INSTITUT FÜR INFORMATIK

Improving Lazy Non-Deterministic Computations by Demand Analysis

Michael Hanus

Bericht Nr. 1209
Juni 2012
ISSN 2192-6247

CHRISTIAN-ALBRECHTS-UNIVERSITÄT

ZU KIEL

Institut für Informatik der
Christian-Albrechts-Universität zu Kiel
Olshausenstr. 40
D – 24098 Kiel

Improving Lazy Non-Deterministic Computations by Demand Analysis

Michael Hanus

Bericht Nr. 1209
Juni 2012
ISSN 2192-6247

e-mail: mh@informatik.uni-kiel.de

Improving Lazy Non-Deterministic Computations by Demand Analysis

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

Technical Report 1209, June 2012

Abstract

Functional logic languages combine lazy (demand-driven) evaluation strategies from functional programming with non-deterministic computations from logic programming. The lazy evaluation of non-deterministic subexpressions results in a demand-driven exploration of the search space: if the value of some subexpression is not required, the complete search space connected to it is not explored. On the other hand, this improvement could cause efficiency problems if unevaluated subexpressions are duplicated and later evaluated in different parts of a program. In order to improve the execution behavior in such situations, we propose a program analysis that guides a program transformation to avoid such inefficiencies. We demonstrate the positive effects of this program transformation with KiCS2, a recent highly efficient implementation of the functional logic programming language Curry.

1 Motivation

Functional logic languages support the most important features of functional and logic programming in a single language (see [11, 33] for recent surveys). They provide higher-order functions and demand-driven evaluation from functional programming as well as logic programming features like non-deterministic search and computing with partial information (logic variables). This combination led to new design patterns [8, 12], better abstractions for application programming (e.g., programming with databases [19, 27], GUI programming [30], web programming [31, 32, 35], string parsing [23]), and new techniques to implement programming tools, like partial evaluators [3] or test case generators [28, 50].

The implementation of functional logic languages is challenging due to the combination of the various language features. For instance, one can

- design new abstract machines appropriately supporting these operational features and implementing them in some (typically, imperative) language, like C [43] or Java [13, 37],

- compile into logic languages like Prolog and reuse the existing backtracking implementation for non-deterministic search as well as logic variables and unification for computing with partial information [7, 41], or

- compile into non-strict functional languages like Haskell and reuse the implementation of lazy evaluation and higher-order functions [21, 22].

The latter approach requires the implementation of non-deterministic computations in a deterministic language but 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 [21].

In this paper we consider KiCS2 [20], a new system that compiles functional logic programs of the source language Curry [38] into purely functional Haskell programs. However, the techniques presented in this paper can also be applied to similar implementations, like KiCS [22] or ViaLOIS [14]. KiCS2 can compete with or outperform other existing implementations of Curry [20]. In particular, deterministic parts of a program are much faster executed than in Prolog-based Curry implementations. Non-determinism is implemented in KiCS2 by representing all non-deterministic results of a computation as a data structure. This structure is traversed by operations implementing the search for solutions. Thus, different search strategies are supported by KiCS2. This flexibility might cause efficiency problems in some situations due to the duplication of unevaluated subexpressions (see below for a more detailed explanation). Therefore, we propose a new technique to improve such problematic situations based on the following steps:

1. The run-time behavior of the program is analyzed. In particular, information about demanded arguments and the non-determinism behavior is approximated.

2. The information obtained from this analysis is used to transform the source program. In particular, the computation of a non-deterministic subexpression is enforced earlier when its value is definitely demanded.

In the next section, we review the source language Curry and the features considered in this paper. Section 3 sketches the basic implementation scheme of KiCS2. Section 4 discusses the potential problems of non-deterministic computations and presents a technique to analyze the demand information that is used in the program transformation presented in Section 5. Some practical results of this transformation are shown in Section 6 before we conclude in Section 7.

2 Functional Logic Programming and Curry

The declarative multi-paradigm language Curry [38] combines features from functional programming (demand-driven evaluation, parametric polymorphism, higher-order functions) and logic programming (computing with partial information, unification, con-

straints). The syntax of Curry is close to Haskell[1] [47]. In addition, Curry allows free (logic) variables in conditions and right-hand sides of defining rules. The operational semantics is based on an optimal evaluation strategy [6] which is a conservative extension of lazy functional programming and logic programming.

A Curry program consists of the definition of data types (introducing *constructors* for the data types) and *operations* on these types. A *value* is an expression without defined operations. Note that, in a functional logic language like Curry, not all definable operations are functions in the classical mathematical sense. There are also operations, sometimes called "non-deterministic functions" [29], which might yield more than one result on the same input. Nevertheless, a Curry program has a purely declarative semantics where non-deterministic operations are modeled as set-valued functions (to be more precise, down-closed partially ordered sets are used as target domains in order to cover non-strictness, see [29] for a detailed account of this model-theoretic semantics).

For instance, Curry contains a *choice* operation defined by:

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

Thus, the expression "`0 ? 1`" has two values: `0` and `1`. If expressions have more than one value, one wants to select intended values according to some constraints, typically in conditions of program rules. A *rule* has the form "$f\ t_1 \ldots t_n$ `|` c `=` e" where the (optional) condition c is a *constraint*, i.e., an expression of the built-in type `Success`. For instance, the trivial constraint `success` is a value of type `Success` that denotes the always satisfiable constraint. Thus, we say that a constraint c is *satisfied* if it can be evaluated to `success`. An *equational constraint* e_1 `=:=` e_2 is satisfiable if both sides e_1 and e_2 are reducible to unifiable values. Furthermore, if c_1 and c_2 are constraints, c_1 `&` c_2 denotes their concurrent conjunction (i.e., both argument constraints are concurrently evaluated).

As a simple example, consider the following Curry program which defines a data type for Boolean values, a polymorphic data type for lists, and operations to compute the concatenation of lists and the last element of a list:[2]

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

-- "++" is a right-associative infix operator
(++) :: [a] → [a] → [a]
[]     ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)

last :: [a] → a
last xs | (ys ++ [z]) =:= xs
        = z                    where ys,z free
```

[1] Variables and function names 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 ("$f\ e$").

[2] Note that lists are a built-in data type with a more convenient syntax, e.g., one can write `[x,y,z]` instead of `x:y:z:[]` and `[a]` instead of the list type "`List a`".

Logic programming is supported by admitting function calls with free variables (e.g., `(ys++[z])` in the rule defining `last`) and constraints in the condition of a defining rule. In contrast to Prolog, free variables need to be declared explicitly to make their scopes clear (e.g., "`where ys,z free`" in the example). A conditional rule is applicable if its condition is satisfiable. Thus, the rule defining `last` states in its condition that `z` is the last element of a given list `xs` if there exists a list `ys` such that the concatenation of `ys` and the one-element list `[z]` is equal to the given list `xs`.

As mentioned above, operations can be non-deterministic:

```
aBool = True ? False
```

Using such non-deterministic operations as arguments might cause a semantical ambiguity which has to be fixed. Consider the operations

```
not True  = False
not False = True

xor True  x = not x
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 derivation

$$\texttt{xorSelf aBool} \to \texttt{xor aBool aBool} \to \texttt{xor True aBool} \to \texttt{xor True False} \to \texttt{not False} \to \texttt{True}$$

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, González-Moreno et al. [29] proposed the rewriting logic CRWL as a logical (execution- and strategy-independent) foundation for declarative programming with non-strict and non-deterministic operations. CRWL specifies the *call-time choice* semantics [40], where values of the arguments of an operation are determined before the operation is evaluated. This can be enforced in a lazy strategy 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

In order to provide a precise definition of the semantics of non-deterministic and non-strict operations, we assume a given program $\mathcal{P}$ and extend standard expressions so that they can also contain the special symbol $\bot$ to represent *undefined or unevaluated values*. A *partial value* is a value containing occurrences of $\bot$. A *partial constructor substitution* is a substitution that replaces variables by partial values. Then we denote by

$$[\mathcal{P}]_\bot = \{\sigma(l) = \sigma(r) \mid l = r \in \mathcal{P}, \sigma \text{ partial constructor substitution}\}$$

the set of all *partial constructor instances* of the program rules. A *context* $\mathcal{C}[\cdot]$ is an expression with some "hole". Then the reduction relation used in this paper is defined as

follows:[3]

$$\mathcal{C}[f\ t_1 \ldots t_n] \twoheadrightarrow \mathcal{C}[r] \quad \text{if } f\ t_1 \ldots t_n = r \in [\mathcal{P}]_\perp$$
$$\mathcal{C}[f\ e_1 \ldots e_n] \twoheadrightarrow \mathcal{C}[\perp] \quad \text{if } f \text{ is a defined operation}$$

The first rule models the call-time choice: if a rule is applied, the actual arguments of the operation must have been evaluated to partial values. The second rule models non-strictness where unevaluated operations are replaced by an undefined value (which is intended if the value of this subexpression is not demanded). As usual, $\overset{*}{\twoheadrightarrow}$ denotes the reflexive and transitive closure of this reduction relation. A partial value t is called a *normal form* of e if $e \overset{*}{\twoheadrightarrow} t$. Note that the derivation for "`xorSelf aBool`" shown above is not possible w.r.t. $\twoheadrightarrow$. The equivalence of this rewrite relation and CRWL is shown in [42, 34].

Although the combination of non-deterministic operations and lazy evaluation requires more efforts on the implementation side (see below), it might lead to search space reductions as the following example shows. An operation that non-deterministically inserts an element at an arbitrary position into a list can be defined by

```
insert x []     = [x]
insert x (y:ys) = (x:y:ys) ? (y:insert x ys)
```

Based on this operation, we can define a permutation of a list by

```
perm []     = []
perm (x:xs) = insert x (perm xs)
```

Thus, we can define list sorting by selecting a sorted permutation by ("`<=:`" denotes the less-than-or-equal-to constraint):

```
psort xs = checkSorted (perm xs)
checkSorted ys | sorted ys = ys

sorted []       = success
sorted [_]      = success
sorted (x:y:ys) = x<=:y & sorted (y:ys)
```

Although this definition seems quite similar to the classical generate-and-test solution where *all* permutations are enumerated and tested, it has a much more efficient behavior in Curry. Since the non-deterministic expression `(perm xs)` is evaluated by the demand of the operation `sorted`, many permutations are not generated. For instance, if `perm xs` is evaluated to `9:8:perm`..., the constraint `9<=:8` fails so that the computation of the remaining part of the permutation (which can result in $(n-2)!$ different permutations if n is the length of the list `xs`) is discarded. Thus, this "test-of-generate" definition has a substantially lower complexity than the traditional generate-and-test solution.

[3] Conditional rules are not considered in the reduction relation since they can be eliminated [4] by transforming each conditional rule "l `|` c `=` e" into "l `=` `cond` c e" where the operation `cond` is defined by "`cond success x = x`".

We do not discuss the implementation of free (logic) variables in the following. This is justified by the fact that logic variables, denoting arbitrary but unknown values, can be replaced by generators, i.e., operations that non-deterministically evaluate to all possible ground values of the type of the free variable. For instance, the operation `aBool` is a generator for Boolean values so that one can transform the expression "`not x`", where `x` is a free variable, into "`not aBool`". It has been shown [9, 26] that computing with logic variables by narrowing [48, 51] and computing with generators by rewriting are equivalent, i.e., compute the same values. Since such generators are standard non-deterministic operations, they are translated like any other operation.

We have seen that the demand-driven evaluation of non-deterministic arguments can have a positive impact on the overall complexity. However, there are also situations where the complexity might increase. Since this depends on the implementation, we discuss some recent implementation techniques in the following.

3 Compiling Non-Deterministic Programs

In this section, we sketch the implementation of non-deterministic programs in a purely functional language. This translation scheme is used by KiCS2 to compile Curry programs into Haskell programs. More details can be found in [17, 18, 20].

As mentioned in the introduction, we are interested in an implementation supporting different, in particular, complete search strategies. Thus, implementations based on a particular search strategy, like backtracking, which can also be found in approaches to support non-deterministic computations in functional programs [24, 39], are too limited. To provide various, also user-definable, search strategies, we explicitly represent all non-deterministic results of a computation 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 as defined above is translated into the Haskell data type[4]

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

In order to implement the call-time choice semantics discussed in Sect. 2, each `Choice` constructor has an additional argument. For instance, the evaluation of `xorSelf aBool` duplicates the argument operation `aBool`. Thus, 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`. In order to get unique identifiers on demand, we pass a (conceptually infinite) set of identifiers, also called *identifier supply*, to each operation.[5] Hence, each `Choice` created during run time can pick its unique identifier from this set. For this purpose, we assume

[4]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`.

[5]Note that the target program should be free of side effects in order to enable various search strategies, including parallel ones.

a type `IDSupply`, representing an infinite set of identifiers, with operations

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

`initSupply` creates such a set (at the beginning of an execution), `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 [15] so that KiCS2 is parametric over concrete implementations of `IDSupply`. A simple one can be based on unbounded integers, see [20].

Now, the correct handling of the call-time choice semantics can be obtained by adding an additional argument of type `IDSupply` to each operation. For instance, the operation `aBool` defined above is translated into:

```
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 `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.[6] For instance, the operation `xor` is extended by an identifier supply and further matching rules:

```
xor :: Bool  → Bool  → IDSupply  → Bool
xor True             x s = not x s
xor False            x s = x
xor (Choice i x1 x2) x s = Choice i (xor x1 x s) (xor x2 x s)
xor Fail             x s = Fail
```

The third rule transforms a non-deterministic argument into a non-deterministic result, i.e., a non-deterministic choice is moved one level up. This is also called a "pull-tab" step

[6] To obtain a simple compilation scheme, KiCS2 transforms source programs into uniform programs [20] where pattern matching is restricted to a single argument. This is always possible by introducing auxiliary operations.

[5]. The final rule returns `Fail` if the matching argument is already a failed computation (failure propagation).

In our concrete example, we assume that choice identifiers are implemented as integers [20]. Thus, if we evaluate the expression `(main 1)` w.r.t. the transformed rules defining `xor`, we obtain the result

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

Hence, the result is non-deterministic and contains three choices with identical identifiers. 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` cannot be extracted.

As one can see, the implementation is modularized in two phases that are interleaved by the lazy evaluation strategy of the target language: any expression is evaluated to a tree representation of all its values and the main user interface (responsible for printing all results) extracts the correct values from this tree structure. As a consequence, one can easily implement various search strategies to extract these values as different tree traversal strategies. Due to the overall lazy evaluation strategy, infinite search spaces does not cause a complication. For instance, if one is interested only in a single solution, one can extract some value even if the computed choice structure is conceptually infinite.

This implementation also provides many opportunities for optimizations. For instance, if operations are deterministic, i.e., cannot introduce `Choice` constructors, it is not necessary to pass an identifier supply to the operation. The benchmarks presented in [20] show that this implementation outperforms all other Curry implementations for deterministic operations, and, for non-deterministic operations, outperforms Prolog-based implementations of Curry and can compete with MCC [43], a Curry implementation that compiles to C.

4 Demand Analysis

The translation scheme presented in the previous section leads to an implementation with a good efficiency (e.g., KiCS2 is used for larger applications in the area of web programming). It is also used in a slightly modified form in another recent compact compiler for functional logic languages [14]. However, there are situations where this scheme cause efficiency problems. For instance, consider the evaluation of the expression `(main 1)` (for simplicity, we do not show the sharing of subexpressions done by the lazy evaluation strategy):

```
main 1 →* xorSelf (aBool 2) 3
```

```
      →* xor (aBool 2) (aBool 2) 3
      →* xor (Choice 2 True False) (Choice 2 True False) 3
      →* Choice 2 (Choice 2 False True) (Choice 2 True False)
```

As one can see, the (initially) single occurrence of the non-deterministic operation `aBool`, whose evaluation introduces a `Choice` constructor, is duplicated so that it results (in combination with the pull-tab step) in three `Choice` constructors. Since the overall strategy to extract values from choice structures has to traverse this choice structure, this might lead to an explosion of the search space in some cases (see benchmarks in Section 6).

A careful analysis shows that this problem stems from the lazy evaluation strategy. Hence, an improvement might be possible by changing the evaluation strategy. The operation `xorSelf` always demands the value of its argument in order to apply some reduction rule. Thus, one can also try to evaluate the argument *before* an attempt to evaluate `xorSelf`. Such a kind of call-by-value or strict evaluation can be achieved by introducing a *strict application* operation "`sApply`" implemented in the target code as follows:

```
sApply f (Choice i x1 x2) s = Choice i (sApply f x1 s) (sApply f x2 s)
sApply f Fail             s = Fail
sApply f x                s = f x s
```

Hence, `sApply` enforces the evaluation of the argument (to an expression without a defined operation at the top, also called *head normal form*) before the operation is applied. In particular, if the argument is a non-deterministic choice, it is moved outside the application. This operation is available as a predefined infix operation "`$!`" in Curry. Now consider what happens if we redefine `main` by

```
main = xorSelf $! aBool
```

and evaluate the translated `main` expression:

```
main 1 →* sApply xorSelf (aBool 2) 3
       →* sApply xorSelf (Choice 2 True False) 3
       →* Choice 2 (xorSelf True 3) (xorSelf False 3)
       →* Choice 2 (xor True True 3) (xor False False 3)
       →* Choice 2 False False
```

Hence, the computed choice structure does not contain duplicated `Choice` constructors, as desired. Similarly, one can also define a *normal form application* operation "`$!!`" which applies an operation to the completely evaluated argument. Thus, a call like "f `$!!` e" would completely evaluate e so that all choices made in the evaluation of e are moved outside the application of f to e. Although this seems reasonable, it destroys one advantage of combining lazy and non-deterministic computations. For instance, reconsider permutation sort as defined above. If we change the definition of `psort` to

```
psort xs = checkSorted $!! (perm xs)
```

the complete evaluation of the argument `(perm xs)` results in the enumeration of all permutations so that we obtain the complexity of the simple generate-and-test solution.

Even worse, we might loose completeness by an unrestricted use of "`$!`" or "`$!!`". For instance, consider the definition

```
ok x = True
loop = loop
```

Then "`ok loop`" has the value `True` but the evaluation of "`ok $! loop`" does not terminate.

As a consequence, we need some information about the demand of operations in order to insert strict applications only for demanded arguments. This seems quite similar to strictness information in purely functional programming [46]. However, the techniques developed there cannot be applied to functional logic programs. For instance, consider the operation `f` defined by

```
f 0 = 0
f x = 1
```

As a functional program, `f` is strict since the first rule demands its argument. As a functional logic program, `f` does not strictly demand its argument: due to the non-deterministic semantics, all rules can be used to compute a result so that we can apply the second rule to evaluate `(f loop)` to the value `1`.

These considerations show that we need a notion of demand specific for functional logic programs. Using the rewrite relation $\twoheadrightarrow$ introduced above, we say that a unary operation[7] f *demands* its argument if $\bot$ is the only normal form of $(f\ \bot)$. Thus, if a demanded argument is not reducible to some expression with a constructor at the root, the application is always undefined. This justifies the use of the strict application operation "`$!`" to demanded arguments.

Hence, we are left with the problem of detecting demanded arguments in a program. Since this property is undecidable in general, we can try to approximate it by some program analysis. Early work on analyzing the behavior of functional logic programs [36, 45, 53] tried to approximate narrowing derivations for confluent term rewriting systems so that it is not applicable in our more general framework of non-deterministic operations. A more appropriate analysis can be based on a fixpoint characterization of CRWL rewriting [1, 44]. An analysis to approximate call patterns w.r.t. CRWL rewriting has been presented in [34]. Since the undefined value $\bot$ is a specific pattern, we can use a variant of this analysis to approximate demanded arguments. Thus, we summarize the main techniques and results of this analysis in the following.

Since we want to approximate the input/output relation of operations, an *interpretation* I is some set of equations

$$I = \{f\ t_1 \ldots t_n) \doteq t \mid f\ n\text{-ary operation}, t_1, \ldots, t_n, t \text{ are partial values}\}$$

[7] The extension to operations with more than one argument is straightforward.

The *evaluation of an expression e w.r.t. I* is a mapping $eval_I$ from expressions into sets of partial values defined by (where C and f denotes a constructor and an operation symbol, respectively):

$$\begin{aligned} eval_I(x) &= \{x\} \\ eval_I(C\ e_1 \ldots e_n) &= \{C\ t_1 \ldots t_n \mid t_i \in eval_I(e_i), i = 1, \ldots, n\} \\ eval_I(f\ e_1 \ldots e_n) &= \{\bot\} \cup \{t \mid t_i \in eval_I(e_i), i = 1, \ldots, n, f\ t_1 \ldots t_n \doteq t \in I\} \end{aligned}$$

Hence, an operation is approximated as undefined or evaluated with the information provided by the interpretation.

For the demand analysis, we are interested in the behavior of operations when they are called with undefined arguments. Thus, it is not necessary to compute the complete semantics of a program but it is sufficient to compute the behavior w.r.t. a given set of *initial calls* $\mathcal{M}$ containing elements of the form $f\ t_1 \ldots t_n$ where f is an operation and $t_1, \ldots, t_n$ are partial values. Then we define the transformation $T_\mathcal{M}$ on interpretations I by

$$\begin{aligned} T_\mathcal{M}(I) = \{s \doteq \bot \mid s \in \mathcal{M}\} \cup \{s \doteq r' \mid s \doteq t \in I,\ s = r \in [\mathcal{P}]_\bot,\ r' \in eval_I(r)\} \\ \cup \{f\ t_1 \ldots t_n \doteq \bot \mid s \doteq t \in I,\ s = r \in [\mathcal{P}]_\bot, f\ e_1 \ldots e_n \text{ is a subterm of } r, \\ t_i \in eval_I(e_i),\ i = 1, \ldots, n\} \end{aligned}$$

Intuitively, the transformation $T_\mathcal{M}$ adds to the set of initial calls in each iteration

1. better approximations of the rules' right-hand sides ($s \doteq r'$) and
2. new function calls occurring in right-hand sides ($f\ t_1 \ldots t_n \doteq \bot$).

Here, "better" should be interpreted w.r.t. the usual approximation ordering $\sqsubseteq$ where $\bot$ is the minimal element. As usual, we define

$$\begin{aligned} T_\mathcal{M} \uparrow 0 &= \emptyset \\ T_\mathcal{M} \uparrow k &= T_\mathcal{M}(T_\mathcal{M} \uparrow (k-1)) \quad (\text{for } k > 0) \end{aligned}$$

Since the mapping $T_\mathcal{M}$ is continuous on the set of all interpretations, the least fixpoint $C_\mathcal{M} = T_\mathcal{M} \uparrow \omega$ exists. The following theorem states the correctness of this fixpoint semantics w.r.t. CRWL rewriting.

Theorem 1 ([34]) *1. If $s \doteq t \in C_\mathcal{M}$, then $s \overset{*}{\twoheadrightarrow} t$.*

2. If $s \in \mathcal{M}$ and t is a partial value with $s \overset{*}{\twoheadrightarrow} t$, then $s \doteq t \in C_\mathcal{M}$.

We call an equation $s \doteq t \in I$ *maximal* in I if there is no $s \doteq t' \in I$ with $t' \neq t$ and $t \sqsubseteq t'$. The set of all maximal elements of an interpretation I is denoted by $max(I)$. Maximal elements can be used to characterize a demanded argument, as the following result shows.

Proposition 2 *Let f be a unary operation and $f\ \bot \in \mathcal{M}$. If $f\ \bot \doteq \bot \in max(C_{\mathcal{M}})$, then f demands its argument.*

Proof: Assume $f\ \bot \doteq \bot \in max(C_{\mathcal{M}})$ and f does not demand its argument. Hence, by definition of a demanded argument, there is a normal form t of $(f\ \bot)$ with $t \neq \bot$. Since $f\ \bot \stackrel{*}{\twoheadrightarrow} t$ and t is a partial value, Theorem 1 implies $f\ \bot \doteq t \in C_{\mathcal{M}}$. This contradicts the assumption that $f\ \bot \doteq \bot$ is maximal in $C_{\mathcal{M}}$. ■

The proposition suggests that one should analyze the least fixpoint w.r.t. a set of initial calls having $\bot$ at argument positions. In order to obtain a computable approximation of the least fixpoint, we use the theory of abstract interpretation [25] and define appropriate abstract domains and abstract operations (like abstract constructor application and abstract matching) to compute an abstract fixpoint.

Interesting finite abstractions of partial values are sets of terms up to a particular depth k, e.g., as already used in the abstract diagnosis of functional programs [2], abstraction of term rewriting systems [16], or call pattern analysis of functional logic programs [34]. Due to its quickly growing size, this domain is mainly useful in practice for depth $k = 1$. In the domain of depth-bounded terms, subterms that exceed the given depth k are replaced by the specific constant $\top$ that represents any term, i.e., the abstract domain of *depth-k terms* consists of partial values up to a depth k extended by the constant $\top$. For instance, `False:`$\top$ is a depth-2 term. If one defines abstract constructor applications (by applying the constructor and cutting subterms deeper than k) and an abstract matching of linear constructor terms against depth-k terms (see [34] for details), one can compute an abstract least fixpoint which approximates the least fixpoint of concrete computations.

For instance, consider the operations "`?`", `not`, `xor`, and `xorSelf` defined above. In order to approximate their demanded arguments, we define a set of initial calls where one argument is $\bot$ and all other arguments are $\top$:

$$\mathcal{M} = \{\bot\texttt{?}\top,\ \top\texttt{?}\bot,\ \texttt{not}\ \bot,\ \texttt{xor}\ \bot\ \top,\ \texttt{xor}\ \top\ \bot,\ \texttt{xorSelf}\ \bot\}$$

Then the abstract least fixpoint w.r.t. $\mathcal{M}$ (note that the depth k is not relevant in this example) contains the following abstract equations:

$$\bot\texttt{?}\top \doteq \top,\ \top\texttt{?}\bot \doteq \top,\ \texttt{not}\ \bot \doteq \bot,\ \texttt{xor}\ \bot\ \bot \doteq \bot,\ \texttt{xor}\ \bot\ \top \doteq \bot,\ \texttt{xor}\ \top\ \bot \doteq \bot,$$
$$\texttt{xorSelf}\ \bot \doteq \bot$$

Since all these elements are also maximal, we can deduce by Proposition 2 that all arguments of `not`, `xor`, and `xorSelf` are demanded whereas "`?`" has no demanded argument. Of course, the analysis becomes more interesting in the case of recursive functions. We omit further examples here but refer to Section 6 for some benchmarks.

Our demand analysis can be extended in various ways. For instance, higher-order features can be covered by transforming higher-order applications into calls to an "apply" operation that implements the application of an arbitrary function occurring in the program to an expression [52]. This technique is also known as "defunctionalization"

[49]. Primitive operations, like arithmetic functions, usually demand all their arguments. Thus, their behavior can be approximated by returning the result $\bot$ if some argument is $\bot$, and otherwise $\top$ is returned.

5 Program Transformation

We want to improve the non-determinism behavior of functional logic programs by transforming them according to the ideas sketched in the previous section. As already discussed, this can be done by adding strict applications to demanded arguments that are non-deterministic. A method to approximate demanded arguments has already been shown. The approximation of non-deterministic expressions is much simpler. For this purpose, we define an operation as *non-deterministic* if it contains a call to "`?`" or a free variable in some of its defining rules, or if it depends directly or indirectly on some non-deterministic operation. Thus, this property can be computed using the defining rules and their program dependency graph.

Based on this information, we can classify expressions: an expression is non-deterministic if it contains some non-deterministic operation. Now we perform the following transformation of the source program: if there is some application `(`f e`)` in some rule, where e is non-deterministic and the argument of f is demanded, replace this application by `(`f `$!` e`)`. For instance, the program rule

```
main = xorSelf aBool
```

will be transformed into

```
main = xorSelf $! aBool
```

since the argument of `xorSelf` is demanded (as approximated above) and the argument `aBool` is non-deterministic. The extension of this transformation to operations with more than one argument is straightforward.

The effect of this transformation will be shown in the next section by some benchmarks.

6 Benchmarks

We have implemented (in Curry) the program transformation shown above in a first prototype in order to get some ideas about its effectiveness. The program analyzer uses the depth-k domain to approximate demanded arguments. In order to provide an efficient analysis, only maximal abstract elements are stored in the current interpretation and the fixpoint iteration is done by an iteration using working lists. The non-determinism information is approximated in a separate analysis. The analysis results are used to guide the program transformation sketched above which produces the optimized Curry program.

Since our prototype does not support all features of Curry (e.g., no I/O), we have tested it only on smaller benchmark programs. Since our transformation is intended to

Benchmark	ViaLOIS (original)	KiCS2 (original)	KiCS2 (optimized)
`last2`	n/a	1.34	0.94
`last6`	n/a	2.72	0.94
`addNum2`	1.25	1.54	0.01
`addNum5`	22.08	8.58	0.01
`addPair`	1.36	1.54	0.01
`addTriple`	4.45	3.65	0.01
`half2`	2.18	3.78	1.44
`half5`	4.97	6.37	1.44
`dupList2`	n/a	3.34	0.11
`dupList5`	n/a	52.49	0.11
`select`	22.51	6.37	0.01
`queens`	n/a	36.62	1.26
`psort`	4.08	4.98	4.78

Table 1: Benchmarks comparing original and optimized programs

improve non-deterministic programs, we have selected programs where non-deterministic operations occur as arguments.

The benchmarks were executed on a Linux machine running Linux (Ubuntu 11.10) with an Intel Core i5 (2.53GHz) processor. We omit the analysis times since they are less than 10 milliseconds for all presented examples. We tested two recent Curry implementations that are based on the idea to present non-deterministic values in a data structure: KiCS2 [20] with the Glasgow Haskell Compiler (GHC 7.0.3, option -O2) as its back end, and ViaLOIS [14] with the OCaml native-code compiler (version 3.12.0) as its back end. Table 1 shows the run times (in seconds) of a compiled executable for different programs. The programs `last2` and `last6` compute the last element of a list (of 10,000 elements) and add it two and six times to itself, respectively. `addNum2` and `addNum5` non-deterministically choose a number (out of 2000) and add it two and five times, respectively. Similarly, `addPair` and `addTriple` non-deterministically create a pair and triple of the same elements and add the components. `half2` and `half5` compute the half of a number n (here: 2000) by solving the equation `x+x=:=`n and add the result two and five times, respectively. `dupList2` and `dupList5` check a list `xs` (of 2000 elements) whether it is a duplicated list by solving the equation `ys+ys=:=xs` and concatenating `ys` two and five times, respectively. `select` non-deterministically selects an element in a list and returns the element and a list computed by deleting the selected element. `queens` computes all safe placements of eight queens on a chessboard by enumerating all placements and non-deterministically checking whether two queens can attack each other. In this example, the duplication of choices stems from lazy pattern matching, as pointed out in [17, Sect. 6.9]. Finally, `psort` is the permutation sort example shown above applied to a list of 14 elements. The appendix contains the complete code of all benchmarks.

Since ViaLOIS is in an experimental state, it does not support all features of Curry

(in particular, free variables of type integer are not supported) so that some benchmarks are not executable with ViaLOIS (marked by "n/a"). For the same reason, ViaLOIS does not support the primitive operation "`$!`" necessary for the optimization presented in this paper. Thus, the optimized programs are only executed with KiCS2. As one can see, the improvements obtained by our optimization are quite relevant for the considered class of programs. Only the improvement for `psort` is small since we cannot strictly evaluate the complete permutation, as discussed in Section 4.

7 Conclusions

We have shown a program transformation to improve the efficiency of non-deterministic computations in implementations of functional logic languages with a demand-driven strategy. If such implementations support a variety of search strategies, in particular, complete strategies, they often present the computation space in some tree structure which is explored by the search strategy [14, 20, 21]. This has the risk that non-deterministic structures are duplicated which increases the complexity of traversing the resulting structures. In order to overcome this disadvantage, we presented an analysis to approximate demanded arguments and use this information to evaluate non-deterministic arguments in a strict manner. We have also shown results from a prototypical implementation of this approach.

Since this work is based on techniques from various domains ranging from implementations of declarative languages to program analysis frameworks for such languages, there is a lot of related work. Since we already discussed related approaches throughout this paper, we omit a further discussion here. For future work, our demand analysis should be extended to enable the analysis of complete applications. This requires the appropriate approximation of all primitive operations, including I/O operations, and a modular analysis to be applied to larger programs. Furthermore, the use of other abstract domains, like rational trees, that can also approximate the demand of arbitrary large structures (e.g., lists) is another interesting topic fur future work. However, the permutation sort example shows that the presence of failures in non-deterministic computations need a careful treatment to avoid an increase of the search space.

References

[1] J.M. Almendros-Jiménez and A. Becerra-Terón. A Framework for Goal-Directed Bottom-Up Evaluation of Functional Logic Programs. In *Proc. of the 5th International Symposium on Functional and Logic Programming (FLOPS 2001)*, pp. 153–169. Springer LNCS 2024, 2001.

[2] M. Alpuente, M. Comini, S. Escobar, M. Falaschi, and S. Lucas. Abstract Diagnosis of Functional Programs. In *Proc. of the 12th Int'l Workshop on Logic-Based Program Synthesis and Transformation (LOPSTR 2002)*, pp. 1–16. Springer LNCS 2664, 2002.

[3] M. Alpuente, M. Falaschi, and G. Vidal. Partial Evaluation of Functional Logic Programs. *ACM Transactions on Programming Languages and Systems*, Vol. 20, No. 4, pp. 768–844, 1998.

[4] S. Antoy. Constructor-based Conditional Narrowing. In *Proc. of the 3rd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP 2001)*, pp. 199–206. ACM Press, 2001.

[5] S. Antoy. On the Correctness of Pull-Tabbing. *Theory and Practice of Logic Programming*, Vol. 11, No. 4-5, pp. 713–730, 2011.

[6] S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. *Journal of the ACM*, Vol. 47, No. 4, pp. 776–822, 2000.

[7] 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.

[8] 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.

[9] 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.

[10] S. Antoy and M. Hanus. Set Functions for Functional Logic Programming. In *Proceedings of the 11th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'09)*, pp. 73–82. ACM Press, 2009.

[11] S. Antoy and M. Hanus. Functional Logic Programming. *Communications of the ACM*, Vol. 53, No. 4, pp. 74–85, 2010.

[12] 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.

[13] 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.

[14] S. Antoy and A. Peters. Compiling a Functional Logic Language: The Basic Scheme. In *Proc. of the Eleventh International Symposium on Functional and Logic Programming*, pp. 17–31. Springer LNCS 7294, 2012.

[15] L. Augustsson, M. Rittri, and D. Synek. On generating unique names. *Journal of Functional Programming*, Vol. 4, No. 1, pp. 117–123, 1994.

[16] D. Bert and R. Echahed. Abstraction of Conditional Term Rewriting Systems. In *Proc. of the 1995 International Logic Programming Symposium*, pp. 147–161. MIT Press, 1995.

[17] B. Braßel. *Implementing Functional Logic Programs by Translation into Purely Functional Programs*. PhD thesis, Christian-Albrechts-Universität zu Kiel, 2011.

[18] 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.

[19] B. Braßel, M. Hanus, and M. Müller. High-Level Database Programming in Curry. In *Proc. of the Tenth International Symposium on Practical Aspects of Declarative Languages (PADL'08)*, pp. 316–332. Springer LNCS 4902, 2008.

[20] 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.

[21] 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.

[22] 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.

[23] R. Caballero and F.J. López-Fraguas. A Functional-Logic Perspective of Parsing. In *Proc. 4th Fuji International Symposium on Functional and Logic Programming (FLOPS'99)*, pp. 85–99. Springer LNCS 1722, 1999.

[24] K. Claessen and P. Ljunglöf. Typed Logical Variables in Haskell. In *Proc. ACM SIGPLAN Haskell Workshop*, Montreal, 2000.

[25] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *Proc. of the 4th ACM Symposium on Principles of Programming Languages*, pp. 238–252, 1977.

[26] J. de Dios Castro and F.J. López-Fraguas. Extra variables can be eliminated from functional logic programs. *Electronic Notes in Theoretical Computer Science*, Vol. 188, pp. 3–19, 2007.

[27] S. Fischer. A Functional Logic Database Library. In *Proc. of the ACM SIGPLAN 2005 Workshop on Curry and Functional Logic Programming (WCFLP 2005)*, pp. 54–59. ACM Press, 2005.

[28] S. Fischer and H. Kuchen. Systematic generation of glass-box test cases for functional logic programs. In *Proceedings of the 9th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'07)*, pp. 63–74. ACM Press, 2007.

[29] 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.

[30] 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.

[31] 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.

[32] M. Hanus. Type-Oriented Construction of Web User Interfaces. In *Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'06)*, pp. 27–38. ACM Press, 2006.

[33] M. Hanus. Multi-paradigm Declarative Languages. In *Proceedings of the International Conference on Logic Programming (ICLP 2007)*, pp. 45–75. Springer LNCS 4670, 2007.

[34] M. Hanus. Call Pattern Analysis for Functional Logic Programs. In *Proceedings of the 10th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'08)*, pp. 67–78. ACM Press, 2008.

[35] M. Hanus and S. Koschnicke. An ER-based Framework for Declarative Web Programming. In *Proc. of the 12th International Symposium on Practical Aspects of Declarative Languages (PADL 2010)*, pp. 201–216. Springer LNCS 5937, 2010.

[36] M. Hanus and S. Lucas. A Semantics for Program Analysis in Narrowing-Based Functional Logic Languages. In *Proc. 4th Fuji International Symposium on Functional and Logic Programming (FLOPS'99)*, pp. 353–368. Springer LNCS 1722, 1999.

[37] 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.

[38] M. Hanus (ed.). Curry: An Integrated Functional Logic Language (Vers. 0.8.2). Available at `http://www.curry-language.org`, 2006.

[39] R. Hinze. Prolog's control constructs in a functional setting - Axioms and implementation. *International Journal of Foundations of Computer Science*, Vol. 12, No. 2, pp. 125–170, 2001.

[40] H. Hussmann. Nondeterministic Algebraic Specifications and Nonconfluent Term Rewriting. *Journal of Logic Programming*, Vol. 12, pp. 237–255, 1992.

[41] F. López-Fraguas and J. Sánchez-Hernández. TOY: A Multiparadigm Declarative System. In *Proc. of RTA'99*, pp. 244–247. Springer LNCS 1631, 1999.

[42] F.J. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández. A Simple Rewrite Notion for Call-time Choice Semantics. In *Proceedings of the 9th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'07)*, pp. 197–208. ACM Press, 2007.

[43] 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.

[44] J.M. Molina-Bravo and E. Pimentel. Modularity in Functional-Logic Programming. In *Proc. of the Fourteenth International Conference on Logic Programming (ICLP'97)*, pp. 183–197. MIT Press, 1997.

[45] J.J. Moreno-Navarro, H. Kuchen, J. Mariño-Carballo, S. Winkler, and W. Hans. Efficient Lazy Narrowing using Demandedness Analysis. In *Proc. of the 5th International Symposium on Programming Language Implementation and Logic Programming*, pp. 167–183. Springer LNCS 714, 1993.

[46] A. Mycroft. The theory and practice of transforming call-by-need into call-by-value. In *Proc. International Symposium on Programming*, pp. 269–281. Springer LNCS 83, 1980.

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

[48] U.S. Reddy. Narrowing as the Operational Semantics of Functional Languages. In *Proc. IEEE Internat. Symposium on Logic Programming*, pp. 138–151, Boston, 1985.

[49] J.C. Reynolds. Definitional Interpreters for Higher-Order Programming Languages. In *Proceedings of the ACM Annual Conference*, pp. 717–740. ACM Press, 1972.

[50] C. Runciman, M. Naylor, and F. Lindblad. Smallcheck and lazy smallcheck: automatic exhaustive testing for small values. In *Proc. of the 1st ACM SIGPLAN Symposium on Haskell*, pp. 37–48. ACM Press, 2008.

[51] 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.

[52] D.H.D. Warren. Higher-order extensions to Prolog: are they needed? In *Machine Intelligence 10*, pp. 441–454, 1982.

[53] F. Zartmann. Denotational Abstract Interpretation of Functional Logic Programs. In *Proc. of the 4th International Symposium on Static Analysis (SAS'97)*, pp. 141–156. Springer LNCS 1302, 1997.

A Benchmark Code

In the following we show the source code of all benchmarks used in Section 6.

A.1 Benchmark `last`

```
[]     ++ ys = ys
(x:xs) ++ ys = x : xs++ys

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

last2 xs = let x = last xs in x+x

last6 xs = let x = last xs in x+x+x+x+x+x
```

A.2 Benchmark `add...`

```
-- an arbitrary number between 1 and n:
ndnum n = if (n==1) then 1 else (n ? ndnum (n-1))

addNum2 n = let x = ndnum n in x+x

addNum5 n = let x = ndnum n in x+x+x+x+x


-- arbitrary tuples between 1 and n:
ndpair   n = if (n==1) then (1,1)   else ((n,n)   ? ndpair (n-1))
ndtriple n = if (n==1) then (1,1,1) else ((n,n,n) ? ndtriple (n-1))

addPair   n = x+y   where (x,y) = ndpair n

addTriple n = x+y+z where (x,y,z) = ndtriple n
```

A.3 Benchmark `half`

```
data Peano = O | S Peano

toPeano :: Int → Peano
```

```
toPeano n = if n==0 then 0 else S (toPeano (n-1))

fromPeano :: Peano → Int
fromPeano 0     = 0
fromPeano (S x) = fromPeano x + 1

equal :: Peano → Peano → Bool
equal 0     0     = True
equal (S p) (S q) = equal p q
equal (S _) 0     = False
equal 0     (S _) = False

add :: Peano → Peano → Peano
add 0     p = p
add (S p) q = S (add p q)

half y | equal (add x x) (toPeano y) = fromPeano x where x free

half2 n = let x = half n in x+x

half5 n = let x = half n in x+x+x+x+x
```

A.4 Benchmark `dupList`

```
[]     ++ ys = ys
(x:xs) ++ ys = x : xs++ys

length []     = 0
length (_:xs) = 1 + length xs

findDuplicate s | x++x =:= s = x  where x free

dupList2 n = let x = findDuplicate n in length (x++x)

dupList5 n = let x = findDuplicate n in length (x++x++x++x++x)
```

A.5 Benchmark `select`

```
someOf (x:xs) = x ? someOf xs

del x (y:ys) = if x==y then ys
                       else y : del x ys
```

```
sum []     = 0
sum (x:xs) = x + sum xs

sumUp xs m = m + sum (del m xs)

select xs = sumUp xs (someOf xs)

main = select [1..100]
```

A.6 Benchmark queens

```
import SetFunctions

insert x []     = [x]
insert x (y:ys) = x:y:ys ? y:insert x ys

perm []     = []
perm (x:xs) = insert x (perm xs)

list n m = if n==m then [m]
                   else n : list (n+1) m

-- a placement is safe if it is a permutation without a capture:
queens n = safe (perm (list 1 n))

-- we use set functions [10] to check whether that there is no capture
-- for the given placement:
safe p | isEmpty (set1 unsafe p) = p

-- a position is unsafe if there is a diagonal capture:
unsafe xs = capture (membersWithDelta xs)

capture (i,lenZ,j) | abs (i-j)-1 == lenZ = success

abs i = if i<0 then 0-i else i

-- compute some element of a list with its position:
memberWithIndex :: [a] → (a,Int)
memberWithIndex xs = memberWithIndex' 0 xs
memberWithIndex' i (x:xs) = (x,i) ? memberWithIndex' (i+1) xs

-- compute some element of a list with the list of subsequent elements:
memberWithRest :: [a] → (a,[a])
```

```
memberWithRest (x:xs) = (x,xs) ? memberWithRest xs

-- compute two elements of a list together with their distance
-- (the assertion x+i+y>0 is added for more demand since the
-- current demand analysis is not general enough for lets)
membersWithDelta l | x+i+y>0 = (x,i,y)
  where (x,xs) = memberWithRest l
        (y,i)  = memberWithIndex xs

main = queens 8
```

A.7 Benchmark psort

```
import Constraint

insert x []     = [x]
insert x (y:ys) = (x:y:ys) ? (y:insert x ys)

perm []     = []
perm (x:xs) = insert x (perm xs)

psort xs = checkSorted (perm xs)
checkSorted ys | sorted ys = ys

sorted []       = success
sorted [_]      = success
sorted (x:y:ys) = x<=:y & sorted (y:ys)

main = psort [14,13..1]
```