Implementing Relational Specifications in a Constraint Functional Logic Language

Rudolf Berghammer and Sebastian Fischer[1]

Institut für Informatik
Universität Kiel
Olshausenstraße 40, 24098 Kiel, Germany

Abstract

We show how the algebra of (finite, binary) relations and the features of the integrated functional logic programming language Curry can be employed to solve problems on relational structures (like orders, graphs, and Petri nets) in a very high-level declarative style. The functional features of Curry are used to implement relation algebra and the logic features of the language are combined with BDD-based solving of boolean constraints to obtain a fairly efficient implementation of a solver for relational specifications.

Keywords: Functional programming, constraint solving, Curry, relation algebra, relational specifications

1 Introduction

For many years, relation algebra has widely been used by mathematicians and computer scientists as a convenient means for problem solving. Its use in Computer Science is mainly due to the fact that many datatypes and structures (like graphs, hyper-graphs, orders, lattices, Petri nets, and data bases) can be modeled via relations, problems on them can be specified naturally by relation-algebraic expressions and formulae, and problem solutions can benefit from relation-algebraic reasoning and computations. A lot of examples and references to relevant literature can be found, e.g., in [17,4,6,13].

In fortunate cases a relational specification is executable as it stands, i.e., is an expression that describes an algorithm for computing the specified object. Then we have the typical situation where a tool like RelView [2] for mechanizing relational algebra is directly applicable. But in large part relational specifications are non-algorithmic as they implicitly specify the object to be computed by a set of properties. Here the standard approach is to intertwine a certain program development method with relation-algebraic calculations to obtain an algorithm (a

[1] Email: {rub,sebf}@informatik.uni-kiel.de

relational program) that implements a relational specification (see [15,1] for some typical examples). Rapid prototyping at the specification level is not applied. As a consequence, specification errors usually are discovered at later stages of the program development.

To avoid this drawback, we propose an alternative approach to deal with relational specifications (and programs). We formulate them by means of the functional logic programming language Curry [8]. Concretely, this means that we use the operational features of this language for implementing relation algebra. On that score our approach is similar to [16,11], the only implementations of relation algebra in a functional language we are aware of. But, exceeding [16,11], we use the logical problem-solving capabilities of Curry for formulating relational specifications and nondeterministically searching for solutions. As we will demonstrate, this allows to prototype a lot of implicit relational specifications. To enhance efficiency, we employ a boolean constraint solver that is integrated into the Curry language. The integration of constraint solving over finite domains and real numbers into functional logic languages has been explored in [12,7]. We are not aware of other approaches that integrate boolean constraints into a functional logic language or combine them with relation algebra to express constraints over relations. The implementation of relation algebra in Curry enables to formulate relational programs within this language. In respect thereof, we even can do more than RelView since Curry is a general purpose language in contrast to the rather restricted language of RelView.

The remainder of this paper is organized as follows. Sections 2 and 3 provide some preliminaries concerning relation algebra and the programming language Curry. In Section 4 we show how the functional features of Curry can be employed for elegantly implementing the constants and operations of relation algebra and, based on this, the logical features of the language can be employed for directly expressing relational problem specifications. Some examples for our approach are presented in Section 5, where we also report on results of practical experiments. Section 6 contains concluding remarks.

2 Relation-algebraic Preliminaries

In the following, we first introduce the basics of relation algebra. Then we show how specific relations, viz. vectors and points, can be used to model sets. For more details concerning relations, see, e.g., [17,4].

2.1 Relation Algebra

We write $R : X \leftrightarrow Y$ if R is a relation with domain X and range Y, i.e., a subset of $X \times Y$. If the sets X and Y of R's *type* $X \leftrightarrow Y$ are finite and of size m and n, respectively, we may consider R as a boolean $m \times n$ matrix. Since a boolean matrix interpretation is well suited for many purposes, in the following we often use matrix terminology and notation. Especially we speak about rows and columns and write $R_{x,y}$ instead of $\langle x, y\rangle \in R$ or $x\,R\,y$. We assume the reader to be familiar with the basic operations on relations, viz. R^T (*inversion*, *transposition*), $\overline{R}$ (*complement*, *negation*), $R \cup S$ (*union*, *join*), $R \cap S$ (*intersection*, *meet*), and RS (*composition*, *multiplication*, denoted by juxtaposition), the predicate $R \subseteq S$ (*inclusion*), and the special relations O (*empty relation*), L (*universal relation*), and I (*identity relation*).

2.2 Modeling of Sets

There are some relation-algebraic possibilities to model sets. In this paper we will use *(row) vectors*, which are relations v with $v = \mathsf{L}v$. Since for a vector the domain is irrelevant, we consider in the following mostly vectors $v : \mathbf{1} \leftrightarrow X$ with a specific singleton set $\mathbf{1} := \{\perp\}$ as domain and omit in such cases the first subscript, i.e., write v_x instead of $v_{\perp,x}$. Such a vector can be considered as a boolean matrix with exactly one row, i.e., as a boolean row vector or a (linear) list of truth values, and *represents* the subset $\{x \in X \mid v_x\}$ of X.

A non-empty vector v is said to be a *point* if $v^{\mathsf{T}}v \subseteq \mathsf{I}$, i.e., v is a non-empty *functional* relation. This means that it represents a singleton subset of its domain or an element from it if we identify a singleton set with the only element it contains. Hence, in the boolean matrix model a point $v : \mathbf{1} \leftrightarrow X$ is a boolean row vector in which exactly one component is true.

3 Functional Logic Programming with Curry

The functional logic programming language Curry [8,10] aims at integrating different declarative programming paradigms into a single programming language. It can be seen as a syntactic extension of Haskell [14] with partial data structures and a different evaluation strategy. The operational semantics of Curry is based on lazy evaluation combined with a possible instantiation of free variables. On ground terms the operational model is similar to lazy functional programming, while free variables are nondeterministically instantiated like in logic languages. Nested expressions are evaluated lazily, i.e., the leftmost outermost function call is selected for reduction in a computation step. If in a reduction step an argument value is a free variable and demanded by an argument position of the left-hand side of some rule, it is either instantiated to the demanded values nondeterministically or the function call suspends until the argument is bound by another concurrent computation. Binding free variables is called *narrowing*; suspending calls on free variables is called *residuation*. Curry supports both strategies because which of them is right depends on the intended meaning of the called function.

3.1 Datatypes and Function Declarations

Curry supports algebraic datatypes that can be defined by the keyword `data` followed by a list of constructor declarations separated by the symbol "`|`". For example, the following two declarations introduce the predefined datatypes for boolean values and polymorphic lists, respectively, where the latter usually is written as `[a]`:

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

Later we will also use the following two functions on lists:

```
any   :: (a -> Bool) -> [a] -> Bool
null  :: [a] -> Bool
```

The first function returns `True` if its second argument contains an element that satisfies the given predicate and the second function checks whether the given list is empty.

Type synonyms can be declared with the keyword `type`. For example, the following definition introduces matrices with entries from a type `a` as lists of lists:

```
type Matrix a = [[a]]
```

Curry functions can be written in prefix or infix notation and are defined by rules that are evaluated nondeterministically. The four declarations

```
not :: Bool -> Bool
not True  = False
not False = True

(&&), (||) :: Bool -> Bool -> Bool
True  && b = b
False && _ = False

True  || _ = True
False || b = b

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

introduce three well-known boolean combinators (conjunction and disjunction as infix operations) and list concatenation (also as infix operation).

3.2 Nondeterministic Search

The logic features of Curry can be employed to nondeterministically search for solutions of constraints. Constraints are represented in Curry as values of the specific type `Success`. The always satisfied constraint is denoted by `success` and two constraints can be combined into a new one with the following concurrent conjunction operator:

```
(&) :: Success -> Success -> Success
```

To constrain a boolean expression to be satisfied one can use the following function that maps the boolean constant `True` to `success` and is undefined for the boolean constant `False`:

```
satisfied :: Bool -> Success
satisfied True = success
```

Based on this function, the following function takes a predicate and nondeterministically computes a solution for the predicate using narrowing:

```
find :: (a -> Bool) -> a
find p | satisfied (p x) = x where x free
```

Here the part of the rule between the two symbols "`|`" and "`=`" is called a guard and must be satisfied to apply the rule. Furthermore, the local declaration `where x free` declares `x` to be an unknown value. Based on this, the function `find` can be used to solve boolean formulae. For example, given the definition

```
one :: (Bool,Bool) -> Bool
one (x,y) = x && not y || not x && y
```

for the boolean formula $(x \wedge \neg y) \vee (\neg x \wedge y)$, the call `(find one)` evaluates to `(True,False)` or `(False,True)`. This means that the formula holds iff x is assigned to the truth value *true* and y is assigned to the truth value *false* or x is assigned to *false* and y is assigned to *true*.

3.3 Boolean Constraint Solving

Boolean formulae can be solved more efficiently using binary decision diagrams [5]. Therefore, the PAKCS [9] implementation of Curry contains a specific library CLPB that provides **C**onstraint **L**ogic **P**rogramming over **B**ooleans based on BDDs. In this library, boolean constraints are represented as values of type `Boolean`. There are two constants, viz. the always satisfied constraint and the never satisfied constraint:

```
true  :: Boolean
false :: Boolean
```

Besides these constants, the library CLPB exports a lot of functions on boolean constraints. For example, there are the following nine functions corresponding to the boolean lattice structure of `Boolean`, where the meaning of the function `neg` and the operations `(.&&)`, `(.||)`, `(.==)`, and `(./=)` is obvious and the remaining operations denote the comparison relations on `Boolean` with the constant `false` being defined strictly smaller than the constant `true`:

```
neg :: Boolean -> Boolean

(.&&), (.||), (.==), (./=), (.<), (.<=), (.>), (.>=)
  :: Boolean -> Boolean -> Boolean
```

Decisive for the applications we will discuss later is the CLPB-function

```
satisfied :: Boolean -> Success
```

that nondeterministically yields a solution if the argument is a satisfiable boolean constraint by possibly instantiating free variables in the constraint. This function is far more efficient than the narrowing-based version presented in Section 3.2.

4 Implementation of Relation Algebra

In this section we sketch an implementation of relation algebra over finite, binary, relations in the Curry language. We will represent relations as boolean matrices. This allows to employ the higher-order features of Curry for an elegant formulation of the relation-algebraic operations, predicates, and constants. As we will also demonstrate, relational constraints can be integrated seamlessly into Curry because we can use free variables to represent unknown parts of a relation. Based on this, the nondeterministic features of Curry permit us to formulate the search for unknown relations that satisfy a given predicate in a natural way.

4.1 Functional Combinators

The functional features of Curry serve well to implement relation algebra. Relations can be easily modeled as algebraic datatype and relational operations, predicates, and constants can be defined as Curry-functions over this datatype. We only consider relations with finite domain and range. As already mentioned in Section 2.1, such a relation can be represented as boolean matrix. Guided by the type of matrices introduced in Section 3.1, we define the following type for relations:

```
type Rel = [[Boolean]]
```

Note that we use the type `Boolean` instead of `Bool` for the matrix elements. This allows to apply the more efficient constraint solver `satisfied` of Section 3.3 for solving relational problems. Furthermore, note that in our implementation a vector corresponds to a list which contains exactly one list.

The dimension (i.e., the number of rows and columns, respectively) of a relation can be computed using the function

```
dim :: Rel -> (Int,Int)
```

and an empty relation, universal relation, and identity relation of a certain dimension can be specified by the functions

```
O, L  :: (Int,Int) -> Rel
I     :: Int -> Rel.
```

The definitions of these three functions are straightforward and, therefore, omitted. Next, we consider the inverse of a relation. In the matrix model of relation algebra it can be computed by transposing the corresponding matrix, and in Curry this transposition looks as follows:

```
inv :: Rel -> Rel
inv xs | any null xs  = []
       | otherwise    = map head xs : inv (map tail xs)
```

Here the predefined function `map :: (a -> b) -> [a] -> [b]` maps a function over the elements of a list and the predefined functions `head` and `tail` compute the head and the tail of a non-empty list, respectively.

The complement of a relation can be easily computed by negating the matrix entries. In Curry this can be expressed by the following declaration, where the function `map` has to be used twice since we have to map over a matrix which is a list of lists:

```
comp :: Rel -> Rel
comp = map (map neg)
```

Union and intersection are implemented using boolean functions that combine the matrices element-wise using disjunction and conjunction, respectively:

```
(.|.), (.&.) :: Rel -> Rel -> Rel
(.|.) = elemWise (.||)
(.&.) = elemWise (.&&)
```

Here the function `elemWise` is defined as

```
elemWise :: (a -> b -> c) -> [[a]] -> [[b]] -> [[c]]
elemWise = zipWith . zipWith
```

using the predefined functions

```
(.)     :: (b -> c) -> (a -> b) -> (a -> c)
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
```

for function composition and list combination, respectively. Finally, relational composition can be implemented as multiplication of boolean matrices. A corresponding Curry function looks as follows:

```
(.*.) :: Rel -> Rel -> Rel
xs .*. ys =  [  [ foldr1 (.||) (zipWith (.&&) xrow ycol)
                | ycol <- inv ys ]
             | xrow <- xs ]
```

Here we employ list comprehensions to enhance readability. Furthermore, we use the predefined function `foldr1 :: (a -> a -> a) -> [a] -> a` that combines all elements of the given list (second argument) with the specified binary operator (first argument).

4.2 Relational Constraints

In Section 2.1 we introduced one more basic combinator on relations, namely relational inclusion. It differs from the other constructions because it does not compute a new relation but yields a truth value. For the applications we have in mind, we understand relational inclusion as a boolean constraint over relations, i.e., a function that takes two relations of the same dimension and yields a value of type `Boolean`. Its Curry implementation is rather straightforward by combining the already used functions `foldr1` and `elemWise` with the predefined function `concat :: [[a]] -> [a]` that concatenates a list of lists into a single list.

```
(.<=.) :: Rel -> Rel -> Boolean
xs .<=. ys = foldr1 (.&&) (concat (elemWise (.<=) xs ys))
```

Replacing the operation `(.<=)` (the implication on constraints) in this declaration by the equivalence test `(.==)` on constraints immediately leads to the following function for testing equality of relations:

```
(.==.) :: Rel -> Rel -> Boolean
xs .==. ys = foldr1 (.&&) (concat (elemWise (.==) xs ys))
```

This definition is slightly more efficient than using `(.<=.)` twice to express equality. Hence, we prefer it over the following simpler definition:

```
(.==.) :: Rel -> Rel -> Boolean
xs .==. ys = xs .<=. ys .&& ys .<=. xs
```

To automatically solve relational constraints we, finally, provide the following function that relies on the function `satisfied` provided by the constraint solver and a function `freeRel` that computes an unknown relation with the specified dimensions:

```
find :: (Int,Int) -> (Rel -> Boolean) -> Rel
find d p | satisfied (p rel) = rel where rel = freeRel d

freeRel :: (Int,Int) -> Rel
freeRel (m,n) = map (map freeVar) (replicate m (replicate n ()))
 where freeVar () = let x free in x
```

The predefined function `replicate :: Int -> a -> [a]` computes a list of given length that contains only the specified element. The function `find` is the key to many solutions of relational problems using Curry since it takes a predicate over a relation and nondeterministically computes solutions for the predicate. A generalization of `find` to predicates over more than one relation is obvious, but for the problems we will consider in this paper this simple version suffices.

5 Applications and Results

Now, we present some example applications. We also report on the results of our practical experiments with the PAKCS implementation of Curry on a PowerPC G4 processor running at 1.33 GHz with 768 MB DDR SDRAM main memory. Unfortunately, the current PAKCS system is not able to use more than 256 MB of main memory which turned out to be a limitation for some examples.

5.1 Least Elements

In the following first example we present an application of our library that does not rely on the logic features of Curry. We implement the relational specification of a least element of a set with regard to an ordering relation. This specification is not given as a predicate but as a relation-algebraic expression.

Let $R : X \leftrightarrow X$ be an ordering relation on the set X and $v : \mathbf{1} \leftrightarrow X$ be a vector that represents a subset V of X. Then the vector $v \cap \overline{v\,\overline{R^\mathsf{T}}} : \mathbf{1} \leftrightarrow X$ is either empty or a point. In the latter case it represents the least element of V with regard to R since the equivalence

$$\begin{aligned}
(v \cap \overline{v\,\overline{R^\mathsf{T}}}\,)_x &\iff v_x \land \neg\exists\, y : v_y \land \overline{R^\mathsf{T}}_{y,x} \\
&\iff v_x \land \forall\, y : v_y \rightarrow R_{x,y} \\
&\iff x \in V \land \forall\, y : y \in V \rightarrow R_{x,y}
\end{aligned}$$

holds for all $x \in X$. Based on the relational specification $v \cap \overline{v\,\overline{R^{\mathsf{T}}}}$, in Curry the least element of a set/vector `v` with regard to an ordering relation `R` can be computed by the following function:

```
leastElement :: Rel -> Rel -> Rel
leastElement R v = v .&. comp (v .*. comp (inv R))
```

The body of this function is a direct translation of the relational specification into the syntax of Curry.

5.2 Linear Extensions

As an example for a relational specification given as a predicate, we consider linear extensions of an ordering relation. A relation $R : X \leftrightarrow X$ is an ordering relation on X if it is reflexive, transitive, and antisymmetric. It is well-known how to express these properties relation-algebraically. Reflexivity is described by $\mathsf{I} \subseteq R$, transitivity by $RR \subseteq R$, and antisymmetry by $R \cap R^{\mathsf{T}} \subseteq \mathsf{I}$. Hence, we immediately obtain the Curry-predicate

```
ordering :: Rel -> Boolean
ordering R = refl R .&& trans R .&& antisym R
 where
  refl    r = I (fst (dim r)) .<=. r
  trans   r = r .*. r .<=. r
  antisym r = r .&. inv r .<=. I (fst (dim r))
```

for testing relations to be ordering relations, where the predefined function `fst` selects the first element of a pair.

A linear extension of an ordering relation $R : X \leftrightarrow X$ is an ordering relation $R' : X \leftrightarrow X$ that includes R and is linear, i.e., $R'_{x,y}$ or $R'_{y,x}$ holds for all $x, y \in X$. The latter means that $R \cup R^{\mathsf{T}} = \mathsf{L}$. Hence, in Curry a linear extension `R'` of `R` can be specified by the following predicate:

```
linext :: Rel -> Rel -> Boolean
linext R R' = R .<=. R' .&& ordering R' .&& linear R'
 where
  linear r = r .|. inv r .==. L (dim r)
```

To compute a linear extension of an ordering relation, we directly can employ the function `find` introduced in Section 4.2. The result is the following function that takes an ordering relation as argument and nondeterministically returns a linear extension of the given relation.

```
linearExtension :: Rel -> Rel
linearExtension R = find (dim R) (linext R)
```

Using encapsulated search [3], the function `linearExtension` can be employed to enumerate all linear extensions of an ordering relation. We do not need to specify sets of linear extensions relation algebraically to compute them. The nondeterministic features of Curry permit us to use a simple specification for *one* linear extension

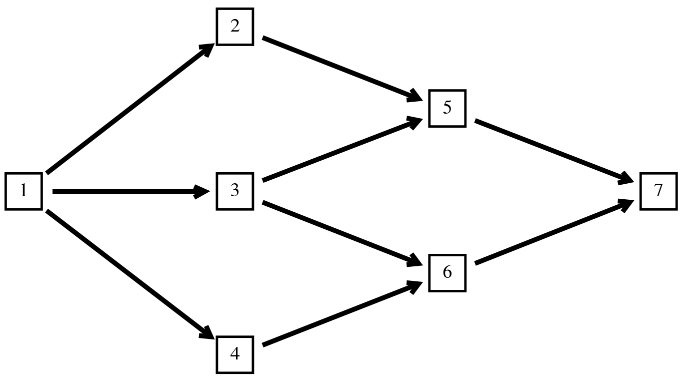

Fig. 1. Dependency structure of a set of tasks

to compute *all* of them. Enumerating linear extensions is of great interest to computer scientists because of its relationship to sorting and scheduling problems. For example, the NP-hard problem of computing a possible scheduling for a distributed system with dependencies given as ordering relation R obviously can be solved by computing all linear extensions of R and picking a best linear extension.

Since the relational constraint solving facilities are implemented as a Curry library, they can be used within functional logic programs to solve specific subproblems that can be elegantly expressed using relation algebra. Especially, one can combine solving of relational constraints and encapsulated search with a functional program that computes optimal schedulings. Similar to an example presented in [12], we consider a set of tasks with dependencies depicted as a directed graph in Figure 1. Based on `linearExtension`, a function that computes all linear extensions of an ordering relation `R` can be easily defined as follows:

```
allLinearExtensions :: Rel -> [Rel]
allLinearExtensions R = findall (=:=linearExtension R)
```

Here `(=:=) :: a -> a -> Success` is the built-in constraint equality and partially applied to get a predicate on relations and `findall :: (a -> Success) -> [a]` encapsulates the search and returns a list of all values that satisfy the given predicate. We use it despite its deficiencies discussed in [3] because it suffices for our purposes.

If the given ordering relation represents the dependencies of tasks in a distributed system, each linear extension represents a possible scheduling expressed as relation of type `Rel`. To rate a scheduling with regard to some quality factor, it would be more convenient to represent it as an ordered list of tasks. The conversion is accomplished by the following function that relies on the predefined function `sortBy :: (a -> a -> Bool) -> [a] -> [a]` that sorts a list according to an ordering predicate. The function `evaluate :: Boolean -> Bool` converts between boolean constraints and values of type `Bool` and `(xs !! n)` selects the `n`-th element of the list `xs`.

```
linearOrderToList :: Rel -> [Int]
linearOrderToList R = sortBy leq (take (length R) [1..])
 where leq x y = evaluate (R !! (x-1) !! (y-1))
```

narrowing

sec	$k = 3$	$k = 4$	$k = 5$	$k = 6$	$k = 7$
$n = 2$	0.04	0.1	0.15	0.37	0.96
$n = 3$	0.1	0.19	0.49	1.36	3.59
$n = 4$	0.22	0.39	1.19	3.32	8.85

constraint solving

sec	$k = 5$	$k = 10$	$k = 20$	$k = 30$	$k = 40$
$n = 2$	0.05	0.09	0.3	0.64	1.16
$n = 3$	0.07	0.12	0.48	1.1	1.96
$n = 4$	0.13	0.22	0.87	1.97	3.62

Fig. 2. Narrowing vs. constraint solving

For the example depicted in Figure 1 there are 16 possible schedulings. We can rate a schedule by the time tasks have to wait for others by accumulating for each task the run times of all tasks that are scheduled before and computing the sum of all these delays. If we assign a run time of 2 time units to tasks 2 and 4 and a run time of 1 time unit to all other tasks, the list `[1,3,2,5,4,6,7]` represents an optimal scheduling.

5.3 *Maximal Cliques*

As another example for a relational specification given as predicate, we consider specific sets of nodes of a graph. The adjacency matrix of a graph g is a relation $R : X \leftrightarrow X$ on the set X of nodes of g. For our example we restrict us to undirected graphs, i.e., we assume the relation R to be irreflexive ($R \subseteq \overline{\mathsf{I}}$) and symmetric ($R = R^{\mathsf{T}}$). A subset C of X is called a clique, if for all $x, y \in C$ from $x \neq y$ it follows $R_{x,y}$. If $x \neq y$ and $R_{x,y}$ are even equivalent for all $x, y \in C$, then C is a maximal clique.

Similar to the simple calculation in Section 5.1 we can show that a vector $v : X \leftrightarrow X$ represents a maximal clique of the undirected graph g iff $v\,\overline{R \cup \mathsf{I}} = \overline{v}$. Hence, a maximal clique of an undirected graph can be specified in Curry as follows, which exactly reflects the relational specification:

```
maxclique :: Rel -> Rel -> Boolean
maxclique R v = v .*. comp (R .|. I (fst (dim R))) .==. comp v
```

As a consequence, a single maximal clique or even the set of all maximal cliques of an undirected graph can be computed via

```
maximalClique :: Rel -> Rel
maximalClique R = find (1, fst (dim R)) (maxclique R)
```

using the function `find` similar to Section 5.2.

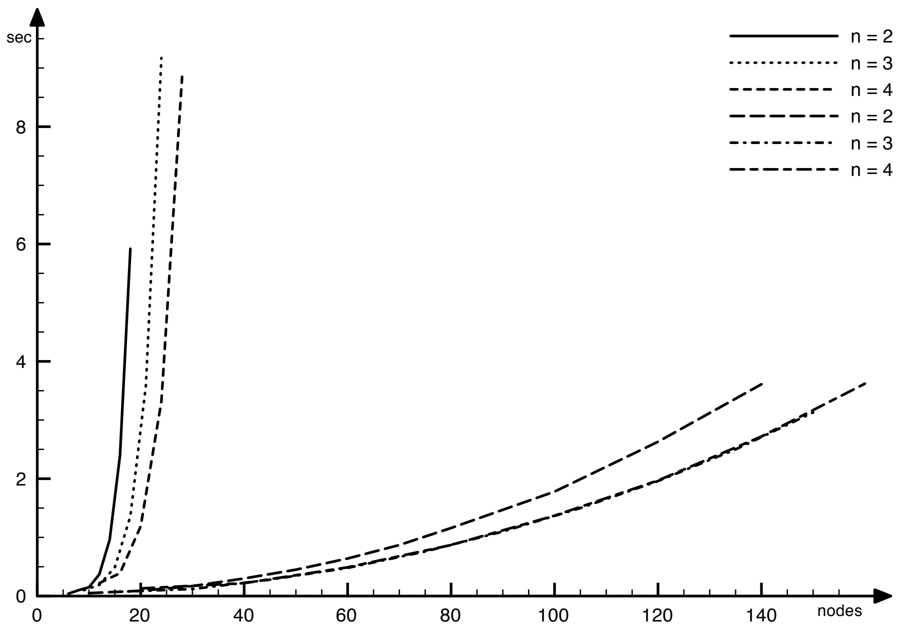

Fig. 3. Computing maximal cliques

5.4 Discussion

Of course, with regard to efficiency, our approach to execute relational specifications cannot compete with specific algorithms for the problems we have considered. It should be pointed out that our intention is not to support the implementation of highly efficient algorithms. We rather strive for automatic evaluation of relational specifications with *minimal programming effort* and reasonable performance for small problem instances. Therefore, we compared our approach to a narrowing-based implementation, that does not rely on a constraint solver but uses the hand-coded function `satisfied` introduced in Section 3.2. The results of our experiments show that using a constraint solver significantly increases the performance while it preserves the declarative formulation of programs.

To compare the constraint-based implementation with the narrowing-based one we especially used the last example and computed maximal cliques in undirected graphs of different size. Problem instances that are published in the Web and generally used to benchmark specific algorithms for computing cliques could not be solved by our approach with reasonable effort. Therefore, for our benchmarks we generated our own problem instances as the disjoint union of n complete (loopless) graphs with k nodes each, and searched for all maximal cliques in these specific graphs. The run times of our benchmarks for different values of k and n are given in the two tables depicted in Figure 2. To check correctness was easy since in each case a maximal clique consists of the set of k nodes of a copy of the complete graphs we started with and there are exactly n maximal cliques.

The run time of the constraint-based implementation increases moderately compared to the narrowing based implementation. To visualize this difference more clearly, the results of the tables are depicted graphically in Figure 3. We could not compute cliques in larger graphs because the constraint solver turned out to be very memory consuming and fails with a resource error for larger problem instances. The instances that can be solved are solved reasonably fast – the maximal cliques of a graph with 160 nodes are computed in less than 4 seconds. Note that, conceptually, the huge number $2^{160} = 1461501637330902918203684832716283019655932542976$ of sets of nodes has to be checked in a graph with 160 nodes.

6 Conclusion

In this paper we have demonstrated how the functional logic programming language Curry can be used to implement relation algebra and to prototype relational specifications. We have used the functional features of Curry for elegantly implementing relations and the most important operations on them. Then the execution of explicit specifications corresponds to the evaluation of expressions. For the execution of implicit specifications we employed a boolean constraint solver available in the PAKCS system which proved to be head and shoulders above a narrowing-based approach. Without presenting an example, it should be clear that our approach also allows the formulation of general relational algorithms (like the computation of the transitive closure R^+ of R as limit of the chain $\mathsf{O} \subseteq f_R(\mathsf{O}) \subseteq f_R(f_R(\mathsf{O})) \subseteq \ldots$, where $f_R(X) = R \cup XX$) as Curry-programs.

By implementing a solver for relational specifications using Curry, we described an application of the integration of different programming paradigms. The involved paradigms are the following:

- Relation algebra – to formulate specifications.
- Functional programming – to express relations as algebraic datatype and relational combinators as functions over this datatype.
- Constraint solving – to efficiently solve constraints over relations.
- Free variables and built-in nondeterminism – to express unknown relations and different instantiations in a natural way.

Using our library, relational specifications can be checked in a high-level declarative style with minimal programming effort. We have demonstrated that different programming paradigms can benefit from each other. Functional programming can be made more efficient using constraint solving facilities and constraint programming can be made more readable by abstraction mechanisms provided by functional programming languages. Especially, higher-order functions and algebraic datatypes serve well to implement constraint generation on a high level of abstraction. Functional logic languages allow for a seamless integration of functional constraint generation and possibly nondeterministic constraint solving with instantiation of unknown values.

Since the underlying constraint solver uses BDDs to represent boolean formulae, constraints over relations are also represented as BDDs. Unlike RelView we do not represent relations as BDDs but use a matrix representation. For future work we plan to investigate, whether the ideas behind the BDD representation of relations employed in RelView can be combined with the BDD representation of relational constrains. Such a combination could result in a more efficient implementation for two reasons: Firstly, applications that use our library to implement relational *al-*

gorithms where many relational expressions need to be evaluated benefit because operations on relations can be implemented more efficiently on BDDs than on matrices. Secondly, even in applications were a *specification* only has to be evaluated once before it is instantiated by the constraint solver, we can benefit if the BDD-based representation of relations uses less memory than the matrix representation. As another topic for future work, we plan to consider a slightly different interface to our library that hides the dimensions of relations. Specifying dimensions of relations is tedious and error prone. They could be handled explicitly in the datatype for relations and propagated by the different relational combinators. The challenge will be to ensure correct guessing of unknown relations without extra specifications by the programmer.

References

[1] Berghammer, R. and T. Hoffmann, *Relational depth-first-search with applications*, Information Sciences **139** (2001), pp. 167–186.

[2] Berghammer, R. and F. Neumann, RelView *– An OBBD-based Computer Algebra system for relations*, in: *Proc. Int. Workshop on Computer Algebra in Scientific Computing* (2005), pp. 40–51.

[3] Braßel, B., M. Hanus and F. Huch, *Encapsulating non-determinism in functional logic computations*, Journal of Functional and Logic Programming **2004** (2004).

[4] Brink, C., W. Kahl and G. Schmidt, editors, "Relational Methods in Computer Science," Advances in Computing, Springer, 1997.

[5] Bryant, R., *Graph-based algorithms for boolean function manipulation*, IEEE Transactions on Computers **C35** (1986), pp. 677–691.

[6] de Swart, H., E. Orlowska, G. Schmidt and M. Roubens, editors, "Theory and Applications of Relational Structures as Knowledge Instruments," Lecture Notes in Computer Science **2929**, Springer, 2003.

[7] Fernández, A., M. Hortalá-González and F. Sáenz-Pérez, *Solving combinatorial problems with a constraint functional logic language*, in: *Proc. 5th Int. Symposium on Practical Aspects of Declarative Languages (PADL 2003)* (2003), pp. 320–338.

[8] Hanus, M., *The integration of functions into logic programming: From theory to practice*, Journal of Logic Programming **19&20** (1994), pp. 583–628.

[9] Hanus, M. et al., *PAKCS: The Portland Aachen Kiel Curry System (version 1.7.1)*, Available at URL `http://www.informatik.uni-kiel.de/~pakcs/` (2003).

[10] Hanus, M. et al., *Curry: An integrated functional logic language (version 0.8.2)*, Available at URL `http://www.informatik.uni-kiel.de/~curry` (2006).

[11] Kahl, W., *Semigroupoid interfaces for relation-algebraic programming in Haskell*, in: R. A. Schmidt, editor, *Relations and Kleene Algebra in Computer Science*, Lecture Notes in Computer Science **4136**, 2006, pp. 235–250.

[12] Lux, W., *Adding linear constraints over real numbers to Curry*, in: *Proc. 5th Int. Symposium on Functional and Logic Programming (FLOPS 2001)* (2001), pp. 185–200.

[13] MacCaull, W., M. Winter and I. Düntsch, editors, "Proc. Int. Seminar on Relational Methods in Computer Science," Lecture Notes in Computer Science **3929**, Springer, 2006.

[14] Peyton Jones, S., editor, "Haskell 98 Language and Libraries—The Revised Report," Cambridge University Press, 2003.

[15] Ravelo, J., *Two graph-algorithms derived*, Acta Informatica **36** (1999), pp. 489–510.

[16] Schmidt, G., *A proposal for a multilevel relational reference language*, Journal of Relational Methods in Computer Science **1** (2004), pp. 314–338.

[17] Schmidt, G. and T. Ströhlein, "Relations and Graphs – Discrete Mathematics for Computer Scientists," Springer, 1993.