LNCS 1631, 1999.
20. W. Lux. Adding Linear Constraints over Real Numbers to Curry. In *Proc. of the 5th International Symposium on Functional and Logic Programming (FLOPS 2001)*, pp. 185–200. Springer LNCS 2024, 2001.
21. K. Marriott and P.J. Stuckey. *Programming with Constraints.* MIT Press, 1998.

22. S.L. Peyton Jones and J. Hughes. Haskell 98: A Non-strict, Purely Functional Language. http://www.haskell.org, 1999.
23. M. Rodríguez-Artalejo. Functional and Constraint Logic Programming. In *Constraints in Computational Logics: Theory and Applications (CCL'99)*, pp. 202–270. Springer LNCS 2002, 2001.
24. P. Wadler. How to Declare an Imperative. *ACM Computing Surveys*, Vol. 29, No. 3, pp. 240–263, 1997.