Eliminating Irrelevant Non-determinism in Functional Logic Programs

Sergio Antoy[1] Michael Hanus[2]

[1] Computer Science Dept., Portland State University, Oregon, U.S.A.
antoy@cs.pdx.edu

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

Abstract. Functional logic programming languages support non-deterministic search and a flexible use of defined operations by applying them to unknown values. The use of these features has the risk that equal values might be computed several times or I/O computations could fail due to non-deterministic subcomputations. To detect such problems at compile time, we present a method to locate non-deterministic operations. If the non-determinism caused by some operation is semantically not relevant, the programmer can direct the compiler to produce only one result of a computation. If all the results of the computations are equal, this directive preserves the semantics and improves the operational behavior of programs. We define the declarative meaning of such annotations and propose both testing and verification techniques that respectively increase the confidence or formally prove that the non-determinism of an operation is irrelevant.

1 Introduction

Functional logic languages combine the most important features of functional and logic programming in a single language (see [8, 24] for recent surveys). In particular, the functional logic language Curry [26] conceptually extends Haskell with common features of logic programming, i.e., non-determinism, free variables, and constraint solving. Non-determinism is useful in programming to write a specification of a task instead of coding all the details of the task's solution. For instance, consider the selection sort algorithm where the smallest element is placed in front of the sorted remaining elements. In Curry, one can easily specify the smallest element of a list of integers by

```
min :: [Int] → Int
min xs@(_++[x]++_) | all (x<=) xs = x
```

Here we use a *functional pattern*, i.e., an expression with evaluable functions at pattern positions [5], to express that `x` is any element of the input list, and an *as pattern* (known from Haskell) to refer to the complete input list by `xs`. If the condition that `x` is not greater than any element of the input list `xs` is satisfied, we return the selected element `x` as the smallest one. Operation `min` shows an example of don't care non-determinism. Its definition through a functional pattern is elegant and declarative, but a consequence is that if there are repeated occurrences of the minimum in the argument, the minimum is returned multiple times. Of course, we don't care which occurrence is returned since they are all equal.

With this definition of `min`, the implementation of sorting a list is straightforward (`delete x xs` returns the list `xs` without the first occurrence of `x`):

```
selSort []        = []
selSort xs@(_:_) = m : selSort (delete m xs) where m = min xs
```

Although this implementation of sorting a list is correct, it has a potential drawback when used in larger applications. To ensure a declarative style of computations, Curry adopts the monadic I/O approach of Haskell. Hence, an application program computes an I/O action, i.e., a transformation on a state of a "world" (including physical resources like a terminal or file system), that is applied to a concrete world when the program is executed. Since it is impossible to copy the world to apply a non-deterministic I/O action to these copies, the computed I/O action must be unique [24]. For instance, the execution of the call ("`?`" denotes a non-deterministic choice between two values)

```
print 1 ? print 2
```

leads to a run-time error ("non-determinism in I/O"). This is intended, since it is intentionally unspecified whether one should show `1` or `2` on the display. As a consequence, non-deterministic computations need to be encapsulated when using them in applications performing I/O. Encapsulating non-determinism means producing the set of every possible non-deterministic result of a computation, hence a deterministic result. Thus, if the call "`print (selSort [1,3,2,1])`" is evaluated without encapsulating the argument, we obtain a non-determinism error. This is due to the fact that the list contains two smallest elements so that the auxiliary operation `min` yields two (equal) results.

The same problem might occur even if only one non-deterministic branch of a computation leads to a result. For instance, consider the computation of the last element of a list by an inverse use of list concatenation:

```
last (_ ++ [x]) = x
```

Although `last` yields at most one result for a given list, its use in the context of an I/O operation causes a run-time error since one cannot decide which of the alternative I/O actions eventually yields a result.

These are not artificial examples. Such problems occurred to us several times when putting together applications consisting of more than one hundred modules and thousands of operations. Therefore, we develop a practical solution to it. As known from lazy functional languages, the source of run-time errors is not easy to locate from the run-time stack available when an error actually occurs. Therefore, we propose a compile-time analysis to locate potential calls to non-deterministic operations from a main operation. In this way, a programmer can examine these operations. If an operation computes, for a given argument, a single result multiple times, we propose to annotate such operations as *deterministic*. This information is used by a compiler to return the result only once since any recomputation would provide no additional information. This yields an improved operational behavior (reduction of the computation space) and avoids the kinds of non-determinism errors sketched above. In our example, we simply annotate the operation `min` as deterministic to avoid the non-determinism error. By the use of determinism annotations, we combine the compact and comprehensible specification of operations with a reasonable operational behavior.

This paper investigates the source of non-determinism in a program, introduces a new concept of deterministic operation and defines its semantics. The semantic properties of deterministic operations allow us to implement them more efficiently. Moreover, we discuss methods to check these properties. In the next section, we review the main concepts of functional logic programming and Curry. A compile-time method to locate non-deterministic operations relevant in application programs is presented in Sect. 3. The concept of deterministic operation is introduced in Sect. 4. The implementation of our program analysis and deterministic operations is shown in Sect. 5 and evaluated by some benchmarks. Options to check the formal properties of deterministic operations are discussed in Sect. 6 before we relate our proposal to other work and conclude.

2 Functional Logic Programming and Curry

We briefly review those elements of functional logic languages and Curry that are necessary to understand the contents of this paper. More details can be found in surveys on functional logic programming [8, 24] and in the language report [26].

Curry is a declarative multi-paradigm language combining in a seamless way features from functional and logic programming. The syntax of Curry is close to Haskell [32]. In addition to Haskell, Curry allows *free* (*logic*) *variables* in conditions and right-hand sides of rules. These variables must be explicitly declared unless they are anonymous. Function calls can contain free variables, in particular variables without a value at call time. These calls are evaluated lazily where free variables as demanded arguments are non-deterministically instantiated [4].

Moreover, the patterns of a defining rule are expanded with respect to traditional functional languages. As a matter of convenience, patterns can be non-linear, i.e., they might contain multiple occurrences of some variable, which is an abbreviation for equalities between these occurrences. Patterns can also be *functional* [5] to more easily and directly define functions. A functional pattern is a pattern containing defined operations (and not only data constructors and variables) occurring in an argument of the left-hand side of a rule. Such a pattern abbreviates the set of all standard patterns to which the functional pattern can be evaluated (by narrowing). For example, functional patterns have been proved powerful to express arbitrary selections in tree structures, e.g., in XML documents [23]. Details about their semantics and a constructive implementation of functional patterns by a demand-driven unification procedure can be found in [5].

Example 1. The following simple program shows the functional and logic features of Curry. It defines an operation "++" to concatenate two lists, which is identical to the Haskell encoding. The second operation, dup, returns some list element having at least two occurrences:[1]

```
(++) :: [a] → [a] → [a]          dup :: [a] → a
[]     ++ ys = ys                dup xs | xs == _++[x]++_++[x]++_
(x:xs) ++ ys = x : (xs ++ ys)           = x    where x free
```

The condition of the rule defining dup is solved by instantiating x and the anonymous free variables "_". This evaluation method corresponds to narrowing [33, 34], but Curry

[1] Note that Curry requires the explicit declaration of free variables, as x in the rule of dup, to ensure checkable redundancy.

narrows with possibly non-most-general unifiers to ensure the optimality of computations [4]. Using a functional pattern, the definition of `dup` is simply phrased as:

```
dup (_++[x]++_++[x]++_) = x
```

Note that `dup` is a *non-deterministic operation* since it might deliver more than one result for a given argument, e.g., the evaluation of `dup [1,2,2,1]` yields the values `1` and `2`. Non-deterministic operations, which are interpreted as mappings from values into sets of values [21], are an important feature of contemporary functional logic languages. Hence, there is also a predefined *choice* operation:

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

Thus, the expression "`0 ? 1`" evaluates to `0` and `1` with the value non-deterministically chosen.

Default rules, which have recently been proposed [10], are useful in combination with functional patterns in order to express cases where a functional pattern, which often corresponds to an infinite set of standard patterns, is not applicable. Any operation can have a single default rule. To avoid syntactic extensions, default rules are marked by adding the suffix `'default` to the operation's name. The default rule is applied if no standard rule is applicable (see [10] for a precise definition in the context of non-deterministic values and free variables). For instance, by slightly modifying the operation `dup`, we can easily define a predicate `isSet` which checks whether a given list represents a set, i.e., does not contain duplicates:

```
isSet (_++[x]++_++[x]++_) = False
isSet'default _           = True
```

Set functions [7] allow the encapsulation of non-deterministic computations in a strategy-independent manner. For each defined function f, $f_{\mathcal{S}}$ denotes the corresponding set function. $f_{\mathcal{S}}$ encapsulates the non-determinism caused by evaluating f apart from the non-determinism originating from the evaluation of the arguments to which f is applied. For instance, consider the operation `decOrInc` defined by

```
decOrInc x = (x-1) ? (x+1)
```

Then "`decOrInc`$_{\mathcal{S}}$ `3`" evaluates to (an abstract representation of) the set $\{2,4\}$, i.e., the non-determinism caused by `decOrInc` is encapsulated into a set. However, "`decOrInc`$_{\mathcal{S}}$ `(2 ? 5)`" evaluates to two different sets $\{1,3\}$ and $\{4,6\}$, i.e., the non-determinism caused by the argument is not encapsulated.

3 Location of Non-deterministic Operations

To avoid potential problems with non-deterministic operations, first we have to locate them in a source program. In this section we present our method for this.

Definition 1 (NDD operation). *An operation is* non-deterministically defined *(*NDD*) if its defining rules are not inductively sequential*[2] *or some of the defining rules contain free variables.*

[2] The defining rules are inductively sequential if their patterns are just case distinctions on the constructors (see [2] for a precise definition). A consequence of this definition is that operations defined by functional patterns are NDD.

The operation `dup` defined in Example 1 shows why the occurrence of free variables in rules might lead to non-deterministic operations even if the left-hand sides are inductively sequential. This is due to the fact that free variables are equivalent to non-deterministic operations that generate all the values [6] of a given type. For instance, the choice operator "`?`" defined above by rules with overlapping left-hand sides can also be defined by rules with non-overlapping left-hand sides and a free variable:

```
x ? y = choose b x y            choose True  x y = x
          where b free          choose False x y = y
```

Note that Def. 1 only approximates non-deterministic evaluations. For instance, the use of the operation `f` defined by

```
f = if True then [] else ys++ys  where ys free
```

will never lead to a choice in a computation, although we classify it as non-deterministically defined. However, our approximation is syntactically decidable.

If NDD operations are not invoked during a functional logic computation, then this computation is deterministic, i.e., there is no alternative outcome for the same initial expression. This can be easily proved by induction on the steps of an evaluation sequence, e.g., using the small-step operational semantics defined in [1]. Hence, in order to detect the sources of potentially non-deterministic computations, we have to find NDD operations. However, in a large application with many libraries, not all NDD operations are relevant since they might not be called or their calls are encapsulated in set functions. It would not be helpful to report all NDD operations occurring in a program. Instead, we want to know only those NDD operations that are called (directly or indirectly) from the main expression starting the application. For this purpose, we need a dependency analysis which assigns to each operation the set of all relevant NDD operations.

Definition 2 (Relevant NDD operations). *Let $\mathcal{F}$ be the set of all defined operations in a program. For all $f \in \mathcal{F}$ we denote by $f^{\mathcal{R}} \subseteq \mathcal{F}$ the set of NDD operations that are* relevant for f*: $f^{\mathcal{R}}$ is the smallest set such that the following properties hold:*

- *If f is an NDD operation, then $f^{\mathcal{R}} = \{f\}$.*
- *If f is a set function, then $f^{\mathcal{R}} = \varnothing$.*
- *Otherwise: $g^{\mathcal{R}} \subseteq f^{\mathcal{R}}$ for all $g \in \mathcal{F}$ occurring in some rule defining f.*

The first property ignores further NDD operations called by f if f itself is NDD. Hence, we return only the "first" NDD operation. In all our practical examples (see Sect. 5.2), this is sufficient to spot the NDD operation that is actually relevant for the overall non-deterministic behavior. Our implementation supports also the computation of the transitive closure of relevant NDD operations, but this often returns too much information.

Relevant NDD operations can be computed by a standard fixpoint analysis. The fixpoint computation always terminates since $\mathcal{F}$ is finite so that the abstract domain is finite. The implementation and practical results of this analysis are discussed in Sect. 5.

4 Deterministic Operations

When we locate a relevant NDD operation in an application, we can avoid its non-deterministic behavior if it is semantically not relevant, like in the operations `min` or `isSet` defined above. We call operations with semantically irrelevant non-determinism *deterministic*. Informally, an operation is deterministic if *different* values cannot be obtained for the *same* input. However, the operation can compute the same value multiple times. Deterministic operations can also fail if the input is not appropriate. For instance, the operation `head` defined as

```
head (x:xs) = x
```

is deterministic, although it does not yield a result for the empty list.

In order to mark a defined operation as deterministic, we annotate its determinism status in the last arrow of its type signature (this is partially inspired by the notation of deterministic and non-deterministic operations used in the semantic models of functional logic programming in [21]). Thus, we express the determinism status of the operations `min` and `last` by the following type signatures:

```
min  :: [Int] →DET Int                last :: [a] →DET a
```

From a declarative point of view, such an annotation is correct, i.e., an operation is deterministic, if all the results computed by this operation for a given input are equal. Since we are in a context of a lazy non-deterministic language where arguments, even if they are ground expressions, might denote several or also infinite values, a precise definition needs more care. For instance, consider the identity operation

```
id :: a  →  a
id x = x
```

Intuitively, `id` is a deterministic operation since it does not introduce any non-determinism. However, the ground (i.e., variable free) call `id (0?1)` yields two different results: `0` and `1`. Note that these non-deterministic results are caused by the arguments and not by `id` itself.

In order to deal with such subtleties, we need a formal model of the semantics of functional logic programs. The difficulties of combining non-deterministic operations with a demand-driven evaluation model have been pioneered in [21]. The authors proposed the *call-time choice* semantics [28] as a reasonable model, which has been adapted to contemporary functional logic languages. , such as Curry [26] or TOY [30]. The authors defined the rewriting logic CRWL as a logical foundation for declarative programming with non-strict and non-deterministic operations. Conceptually, values of arguments of an operation are determined before the operation is evaluated. In a lazy strategy, this is naturally obtained by sharing. Since standard term rewriting does not conform to the intended call-time choice semantics, other notions of rewriting are necessary to formalize this idea. In this paper we use the simple reduction relation of [31] which we review in the following.

Expressions occurring in a program contain *operations*, *constructors* (introduced in data type declarations), and *variables* (arguments of operations or free variables). The goal of a computation is to obtain a value of some expression, where a *value* is an expression that does not contain any operation. To cover demand-driven or non-strict computations, expressions can also contain the special symbol $\perp$ to represent an *undefined or unevaluated value*. A *partial value* is a value which might contain occurrences of $\perp$. A *partial constructor substitution* is a substitution that replaces variables by partial values. A *context* $\mathcal{C}[\cdot]$ is an expression with some "hole". Then the reduction relation we use throughout this paper is defined as follows:

$$\begin{array}{rcll} \mathcal{C}[\sigma(f\ t_1 \ldots t_n)] & \twoheadrightarrow & \mathcal{C}[\sigma(r)] & (f\ t_1 \ldots t_n = r \text{ program rule, } \sigma \text{ partial constr. subst.}) \\ \mathcal{C}[e] & \twoheadrightarrow & \mathcal{C}[\bot] & (e \text{ expression}) \end{array}$$

The first rule models 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 by allowing the evaluation of any subexpression to an undefined value (which is intended if the value of this subexpression is not demanded). As usual, $\twoheadrightarrow^*$ denotes the reflexive and transitive closure of this reduction relation. The equivalence of this rewrite relation and CRWL is shown in [31]. Conditional rules are not considered since they can be seen as syntactic sugar for conditional expressions [3]. We recall that two expressions e_1 and e_2 are *equal* iff they can be reduced to the same *value*, i.e., there exists a value t such that $e_1 \twoheadrightarrow^* t$ and $e_2 \twoheadrightarrow^* t$.

Now we can formally define the meaning of deterministic operations.

Definition 3 (Deterministic operation). *An n-ary operation f is* deterministic*, i.e., a determinism annotation f :: $\tau_1 \to \cdots \tau_n \to_{\scriptscriptstyle DET} \tau'$ is correct, iff, for all partial values $t_1, \ldots, t_n$ and evaluations $f\ t_1, \ldots, t_n \twoheadrightarrow^* r$ and $f\ t_1, \ldots, t_n \twoheadrightarrow^* r'$ with values r and r', $r = r'$ holds.*

Clearly, the declaration "`id :: a` $\to_{\scriptscriptstyle DET}$ `a`" is correct, but "`dup :: [a]` $\to_{\scriptscriptstyle DET}$ `a`" (where `dup` was defined in Example 1) would not be correct, since "`dup [1,2,2,1]`" evaluates to `1` and `2`. It is not obvious whether the annotation "`min :: [Int]` $\to_{\scriptscriptstyle DET}$ `Int`" is correct. We discuss methods to check the correctness of determinism annotations in Sect. 6.

Below, we motivate two crucial design decisions of the definition, namely why a deterministic operation must have unique result *values* and not just unique head normal forms or partial values and why arguments can be evaluated up to partial values rather then values, i.e., fully evaluated.

An alternative to unique *values* is unique head normal forms. Often, the latter is a target of computations in non-strict languages. Head normal form would be inappropriate since non-determinism may show up under the head, as in

```
f x = [x, 0 ? 1]
```

The expression "`f 0`" evaluates to the single head normal form `[0,0?1]` but to two different values `[0,0]` and `[0,1]`. Hence, the number of different results of an expression might depend on the degree of its evaluation, which is unfortunate when reasoning about programs in a declarative manner, i.e., without considering an evaluation strategy. As a consequence of this design, we must evaluate a deterministic operation application completely, i.e., to normal form, before we omit all other alternative choices.

Likewise, unique partial values would be inappropriate since most expressions might have different partial result values. For example, consider the operation `id` defined above. Then "`id 1` $\twoheadrightarrow^*$ `1`" and "`id 1` $\twoheadrightarrow^*$ $\bot$" are two derivations with different partial result values. Hence, if we change Def. 3 so that we require unique *partial* result values, `id` would not be deterministic.

The requirement that arguments can be partial values is appropriate since this allows us to prune the computation space even if an argument has not been fully evaluated. For instance, consider the following contrived non-deterministic definition of computing the first element of a list and the definition of an infinite list:

```
head (x:xs) = x                    ones = 1 : ones
head (x:xs) = id x
```

Note that `head` is a deterministic operation according to our definition. The evaluation of the expression `head ones` demands the head normal form of the argument, which is `1:ones`. If we apply the first rule of `head`, we obtain the result value `1` and any attempt to compute another value can be dropped because `1:ones` $\twoheadrightarrow^*$ `1:`$\bot$ and `head` is deterministic. Requiring values for the arguments in Def. 3 would have the consequence that the evaluation of `head [1]` yields one result whereas the evaluation of `head ones` yields two identical results.

Nevertheless, to check deterministic operations, it is sufficient to consider their behavior on values provided that each data type is *sensible*, i.e., has at least one value:

Proposition 1. *Assume types are sensible and that f is an n-ary operation such that, for all values $t_1, \ldots, t_n$ and evaluations $f\ t_1, \ldots, t_n \twoheadrightarrow^* r$ and $f\ t_1, \ldots, t_n \twoheadrightarrow^* r'$ with values r and r', $r = r'$ holds. Then f is a deterministic operation.*

Proof. Let $t'_1, \ldots, t'_n$ be partial values and $e' = f\ t'_1, \ldots, t'_n$. Assume $e' \twoheadrightarrow^* r$ and $e' \twoheadrightarrow^* r'$ with values r and r'. We must prove that $r = r'$. We transform each partial value t'_i into some value t_i by replacing each occurrences of $\bot$ in t'_i with some value of the appropriate type (which exists by assumption). Let $e = f\ t_1, \ldots, t_n$. Since the symbol $\bot$ does not occur in the program rules, the applicability of a program rule is not affected by the occurrences of $\bot$-symbols in the expression to be derived. Hence $e \twoheadrightarrow^* r$ by executing the same steps (i.e., same positions and rules) executed in $e' \twoheadrightarrow^* r$, and likewise $e \twoheadrightarrow^* r'$. By the assumption of the proposition, $r = r'$. Hence f is deterministic. □

We could have used this property, where the requirements on arguments and results are more symmetric, as the definition of deterministic operations. However, this would unnecessarily restrict the cutting of the search space by deterministic operations, as discussed above.

Our notion of deterministic operations is intended to be a constructive approximation of determinism in functional logic programs. Hence, we do not cover all potential determinism, in particular for non-terminating operations. To see why we cut the search space only if we compute a result value (and not a partially evaluated expression), consider the operation

```
inf x = if p x then x : inf (x+1)
                else (42 ? x) : inf (x+1)
```

where p is some predicate on integers. Intuitively, the operation `inf` does not branch if p is always satisfied. If p is not satisfied only on the argument `42`, the evaluation of `inf` branches but does not compute different results. In other cases, `inf` might compute different results. In particular, the non-deterministic branching might occur “arbitrarily late” during the evaluation of a call to `inf` so that there is no point to cut the computation space during the computation of an infinite structure. Thus, we decided to restrict the determinism property to finite result values.

We demonstrate the advantages of determinism annotations for application programming by further examples.

Example 2. We define an operation to sort a list by switching two adjacent elements which are out of order (a generalization of bubble sort):

```
bsort :: [Int] →DET [Int]
bsort (xs++[x,y]++ys) | x>y = bsort (xs++[y,x]++ys)
bsort'default xs = xs
```

The functional pattern in the first rule frees the programmer from specifying a concrete strategy to find a pair which should be swapped. Actually, the sort operation works with any strategy to select such pairs. The determinism annotation has the effect that all attempts to compute further values after the first sorted list are discarded. Without this annotation, we obtain 16 (equal) result values for the call `bsort [4,3,2,1]`, 768 results for `bsort [5,4,3,2,1]`, and 292864 results for `bsort [6,5,4,3,2,1]`.

Example 3. The *Dutch National Flag* problem [19] has been proposed in a simple form to discuss the termination of rewriting [17]. As already shown in [9], Curry allows a quite direct formulation by exploiting functional patterns and a default rule for the termination condition. Since this formulation produces many identical solutions, we mark it as deterministic and obtain the following reasonable implementation:

```
dnf :: [Color] →DET [Color]
dnf (x++[White]++y++[Red  ]++z) = dnf (x++[Red  ]++y++[White]++z)
dnf (x++[Blue ]++y++[Red  ]++z) = dnf (x++[Red  ]++y++[Blue ]++z)
dnf (x++[Blue ]++y++[White]++z) = dnf (x++[White]++y++[Blue ]++z)
dnf'default flag = flag
```

Example 4. The simplification of symbolic arithmetic expressions has been used in [5] to demonstrate the power of functional patterns. The task is to simplify arithmetic expressions like $1 * (x + 0)$ to x. Based on the definition of a replacement operation `replace`, where "`replace` e p t" is equivalent to the notation $e[t]_p$ commonly used in term rewriting [18], and a non-deterministic operation `evalTo` which evaluates to expressions equivalent to the argument, [3] [5] defines a one-step simplification operation as

```
simplifyStep (replace c p (evalTo x)) = replace c p x
```

Furthermore, it is remarked in [5] that "the application of repeated simplification steps to an expression until no more simplification steps are available can be controlled by Curry's search primitives." The concrete code for this task is not shown since the coding via search primitives is a bit cumbersome: one has to compute the set of all the results of a simplification step, check whether this set is empty and, if not, select one step and proceed with the simplification. Using default rules and determinism annotations, the code for completely simplifying expressions becomes quite agile:

```
simplify :: Exp →DET Exp
simplify (replace c p (evalTo t)) = simplify (replace c p t)
simplify'default e = e
```

The correctness of the determinism annotation of `simplify` depends on the confluence of the simplification rules specified by `evalTo`.

[3] The implementation of these operations and the structure of expressions is reviewed in Appendix A.

5 Practical Aspects

5.1 Implementation

We have implemented the analysis of relevant NDD operations described in Sect. 3 with the Curry analysis framework CASS [25]. CASS provides the infrastructure to analyze larger applications in a modular and incremental manner. Our actual analysis does not return only the relevant NDD operations but also the call sequence (limited to a fixed maximal length to keep the abstract domain finite) leading to relevant NDD operations from a main expression. This context information could be helpful to decide at which point non-determinism should be encapsulated.

To implement the reduction of the computation space by deterministic operations, we use existing features of functional logic languages. In particular, deterministic operations are implemented by a preprocessing approach that requires no language extension. The actual preprocessor is available and integrated into the compilation chain of the Curry systems PAKCS [27] and KiCS2 [12].

To support the possibility to annotate deterministic operations similarly to the notation used before, we introduce a type synonym:

```
type DET a = a
```

Hence, we can put the type constructor `DET` around any type without changing its meaning. For instance, we can write the type annotation of Example 4 as

```
simplify :: Exp ->DET Exp
```

Our preprocessor reads a Curry program and looks for such occurrences of `DET`. Since a deterministic operation is intended to compute only a single value for a given argument and ignore all others, we use set functions [7] to compute and select one value. Since the result sets are evaluated lazily, the computation of further elements is automatically precluded if we access only one element. Therefore, the following transformation is sufficient. If the preprocessor finds a function definition of the form (where $\overline{t_n}$ denotes a sequence of elements $t_1 \dots t_n$)

$$\begin{array}{l} f \ :: \ \tau_1 \ \rightarrow \dots \ \rightarrow \tau_n \ \rightarrow_{\text{DET}} \ \tau \\ f \ \overline{t_n^1} \ \mid \ c_1 \ = \ e_1 \\ \vdots \\ f \ \overline{t_n^k} \ \mid \ c_k \ = \ e_k \end{array}$$

then it is transformed into

$$\begin{array}{ll} f \ :: \ \tau_1 \rightarrow \ \cdots \ \rightarrow \tau_n \rightarrow \tau & f^{ND} \ :: \ \tau_1 \rightarrow \ \cdots \ \rightarrow \tau_n \rightarrow \tau \\ f \ \overline{x_n} \ = \ \texttt{selectValue} \ (f^{ND}_{\mathcal{S}} \ \overline{x_n}) & f^{ND} \ \overline{t_n^1} \ \mid \ c_1 \ = \ e_1 \\ & \vdots \\ & f^{ND} \ \overline{t_n^k} \ \mid \ c_k \ = \ e_k \end{array}$$

where f^{ND} is a new identifier and $\overline{x_n}$ are pairwise distinct variables. Hence, the original operation is replaced by a call to its set function where some element of the set is returned by the operation `selectValue`.[4] Due to this transformation, determinism

[4] Note that this operation on value sets returns *some* value from the set and ignores the others, i.e., it implements "don't care" non-determinism.

annotations have similarities to strictness annotations ("`seq`") in Haskell: they change the semantics in order to get a more efficient operational behavior.

Note that if the arguments of a deterministic operations are non-deterministic and have several values, the search space is cut for each value separately. This is due to the fact that set functions encapsulate the non-determinism of the function definition but not the non-determinism of the arguments (see Sect. 2). For instance, consider the operation `list2set` which transforms a list into a set by removing duplicated elements (also known as `nub` in Haskell but specified without a concrete strategy to find duplicates):

```
list2set :: [a] →DET [a]
list2set (xs++[e]++ys++[e]++zs) = list2set (xs++[e]++ys++zs)
list2set'default xs = xs
```

Then the call

```
list2set [True, True?False, True]
```

evaluates to two results: `[True]` and `[True,False]`. Thanks to the determinism annotation, the result `[True]` is computed once whereas it would be computed three times without the determinism annotation. One can even call deterministic functions with unknown arguments. For instance, `list2set xs == [True,False]` is solved by non-deterministically instantiating `xs` to `[True,False]`, `[True,False,False]`, `[True,False,True]`, and so on.[5] This shows that a determinism annotation does not imply that the operation can only be used in a purely functional manner, i.e., to compute an output value from a given input value, but deterministic operations compute at most one result for each given input value, which can still be guessed. This makes deterministic operations more powerful than Prolog's cut operator.

5.2 Benchmarking

To evaluate our analysis on non-trivial examples, we applied it to some existing applications where I/O non-determinism errors occurred during their development. Since our analysis was not available at that time, we manually located them in a time-consuming process. For our current test, we re-introduced the problematic definitions (mainly due to the use of functional patterns) in these applications. Our current analysis precisely returned these NDD operations as relevant for the main operation of the applications. The applications we tested are the KiCS2 compiler, CurryCheck (discussed in Sect. 6.1), the Curry preprocessor (partially described above), and a web-based information system for the curricula in the department of computer science in Kiel. For the benchmarks, we used the Curry implementation KiCS2 (Version 0.5.1) [12] with the Glasgow Haskell Compiler (GHC 7.6.3, option -O2) as its back end on a Linux machine (Debian 8.5) with an Intel Core i7-4790 (3.60Ghz) processor and 8GiB of memory.

Figure 1 shows the size of these applications and their analysis times. The table shows the number of modules, the size (in KB) and the number of lines of the source code (including all imported libraries), the time to analyze the complete application for the first time, and the time to re-analyze the complete application after fixing the problem (in seconds). Note that CASS performs a modular and incremental analysis, i.e.,

[5] This behavior is specific to KiCS2. PAKCS suspends on this equation since it has a more restricted implementation of set functions.

Application	# modules	program size	# source lines	initial	re-analysis
KiCS2 Compiler	63	651	17521	7.72	2.36
Curry Preprocessor	110	1040	26085	12.58	2.69
CurryCheck	52	538	14357	5.33	1.80
Curricula Web System	97	1056	26634	15.27	7.42

Fig. 1. Benchmarks: analysis of relevant NDD operations

Expression	nondet	det
`isSet (take 200 (repeat [1..10]))`	1.46	0.00
`last [1..20000]`	0.05	0.24
`selSort ([1..10]++[1..10])`	0.48	0.00
`bsort [5,4,3,2,1]`	2.25	0.00
`list2set [1,2,3,4,5,6,7,7,6,5,4,3,2,1]`	53.96	0.02
`dnf [White,Red,White,Blue,Red,Blue,White]`	0.89	0.00
`simplify <expression with 17 nodes>`	4.33	0.00

Fig. 2. Benchmarks: assessing the effect of determinism annotations

if some module has been analyzed, it stores the analysis information and re-analyzes a module only if the module or some of its (direct or indirect) imported modules have been changed. Hence, the initial analysis time is the worst-case analysis time which rarely occurs in practice. The re-analysis time clearly shows the advantage of this incremental analysis method. Altogether, the benchmarks demonstrate that our analysis method is effective and efficient enough for realistic applications.

In order to assess the practical consequences of determinism annotations, we compared the run times of some examples with and without determinism annotations on the same architecture used in the previous benchmarks. The timings were performed with the Unix *time* command measuring the execution time to compute all solutions (in seconds) of a compiled executable for each benchmark as a mean of three runs. The programs used for the benchmarks are the examples presented in the previous sections.

Figure 2 shows the execution times for evaluating the given expression without ("nondet" column) and with ("det" column) a determinism annotation (where "0.00" means less than 10 ms). Obviously, one can obtain arbitrarily large speedups by increasing the size of the input. Nevertheless, the numbers indicate that a non-deterministic implementation where we don't care about strategies to solve intermediate problems, like selecting appropriate list elements, is reasonable if the overall operation is deterministic. The example `last` shows that determinism annotations can also come with some cost since the machinery to encapsulate search with set functions is not for free. However, it should be noted that this comparison is also somehow artificial since a non-encapsulated top-level non-determinism is compared with an encapsulated computation. In practice, where the application program performs I/O operation on the top-level, all intermediate non-deterministic computations need to be encapsulated as discussed in Sect. 1.

6 Checking Deterministic Operations

If we add determinism annotations to operations that are not deterministic according to Def. 3, we lose completeness. Since the determinism property is undecidable in general, we cannot expect an automatic tool to verify this property. On the other hand, accepting only those determinism annotations where the determinism property can be verified by some sufficient criteria would be too restrictive. Therefore, the preprocessor outputs for each operation with a determinism annotation a proof obligation as a reminder and puts the task of verifying this property into the hands of the programmer. In this section, we discuss some methods to check or verify the correctness of determinism annotations.

6.1 Testing Deterministic Operations

A first approach to get confidence in the correctness of determinism annotations is testing. Testing can be quite powerful if one tests program properties, i.e., predicates, with a lot of test data. A well known example of such a property-based test framework is Haskell's QuickCheck tool [15] which generates random test data to test given properties. CurryCheck is a similar new tool for Curry programs distributed with the Curry systems PAKCS and KiCS2. It uses EasyCheck [14] for test data generation but automates property testing with additional features. In particular, CurryCheck automatically tests the correctness of determinism annotations as follows. If CurryCheck finds an annotation

f :: $\tau_1 \;\to\; \ldots \;\to\; \tau_n \;\to_{\text{DET}}\; \tau$

CurryCheck removes the determinism annotation (actually, it copies the code of f without the determinism annotation, since the annotated operation might be used in some other property) and adds the following property (where the property "e #< n" is satisfied if the *set* of all values of e contains less than n elements):

fIsDeterministic :: $\tau_1 \;\to\; \cdots \to\; \tau_n \;\to$ Prop
fIsDeterministic $x_1 \ldots x_n$ = $f\, x_1 \ldots x_n$ #< 2

This property is tested by systematically enumerating values for $x_1, \ldots, x_n$. Although this enumeration is exhaustive only for finite domains, checking determinism properties by testing is a quite useful tool in practice if the test cases are numerous and well distributed. These test cases are provided by the underlying EasyCheck library.

6.2 Proving Determinism Annotations

To show the correctness of determinism annotations also for infinite sets of input values, formal proofs are required. We discuss in this section methods to construct such proofs for particular examples.

A method to determine the determinism of an operation borrows from the theory of rewriting [37]. We denote by $\to$ the standard rewrite relation on terms and by $\to^*$ its reflexive and transitive closure. Then we can use the following proposition to verify determinism annotations by rewriting:

Proposition 2. *Assume that each data type is sensible and f is an n-ary operation so that, for all values $t_1, \ldots, t_n$ and rewrite derivations $f\ t_1, \ldots, t_n \to^* r$ and $f\, t_1, \ldots, t_n \to^* r'$ with values r and r', $r = r'$ holds. Then f is deterministic.*

Proof. Assume that the preconditions hold and $e \twoheadrightarrow^* r$ and $e \twoheadrightarrow^* r'$ for $e = f\ t_1, \ldots, t_n$. Since $\rightarrow^*$ over-approximates $\twoheadrightarrow^*$ ([31, Theorems 7 and 8]), there are rewrite derivations $e \rightarrow^* r$ and $e \rightarrow^* r'$. By assumption, $r = r'$. Hence f is deterministic by Prop. 1. □

Note that the converse of this proposition does not hold: The operation `f` defined as

```
f x = square (x ? (0-x))  where square x = x*x
```

is deterministic in the sense of Def. 3 but the expression `f 3` has the following rewrite derivations (among others):

```
f 3 →* 3 * (3 ? (0-3))  →  3 * 3   →  9
f 3 →* 3 * (3 ? (0-3))  →  3 * -3  →  -9
```

There are many cases where Prop. 2 can be applied to verify determinism annotations. For instance, within the context of rewriting, determinism coincides with *confluence*, the property that the end result of a complete sequence of applications of the rules does not depend on the order in which the rules were applied. *Weak orthogonality* is a sufficient condition to ensure confluence, hence determinism. First we briefly recall this concept, then we show its application to Example 4.

Given a binary relation $\rightarrow$ on a set A of "objects" and an element $a \in A$, we say that a is *confluent* iff for all $b, c \in A$, if $a \rightarrow^* b$ and $a \rightarrow^* c$ then there exists some $d \in A$ such that $b \rightarrow^* d$ and $c \rightarrow^* d$. If every element of A is confluent, then A is called confluent (or also Church-Rosser). Confluence captures determinism in that no element can have two distinct normal forms or values. When the objects of A are terms, there is a simple syntactic condition, called weak orthogonality, that ensures confluence. A rewrite system R is *weakly orthogonal* iff the following two conditions holds: (1) the rules of R are *left-linear*, i.e., no variable in the left-hand side is repeated, and (2) any critical pair (t, s) is trivial, i,.e. $t = s$ syntactically. We refer to [37, Def. 2.7.9] for the definition of *critical pair*, which is quite technical, but in the following paragraph we show an application of these concepts to one of our examples.

The simplification of an expression, as in Example 4, can be seen as a rewrite computation. A rule, $l \rightarrow r$, of this computation is constructed as follows: l is an alternative in the right-hand side of the definition of `evalTo` (see Appendix A) and r is the variable e, for example `Add (Lit 0) e` $\rightarrow$ `e`. An inspection of the rules shows that they are left-linear. If the left-hand sides of two rules do not overlap, as in a rule simplifying addition and a rule simplifying multiplication, then the rules can be applied independently of each other and the order in which they are applied does not affect the result. If the left-hand sides overlap, then we consider their most general common instance and rewrite this instance with each rule. The two results form a critical pair. For example, the two rules of addition overlap, their most common general instance is `Add (Lit 0) (Lit 0)`, and the critical pair is (`Lit 0`, `Lit 0`). Since the components of the pair are equal, the pair is trivial. Since all the critical pairs of this system are trivial, the system is weakly orthogonal, hence confluent, hence deterministic.

A second approach to ensure the determinism of an operation relies on the characteristics of the operation definition. For example, consider the sort operation `bsort` defined earlier. A call to `bsort` t, where t is a list of elements, has either of two outcomes: (1) the call result in a recursive call `bsort` t' where t' is a permutation of t, or (2) the call outputs t, the argument of `bsort`. The latter occurs only when there are no elements out of order in the argument. Since there is only one permutation of t with this property, this permutation is the only value that `bsort` t can ever produce. Hence `bsort` is deterministic. The pattern exemplified by `bsort` is not uncommon, e.g., the Dutch National Flag problem is similar, hence this is a simple and useful technique for determinism proofs.

One could also use proof assistants to show determinism properties. Due to the presence of (don't know) non-determinism in Curry programs, this requires the formal representation of the rewriting logic, as sketched in Sect. 4, in the logic of the proof assistant, as proposed in [16]. However, in simpler examples, it suffices to show properties about functional computations to show the correctness of determinism annotations. For instance, to show the determinism of the operation `last`, we have to show that every concatenation used in the pattern of `last` produces the same last element. This proof obligation can be formally written as

$$\forall l, l1, l2, x1, x2 : (l == l1 \mathbin{++} [x1] \;\;\wedge\;\; l == l2 \mathbin{++} [x2]) \implies x1 == x2$$

Since the involved operations "==" and "++" are defined in a purely functional manner, we could apply proof assistants for functional programs to verify this property. Actually, we formally verified this property with Agda (see Appendix B), a dependently typed functional programming language where proofs are written in a functional programming style [36]. The similarity of Agda with Haskell eases the translation of Curry programs into Agda. Actually, [11] describes a method to prove properties of non-deterministic computations by translating Curry programs into Agda programs. Using this method, one can mechanically prove that `min` (see Sect. 1) is deterministic by verifying its correspondence to a deterministic definition of a minimum function (see Appendix C).

7 Related Work

The use of deterministic operations to improve the operational behavior of functional logic computations has a long history. For instance, the SLOG system [20] used simplification with program rules and inductive axioms to reduce the search space. Similarly, the more general language ALF exploited deterministic rewrite computations interspersed in narrowing steps to obtain efficient functional logic computations [22]. A more dynamic use of deterministic computations was proposed in [29] where the "dynamic cut" as an alternative to Prolog's static cut has been introduced. In contrast to the static cut operator in Prolog, whose disadvantages were already discussed in Sect. 4, all these proposals aim at keeping the completeness of functional logic computations. In contrast to our proposal, these older proposals did not characterize a separate set of deterministic operations since all operations are deterministic due to a confluence requirement of the involved programs.

This view changed with the introduction of a new semantic foundation of functional logic programming presented in [21]. There, the notion of non-deterministic functions was introduced in functional logic programs and deterministic and non-deterministic functions are distinguished on a semantic level. The authors used these two kinds of functions to define the intended models of functional logic programs. Deterministic functions characterize homomorphisms and interpret data constructors, whereas user-defined operations are always interpreted as non-deterministic functions so that they are evaluated in a non-deterministic manner.

Improving computations for deterministic operations in the presence of non-deterministic operations has also been addressed in [13]. The authors transferred the idea of dynamic determinism detection in functional logic programs introduced in [29] to functional logic programs with non-deterministic operations. Dynamic determinism detection is based on the idea to check variable bindings of actual arguments inside an operation and omit alternatives (as with Prolog's cut) if arguments are not bound during the evaluation of the operation. Although this has some similarities with our approach, it is less general. Due to the use of set functions, we can still call deterministic operations with free variables and compute bindings for them in order to cut the search space in computations with individual bindings. Moreover, [13, 29] have strong criteria on operations where dynamic determinism detection is applied (in particular, no extra variables in right-hand sides) so that it is not applicable to most of our examples.

The declarative language Mercury[6] also supports monadic I/O as well as non-deterministic computations. To annotate predicates where only one of possibly several solutions are needed, the user can use committed choice annotations (`cc_nondet`, `cc_multi`) to suppress the computation of several solutions. Since the Mercury compiler checks these annotations, their usage is restricted in contrast to our semantic-based notion of deterministic operations.

A well-known method in logic programming to restrict the search space is the "cut" operator of Prolog, which is also intended to mark deterministic computations and omit the computation of further results. Although our requirement to compute complete values before omitting parts of the computation space looks stronger than the cut operator, it helps to ensure referential transparency, i.e., a strategy-independent interpretation of computations. Compared to Prolog's cut operator, our concept of deterministic operations does not destroy operational completeness. For instance, consider a simplifier for arithmetic expressions which might contain the following clauses:

```
simp(*(0,X),0).
simp(*(X,0),0).
```

Since the goal

```
?- simp(*(0,0),S).
```

yields the binding `S=0` twice, we could be attempted to avoid this superfluous non-determinism by putting a cut at the end of the first clause:

```
simp(*(0,X),0) :- !.
simp(*(X,0),0).
```

This seems to work at first glance but fails if we try to generate expressions that yield a given simplified form. For instance, the goal

```
?- X=1, simp(*(X,Y),0).
```

succeeds, whereas the logical equivalent goal

```
?- simp(*(X,Y),0), X=1.
```

[6] www.mercurylang.org

fails. Trying to move the cut from the clause to the call side does not help. We might try to define different predicates for different modes, but this is not an appropriate solution from a declarative point of view. As we will see later, our deterministic operations "cut" the search space only if there is enough information about the arguments, which avoids the problems sketched above. In this sense, the annotation of deterministic operations corresponds to a safe use of cuts in Prolog, i.e., they correspond to the informal concept of "green cuts" [35].

8 Conclusions

We presented a method to detect relevant non-deterministic operations in Curry applications and proposed the use of deterministic operations to improve their operational behavior. We characterized deterministic operations semantically w.r.t. their input/output behavior, i.e., deterministic operations might yield multiple results under the standard semantics but all results are equal for a given input. We showed that one can exploit this property by cutting the computation space for such operations if the arguments are sufficiently evaluated. In this way, we do not only improve their operational behavior, but one can also avoid run-time problems if these operations are used inside I/O operations, which always require deterministic subcomputations.

We demonstrated with various examples that deterministic operations frequently occur in functional logic programs. Actually, they occur whenever a task like selecting list elements or subterms, or applying transformation rules can be more easily expressed in a non-deterministic manner.

We also discussed how determinism properties can be checked, since they are decidable only in simple cases. One can automatically test these properties with advanced testing tools which might also prove a property if the set of possible argument values is finite. We sketched also proof techniques for determinism annotations. Developing better proof techniques with mechanical support is an interesting topic for future research.

Acknowledgments. This material is based in part upon work supported by the National Science Foundation under Grant No. 1317249.

References

1. E. Albert, M. Hanus, F. Huch, J. Oliver, and G. Vidal. Operational semantics for declarative multi-paradigm languages. *Journal of Symbolic Computation*, 40(1):795–829, 2005.
2. S. Antoy. Definitional trees. In *Proc. of the 3rd International Conference on Algebraic and Logic Programming*, pages 143–157. Springer LNCS 632, 1992.
3. S. Antoy. Constructor-based conditional narrowing. In *Proc. of the 3rd Int. ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP 2001)*, pages 199–206, 2001.
4. S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. *Journal of the ACM*, 47(4):776–822, 2000.
5. S. Antoy and M. Hanus. Declarative programming with function patterns. In *Proc. LOPSTR 2005*, pages 6–22. Springer LNCS 3901, 2005.

6. S. Antoy and M. Hanus. Overlapping rules and logic variables in functional logic programs. In *Proc. ICLP 2006*, pages 87–101. Springer LNCS 4079, 2006.
7. S. Antoy and M. Hanus. Set functions for functional logic programming. In *Proc. PPDP 2009*, pages 73–82. ACM Press, 2009.
8. S. Antoy and M. Hanus. Functional logic programming. *Comm. of the ACM*, 53(4):74–85, 2010.
9. S. Antoy and M. Hanus. Curry without Success. In *Proc. of the 23rd International Workshop on Functional and (Constraint) Logic Programming (WFLP 2014)*, volume 1335 of *CEUR Workshop Proceedings*, pages 140–154. CEUR-WS.org, 2014.
10. S. Antoy and M. Hanus. Default rules for Curry. In *Proc. PADL 2016*, pages 65–82. Springer LNCS 9585, 2016.
11. S. Antoy, M. Hanus, and S. Libby. Proving non-deterministic computations in Agda. In *Proc. 24th Int. Workshop on Functional and Logic Programming (WFLP 2016)*. EPTCS, 2016.
12. B. Braßel, M. Hanus, B. Peemöller, and F. Reck. KiCS2: A new compiler from Curry to Haskell. In *Proc. WFLP 2011*, pages 1–18. Springer LNCS 6816, 2011.
13. R. Caballero and F.J. López-Fraguas. Improving deterministic computations in lazy functional logic languages. *Journal of Functional and Logic Programming*, 2003, 2003.
14. J. Christiansen and S. Fischer. EasyCheck - test data for free. In *Proc. FLOPS 2008*, pages 322–336. Springer LNCS 4989, 2008.
15. K. Claessen and J. Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proc. ICFP 2000*, pages 268–279. ACM Press, 2000.
16. J.M. Cleva, J. Leach, and F.J. López-Fraguas. A logic programming approach to the verification of functional-logic programs. In *Proc. PPDP 2004*, pages 9–19. ACM Press, 2004.
17. N. Dershowitz. Termination of rewriting. *J. Symbolic Computation*, 3(1/2):69–116, 1987.
18. N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B*, pages 243–320. Elsevier, 1990.
19. E.W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.
20. L. Fribourg. Slog: A logic programming language interpreter based on clausal superposition and rewriting. In *Proc. IEEE Int. Symp. on Logic Programming*, pages 172–184, 1985.
21. 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*, 40:47–87, 1999.
22. M. Hanus. Efficient implementation of narrowing and rewriting. In *Proc. Int. Workshop on Processing Declarative Knowledge*, pages 344–365. Springer LNAI 567, 1991.
23. M. Hanus. Declarative processing of semistructured web data. In *Technical Communications of the 27th Int. Conference on Logic Programming*, volume 11, pages 198–208. LIPIcs, 2011.
24. M. Hanus. Functional logic programming: From theory to Curry. In *Programming Logics - Essays in Memory of Harald Ganzinger*, pages 123–168. Springer LNCS 7797, 2013.
25. M. Hanus and F. Skrlac. A modular and generic analysis server system for functional logic programs. In *Proc. PEPM 2014*, pages 181–188. ACM Press, 2014.
26. M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.9.0). Available at `http://www.curry-language.org`, 2016.
27. M. Hanus (et al.). PAKCS: The Portland Aachen Kiel Curry System. Available at `http://www.informatik.uni-kiel.de/~pakcs/`, 2016.
28. H. Hussmann. Nondeterministic algebraic specifications and nonconfluent term rewriting. *Journal of Logic Programming*, 12:237–255, 1992.
29. R. Loogen and S. Winkler. Dynamic detection of determinism in functional logic languages. *Theoretical Computer Science 142*, pages 59–87, 1995.
30. F. López-Fraguas and J. Sánchez-Hernández. TOY: A multiparadigm declarative system. In *Proc. of RTA'99*, pages 244–247. Springer LNCS 1631, 1999.

31. F.J. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández. A simple rewrite notion for call-time choice semantics. In *Proc. PPDP 2007*, pages 197–208. ACM Press, 2007.
32. S. Peyton Jones, editor. *Haskell 98 Language and Libraries—The Revised Report*. Cambridge University Press, 2003.
33. U.S. Reddy. Narrowing as the operational semantics of functional languages. In *Proc. IEEE Internat. Symposium on Logic Programming*, pages 138–151, Boston, 1985.
34. J.R. Slagle. Automated theorem-proving for theories with simplifiers, commutativity, and associativity. *Journal of the ACM*, 21(4):622–642, 1974.
35. L. Sterling and E. Shapiro. *The Art of Prolog*. MIT Press, 1986.
36. A. Stump. *Verified Functional Programming in Agda*. ACM and Morgan & Claypool, 2016.
37. Terese. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2003.

A Simplification of Arithmetic Expressions

The simplification of symbolic arithmetic expressions has been used in [5] to demonstrate the power of functional patterns. Such a simplifier should simplify arithmetic expressions like $1 * (x + 0)$ to x. For this purpose, the structure of arithmetic expressions is defined by

```
data Exp = Lit Int
         | Var String
         | Add Exp Exp
         | Mul Exp Exp
```

In order to replace a subterm at some position, we define an operation `replace` such that the call "`replace` e p t" is equivalent to the notation $e[t]_p$ commonly used in term rewriting [18], where a position p is represented as a sequence of numbers:

```
replace :: Exp  → [Int]  → Exp  → Exp
replace _         []    t = t
replace (Add l r) (1:p) t = Add (replace l p t) r
replace (Add l r) (2:p) t = Add l (replace r p t)
replace (Mul l r) (1:p) t = Mul (replace l p t) r
replace (Mul l r) (2:p) t = Mul l (replace r p t)
```

The various possibilities to simplify an expression e are defined by a non-deterministic operation `evalTo` that evaluates to expressions that are equivalent to e (where only a few possibilities are shown here):

```
evalTo :: Exp  → Exp
evalTo e = Add (Lit 0) e
         ? Add e (Lit 0)
         ? Mul (Lit 1) e
         ? Mul e (Lit 1)
```

Exploiting functional patterns, a one-step simplification operation can be defined as follows:

```
simplifyStep (replace c p (evalTo x)) = replace c p x
```

Thus, if there is a context c and a position p such that the actual argument has the form $c[e]_p$, where e is simplifiable according to the definition of `evalTo`, then this subterm is replaced by its simplified form.

B Proof: `last` is deterministic

This section contains the Agda program which proves that `last` is a deterministic operation. For this purpose, we show that every concatenation used in the pattern of `last` determines the same last element. To be more precise, we show that, if two concatenations produces the same list (i.e., argument of `last`), the selected elements are equal. More formally, we prove:

$\forall l1, l2, x1, x2 : (l1 \mathbin{++} [x1] == l2 \mathbin{++} [x2]) \implies x1 == x2$

This is mechanically proved with the following Agda program (where we omit the module header and imports):

```
-- Auxiliary statements: an empty list cannot be equal to a list with
-- an element at the end:
nonemptylast :  ∀ (l : 𝕃 ℕ)(x : ℕ) → =𝕃 _=ℕ_ [] (l ++ [ x ]) ≡ ff
nonemptylast [] x = refl
nonemptylast (z :: l) x = refl

nonemptylastr :  ∀ (l : 𝕃 ℕ)(x : ℕ) → =𝕃 _=ℕ_ (l ++ [ x ]) [] ≡ ff
nonemptylastr [] x = refl
nonemptylastr (z :: l) x = refl

-- Main property formulated as Boolean implication:
lasteqp : ∀ (l1 l2 : 𝕃 ℕ)(x1 x2 : ℕ)
            → (=𝕃 _=ℕ_ (l1 ++ [ x1 ]) (l2 ++ [ x2 ])) imp (x1 =ℕ x2) ≡ tt
lasteqp [] [] x1 x2 rewrite &&-tt (x1 =ℕ x2) | imp-same (x1 =ℕ x2) = refl
lasteqp [] (x :: l2) x1 x2
  rewrite nonemptylast l2 x2 | &&-ff (x1 =ℕ x) = refl
lasteqp (x :: l1) [] x1 x2
  rewrite nonemptylastr l1 x1 | &&-ff (x =ℕ x2) = refl
lasteqp (z1 :: l1) (z2 :: l2) x1 x2 with (z1 =ℕ z2)
lasteqp (z1 :: l1) (z2 :: l2) x1 x2 | tt
  rewrite lasteqp l1 l2 x1 x2 = refl
lasteqp (z1 :: l1) (z2 :: l2) x1 x2 | ff = refl

-- Main property formulated as propositional implication:
lasteq : ∀ (l1 l2 : 𝕃 ℕ)(x1 x2 : ℕ)
            → =𝕃 _=ℕ_ (l1 ++ [ x1 ]) (l2 ++ [ x2 ]) ≡ tt
            → x1 =ℕ x2 ≡ tt
lasteq l1 l2 x1 x2 p = imp-mp (lasteqp l1 l2 x1 x2) p
```

C Proof: `min` is deterministic

This section contains the Agda program which proves that `min` is a deterministic operation. This is done by proving that a result computed by `min` is identical to the minimum of the list defined in a purely functional manner. Thus, we also show that the non-deterministic specification of the minimum is identical to a more efficient functional implementation. To distinguish the non-deterministic and the deterministic definition of `min`, they are called `min-nd` and `min-d`, respectively, in the proof.

We model the non-determinism by a translation into Agda which is called "planned choices" in [11]. The mechanical proof is non-trivial and requires various lemmas. In particular, the proof is split into showing that the non-deterministically selected element is smaller than, or equal to, all other list elements, but it cannot be strictly smaller than all others since it is itself an element from the list.

```
open import bool

module nd-minlist-is-correct
  (Choice : Set)
  (choose : Choice → 𝔹)
  (lchoice : Choice → Choice)
  (rchoice : Choice → Choice)
 where

open import eq
open import bool-thms
open import bool-thms2
open import nat
open import nat-thms
open import list
open import maybe
open import inspect

----------------------------------------------------------------------
-- Some auxiliaries:

-- We define our own less-or-equal since the standard definition with
-- if-then-else produces too many branches:

_<=_ : ℕ → ℕ → 𝔹
0 <= y = tt
(suc x) <= 0 = ff
(suc x) <= (suc y) = x <= y

-- Some properties about less-or-equal:

<=-refl : ∀ (x : ℕ) → x <= x ≡ tt
<=-refl 0 = refl
<=-refl (suc x) = <=-refl x

<=-trans : ∀ (x y z : ℕ) → x <= y ≡ tt → y <= z ≡ tt → x <= z ≡ tt
<=-trans zero y z p1 p2 = refl
<=-trans (suc x) zero z p1 p2 = 𝔹-contra p1
<=-trans (suc x) (suc y) zero p1 p2 = 𝔹-contra p2
<=-trans (suc x) (suc y) (suc z) p1 p2 = <=-trans x y z p1 p2

<=-< : ∀ (x y : ℕ) → x <= y ≡ ff → y < x ≡ tt
<=-< zero x ()
<=-< (suc x) zero p = refl
<=-< (suc x) (suc y) p = <=-< x y p

<-<= : ∀ (x y : ℕ) → x < y ≡ tt → y <= x ≡ ff
<-<= x zero p rewrite <-0 x = 𝔹-contra p
<-<= zero (suc y) p = refl
<-<= (suc x) (suc y) p = <-<= x y p

<-<=-ff : ∀ (x y : ℕ) → x < y ≡ ff → y <= x ≡ tt
<-<=-ff zero zero p = refl
<-<=-ff zero (suc y) p = 𝔹-contra (sym p)
<-<=-ff (suc x) zero p = refl
<-<=-ff (suc x) (suc y) p = <-<=-ff x y p
```

```
<-<=-trans : ∀ (x y z : ℕ) → x < y ≡ tt → y <= z ≡ tt → x < z ≡ tt
<-<=-trans zero zero z p1 p2 = 𝔹-contra p1
<-<=-trans zero (suc y) zero p1 p2 = 𝔹-contra p2
<-<=-trans zero (suc y) (suc z) p1 p2 = refl
<-<=-trans (suc x) zero z p1 p2 = 𝔹-contra p1
<-<=-trans (suc x) (suc y) zero p1 p2 = 𝔹-contra p2
<-<=-trans (suc x) (suc y) (suc z) p1 p2 = <-<=-trans x y z p1 p2

-----------------------------------------------------------------------
-- More lemmas about ordering relations:

leq-if : ∀ (x y z : ℕ)
      → y <= x && y <= z ≡ (if x <= z then y <= x else y <= z)
leq-if x y z with inspect (y <= x)
leq-if x y z | it tt p1 with inspect (x <= z)
... | it tt p2 rewrite p1 | p2 | <=-trans y x z p1 p2 = refl
... | it ff p2 rewrite p1 | p2 = refl
leq-if x y z | it ff p1 with inspect (x <= z)
... | it tt p2 rewrite p1 | p2 = refl
... | it ff p2 rewrite p1 | p2
           | <-<= z y (<-trans {z} {x} {y} (<=-< x z p2) (<=-< y x p1)) = refl

le-if : ∀ (x y z : ℕ)
     → y < x && y < z ≡ (if x <= z then y < x else y < z)
le-if x y z with inspect (y < x)
le-if x y z | it tt p1 with inspect (x <= z)
... | it tt p2 rewrite p1 | p2 | <-<=-trans y x z p1 p2 = refl
... | it ff p2 rewrite p1 | p2 = refl
le-if x y z | it ff p1 with inspect (x <= z)
... | it tt p2 rewrite p1 | p2 = refl
... | it ff p2 rewrite p1 | p2
         | <-asym {z} {y} (<-<=-trans z x y (<=-< x z p2) (<-<=-ff y x p1))
  = refl

-----------------------------------------------------------------------
-- A lemma relating equality and orderings:

=ℕ-not-le : ∀ (m n : ℕ) → m =ℕ n ≡ ˜ (m < n) && m <= n
=ℕ-not-le zero zero = refl
=ℕ-not-le zero (suc m) = refl
=ℕ-not-le (suc n) zero = refl
=ℕ-not-le (suc n) (suc m) = =ℕ-not-le n m

-----------------------------------------------------------------------
-- This is the translation of the Curry program defining min in
-- a deterministic and a non-deterministic manner:

-- Check whether all elements of a list satisfy a given predicate:
all : {A : Set} → (A → 𝔹) → 𝕃 A → 𝔹
all _ [] = tt
all p (x :: xs) = p x && all p xs

-- Deterministic min-d (the second argument is a proof that
-- the input list is non-empty):
min-d : (l : 𝕃 ℕ) → is-empty l ≡ ff → ℕ
min-d []        ()
min-d (x :: []) _ = x
min-d (x :: y :: xs) _ = let z = min-d (y :: xs) refl
                         in if x <= z then x else z

-- Non-deterministic selection of some element from a list:
select : {A : Set} → Choice → 𝕃 A -> maybe A
select _  []        = nothing
select ch (x :: xs) = if choose ch then just x
                                   else select (lchoice ch) xs

-- Non-deterministically select elements satisfying a property from a list:
select-with : {A : Set} → Choice → (A → 𝔹) → 𝕃 A → maybe A
select-with _  p [] = nothing
select-with ch p (x :: xs) =
  if choose ch then (if p x then just x else nothing)
             else select-with (lchoice ch) p xs

-- Non-deterministic minimum definition:
min-nd : Choice → (xs : 𝕃 ℕ) → maybe ℕ
min-nd ch xs = select-with ch (λ x → all (_<=_ x) xs) xs

----------------------------------------------------------------------

-- Proof of the correctness of the operation select-with:
select-with-correct : ∀ {A : Set}
                   → (ch : Choice) (p : A → 𝔹) (xs : 𝕃 A) (z : A)
                   → select-with ch p xs ≡ just z → p z ≡ tt
select-with-correct ch p [] z ()
select-with-correct ch p (x :: xs) z u with choose ch
select-with-correct ch p (x :: xs) z u | tt with inspect (p x)
select-with-correct ch p (x :: xs) z u | tt | it tt v rewrite v | u | down-≡ u
 = v
select-with-correct ch p (x :: xs) z u | tt | it ff v rewrite v with u
select-with-correct ch p (x :: xs) z u | tt | it ff v | ()
select-with-correct ch p (x :: xs) z u | ff =
  select-with-correct (lchoice ch) p xs z u

----------------------------------------------------------------------
-- First step:
-- if y smaller than all elements, y is smaller than the minimum:
all-leq-min : ∀ (y x : ℕ) (xs : 𝕃 ℕ)
           → all (_<=_ y) (x :: xs) ≡ y <= min-d (x :: xs) refl
all-leq-min y x [] = &&-tt (y <= x)
```

```
all-leq-min y x (z :: zs)
  rewrite all-leq-min y z zs
    | ite-arg (_<=_ y) (x <= min-d (z :: zs) refl) x (min-d (z :: zs) refl)
 = leq-if x y (min-d (z :: zs) refl)

-- Now we can prove:
-- if min-nd selects an element, it is smaller than the minimum:
min-nd-select-min-d : ∀ (ch : Choice) (x : ℕ) (xs : 𝕃 ℕ) (z : ℕ)
      → min-nd ch (x :: xs) ≡ just z → z <= min-d (x :: xs) refl ≡ tt
min-nd-select-min-d ch x xs z u
 rewrite sym (all-leq-min z x xs)
       | select-with-correct ch (λ y → all (_<=_ y) (x :: xs)) (x :: xs) z u
 = refl

----------------------------------------------------------------------
-- Next step: if y smaller than all elements, y is smaller than the minimum:
all-less-min : ∀ (y x : ℕ) (xs : 𝕃 ℕ)
           → all (_<_ y) (x :: xs) ≡ y < min-d (x :: xs) refl
all-less-min y x [] rewrite &&-tt (y < x) = refl
all-less-min y x (z :: zs)
  rewrite all-less-min y z zs
    | ite-arg (_<_ y) (x <= min-d (z :: zs) refl) x (min-d (z :: zs) refl)
 = le-if x y (min-d (z :: zs) refl)

-- Next we prove that the element selected by min-nd cannot be smaller
-- than the minimum.

-- For this purpose, we prove an auxiliary lemma:
-- If an element is selected from a list, it cannot be smaller than all elements
select-with-all<-ff : ∀ (ch : Choice) (p : ℕ → 𝔹) (xs : 𝕃 ℕ) (z : ℕ)
                  → select-with ch p xs ≡ just z → all (_<_ z) xs ≡ ff
select-with-all<-ff ch _ [] z ()
select-with-all<-ff ch p (x :: xs) z u with (choose ch)
select-with-all<-ff ch p (x :: xs) z u | tt with (p x)
select-with-all<-ff ch p (x :: xs) z u | tt | tt rewrite down-≡ u | <-irrefl z
  = refl
select-with-all<-ff ch p (x :: xs) z () | tt | ff
select-with-all<-ff ch p (x :: xs) z u | ff
  rewrite select-with-all<-ff (lchoice ch) p xs z u | &&-ff (z < x) = refl

-- Now we can prove: if min-nd selects an element, it cannot be smaller
-- than all other elements:
min-nd-select-all<-ff : ∀ (ch : Choice) (xs : 𝕃 ℕ) (z : ℕ)
      → min-nd ch xs ≡ just z → all (_<_ z) xs ≡ ff
min-nd-select-all<-ff ch xs z u
 rewrite select-with-all<-ff ch (λ y → all (_<=_ y) xs) xs z u
= refl

----------------------------------------------------------------------
```

```
-- The main theorem: each result of min-nd is the mininum computed by min-d:
min-nd-theorem : ∀ (ch : Choice) (x : ℕ) (xs : 𝕃 ℕ) (z : ℕ)
           → min-nd ch (x :: xs) ≡ just z → z =ℕ min-d (x :: xs) refl ≡ tt
min-nd-theorem ch x xs z u
 rewrite
   =ℕ-not-le z (min-d (x :: xs) refl)  -- split equality into no less and leq
 | min-nd-select-min-d ch x xs z u       -- min-nd selects leq min. elements
 | sym (all-less-min z x xs)             -- less-than min. elements satisfy all<
 | min-nd-select-all<-ff ch (x :: xs) z u --min-nd can't select any all< element
 = refl

------------------------------------------------------------------------
```