Implementing Equational Constraints in a Functional Language

Bernd Braßel, Michael Hanus, Björn Peemöller, and Fabian Reck

Institut für Informatik, CAU Kiel, D-24098 Kiel, Germany
{bbr|mh|bjp|fre}@informatik.uni-kiel.de

Abstract. KiCS2 is a new system to compile functional logic programs of the source language Curry into purely functional Haskell programs. The implementation is based on the idea to represent the search space as a data structure and logic variables as operations that generate their values. This has the advantage that one can apply various, in particular, complete search strategies or even user-defined strategies to compute solutions. However, the generation of all values for logic variables might be inefficient for applications that exploit constraints on partially known values. To overcome this drawback, we propose new techniques to implement equational constraints in this framework. In particular, we show how unification modulo function evaluation and functional patterns can be added without sacrificing the efficiency of the kernel implementation.

1 Introduction

Functional logic languages combine the most important features of functional and logic programming in a single language (see [6,16] for recent surveys). In particular, they provide higher-order functions and demand-driven evaluation from functional programming together with logic programming features like non-deterministic search and computing with partial information (logic variables). This combination has led to new design patterns [3,7] and better abstractions for application programming, but it also gave rise to new implementation challenges.

In order to implement a functional logic language, one can develop a suitable abstract machine and implement it in some (typically, imperative) language, like C [24] or Java [8,20]. One could also compile into logic languages like Prolog and reuse existing backtracking implementations for non-deterministic search as well as logic variables and unification for computing with partial information [2,23]. More recent approaches [10,12,13] compile functional logic programs into non-strict functional programs to reuse the implementation of lazy evaluation and higher-order functions. Although this requires the implementation of non-deterministic computations in a deterministic language, it has the advantage that the explicit handling of non-determinism allows for various search strategies, like depth-first, breadth-first, parallel, or iterative deepening, instead of committing to a fixed (incomplete) strategy like backtracking [12].

This paper is related to the latter implementation approach. In particular, we consider KiCS2 [11], a new system that compiles functional logic programs of the source language Curry [21] into purely functional Haskell programs. KiCS2 is based on the

idea to represent the search space, i.e., all non-deterministic results of a computation, as a data structure that can be traversed by operations implementing various strategies. Logic variables are replaced by generators, i.e., operations that non-deterministically evaluate to all possible ground values of the type of the logic variable. This is justified by the fact that computing with logic variables by narrowing [28,31] and computing with generators by rewriting are equivalent, i.e., yield the same values [5]. Although this implementation technique outperforms other implementations of Curry on deterministic programs and can compete with them on non-deterministic programs (see [11] for benchmarks), the generation of all values for logic variables might be inefficient for applications that exploit constraints on partially known values. For instance, the equality constraint "`X=c(a)`" is solved in Prolog by instantiating the variable `X` to `c(a)`, but the equality constraint "`X=Y`" is solved by binding `X` to `Y` without enumerating any values for `X` or `Y`. In order to obtain a similar behavior in KiCS2, we propose in this paper new techniques to implement equational constraints (in contrast to Prolog, Curry performs unification modulo function evaluation) in a purely functional target language. A purely functional, i.e., side-effect free, implementation is reasonable in order to support different, in particular, parallel or user-defined search strategies. Beyond equational constraints, we also show how functional patterns [4], i.e., patterns containing evaluable operations for more powerful pattern matching than in logic or functional languages, can be implemented in this framework. We show that both extensions lead to efficiency improvements without sacrificing the efficiency of the kernel implementation.

In the next section, we review the source language Curry and the features considered in this paper. Section 3 recapitulates the implementation scheme of KiCS2 originally presented in [11]. Sections 4 and 5 discuss our new extensions to implement unification modulo function evaluation and functional patterns, respectively. Benchmarks demonstrating the usefulness of these extensions are presented in Sect. 6 before we conclude in Sect. 7.

2 Curry Programs

The syntax of the functional logic language Curry [21] is close to Haskell [27], 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. The application of f to e is denoted by juxtaposition ("$f\ e$"). In addition to Haskell, Curry allows free (logic) variables in conditions and right-hand sides of defining rules. Hence, an operation is defined by conditional rewrite rules of the form:

$$f\ t_1 \ldots t_n \ \texttt{|}\ c\ \texttt{=}\ e \quad \texttt{where}\ vs\ \texttt{free} \tag{1}$$

where the *condition* c is optional and vs is the list of variables occurring in c or e but not in the *left-hand side* $f\ t_1 \ldots t_n$.

In contrast to functional programming and similarly to logic programming, operations can be defined by overlapping rules so that they might yield more than one result on the same input. Such operations are also called *non-deterministic*. For instance, Curry offers a *choice* operation that is predefined by the following rules:

```
x ? _ = x
_ ? y = y
```

Thus, we can define a non-deterministic operation `aBool` by

```
aBool = True ? False
```

so that the expression "`aBool`" has two values: `True` and `False`.

If non-deterministic operations are used as arguments in other operations, a semantical ambiguity might occur. Consider the operations

```
not True  = False          xor True  x = not x
not False = True           xor False x = x

xorSelf x = xor x x
```

and the expression "`xorSelf aBool`". If we interpret this program as a term rewriting system, we could have the reduction

```
xorSelf aBool  →  xor aBool aBool  →  xor True aBool
               →  xor True False   →  not False     →  True
```

leading to the unintended result `True`. Note that this result cannot be obtained if we use a strict strategy where arguments are evaluated prior to the function calls. In order to avoid dependencies on the evaluation strategies and exclude such unintended results, the rewriting logic CRWL is proposed in [15] as a logical (execution- and strategy-independent) foundation for declarative programming with non-strict and non-deterministic operations. This logic specifies the *call-time choice* semantics [22], where values of the arguments of an operation are determined before the operation is evaluated. In a lazy strategy, this can be enforced by sharing actual arguments. For instance, the expression above can be lazily evaluated provided that all occurrences of `aBool` are shared so that all of them reduce either to `True` or to `False` consistently.

The condition c in rule (1) typically is a conjunction of *equational constraints* of the form $e_1 =:= e_2$. Such a constraint is satisfiable if both sides e_1 and e_2 are reducible to unifiable data terms. For instance, if the symbol "`++`" denotes the usual list concatenation operation, we can define an operation `last` that computes the last element `e` of a non-empty list `xs` as follows:

```
last xs | ys++[e] =:= xs  = e   where ys, e free
```

Like in Haskell, most rules defining functions are *constructor-based* [26], i.e., in (1) $t_1, \ldots, t_n$ consist of variables and/or data constructor symbols only. However, Curry also allows *functional patterns* [4], i.e., t_i might additionally contain calls to defined operations. For instance, we can also define the last element of a list by the more concise definition

```
last' (xs++[e]) = e
```

Here, the functional pattern `(xs++[e])` states that `(last' t)` is reducible to `e` provided that the argument `t` can be matched against some value of `(xs++[e])` where `xs` and `e` are free variables. By instantiating `xs` to arbitrary lists, the value of `(xs++[e])` is any list having `e` as its last element. Functional patterns are a powerful feature to express arbitrary selections in term structures. For instance, they support a straightforward processing of XML data with incompletely specified or evolving formats [17].

3 The Compilation Scheme of KiCS2

To understand the extensions described in the subsequent sections, we review the translation of Curry programs into Haskell programs as performed by KiCS2. More details about this translation scheme can be found in [10,11].

As mentioned in the introduction, the KiCS2 implementation is based on the explicit representation of non-deterministic results in a data structure. This is achieved by extending each data type of the source program by constructors to represent a choice between two values and a failure, respectively. For instance, the data type for Boolean values defined in a Curry program by

```
data Bool = False | True
```

is translated into the Haskell data type[1]

```
data Bool = False | True | Choice ID Bool Bool | Fail
```

where `Fail` represents a failure and `(Choice i t1 t2)` a non-deterministic value, i.e., a selection of two values `t1` and `t2` that can be chosen by some search strategy. The first argument `i` of type `ID` of a `Choice` constructor is used to implement the call-time choice semantics discussed in Sect. 2. Since the evaluation of `xorSelf aBool` duplicates the argument operation `aBool`, we have to ensure that both duplicates, which later evaluate to a non-deterministic choice between two values, yield either `True` or `False`. This is obtained by assigning a unique identifier (of type `ID`) to each `Choice` constructor. The difficulty is to get a unique identifier on demand, i.e., when some operation evaluates to a `Choice`. We cannot thread an identifier supply, e.g., a counter, through the search tree without fixing an evaluation order. Since we want to compile into *purely* functional programs (in order to enable powerful program optimizations), we can neither use unsafe features with side effects to generate such identifiers. Hence, we follow the idea presented in [9] and pass a (conceptually infinite) set of identifiers, also called *identifier supply*, to each operation so that a `Choice` can pick its unique identifier from this set. For this purpose, we assume a type `IDSupply`, representing an infinite set of identifiers, with operations

```
initSupply  :: IO IDSupply
thisID      :: IDSupply → ID
leftSupply  :: IDSupply → IDSupply
rightSupply :: IDSupply → IDSupply
```

The operation `initSupply` creates such a set (at the beginning of an execution), the operation `thisID` takes some identifier from this set, and `leftSupply` and `rightSupply` split this set into two disjoint subsets without the identifier obtained by `thisID`. There are different implementations available (see below for a simple one) and our system is parametric over concrete implementations of `IDSupply`.

When translating Curry to Haskell, KiCS2 adds to each operation an additional argument of type `IDSupply`. For instance, the operation `aBool` defined in Sect. 2 is translated into:

[1] Actually, our compiler performs some renamings to avoid conflicts with predefined Haskell entities and introduces type classes to resolve overloaded symbols like `Choice` and `Fail`.

```
aBool :: IDSupply → Bool
aBool s = Choice (thisID s) True False
```

Similarly, the operation

```
main :: Bool
main = xorSelf aBool
```

is translated into

```
main :: IDSupply → Bool
main s = xorSelf (aBool (leftSupply s)) (rightSupply s)
```

so that the set `s` is split into a set `(leftSupply s)` containing identifiers for the evaluation of the argument `aBool` and a set `(rightSupply s)` containing identifiers for the evaluation of the operation `xorSelf`.

Since all data types are extended by additional constructors, we must also extend the definition of operations performing pattern matching.[2] For instance, consider the definition of polymorphic lists

```
data List a = Nil | Cons a (List a)
```

and an operation to extract the first element of a non-empty list:

```
head :: List a → a
head (Cons x xs) = x
```

The type definition is then extended as described above:

```
data List a = Nil | Cons a (List a) | Choice ID (List a) (List a) | Fail
```

The operation `head` is extended by an identifier supply and further matching rules:

```
head :: List a → IDSupply → a
head (Cons x xs)      s = x
head (Choice i x1 x2) s = Choice i (head x1 s) (head x2 s)
head _                s = Fail
```

The second rule transforms a non-deterministic argument into a non-deterministic result and the final rule returns `Fail` in all other cases, i.e., if `head` is applied to the empty list as well as if the matching argument is already a failed computation (failure propagation). Since deterministic operations do not introduce new `Choice` constructors, `head` does not use the identifier supply `s`.

To show a concrete example, we use the following implementation of `IDSupply` based on unbounded integers:

```
type IDSupply = Integer
initSupply    = return 1
thisID      n = n
leftSupply  n = 2 * n
rightSupply n = 2 * n + 1
```

If we apply the same transformation to the rules defining `xor` and evaluate the main expression `(main 1)`, we obtain the result

[2] To obtain a simple compilation scheme, KiCS2 transforms source programs into uniform programs [11] where pattern matching is restricted to a single argument. This is always possible by introducing auxiliary operations.

```
Choice 2 (Choice 2 False True) (Choice 2 True False)
```

Thus, the result is non-deterministic and contains three choices, whereby all of them have the same identifier. To extract all values from such a `Choice` structure, we have to traverse it and compute all possible choices but consider the choice identifiers to make consistent (left/right) decisions. Thus, if we select the left branch as the value of the outermost `Choice`, we also have to select the left branch in the selected argument `(Choice 2 False True)` so that `False` is the only value possible for this branch. Similarly, if we select the right branch as the value of the outermost `Choice`, we also have to select the right branch in its selected argument `(Choice 2 True False)`, which again yields `False` as the only possible value. In consequence, the unintended value `True` is not extracted as a result.

The requirement to make consistent decisions can be implemented by storing the decisions already made for some choices during the traversal. For this purpose, we introduce the type

```
data Decision = NoDecision | ChooseLeft | ChooseRight
```

where `NoDecision` represents the fact that the value of a choice has not been decided yet. Furthermore, we assume operations to lookup the current decision for a given identifier or change it (depending on the implementation of `IDSupply`, KiCS2 supports several implementations based on memory cells or finite maps). For a top-level operation that prints all values contained in a choice structure in a depth-first manner, these operations would be of the following types:

```
lookupDecision :: ID  → IO Decision
setDecision    :: ID  → Decision  → IO ()
```

Now the search operation can be defined by the I/O operation below:[3]

```
printValsDFS :: a  → IO ()

printValsDFS Fail             = return ()

printValsDFS (Choice i x1 x2) = lookupDecision i >>= follow
  where
    follow ChooseLeft  = printValsDFS x1
    follow ChooseRight = printValsDFS x2
    follow NoDecision  = do newDecision ChooseLeft  x1
                            newDecision ChooseRight x2

    newDecision d x = do setDecision i d
                         printValsDFS x
                         setDecision i NoDecision

printValsDFS v = print v
```

This operation ignores failures and prints values that are not rooted by a `Choice` constructor. For a `Choice` constructor, it checks whether a decision for this identifier has already been made (note that the initial value for all identifiers is `NoDecision`). If a decision has been made for this choice, it follows this decision. Otherwise, the left al-

[3] Note that this code has been simplified and slightly renamed compared to [11] for readability. The type system of Haskell does not allow this direct definition.

ternative is used and this decision is stored. After printing all values w.r.t. this decision, the decision is undone (like in backtracking) and the right alternative is used and stored.

In general, this operation is applied to the normal form of the main expression (where `initSupply` is used to compute an initial identifier supply passed to this expression). The normal form computation is necessary for structured data, like lists, so that a failure or choice in some part of the data is moved to the root.

Other search strategies, like breadth-first search, iterative deepening, or parallel search, can be obtained by different implementations of this top-level operation to print all values. Instead of printing them, one can also collect the values in a tree-like data structure for further processing. Thus, KiCS2 supports a primitive

```
getSearchTree :: a → IO (SearchTree a)
```

that returns the search tree corresponding to the evaluation of its argument, where the search tree is some computed value, a failure, or a choice between two trees:

```
data SearchTree a = Value a | Fail | Or (SearchTree a) (SearchTree a)
```

This primitive is useful to encapsulate non-deterministic operations and select some result value, e.g., the "first" or the "best" one according to some ordering. Since the search tree is created in a demand-driven manner, the primitive is also applicable to infinite search spaces (in contrast to Prolog's `findall` primitive [25]). Based on this representation, a Curry programmer can define his own search strategies as tree traversals in his source program without any modification of the Curry compiler (see [19] for detailed examples). Note that these kinds of applications demand for a side-effect free implementation of non-deterministic computations (in contrast to traditional Prolog implementations [1])—which is the challenge addressed in this paper.

Since large parts of typical functional logic computations are deterministic, KiCS2 performs an optimization for deterministic operations. If an operation is defined by non-overlapping rules and does not call, neither directly nor indirectly through other operations, an operation defined by overlapping rules, the evaluation of such an operation (like `head`) cannot introduce non-deterministic values. Thus, it is not necessary to pass an identifier supply to the operation. In consequence, only the matching rules are extended by additional cases for handling `Choice` and `Fail` so that the generated code is nearly identical to a corresponding functional program. Actually, the benchmarks presented in [11] show that for deterministic operations this implementation outperforms all other Curry implementations, and, for non-deterministic operations, outperforms Prolog-based implementations of Curry and can compete with MCC [24], a Curry implementation that compiles to C.

As mentioned in the introduction, KiCS2 translates occurrences of logic variables into generators. For instance, the expression "`not x`", where `x` is a logic variable, is translated into "`not (aBool s)`", where `s` is an `IDSupply` provided by the context of the expression. The latter expression is evaluated by reducing the argument `(aBool s)` to a choice between `True` or `False` followed by applying `not` to this choice. This is similar to a narrowing step [28] on "`not x`" that instantiates the variable `x` to `True` or `False`. Since such generators are standard non-deterministic operations, they are translated like any other operation and, therefore, do not require any additional run-time support. However, in the presence of equational constraints, there are methods which

are more efficient than generating all values. These methods and their implementation are discussed in the next section.

4 Equational Constraints and Unification

As known from logic programming, predicates or constraints are important to restrict the set of intended values in a non-deterministic computation. Apart from user-defined predicates, equational constraints of the form $e_1 =:= e_2$ are the most important kind of constraints. We have already seen a typical application of an equational constraint in the operation `last` in Sect. 2.

Due to the presence of non-terminating operations and infinite data structures, "`=:=`" is interpreted as the *strict equality* on terms [14], i.e., the equation $e_1 =:= e_2$ is satisfied iff e_1 and e_2 are reducible to unifiable constructor terms. In particular, expressions that do not have a value are not equal w.r.t. "`=:=`", e.g., the equational constraint "`head [] =:= head []`" is not satisfiable.[4]

According to this definition, "`=:=`" can be considered as a binary function defined by the following rules (we only present the rules for the Boolean and list types, where `Success` denotes the only constructor of the type `Success` of constraints):

```
True   =:= True    =  Success
False  =:= False   =  Success

[]     =:= []      =  Success
(x:xs) =:= (y:ys)  =  x =:= y & xs =:= ys

Success & c  =  c
```

If we translate these operations into Haskell by the scheme presented in Sect. 3, the following rules are added to these rules in order to propagate choices and failures:

```
Fail         =:= _             =  Fail
_            =:= Fail          =  Fail
Choice i l r =:= y             =  Choice i (l =:= y) (r =:= y)
x            =:= Choice i l r  =  Choice i (x =:= l) (x =:= r)
_            =:= _             =  Fail

Fail         & _     =  Fail
Choice i l r & c     =  Choice i (l & c) (r & c)
_            & _     =  Fail
```

Although this is a correct implementation of equational constraints, it might lead to an unnecessarily large search space when it is applied to generators representing logic variables. For instance, consider the following generator for Boolean lists:

```
aBoolList = [] ? (aBool : aBoolList)
```

This is translated into Haskell as follows:

[4] From now on, we use the standard notation for lists, i.e., `[]` denotes the empty list and `(x:xs)` denotes a list with head element `x` and tail `xs`.

```
aBoolList :: IDSupply → [Bool]
aBoolList s = Choice (thisID s) [] (aBool (leftSupply s)
                                        : aBoolList (rightSupply s))
```

Now consider the equational constraint "`x =:= [True]`". If the logic variable `x` is replaced by `aBoolList`, the translated expression "`aBoolList s =:= [True]`" creates a search space when evaluating its first argument, although there is no search required since there is only one binding for `x` satisfying the constraint. Furthermore and even worse, unifying two logic variables introduces an infinite search space. For instance, the expression "`xs =:= ys & xs++ys =:= [True]`" results in an infinite search space when the logic variables `xs` and `ys` are replaced by generators.

To avoid these problems, we have to implement the idea of the well-known unification principle [29]. Instead of enumerating all values for logic variables occurring in an equational constraint, we *bind* the variables to another variable or term. Since we compile into a purely functional language, the binding cannot be performed by some side effect. Instead, we add binding constraints to the computed results to be processed by a search strategy that extracts values from choice structures.

To implement unification, we have to distinguish free variables from "standard choices" (introduced by overlapping rules) in the target code. For this purpose, we refine the definition of the type `ID` as follows:[5]

```
data ID = ChoiceID Integer | FreeID Integer
```

The new constructor `FreeID` identifies a choice corresponding to a free variable, e.g., the generator for Boolean variables is redefined as

```
aBool s = Choice (FreeID (thisID s)) True False
```

If an operation is applied to a free variable and requires its value, the free variable is transformed into a standard choice. For this purpose, we define a simple operation to perform this transformation:

```
narrow :: ID → ID
narrow (FreeID i) = ChoiceID i
narrow x          = x
```

We use this operation in narrowing steps, i.e., in all rules operating on `Choice` constructors. For instance, in the implementation of the operation `not` we replace the rule

```
not (Choice i x1 x2) s = Choice i (not x1 s) (not x2 s)
```

by the rule

```
not (Choice i x1 x2) s = Choice (narrow i) (not x1 s) (not x2 s)
```

to ensure that the resulting choice is not considered a free variable.

As mentioned above, the consideration of free variables is relevant in equational constraints where *binding constraints* are generated. For this purpose, we introduce a type to represent a binding constraint as a pair of a choice identifier and a decision for this identifier:

```
data Constraint = ID :=: Decision
```

[5] For the sake of simplicity, in the following, we consider the implementation of `IDSupply` to be unbounded integers.

Furthermore, we extend each data type by the possibility to add constraints:

```
data Bool   = ... | Guard [Constraint] Bool
data List a = ... | Guard [Constraint] (List a)
```

A single `Constraint` provides the decision for one constructor. In order to support constraints for structured data, a list of `Constraint`s provides the decision for the outermost constructor and the decisions for all its arguments. Thus, `(Guard` cs v`)` represents a *constrained value*, i.e., the value v is only valid if the constraints cs are consistent with the decisions previously made during search. These binding constraints are created by the equational constraint operation "`=:=`": if a free variable should be bound to a constructor, we make the same decisions as it would be done in the successful branch of the generator. In case of Boolean values, this can be implemented by the following additional rules for "`=:=`":

```
Choice (FreeID i) _ _ =:= True  = Guard [i :=: ChooseLeft ] Success
Choice (FreeID i) _ _ =:= False = Guard [i :=: ChooseRight] Success
```

Hence, the binding of a variable to some known value is implemented as a binding constraint for the choice identifier for this variable. However, if we want to bind a variable to another variable, we cannot store a concrete decision. Instead, we store the information that the decisions for both variables, when they are made to extract values, must be identical. For this purpose, we extend the `Decision` type to cover this information:

```
data Decision = ... | BindTo ID
```

Furthermore, we add to the definition of "`=:=`" the rule that an equational constraint between two variables yields a binding for these variables:

```
Choice (FreeID i) _ _ =:= Choice (FreeID j) _ _
  =  Guard [i :=: BindTo j] Success
```

The consistency of constraints is checked when values are extracted from a choice structure, e.g., by the operation `printValsDFS`. For this purpose, we extend the definition of the corresponding search operations by calling a constraint solver for the constraints. For instance, the definition of `printValsDFS` is extended by a rule handling constrained values:

```
...
printValsDFS (Guard cs x) = do consistent <- add cs
                               if consistent then do printValsDFS x
                                                     remove cs
                                             else return ()
...
```

The operation `add` checks the consistency of the constraints `cs` with the decisions made so far and, in case of consistency, stores the decisions made by the constraints. In this case, the constrained value is evaluated before the constraints are removed to allow backtracking. Furthermore, the operations `lookupDecision` and `setDecision` are extended to deal with bindings between two variables, i.e., they follow variable chains in case of `BindTo` constructors.

Finally, with the ability to distinguish free variables (choices with an identifier of the form `(FreeID ...)`) from other values during search, values containing logic variables

can also be printed in a specific form rather than enumerating all values, similarly to logic programming systems. For instance, KiCS2 evaluates the application of `head` to an unknown list as follows:

```
Prelude> head xs where xs free
{xs = (_x2:_x3)} _x2
```

Here, free variables are marked by the prefix `_x`.

5 Functional Patterns

A well-known disadvantage of equational constraints is the fact that "`=:=`" is interpreted as strict equality. Thus, if one uses equational constraints to express requirements on arguments, the resulting operations might be too strict. For instance, the equational constraint in the condition defining `last` (see Sect. 2) requires that `ys++[e]` as well as `xs` must be reducible to unifiable terms so that in consequence the input list `xs` is completely evaluated. Hence, if `failed` denotes an operation whose evaluation fails, the evaluation of `last [failed,True]` has no result. On the other hand, the evaluation of `last' [failed,True]` yields the value `True`, i.e., the definition of `last'` is less strict thanks to the use of functional patterns. Beyond this improved operational behavior, functional patterns can lead to more expressive programs (e.g., matching and unification on infinite structures, pattern matching at arbitrary depth in recursive data structures) and more elegant program patterns (see [4,7,17] for examples).

Conceptually, a functional pattern like `(xs++[e])` abbreviates all values to which it can be evaluated (by narrowing), like `[e]`, `[x1,e]`, `[x1,x2,e]`, and so on. In consequence, the rule defining `last'` abbreviates the following (infinite) set of rules:

```
last' [e] = e
last' [x1,e] = e
last' [x1,x2,e] = e
...
```

Obviously, one cannot implement functional patterns by a transformation into an infinite set of rules. Instead, they are implemented by a specific *lazy unification* procedure "`=:<=`" [4]. For instance, the definition of `last'` is transformed into

```
last' ys | (xs++[e]) =:<= ys  = e   where xs, e free
```

The behavior of "`=:<=`" is similar to "`=:=`", except for the case that a variable in the left argument should be bound to some expression: instead of evaluating the expression to some value and binding the variable to the value, the variable is bound to the *unevaluated* expression (see [4] for more details). Due to this slight change, failures or infinite structures in actual arguments do not cause problems in the matching of functional patterns.

Our proposed implementation of functional patterns in KiCS2 has a structure that is quite similar to that of equational constraints with the exception that variables could be also bound to unevaluated expressions. Only if such variables are later accessed, the expressions they are bound to are evaluated. This can be achieved by adding a further alternative to the type of decisions:

```
data Decision = ... | LazyBind [Constraint]
```

The implementation of the lazy unification operation "`=:<=`" is almost identical to the strict unification operation "`=:=`" as shown in Sect. 4. The only difference is in the rules where a free variable occurs in the left argument. All these rules are replaced by the single rule

```
Choice (FreeID i) _ _ =:<= x
  = Guard [i :=: LazyBind (lazyBind i x)] Success
```

where the auxiliary operation `lazyBind` implements the demand-driven evaluation of the right argument `x`:

```
lazyBind :: ID → a → [Constraint]
lazyBind i True  = [i :=: ChooseLeft]
lazyBind i False = [i :=: ChooseRight]
```

The use of the additional `LazyBind` constructor allows the argument `x` to be stored in a binding constraint without evaluation (due to the lazy evaluation strategy of the target language Haskell). However, it is evaluated by `lazyBind` to head normal form when its binding is required by another part of the computation, whereas the binding constraints for any sub-expression are in turn lazily computed using `lazyBind`.

Similarly to equational constraints, lazy bindings are processed by a solver when values are extracted. In particular, if a variable has more than one lazy binding constraint (which is possible if a functional pattern evaluates to a non-linear term), the corresponding expressions are evaluated and unified according to the semantics of functional patterns [4].

In order to demonstrate the operational behavior of our implementation, we sketch the evaluation of the lazy unification constraint `xs++[e] =:<= [failed,True]` that occurs when the expression `last' [failed,True]` is evaluated (we omit failed branches and some other details). Note that logic variables are replaced by generators, i.e., we assume that `xs` is replaced by `aBoolList 2` and `e` is replaced by `aBool 3`:

```
   aBoolList 2 ++ [aBool 3] =:<= [failed, True]
⇝ [aBool 4, aBool 3] =:<= [failed, True]
⇝ aBool 4 =:<= failed & aBool 3 =:<= True & [] =:<= []
⇝ Guard [ 4 :=: LazyBind (lazyBind 4 failed)
        , 3 :=: LazyBind (lazyBind 3 True)] Success
```

If the value of the expression `last' [failed,True]` is later required, the value of the variable `e` (with the identifier `3`) is in turn required. Thus, `(lazyBind 3 True)` is evaluated to `[3 :=: ChooseLeft]` which corresponds to the value `True` of the generator `(aBool 3)`. Note that the variable with identifier `4` does not occur anywhere else so that the binding `(lazyBind 4 failed)` will never be evaluated, as intended.

6 Benchmarks

In this section we evaluate our implementation of equational constraints and functional patterns by some benchmarks. The benchmarks were executed on a Linux machine running Debian 5.0.7 with an Intel Core 2 Duo (3.0GHz) processor. KiCS2 has been used with the Glasgow Haskell Compiler (GHC 7.0.4, option -O2) as its backend

Expression	==	=:=	=:<=
`last (map (inc 0) [1..10000])`	2.91	0.05	0.01
`simplify`	10.30	6.77	7.07
`varInExp`	2.34	0.24	0.21
`fromPeano (half (toPeano 10000))`	26.67	5.95	11.19
`palindrome`	30.86	14.05	20.26
`horseman`	3.24	3.31	n/a
`grep`	1.06	0.10	n/a

Fig. 1. Benchmarks: comparing different representations for equations

and an efficient `IDSupply` implementation that makes use of `IORef`s. For a comparison with other mature implementations of Curry, we considered PAKCS [18] (version 1.9.2, based on a SICStus-Prolog 4.1.2) and MCC [24] (version 0.9.10). The timings were performed with the time command measuring the execution time (in seconds) of a compiled executable for each benchmark as a mean of three runs. The programs used for the benchmarks, partially taken from [4], are `last` (compute the last element of a list),[6] `simplify` (simplify a symbolic arithmetic expression), `varInExp` (non-deterministically return a variable occuring in a symbolic arithmetic expression), `half` (compute the half of a Peano number using logic variables), `palindrome` (check whether a list is a palindrome), `horseman` (solving an equation relating heads and feet of horses and men based on Peano numbers), and `grep` (string matching based on a non-deterministic specification of regular expressions [6]).

In Sect. 4 we mentioned that equational constraints could also be solved by generators without variable bindings, but this technique might increase the search space due to the possibly superfluous generation of all values. To show the beneficial effects of our implementation of equational constraints with variable bindings, in Fig. 1 we compare the results of using equational constraints (`=:=`) to the results where the Boolean equality operator (`==`) is used (which does not perform bindings but enumerate all values). As expected, in most cases the creation and traversal of a large search space introduced by "==" is much slower than our presented approach with variable bindings. In addition, the example `last` shows that the lazy unification operator (`=:<=`) improves the performance when unifying an expression which has to be evaluted only partially. Using strict unification, all elements of the list are (unnecessarily) evaluated. On the other hand, lazy unification causes some overhead when the expressions are fully evaluated, which is shown by the `fromPeano` and `palindrome` examples. Thus, it is reasonable to use it only if its improved computational power is really required, as intended by its design.

In contrast to the Curry implementations PAKCS and MCC, our implementation of strict unification is based on an explicit representation of the search space instead of backtracking and manipulating a global state containing bindings for logic variables. Nevertheless, the benchmarks in Fig. 2, using equational constraints only, show that it can compete with or even outperform the other implementations. The results show that the implementation of unification of MCC performs best. However, in most cases

[6] "`inc x n`" is a naive addition that `n` times increases its argument `x` by 1.

Expression	KiCS2	PAKCS	MCC
`last (map (inc 0) [1..10000])`	0.05	0.40	0.01
`simplify`	6.77	0.15	0.00
`varInExp`	0.24	0.89	0.07
`fromPeano (half (toPeano 10000))`	5.95	108.88	3.22
`palindrome`	14.05	32.56	1.07
`horseman`	3.31	8.70	0.42
`grep`	0.10	2.88	0.14

Fig. 2. Benchmarks: strict unification in different Curry implementations

Expression	KiCS2	PAKCS
`last (map (inc 0) [1..10000])`	0.01	0.33
`simplify`	7.07	0.27
`varInExp`	0.21	1.87
`fromPeano (half (toPeano 10000))`	11.19	∞
`palindrome`	20.26	∞

Fig. 3. Benchmarks: functional patterns in different Curry implementations

our implementation outperforms the Prolog-based PAKCS implementation, except for some examples. In particular, `simplify` does not perform well due to expensive bindings of free variables to large arithmetic expressions in unsuccessful branches of the search. Further investigation and optimization will hopefully lead to a better performance in such cases.

As MCC does not support functional patterns, the performance of lazy unification is compared with PAKCS only (Fig. 3). Again, our compiler performs well against PAKCS and outperforms it in most cases ("∞" denotes a run time of more than 30 minutes).

7 Conclusions and Related Work

We have presented an implementation of equational constraints and functional patterns in KiCS2, a purely functional implementation of Curry. In addition to the kernel implementation described in [11], we add binding constraints to computed values which are processed when values are extracted at the top level of a computation. Since only new constructors and pattern matching rules for them are added in our implementation, no overhead is introduced for programs without equational constraints, i.e., our implementation does not sacrifice the high efficiency of the kernel implementation shown in [11]. However, if these features are used, they usually lead to a comparably efficient execution, as demonstrated by our benchmarks. Although the benchmarks were small in order to evaluate our unification implementation, it should be noted that KiCS2 is used in larger applications, like the curricula and module information system of our department[7]. In these and similar applications, where large parts are purely functional computations, KiCS2 is 15-20 times faster than PAKCS [18].

[7] `http://www-ps.informatik.uni-kiel.de/~mh/studiengaenge/`

Other implementations of equational constraints in functional logic languages are based on side effects. For instance, PAKCS [18] exploits the implementation of logic variables in Prolog, which are implemented on the primitive level by side effects. MCC [24] compiles into C where a specific abstract machine implements the handling of logic variables. We have shown that our implementation is competitive to those. In contrast to those systems, our implementation supports a variety of "top-level" search strategies, like iterative deepening, breadth-first or parallel search, as well as user-programmable search strategies, where the avoidance of side effects is important.

For future work it might be interesting to add further constraint structures to our implementation, like real arithmetic or finite domain constraints. This might be possible by extending the kinds of constraints of our implementation and solving them by functional programming approaches like [30].

References

1. H. Aït-Kaci. *Warren's Abstract Machine*. MIT Press, 1991.
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. 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)*, pp. 6–22. Springer LNCS 3901, 2005.
5. S. Antoy and M. Hanus. Overlapping Rules and Logic Variables in Functional Logic Programs. In *Proceedings of the 22nd International Conference on Logic Programming (ICLP 2006)*, pp. 87–101. Springer LNCS 4079, 2006.
6. S. Antoy and M. Hanus. Functional Logic Programming. *Communications of the ACM*, Vol. 53, No. 4, pp. 74–85, 2010.
7. 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)*, pp. 19–34. Springer LNCS 6816, 2011.
8. S. Antoy, M. Hanus, J. Liu, and A. Tolmach. A Virtual Machine for Functional Logic Computations. In *Proc. of the 16th International Workshop on Implementation and Application of Functional Languages (IFL 2004)*, pp. 108–125. Springer LNCS 3474, 2005.
9. L. Augustsson, M. Rittri, and D. Synek. On generating unique names. *Journal of Functional Programming*, Vol. 4, No. 1, pp. 117–123, 1994.
10. B. Braßel and S. Fischer. From Functional Logic Programs to Purely Functional Programs Preserving Laziness. In *Proceedings of the 20th International Symposium on Implementation and Application of Functional Languages (IFL 2008)*, pp. 25–42. Springer LNCS 5836, 2008.
11. B. Braßel, M. Hanus, B. Peemöller, and F. Reck. KiCS2: A New Compiler from Curry to Haskell. In *Proc. of the 20th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2011)*, pp. 1–18. Springer LNCS 6816, 2011.
12. B. Braßel and F. Huch. On a Tighter Integration of Functional and Logic Programming. In *Proc. APLAS 2007*, pp. 122–138. Springer LNCS 4807, 2007.

13. B. Braßel and F. Huch. The Kiel Curry System KiCS. In *Applications of Declarative Programming and Knowledge Management*, pp. 195–205. Springer LNAI 5437, 2009.
14. E. Giovannetti, G. Levi, C. Moiso, and C. Palamidessi. Kernel LEAF: A Logic plus Functional Language. *Journal of Computer and System Sciences*, Vol. 42, No. 2, pp. 139–185, 1991.
15. J.C. González-Moreno, M.T. Hortalá-González, F.J. López-Fraguas, and M. Rodríguez-Artalejo. An approach to declarative programming based on a rewriting logic. *Journal of Logic Programming*, Vol. 40, pp. 47–87, 1999.
16. M. Hanus. Multi-paradigm Declarative Languages. In *Proceedings of the International Conference on Logic Programming (ICLP 2007)*, pp. 45–75. Springer LNCS 4670, 2007.
17. M. Hanus. Declarative Processing of Semistructured Web Data. In *Technical Communications of the 27th International Conference on Logic Programming*, volume 11, pp. 198–208. Leibniz International Proceedings in Informatics (LIPIcs), 2011.
18. 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/`, 2010.
19. M. Hanus, B. Peemöller, and F. Reck. Search Strategies for Functional Logic Programming. In *Proc. of the 5th Working Conference on Programming Languages (ATPS'12)*, pp. 61–74. Springer LNI 199, 2012.
20. M. Hanus and R. Sadre. An Abstract Machine for Curry and its Concurrent Implementation in Java. *Journal of Functional and Logic Programming*, Vol. 1999, No. 6, 1999.
21. M. Hanus (ed.). Curry: An Integrated Functional Logic Language (Vers. 0.8.3). Available at `http://www.curry-language.org`, 2012.
22. H. Hussmann. Nondeterministic Algebraic Specifications and Nonconfluent Term Rewriting. *Journal of Logic Programming*, Vol. 12, pp. 237–255, 1992.
23. F. López-Fraguas and J. Sánchez-Hernández. TOY: A Multiparadigm Declarative System. In *Proc. of RTA'99*, pp. 244–247. Springer LNCS 1631, 1999.
24. W. Lux. Implementing Encapsulated Search for a Lazy Functional Logic Language. In *Proc. 4th Fuji International Symposium on Functional and Logic Programming (FLOPS'99)*, pp. 100–113. Springer LNCS 1722, 1999.
25. L. Naish. All Solutions Predicates in Prolog. In *Proc. IEEE Internat. Symposium on Logic Programming*, pp. 73–77, Boston, 1985.
26. M.J. O'Donnell. *Equational Logic as a Programming Language*. MIT Press, 1985.
27. S. Peyton Jones, editor. *Haskell 98 Language and Libraries—The Revised Report*. Cambridge University Press, 2003.
28. U.S. Reddy. Narrowing as the Operational Semantics of Functional Languages. In *Proc. IEEE Internat. Symposium on Logic Programming*, pp. 138–151, Boston, 1985.
29. J.A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, Vol. 12, No. 1, pp. 23–41, 1965.
30. T. Schrijvers, P. Stuckey, and P. Wadler. Monadic Constraint Programming. *Journal of Functional Programming*, Vol. 19, No. 6, pp. 663–697, 2009.
31. J.R. Slagle. Automated Theorem-Proving for Theories with Simplifiers, Commutativity, and Associativity. *Journal of the ACM*, Vol. 21, No. 4, pp. 622–642, 1974.