Type-based Nondeterminism Checking in Functional Logic Programs*

Michael Hanus and Frank Steiner
Institut für Informatik, Christian-Albrechts-Universität Kiel
Olshausenstr. 40, D-24098 Kiel, Germany
{mh,fst}@informatik.uni-kiel.de

ABSTRACT

Functional logic languages combine nondeterministic search facilities of logic languages with features of functional languages, e.g., monadic I/O to provide a declarative method to deal with I/O actions. Unfortunately, monadic I/O cannot be used in programs which split the computation due to nondeterministic reductions. This problem can be avoided if nondeterministic computations are encapsulated by search operators which are available, for instance, in the multi-paradigm language Curry. To support the programmer in identifying nondeterministic parts of a program, we develop a method based on a type and effect system that will find every possible source of nondeterminism. Additionally, such information can be exploited in compilers to optimize deterministically reducible parts of a program.

1. INTRODUCTION

An important feature of logic languages is the ability to deal with nondeterministic computations to compute solutions for partially instantiated goals. This can lead to problems when using I/O operations, because they are usually not backtrackable. For instance in Prolog, output made by a failing branch of the search tree will remain on the screen and disturb the output of a possibly successful computation. In languages supporting more flexible search strategies instead of backtracking, like Curry [10, 16] or Oz [28], the problem becomes more serious since different branches of a nondeterministic computation might be evaluated concurrently. Thus, different branches of the search tree would compete for the input and output devices.

To provide a clean and declarative method of I/O, one can use the monadic I/O concept [30] which was developed for Haskell [27] and adapted in Curry. In this concept, I/O operations are seen as transformations that act on the outside world, which contains the file system, the Internet etc. Since this world can not be copied, nondeterminism in combination with monadic I/O is not allowed and leads to a run-time error in Curry. To avoid this problem, Curry allows to encapsulate nondeterministic computations [15, 28], thus increasing program stability and safety.

The remaining problem is to detect all possible sources for nondeterminism in a program. This can be very difficult even for small programs and often also depends on the form of queries the user may ask. Thus, our aim is to develop a method to detect all possible sources of nondeterminism.

Additionally, the information computed by our program analysis can be used for compiler optimizations. For instance, instructions for checking function arguments and deciding if the actual call of this function reduces deterministically or not can be eliminated for functions which are proven to reduce deterministically. If larger program parts or the complete program do not cause nondeterminism, code for handling nondeterminism, for instance for spawning new computation branches, can be dropped (*dead code elimination*). Such optimizations can be applied for instance to the Curry2Java compiler [14] which is part of our Curry system PAKCS [17] and compiles Curry programs into code for an abstract machine implemented in Java.

Our analysis is based on a type and effect system (see [23] for an overview) which can be seen as an extension of classical type systems known from functional languages. The basic idea is to annotate function types with some effect to describe the run-time behavior of the function. In our case we annotate the names of those functions as an effect that might cause nondeterministic computations. Then our analysis returns all function names that could split the computation when applied during the evaluation of a certain expression (i.e., the goal to be solved).

Note that existing determinism analyses for (functional) logic languages cannot be directly adapted to Curry because the same function might reduce deterministically or not, depending on its arguments. Thus, we need to derive groundness information for arguments in function calls which is not trivial due to the lazy evaluation mechanism in

*The research described in this paper has been partially supported by the German Research Council (DFG) under grant Ha 2457/1-1.

Curry. Existing analyses for strict languages will indeed fail to analyse Curry programs correctly. Consider the following simple example:

```
f 1 x = x
f 2 x = x+x
g 1 = 3
```

and the call `f x (g x)`. A strict analysis will analyse the call `(g x)` before analysing the call to `f`. Thus, it will consider `x` to be bound to `1` by evaluating `g` and then analyse the call `f 1 3` which would deterministically reduce with the first rule. But in Curry, `(g x)` will be evaluated *after* applying a rule to `f`. Thus, in Curry, `f` will indeed be called with an unbound variable as first argument, which will cause a non-deterministic splitting (see Section 2.1 for an explanation of the reduction mechanisms in Curry). Therefore, analyses for languages like Prolog [29, 5], Mercury [18], or HAL [6] do not apply because they do not deal with lazy evaluation. In contrast, analyses proposed for narrowing-based functional logic languages dealing with lazy evaluation cannot handle residuation, which additionally exists in Curry, and rely on the non-ambiguity condition [21] which is too restrictive for Curry programs. Furthermore, these analyses are either applied during run time (like in Babel [21] and partially in K-Leaf [20]), or are unable to derive groundness information for function calls in arguments (like in K-Leaf).

In the next section we will briefly introduce the language Curry which is the object language of our analysis. Note that the analysis itself should be adaptable also to other (functional) logic languages that suffer from similar problems. In Section 3 the type and effect system will be described together with some examples and an informal description of a type inference algorithm. Section 4 discusses some practical results of our first implementation, and Section 5 contains our conclusions and points out some future work. Due to lack of space, some detailed definitions and the proofs of the results are omitted.

2. OVERVIEW OF CURRY

Curry [10, 16] is a multi-paradigm language combining in a seamless way features from functional programming (nested expressions, lazy evaluation, higher-order functions), logic programming (logical variables, partial data structures, built-in search), and concurrent programming (concurrent evaluation of constraints with synchronization on logical variables). It also amalgamates the most important operational principles developed in the area of integrated functional logic languages: "residuation" and "narrowing" (see [9] for a survey on functional logic programming). Thus, various operational models developed for declarative programs can be seen as restrictions of Curry's computation model (see [10] for a detailed discussion).

A Curry program specifies the semantics of expressions, where goals, which occur in logic programming, are particular constraint expressions. Executing a Curry program means simplifying an expression until a value (or solution) is computed. To distinguish between values and reducible expressions, Curry has a strict distinction between (data) *constructors* and *operations* or *defined functions* on these data. Hence, a Curry program consists of a set of type and function declarations. The type declarations define the computational domains (constructors) and the function declarations the operations on these domains.

Curry combines various features known in declarative programming, like higher-order functions, constraints and the possibility to use constraint solvers for various domains, encapsulated search [15, 28], a Hindley/Milner-like polymorphic type system [4], monadic I/O [30] and features for communication and distributed programming [11]. A detailed description of these features can be found in [16]. In the following we will only outline those which are necessary to understand the ideas of our analysis.

2.1 Basic Features of Curry

Values in Curry are, similarly to functional or logic languages, *data terms* constructed from constants and data constructors. For instance, the datatype declarations

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

introduce the datatype `Bool` with the 0-ary constructors (*constants*) `True` and `False`, and the polymorphic type "`List a`" of lists. Natural numbers, which we will use in the examples, are represented in Curry by constants (`0`, `1`, `2`,...) of type `Int`.

A *data term* is a well-typed expression containing variables, constants and data constructors, e.g., `1:2:xs`. *Functions* operate on data terms. Their meaning is specified by *rules* (or *equations*) of the form "l | c = r" (the condition part "| c" is optional) where l is a *pattern*, i.e., l has the form $f\ t_1 \ldots t_n$ with f being a function symbol, $t_1, \ldots, t_n$ data terms and each variable occurs only once, and r is a well-formed *expression* containing function calls, constants, data constructors and variables from l and c. The *condition* c is a *constraint* which optionally contains a list of locally declared variables, i.e., a constraint can have the form `let` $v_1, \ldots, v_k$ `free in` *con* where the variables v_i are only visible in the constraint *con*. Basic constraints are (strict) equations of the form e_1 `=:=` e_2 which are solvable if e_1 and e_2 are reducible to unifiable data terms. Constraints can be composed by the concurrent conjunction operator `&`, i.e., c_1 `&` c_2 will be evaluated by concurrently evaluating c_1 and c_2. If a local variable v of a condition should be visible also in the right-hand side, the rule is written as l | c =r `where` v `free`. A rule can be applied if its condition is solvable. A *head normal form* is a variable, a constant, or an expression of the form $c\ e_1 \ldots e_n$ where c is a data constructor. A *Curry program* is a set of data type declarations and equations.

EXAMPLE 1. *The following rules define the concatenation of lists, a function for computing the last element of a list and a (partial) square function* `sq`*, which we will use in later examples:*

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

last l | let xs free in append xs [x] =:= l  = x
                                         where x free

sq 1 = 1
sq 2 = 4
```

If the equation "`append xs [x] =:= l`*" is solvable, then* `x` *is the last element of the list* `l`. □

From a functional point of view, we are interested in computing the *value* of an expression, i.e., a data term which is equivalent (w.r.t. the program rules) to the initial expression. In logic languages, we want to solve goals, i.e., compute bindings for free variables in an initial expression. Since Curry integrates these two paradigms, it computes *answer pairs* consisting of a substitution and an expression. Due to the nondeterministic features of Curry, an expression may reduce to more than one answer pair, i.e., a reduction step has the general form[1]

$$e \;\Rightarrow\; \sigma_1, e_1 \;\mid \cdots \mid\; \sigma_n, e_n$$

where $n \geq 0$, e, e_1,…,e_n are expressions, $\sigma_1, \ldots, \sigma_n$ are substitutions on the free variables in e, and "|" joins different alternatives to a disjunction. We call the evaluation step *deterministic* if $n = 1$ and *nondeterministic* if $n > 1$. The case $n = 0$ corresponds to a *failure*.

For selecting the next reducible function call (so called *redex*) in an expression that must be evaluated, Curry uses a combination of residuation and *needed narrowing* [1, 10]. This is, roughly speaking, the combination of lazy evaluation with bindings of uninstantiated variables as demanded by the patterns of left-hand sides in the rules. For instance, consider the function `sq` from Example 1: The function call "`sq 2`" is reduced to the value `4` like in any functional language. However, if the argument is an uninstantiated variable, there are two possibilities:

1. If we evaluate `sq` by residuation (in this case `sq` is called *rigid*), the call `sq x` will suspend until `x` is bound to some constructor term that will allow to choose one of the rules for reduction. This is possible by the concurrent evaluation of constraints where a different thread can bind `x` to some value.

2. If we evaluate `sq` by narrowing, as it will be done in all following examples (in this case `sq` is called *flexible*), we bind `x` to all possible patterns of the left-hand sides and continue with all different computation branches independently:

   `sq x` $\overset{*}{\Rightarrow}$ `{x=1} 1 | {x=2} 4`

   Thus, the call `sq x` causes a nondeterministic step in our computation.

In Curry, constraints are evaluated by narrowing (since they correspond to predicates in logic languages), while non-constraint functions are computed using residuation. This behavior can be easily changed by annotations [16].

To make the pattern matching and the rigid/flexible status of functions explicit, we assume that all functions are defined by one rule which left-hand side contains only variables as arguments and the right-hand side contains case-expressions for pattern matching. Thus, expressions have the following form (we assume that lambda abstractions and local declarations are eliminated by lambda lifting [19]):

$$\begin{aligned} e \;\; ::= \;\; & x \mid f\ e_1 \ldots e_n \mid \texttt{let}\ x\ \texttt{free in}\ e \mid \\ & \texttt{case}\ e\ \texttt{of}\ p_1 : e_1; \ldots; p_n : e_n \mid \\ & \texttt{fcase}\ e\ \texttt{of}\ p_1 : e_1; \ldots; p_n : e_n \mid e_1\ \texttt{or}\ e_2 \end{aligned}$$

f is a constructor or defined function and p_i are flat patterns of the form $C\ x_1 \ldots x_n$ where C is a n-ary data constructor. `case` and `fcase` are the rigid and flexible case distinctions, respectively, and `or` denotes a don't-know alternative between two expressions.

The assumption that each function f is defined by a single rule with left-hand side $f\ x_1 \ldots x_n$ does not restrict the class of Curry programs we can handle, since all function definitions can be transformed into this form (higher-order features can be translated into first-order using Warren's method [31]). For instance, the `sq` function can be rewritten as

```
sq x = (f)case x of 1:1; 2:4
```

where `case` is used if `sq` is defined as rigid (i.e., evaluated by residuation), and `fcase` if it is flexible (i.e., evaluated by narrowing). A precise definition of this transformation can be found in [13].

The binary `or` operator is used to translate functions with overlapping left-hand sides. For instance, the rigid function defined by

```
0 * y = 0
x * 0 = 0
```

is translated into the single rule

```
x * y = (case x of 0:0)  or  (case y of 0:0)
```

Nesting the binary `or` operator allows to encode functions with more than two overlapping left-hand sides.

2.2 Nondeterminism and I/O

Nondeterministic computations are problematic if they appear in programs that use I/O. A declarative treatment of input/output, as implemented in Curry, can be obtained by the *monadic I/O* concept [30]. In this concept, an interactive program is considered as a function computing a sequence of actions which are applied to the outside world. An *action* changes the state of the world and possibly returns a result (e.g., a character read from the terminal). For instance, `getChar` reads a character from the standard input whenever it is executed, i.e., applied to a world. Several I/O actions can be composed, e.g., `getChar` can be composed with the action `putChar` (which writes a character to the terminal) by the sequential composition operator `>>=`, i.e.,

```
getChar >>= putChar
```

is a composed action which prints the character typed in the keyboard to the screen (see [30] for more details).

Since the world cannot be copied (note that the world contains at least the complete file system or the complete Internet in web applications), an interactive program having a disjunction as a result makes no sense. For instance consider the program

```
nonsense file x = writeFile file (sq x)
```

[1] See [16] for a definition of the one step relation $\Rightarrow$.

where `writeFile f e` is an action that writes the value of `e` to the file `f`. If we call `nonsense "dummy" x`, a nondeterministic splitting during the evaluation of `sq x` would result in two independent computation branches, both trying to write different values to the same file. This must obviously be avoided because it could, for instance, lead to an inconsistent file state. Thus, a Curry program that uses I/O actions will result in a run-time error whenever a nondeterministic computation is detected. The goal of this paper is to provide a method to detect such kind of programming errors at compile time, so that they can be avoided by encapsulating all possible search between I/O operations (see [15, 28] for a more detailed description of encapsulated search).

3. THE NONDETERMINISM ANALYSIS

In this section we develop a method based on a non-standard type system to check expressions for possible nondeterministic evaluation steps. First we sketch the ideas behind our nondeterminism analysis before we provide and explain the typing rules in detail.

As we have seen in the examples in Section 2, uninstantiated variables as arguments may cause nondeterministic steps when using narrowing. Hence, we propose a type and effect system that allows us to identify the form of arguments, i.e., if they are ground terms or not. This is a non-trivial task because the arguments can be function calls which are evaluated lazily in Curry. Thus, the type of the arguments cannot be derived from their *actual* form but the analysis must take into account their reduction behavior. We will collect the names of the functions that might split the computation as an *effect* of this computation.

In a first step, the program will be typed, either by hand or by a type inference algorithm (see also Section 3.4). For instance, the function definition

```
f x = sq x
```

could be given the following type:

$$A \overset{\{\texttt{f},\texttt{sq}\}}{\longrightarrow} G$$

This type expresses that `f` takes **A**ny argument and returns a **G**round term but during the application of `f` nondeterministic steps may be raised by the functions `f` and `sq`. This is signalized by the effect $\{\texttt{f}, \texttt{sq}\}$ which is annotated above the arrow.

In a second step, we will analyse an expression with respect to the program and obtain results like "the expression reduces deterministically" or "the reduction of the expression raises a nondeterminism caused by the evaluation of the function f." We will explain this in more detail in the next sections.

3.1 The Types and Effects

Type and effect systems can be seen as an extension of classical type systems known from functional languages (see [22, 23] for details). The basic idea is to extend the type annotation for a function with an effect that may occur during the application of this function. For every expression a type and the effect of its reduction can be approximated. Type and effect systems provide for powerful analyses like exception analysis [26], side effect analysis [22] or communication analysis [23, 24].

Since nondeterminism is mostly caused by uninstantiated variables in arguments of flexible functions, we use a non-standard type system to distinguish between ground terms and any terms. Thus, we define *type expressions* as follows:

$$\tau \;=\; G \;\mid\; A \;\mid\; \tau_1 \ldots \tau_n \to \tau$$

G denotes a ground term and A denotes any term. Since a ground term is also any term, we have subtyping [2], i.e., $G < A$ where $\tau_1 < \tau_2$ means that any term of type τ_1 has also type τ_2. For functions it is important to note that covariance holds for the result type but contravariance for the argument types, i.e., $\tau_1 \to \tau_2 \leq \tau_1' \to \tau_2'$ iff $\tau_1' \leq \tau_1, \tau_2 \leq \tau_2'$. This is easy to understand if we think of smaller types as more precise information. The type $A \to G$ is smaller than $G \to G$ and it is more precise, because it provides information about a larger set of inputs. Thus, the larger the input type, the more precise the information, i.e., the smaller the function type. The opposite holds for the output type, i.e., $A \to G$ is smaller than $A \to A$, because the information about the target set is more precise. Our type system allows several types for one expression. For instance, for a function defined by `f x = x` the two types $G \to G$ and $A \to A$ are correct but not related by subtyping.

As an *effect* we want to collect the names of functions which application can cause nondeterministic computations. Therefore, we define the effect φ by

$$\varphi \subseteq \mathcal{F} \cup \{case, or\}$$

where $\mathcal{F}$ is the set of function symbols of a program P. The effect annotated for a function declaration must contain all function names that might reduce in a nondeterministic way during the application of this function. For instance, a type annotation for a function f could be

$$f :: A \overset{\{f,g,h\}}{\longrightarrow} A$$

if in the right-hand side of f a call to the function g causes a nondeterminism, whereas g splits the computation because it calls the function h in its body which raises the nondeterminism. Additionally, f could have another type, for instance

$$f :: G \overset{\emptyset}{\to} G$$

if f does *not* split the computation when applied to a *ground term.*

The aim of our analysis is to calculate type judgements

$$E \vdash e :: \tau/\varphi$$

for an expression e with respect to a program P, where E is a type environment, i.e., a set of type annotations for function and constructor symbols from P and for variables. $E \vdash e :: \tau/\varphi$ means that the expression e has type τ, and during the reduction of e the effect φ may occur. For instance, the type judgement

$$E \vdash f\ x :: A/\{f, g, h\}$$

for the expression $f\ x$ reflects the nondeterministic behaviour of f by returning a non-empty effect. In contrast, the second type $G \xrightarrow{\emptyset} G$ could be used to analyze f 1 and the computed type judgement would be

$$E \vdash f\ 1 :: G/\emptyset,$$

signalizing that the call f 1 reduces deterministically (because function symbols that could split the computation were not found).

The soundness of our approximation, which will be formalized at the end of Section 3.2, ensures that we find every function which can raise a nondeterminism and that terms identified as ground by our analysis will actually reduce to ground terms. Similarly to most program analyses, we cannot compute precise information for all program parts but only approximations of the real program behavior. In our case this means that the computed type might be too imprecise, e.g., A instead of G, and the effect might be too large, i.e., it might contain functions that will never raise a nondeterminism. This imprecision can be reduced by refining the type domain (see Section 5). Nevertheless, our first implementation has shown that the analysis is already quite accurate even for larger examples which occur in practice and especially for purely functional programs (cf. Section 4).

3.2 The Typing Rules

Figure 1 shows the typing rules for our analysis which define the type judgements $E \vdash e :: \tau/\varphi$.

The DECL rule defines when a program rule r is correctly typed w.r.t. a type annotation $\mathcal{A}$ (i.e., $E \vdash_{\mathcal{A}} r$): The type annotation $\mathcal{A}$ for the function f must be given in the environment E, and with the corresponding types for the arguments x_i added to the environment[2] the derived type for the right-hand side must match the type of f. For the effect, two cases are distinguished: If $\varphi \neq \emptyset$ then the right-hand side e possibly raises a nondeterministic computation step. Thus, f could also reduce nondeterministically and so the annotated effect must include φ and $\{f\}$. Otherwise, if $\varphi = \emptyset$, ψ is completely unrestricted. In both cases ψ can contain arbitrary function symbols from the program, because, due to subeffecting (see below), it is still a safe approximation to annotate a larger effect.

The VAR rule is common. If a type is given for x in the environment, this type and an empty effect can be derived. Note that x can be a variable, a function symbol, or a constructor symbol.

The NEWVAR rule handles local (existentially quantified) variables. The newly introduced variable x_{fresh} is uninstantiated, and thus must be given the type A for analysing e. By $e[x/t]$ we denote the replacement of all free occurrences of x in e by t. An occurrence of a variable x in e is free if it does not occur inside a subterm e' of e with $e' =$ `let` x `free in` $\tilde{e}$. A variable x occurs free in e if at least one occurrence of x in e is free.

[2] $E[x_1 :: \tau_1, \dots, x_n :: \tau_n]$ denotes the type environment obtained from E by deleting all existing type annotations for $x_1, \dots, x_n$ and adding the new type annotations in the square brackets.

When applying a function f to some e_i, the effect φ annotated for f might occur. Thus, in the APP rule, φ is returned together with the effects derived for the arguments.

The FCASE rule handles nondeterminism caused by narrowing. If the type A is derived for e and more than two case branches exist, a non-empty effect must be returned to signalize the potential nondeterminism. This is ensured by adding the keyword *case* (which could be indexed by the function name from the left-hand side, as we will do it in subsequent examples) to the effect. Additionally, all effects from the right-hand sides must be collected. Note that for analysing the right-hand sides, the pattern variables are given the type of e: If e reduces to a ground term, so will the terms that the pattern variables are instantiated to. Otherwise, they could be uninstantiated in the subsequent computation and therefore they have the type A. If only one right-hand side has type A, the entire fcase-expression might return this type. Thus, the maximum type of all right-hand sides (w.r.t. the ordering $G < A$) is the type of the fcase-expression.

The CASE rule analyses rigid functions. Thus, the case-expression itself will never split the computation and no additional effect is returned.

Nondeterminism which is caused by overlapping left-hand sides is handled by the OR rule, where the keyword *or* is returned. Like in the (F)CASE rules, the effects of the right-hand sides are collected and the maximal type is returned. Note that by using *case* and *or* keywords, nondeterminism caused by narrowing and by overlapping left-hand sides can be distinguished.

The last rule, SUB, defines subtyping and subeffecting [23]. For our type system, the rule expresses that any expression of type G is also of type A, and that every effect φ can be enlarged without loosing the safety condition. The approximation just becomes more imprecise.

In order to show the correctness of the typing rules, we must define *correct type environments.*

DEFINITION 1. *Let P be a Curry program and E a type environment which contains at least one type annotation for each function and constructor occurring in P.*

1. *Let $r \in P$ be a program rule with left-hand side $f\ x_1 \dots x_n$ and $E_f \subseteq E$ the set of all type annotations for f in E. r is correctly typed w.r.t. E, denoted by $E \vdash r$, iff $E \vdash_{\mathcal{A}} r$ for all $\mathcal{A} \in E_f$.*
2. *E is a correct type environment for P iff $E \vdash r$ for all $r \in P$, and for all type annotations $c :: \tau_1 \dots \tau_n \xrightarrow{\varphi} \tau \in E$, where c is an n-ary constructor, $\tau_i = A$ implies $\tau = A$.*

Thus, a correct type environment E for a program must contain at least one type for each defined function and constructor. Otherwise, the rules that use these constructors and functions cannot be correctly typed. Note that for a

Typing of rules:

$$\text{DECL}\quad \frac{E[x_1::\tau_1,\ldots,x_n::\tau_n]\vdash e::\tau/\emptyset}{E\vdash_{\mathcal{A}} f\ x_1\ldots x_n\ =\ e}\quad \text{if } \mathcal{A}=f::\tau_1\ldots\tau_n\xrightarrow{\psi}\tau\in E$$

$$\text{DECL}\quad \frac{E[x_1::\tau_1,\ldots,x_n::\tau_n]\vdash e::\tau/\varphi}{E\vdash_{\mathcal{A}} f\ x_1\ldots x_n\ =\ e}\quad \text{if } \varphi\neq\emptyset,\ \mathcal{A}=f::\tau_1\ldots\tau_n\xrightarrow{\psi\cup\varphi\cup\{f\}}\tau\in E$$

Typing of expressions:

$$\text{VAR}\quad \frac{}{E\vdash x::\tau/\emptyset}\quad \text{if } x::\tau\in E$$

$$\text{NEWVAR}\quad \frac{E[x_{fresh}::A]\vdash e[x/x_{fresh}]::\tau/\varphi}{E\vdash \texttt{let}\ x\ \texttt{free in}\ e\ ::\ \tau/\varphi}$$

$$\text{APP}\quad \frac{E\vdash e_1::\tau_1/\varphi_1\ \ldots\ E\vdash e_n::\tau_n/\varphi_n\qquad E\vdash f::\tau_1\ldots\tau_n\xrightarrow{\varphi}\tau/\emptyset}{E\vdash f\ e_1\ldots e_n\ ::\ \tau/\bigcup_i\varphi_i\cup\varphi}$$

$$\text{FCASE}\quad \frac{E\vdash e::\tau/\varphi\qquad E[\overline{x_{1m}}::\tau]\vdash e_1::\tau_1/\varphi_1\ \ldots\ E[\overline{x_{nm}}::\tau]\vdash e_n::\tau_n/\varphi_n}{E\vdash \texttt{fcase}\ e\ \texttt{of}\ p_1(\overline{x_{1m}}):e_1;\ldots p_n(\overline{x_{nm}}):e_n\ ::\ max_i(\tau_i)/\bigcup_i\varphi_i\cup\varphi\cup\varphi'}$$

$$\text{where } \varphi'=\begin{cases}\{case\} & \text{if } \tau=A \text{ and } n>1\\ \emptyset & \text{otherwise}\end{cases}$$

$$\text{CASE}\quad \frac{E\vdash e::\tau/\varphi\qquad E[\overline{x_{1m}}::\tau]\vdash e_1::\tau_1/\varphi_1\ \ldots\ E[\overline{x_{nm}}::\tau]\vdash e_n::\tau_n/\varphi_n}{E\vdash \texttt{case}\ e\ \texttt{of}\ p_1(\overline{x_{1m}}):e_1;\ldots p_n(\overline{x_{nm}}):e_n\ ::\ max_i(\tau_i)/\bigcup_i\varphi_i\cup\varphi}$$

$$\text{OR}\quad \frac{E\vdash e_1::\tau_1/\varphi_1\qquad E\vdash e_2::\tau_2/\varphi_2}{E\vdash \texttt{or}(e_1,e_2)\ ::\ max(\tau_1,\tau_2)/\varphi_1\cup\varphi_2\cup\{or\}}$$

$$\text{SUB}\quad \frac{E\vdash e::\tau/\varphi}{E\vdash e::\tau'/\varphi'}\quad \text{if } \tau\leq\tau',\varphi\subseteq\varphi'$$

Note: $\overline{x_{ij}}$ denotes the sequence $x_{i1},\ldots,x_{ij_i}$.

Figure 1: Typing rules

constructor c its result type is always the maximum of its input types (i.e., 0-ary constructors could always have type G), and it is always correct to annotate $\varphi=\emptyset$ because a constructor itself is not reduced (only its arguments) and so does never raise a nondeterministic reduction step.

In the following we use E_e as an abbreviation for $E\,[x_1::A,\ldots,x_n::A]$ where $\{x_1,\ldots,x_n\}$ is the set of variables which occur free in the expression e.

The following lemma states the correctness of the typing rules w.r.t. a single reduction step.

Lemma 1 (Subject reduction). *Let E be a correct type environment for a Curry program and e an expression. If $E_e\vdash e::\tau/\varphi$ and there is a reduction step $e\Rightarrow\sigma_1,e_1\mid\cdots\mid\sigma_n,e_n$ with $n>0$, then $E_{e_i}\vdash e_i::\tau/\varphi$.*

Thus, the type and effect of an expression is invariant under reduction steps. We can easily prove the following correctness results for our framework with this important property:[3]

Theorem 1 (Correctness of typing rules). *Let E be a correct type environment for a Curry program, e an expression and $E_e\vdash e::\tau/\varphi$.*

1. *If $\tau=G$ and e reduces in finitely many steps to a* value *v (i.e., a term without defined function symbols), then v is a ground term.*
2. *If e reduces in finitely many steps to an expression $\tilde{e}$ and $\tilde{e}\Rightarrow\sigma_1,e_1\mid\cdots\mid\sigma_n,e_n$ with $n>1$, then φ contains "case" or "or" depending on the redex of $\tilde{e}$ which caused the nondeterminism.*

From the second property we can directly conclude that e reduces deterministically if $\varphi=\emptyset$.

Thus, we know that every function that might raise a nondeterministic computation step during the reduction of e will be collected in the effect φ by our typing rules, i.e., we will never miss such a function[4]. This is the meaning of a

[3]The reduction semantics of Curry can be found in [16] and (concerning case-expressions) in [13].

[4]Note that the occurrence of *case* (*or*) in the effect guarantees that also the name of the function, in which body the *case* (*or*) expression occurs, is collected in the effect due to the DECL rule. Thus, we know which function definitions to consider for detecting the source of nondeterminism.

safe approximation for our analysis. The same holds for the type, i.e., a term analysed as ground will indeed reduce to a ground term (if its reduction successfully terminates).

3.3 Examples

We want to clarify the ideas of our analysis by providing some simple examples. If the type environment E contains only one type annotation $\mathcal{A}$ for the function in the left-hand side of the rule r, we simply write $E \vdash r$ instead of $E \vdash_{\mathcal{A}} r$.

EXAMPLE 2. *We want to show that the rule*

$$\texttt{f :: } A \overset{\emptyset}{\to} G$$
```
f x = 0
```

is correctly typed. The type expresses that `f` *will reduce deterministically and return a ground term regardless of its input argument. It should be obvious that this type is correct. The initial type environment contains the type for* `f` *and the constructor* `0`*:*

$$E = \{\texttt{f} :: A \overset{\emptyset}{\to} G,\ \texttt{0} :: G\}$$

The correctness of the type is proven by the following derivation:

$$\dfrac{\dfrac{}{E[\texttt{x} :: A] \vdash \texttt{0} :: G/\emptyset}\ \mathit{VAR}}{E \vdash \texttt{f x = 0}}\ \mathit{DECL}$$

With E extended by $\texttt{x} :: A$ according to the type annotation for `f`*, the analysis of the right-hand side returns the ground type and an empty effect, thus matching the target type of* `f` *and the empty effect annotated with* `f`*. Note that due to subtyping and subeffecting, there are more correct types for* `f`*, e.g., $A \overset{\emptyset}{\to} A$, $G \overset{\emptyset}{\to} G$, $G \overset{\emptyset}{\to} A$ and any of these types with every non-empty effect. However, all these types are larger than the most precise type $A \overset{\emptyset}{\to} G$.*

Another simple example is the identity function where the analysis of the right-hand side depends on the results from the left-hand side:

$$\texttt{id :: } G \overset{\emptyset}{\to} G$$
```
id x = x
```

The following derivation, using $E = \{\texttt{id} :: G \overset{\emptyset}{\to} G\}$ as initial environment, proves that the type annotation is correct:

$$\dfrac{\dfrac{}{E[\texttt{x} :: G] \vdash \texttt{x} :: G/\emptyset}\ \mathit{VAR}}{E \vdash \texttt{id x = x}}\ \mathit{DECL}$$

Because E is extended by the type for the input argument of id *($x :: G$), we can derive the type G for the right-hand side, too.* □

Due to space limitations, we do not mark the rules with their names anymore in the following examples.

EXAMPLE 3. *To study an example where nondeterminism occurs, we use the function* `sq` *which was defined in Section 2:*

$$\texttt{sq :: } G \overset{\emptyset}{\to} G$$
$$\texttt{sq :: } A \xrightarrow{\{\texttt{sq}, case_{\texttt{sq}}\}} G$$
```
sq x = fcase x of 1:1; 2:4
```

Two types are defined for `sq`*, claiming that the function will reduce deterministically when applied to a ground term, but might raise a nondeterministic step otherwise. The initial environment contains the two types together with the types for the constructors (we identify the two types by $\mathcal{A}_1$ and $\mathcal{A}_2$):*

$$\begin{aligned} E \ = \ & \{\mathcal{A}_1 :\ \texttt{sq} :: G \overset{\emptyset}{\to} G,\ \mathcal{A}_2 :\ \texttt{sq} :: A \xrightarrow{\{\texttt{sq}, case_{\texttt{sq}}\}} G, \\ & \texttt{1} :: G,\ \texttt{2} :: G,\ \texttt{4} :: G\} \end{aligned}$$

Since we have two types for `sq`*, we must verify two cases of the DECL rule, one for each type (we drop the VAR derivation for the constructor* `4` *because it is the same as for* `1`*):*

$$\dfrac{\dfrac{\dfrac{}{E[\texttt{x} :: G] \vdash \texttt{x} :: G/\emptyset} \quad \dfrac{}{E[\texttt{x} :: G] \vdash \texttt{1} :: G/\emptyset}}{E[\texttt{x} :: G] \vdash \texttt{fcase x of 1:1;2:4} :: G/\emptyset}}{E \vdash_{\mathcal{A}_1} \texttt{sq x = fcase x of 1:1; 2:4}}$$

$$\dfrac{\dfrac{\dfrac{}{E[\texttt{x} :: A] \vdash \texttt{x} :: A/\emptyset} \quad \dfrac{}{E[\texttt{x} :: A] \vdash \texttt{1} :: G/\emptyset}}{E[\texttt{x} :: A] \vdash \texttt{fcase x of 1:1;2:4} :: G/\{case_{\texttt{sq}}\}}}{E \vdash_{\mathcal{A}_2} \texttt{sq x = fcase x of 1:1; 2:4}}$$

Using the first type of `sq`*,* `x` *has type G and therefore the FCASE rule returns an empty effect for the right-hand side, thus matching the type of* `sq`*. With the second type, E is extended by $\texttt{x} :: A$, causing the FCASE rule to return a non-empty effect. Therefore, the DECL rule demands the effect annotated with* `sq` *to contain* `sq` *as well as the effect from the right-hand side, which is satisfied. Thus, the rule is correctly typed because it is correctly typed w.r.t. to both type annotations.*

In contrast, the type annotation $\texttt{sq} :: A \xrightarrow{\emptyset} G$, which claims that `sq` *will reduce in a deterministic way even if we pass any term as argument, is wrong (cf. Section 2.1). This is indeed detected by our analysis:*

$$\texttt{sq :: } A \overset{\emptyset}{\to} G$$
```
sq x = fcase x of 1:1; 2:4
```

Given the initial environment $E = \{\texttt{sq} :: A \overset{\emptyset}{\to} G,\ \texttt{1} :: G,\ \texttt{2} :: G,\ \texttt{4} :: G\}$, the analysis behaves correctly in refuting this type:

$$\dfrac{\dfrac{\dfrac{}{E[\texttt{x} :: A] \not\vdash \texttt{x} :: G/\emptyset} \quad \dfrac{}{E[\texttt{x} :: A] \vdash \texttt{1} :: G/\emptyset}}{E[\texttt{x} :: A] \not\vdash \texttt{ case x of 1:1;2:4} :: G/\emptyset}}{E \not\vdash \texttt{sq x = case x of 1:1;2:4}}$$

Since we cannot derive the type G for `x` *after E has been extended by $\texttt{x} :: A$, the FCASE rule is not able to analyse an empty effect for the right-hand side of* `sq`*. But this would be necessary to match the empty effect annotated for* `sq`*.* □

The next example shows how to analyse function applications.

EXAMPLE 4. *We define a simple function* `f` *that calls* `sq`*, assuming that the two correct types for* `sq` *from Example 3 are specified in the initial environment.*

$$\texttt{f} :: G \xrightarrow{\emptyset} G$$
$$\texttt{f} :: A \xrightarrow{\{\texttt{f},\texttt{sq},case_{\texttt{sq}}\}} G$$

```
f x = sq x
```

$$\texttt{sq} :: G \xrightarrow{\emptyset} G$$
$$\texttt{sq} :: A \xrightarrow{\{\texttt{sq},case_{\texttt{sq}}\}} G$$

```
sq x = fcase x of 1:1; 2:4
```

We drop the types for `sq` *and the natural numbers in* E *to keep it small:*

$$E = \{\mathcal{A}_1 :\ \texttt{f} :: G \xrightarrow{\emptyset} G,\ \mathcal{A}_2 :\ \texttt{f} :: A \xrightarrow{\{\texttt{f},\texttt{sq},case_{\texttt{sq}}\}} G\}$$

We must show the correctness of both types for the rule to be correctly typed:

$$\cfrac{\cfrac{\cfrac{}{E[\texttt{x} :: G] \vdash \texttt{x} :: G/\emptyset} \quad \cfrac{}{E[\texttt{x} :: G] \vdash \texttt{sq} :: G \xrightarrow{\emptyset} G/\emptyset}}{E[\texttt{x} :: G] \vdash \texttt{sq x} :: G/\emptyset}}{E \vdash_{\mathcal{A}_1} \texttt{f x = sq x}}$$

Since $\texttt{x} :: G$ *is added to the environment to match the argument type of* `f`*, we can derive the result type* G *and an empty effect for* `sq x` *by selecting the first type annotation for* `sq` *(cf. Example 3).*

If, according to the annotation $\mathcal{A}_2$*, the type* A *is assigned to* `x`*, the second type annotation for* `sq` *matches and we derive a non-empty effect that is consistent with the annotation for* `f`*:*

$$\cfrac{\cfrac{\cfrac{}{E[\texttt{x} :: A] \vdash \texttt{x} :: A/\emptyset} \quad \cfrac{}{E[\texttt{x} :: A] \vdash \texttt{sq} :: A \xrightarrow{\{\texttt{sq},case_{\texttt{sq}}\}} G/\emptyset}}{E[\texttt{x} :: A] \vdash \texttt{sq x} :: G/\{\texttt{sq}, case_{\texttt{sq}}\}}}{E \vdash_{\mathcal{A}_2} \texttt{f x = sq x}}$$

□

So far, we have only shown how to verify given type annotations for program rules. Usually, the first step in analysing a program will be to find such annotations, either by guessing and checking them as shown above, or by using a type inference algorithm. In the second step, we analyse expressions to be evaluated w.r.t. the program. Very often we might be interested only in one expression, i.e., a `main` function, that starts all calculations. Note that we are finished after the first step, if the `main` functions is unparameterized: If the behaviour of the `main` function does not depend on arguments, then the type annotation inferred for `main` in the first step specifies the runtime behaviour completely.

If we want to evaluate any other expression (e.g., a call to a parameterized `main` function), we must derive its type. Note that we need only the VAR, NEWVAR, APP and SUB rules in this case, because we only have to analyse function/constructor applications. The right hand sides of the rules are not considered anymore, because all information we need is stored in the type annotation of the function.

EXAMPLE 5. *We want to evaluate the expressions* `f (sq 1)` *and* `f (sq x)` *w.r.t. the program and environment from Example 4. For the first expression, the analysis computes the following:*

$$\cfrac{\cfrac{\cfrac{}{E \vdash \texttt{1} :: G/\emptyset} \quad \cfrac{}{E \vdash \texttt{sq} :: G \xrightarrow{\emptyset} G/\emptyset}}{E \vdash \texttt{sq 1} :: G/\emptyset} \quad \cfrac{}{E \vdash \texttt{f} :: G \xrightarrow{\emptyset} G/\emptyset}}{E \vdash \texttt{f (sq 1)} :: G/\emptyset}$$

Since the argument `sq 1` *of* `f` *reduces to a ground term, the first type annotation for* `f` *is selected and an empty effect is returned. Thus, from the type judgement for* `f (sq 1)` *we conclude that the expression will reduce to a ground term without splitting the computation. For the second expression, the initial environment is extended to contain the type* `A` *for all free variables. Then, with* $E_x := E[x :: A]$*, the result is different:*

$$\cfrac{\cfrac{\cfrac{}{E_x \vdash \texttt{x} :: A/\emptyset} \quad \cfrac{}{E_x \vdash \texttt{sq} :: A \xrightarrow{\{\texttt{sq},case_{\texttt{sq}}\}} G/\emptyset}}{E_x \vdash \texttt{sq x} :: G/\{\texttt{sq}, case_{\texttt{sq}}\}} \quad \cfrac{}{E \vdash \texttt{f} :: G \xrightarrow{\emptyset} G/\emptyset}}{E_x \vdash \texttt{f (sq x)} :: A/\{\texttt{sq}, case_{\texttt{sq}}\}}$$

Here the effect of the computed type judgement is not empty. Thus, we are warned that nondeterminism might (and in this case does) occur during the evaluation of this expression. It is important to notice that `f` *is not contained in the computed effect, because the nondeterminism is not caused by the code of the right-hand side of* `f` *but by the evaluation of the argument* `sq x`*. First,* `f` *is reduced with a non-ground argument to its right-hand side* `sq (sq x)`*. But before the outer call of* `sq` *can reduce further, it must evaluate the argument* `sq x`*. This evaluation splits the computation, but returns only ground terms. Therefore, when the outer* `sq` *is finally reduced, its formerly non-ground argument has become ground, and so the right-hand side is evaluated as if* `f` *was called with a ground term. Thus, not* `f`*, but its argument is the real cause for the nondeterminism, and the APP rule correctly returns only the effect collected from the argument. In this way, the effect calculated for (possibly nested) function calls gives quite precise information about the origin of the nondeterminism.*

□

Finally we show that our analysis will detect the problematic behaviour of our motivating example, i.e., the `nonsense` function.

EXAMPLE 6. *The* `nonsense` *function was defined as*

```
nonsense file x = writeFile file (sq x)
```

We already know the type annotations for `sq`*. The smallest correct type for* `writeFile` *is* $A\ A \xrightarrow{\emptyset} G$*, because writing to a file will not split a computation, and* `writeFile` *is defined* rigid *(cf. Section 2.1). Thus, it will wait until its arguments are instantiated and after writing the file it will return a special, internal I/O constructor which is a ground term. With the type for* `writeFile` *given*[5]*, the following is a correct*

[5]The type of certain I/O operations cannot be derived, be-

type environment:

$$
\begin{aligned}
E \;=\; \{ & \texttt{writeFile} :: A\ A \xrightarrow{\emptyset} G, \\
& \texttt{sq} :: G \xrightarrow{\emptyset} G,\ \texttt{sq} :: A \xrightarrow{\{\texttt{sq}, case_{\texttt{sq}}\}} G, \\
& \texttt{1} :: G,\ \texttt{2} :: G,\ \texttt{4} :: G, \\
& \texttt{nonsense} :: A\ G \xrightarrow{\emptyset} G, \\
& \texttt{nonsense} :: A\ A \xrightarrow{\{\texttt{nonsense}, \texttt{sq}, case_{\texttt{sq}}\}} G \}
\end{aligned}
$$

Now we analyse the call `nonsense "dummy" x` *with* $E_x := E[x :: A]$. *We must chose the second type for* `nonsense` *from* E_x *to match the type* A *for the parameter* x*:*

$$
\frac{\dfrac{}{E_x \vdash \texttt{nonsense} :: A\ A \xrightarrow{\{\texttt{nonsense}, \texttt{sq}, case_{\texttt{sq}}\}} G} \qquad \dfrac{}{E_x \vdash \texttt{x} :: A/\emptyset}}{E_x \vdash \texttt{nonsense "dummy" x} :: G/\{\texttt{nonsense}, \texttt{sq}, case_{\texttt{sq}}\}}
$$

Thus, the nondeterminism is detected through the non-empty effect, and a warning due to the combination with I/O functions can be generated. □

3.4 Type Inference

The goal of this work is the development of a correct method to derive information about possible nondeterministic computations in a program. In the previous sections we have shown how we can check given annotations for a program (and afterwards derive type judgements for expressions which should be evaluated w.r.t. to the program). But for the practical application it is tedious to add explicitly all type annotations for the program rules, which demands for a method to infer them automatically. Due to lack of space, we cannot present it in detail but we sketch the basic ideas to construct an inferencer for our type and effect system.

Basically, an inference algorithm can be constructed following the standard techniques for polymorphic type inference [4], i.e., the typing rules in Figure 1 can be also used for type inference by providing a new type variable for the type of each syntactic entity whose type is not yet known. Then the type analysis of an expression leads to the generation of a set of constraints between type expressions to be solved. Without subtyping, these type constraints can be solved by a standard unification procedure for type terms [4]. The only problem is the subtyping rule SUB which is not inductive on the syntax of expressions. For the purpose of type inference, this rule can be eliminated by considering the possibility of subtyping in other rules. In particular, the VAR is changed so that for $x :: \tau \in E$ the type/effect $\alpha/\emptyset$ is inferred for x, where α is a new type variable, and the new subtype constraint $\tau \leq \alpha$ is generated. In this way, the type inference generates a set of equations and inequations between type expressions. In the same pass, all effects can be inferred but here we must allow *conditional effects* of the form "$\alpha \Rightarrow case$" to consider the fact that an effect in the FCASE rule depends on the groundness of the fcase-argument. Such a conditional effect is equivalent to the effect *case* if α is A and to the empty effect if α is G.

Since our subtype structure is very simple ($G \leq G$, $G \leq A$, $A \leq A$, and the contra/covariance rules for function types), inequations between types can be solved by known methods for type inference in the presence of subtyping between basic types [7]: after transforming all inequations into inequations between basic types (by applying the contra/covariance rules for function types), we instantiate the free type variables to their least possible types in order to compute a minimal type and solve all conditional effects.

---

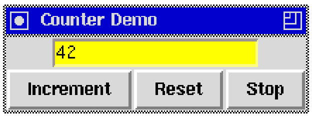
cause they are realized by external functions, i.e., their code is not visible. The annotations for external functions can be specified in a prelude and will be accepted by a type checker and inferencer.

4. PRACTICAL RESULTS

An implementation of the type inferencer is actually under construction, whereas a type checker (following the rules in figure 1) has already been implemented in Curry itself and is available from the authors. All examples described in this paper are correctly analysed by this type checker. Moreover, we have analysed a large set of typical examples, including complex structures like graphical user interfaces, and have received very accurate results.

Of course, there are situations where the analysis will produce imprecise results but the program structures causing such impreciseness are seldom in real applications. For instance, passing an argument like `[x]` to a list-processing function will cause the analysis to consider the argument of type A due to the free variable `x`. Thus, for a function like `append`, which has the types $G\ G \xrightarrow{\emptyset} G$ and $A\ G \xrightarrow{\{\texttt{append}, case_{\texttt{append}}\}} A$ (among others), the analysis will derive a possible nondeterministic behaviour for the call `append [x] []`. Such impreciseness occurs only if a flexible function, i.e., evaluated by narrowing, is considered, its arguments contain free variables inside data structures and these free variables will not cause a nondeterminism later on. Note that especially the last condition will not hold in most practical examples. For instance, analysing the expression `append (1:2:xs) []` will precisely report a nondeterminism, because due to the definition of `append` the free variable `xs` will be passed as first argument to `append` in a recursive call and indeed split the computation. The latter (precisely analysed) situation is more likely to appear, for instance in logic programs, where free variables occur in goals and narrowing is used to search for solutions. On the other hand, the concatenation of two lists containing variables as elements (or similar situations) occurs very seldom in the large set of Curry examples which we have studied.

Even in examples where free variables are used in data structures and bound by narrowing, nondeterminism does not necessarily occur. For instance, to implement graphical user interfaces (GUIs) in a high-level declarative style in Curry [12], the functional features are exploited to define the graphical structure, while the logical features are used to specify the logical dependencies of an interface. Figure 2 shows a simple example (a counter GUI). Note that the GUI specification is passed as a partially instantiated data structure in the second parameter of `runWidget`. In this data structure, the unbound variable `val` is used as a reference (`TkRef val`) to connect the entry field (`TkEntry`), initially containing "42" (`TkText "42"`) with the buttons




```
runWidget "Counter Demo"
  (TkCol [] [
    TkEntry [TkRef val, TkText "42"],
    TkRow [] [TkButton (tkUpdate incrText val) [TkText "Increment"],
              TkButton (tkSetValue val "0")    [TkText "Reset"],
              TkButton tkExit                  [TkText "Stop"]]])
    where val free
```

Figure 2: A specification of a counter GUI

`Increment` and `Reset`. When creating the GUI, the variable `val` will be set to an internal value pointing to the entry field. Now by pressing the `Increment` button, the function call `(tkupdate incrText val)` will be executed to update the value in the counter window by increasing it. At this point, the reference variable `val` is used to identify the field where the value to be increased can be found and where the increased value should be written to. Thus, `val` will be uniquely bound to a pointer value during the setup of the GUI.

The GUI library is completely implemented in Curry [12]. Therefore, programs containing GUI specifications can be analysed together with the GUI library by our nondeterminism analysis. Since the reference variables contained in GUI structures are **uniquely** bound by a narrowing step, our analysis precisely verifies that the program parts handling GUI specifications are deterministic.

This example shows the importance of partially instantiated data structures which combine in a very powerful way functional and logic features, in this case by providing a much more declarative specification for GUIs than in other approaches in functional languages [3]. Unfortunately, it is often difficult to implement these data structures in mode-based logic languages which restrict themselves to just `in` and `out` modes. Even in Mercury, where partially instantiated data structures can be handled in principle, it would require quite difficult and complex structures of nested, parameterized mode declaration to describe the GUI data type structures. Moreover, it is for instance not possible to specify in Mercury that a function works on a list containing ground terms and free variables at the same time, e.g., `[x,2]`. Similar but much more complex situations occur in our Tcl/Tk library. Thus, we could not rely on a mode based analysis to handle any of our GUI-based programs.

A subclass of Curry programs, for which our analysis computes very accurate results, are purely functional subcomputations, which take large parts of most functional logic programs in practice. We consider a computation as purely functional if it behaves like a Haskell computation, i.e., there are no unbound variables at run time and functions are defined so that at most one rule is applicable. In our framework, we can characterize a class of such programs as follows. We call a program *purely functional* if the rules for all functions contain neither `or`-subexpressions (i.e., the original rules have no overlapping left-hand sides, or, in the terminology of [1], they are *inductively sequential*) nor expressions of the form `let...free in` (i.e., free variables are not introduced at run time). This restriction is satisfied by typical functional programs (e.g., note that in Haskell rules with syntactically overlapping left-hand sides are translated into nested case-expressions rather than nondeterministic or-expressions [27]).

We define the *ground type environment* E_G by

$$E_G = \{ f :: \underbrace{G \ldots G}_{n} \xrightarrow{\emptyset} G \mid \text{ for all } n\text{-ary functions or constructors } f \}$$

Then we have the following result:

Proposition 1 (Purely functional programs). *Let P be a purely functional program and E_G a ground type environment for P.*

1. *E_G is a correct type environment.*
2. *For all expressions e without free variables, it is $E_G \vdash e :: G/\emptyset$.*

Thus, purely functional programs (and, similarly, purely functional subcomputations in a program) are precisely analysed as deterministic in our framework.

5. CONCLUSIONS AND FUTURE WORK

We have proposed a method to analyse functional logic programs in order to identify expressions that might raise nondeterministic computations during their evaluation. By the results obtained from such an analysis, the programmer will be able to change the program to make it more robust. For instance, to avoid splitting computations she might remove nondeterminism completely if it was not necessary or a result of a programming error. Otherwise, she can encapsulate the affected program parts, if the nondeterminism is necessary for computing the result. This will avoid run-time errors in programs that use I/O actions (which is the case for almost every larger program) and thus increases program stability. Additionally, compilers can exploit the analysis results to optimize the code for deterministically reducible parts of a program. Although we have presented the typing rules and some examples for the functional logic language Curry, the analysis should be easily adaptable to other functional logic languages (or just logic languages).

For future work we will study how the current set of typing rules needs to be extended to cover features like external functions or search operators. This should not be difficult but has just not been considered yet. Another important point for future work is the efficient implementation of the type and effect inference algorithm since the goal of this work is the development of a fully automatic nondeterminism analysis so that the user is not forced to specify the type and effect annotations by hand (in contrast to mode-based languages like Mercury). Last but not least, we will consider to refine the type domain. The simple Ground-Any-domain, which corresponds to the classical domain used for groundness analysis in logic languages [25], is often sufficient for computing quite exact groundness information. But especially the nondeterminism analysis could benefit for instance from a three element domain, distinguishing between ground terms, terms in head normal form and any terms, or from using regular types [32]. To further improve the preciseness of the analysis, we might also consider to include sharing information in our analysis. At the moment, all arguments of a function call are analysed independently. This does not produce any wrong results, but if the order for analysing arguments would be fixed in some way (which should be possible because the *evaluation order* is encoded in the case rules), information computed while analysing one argument could be used for analysing the next one. In some cases, this could indeed improve the preciseness of the computed results.

6. REFERENCES

[1] S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. To appear in *Journal of the ACM*, 2000. Previous version in *Proc. 21st ACM Symposium on Principles of Programming Languages*, pages 268–279, Portland, 1994.

[2] L. Cardelli. Type systems. In *Allen B. Tucker, Jr. (Editor-in-Chief), The Computer Science and Engineering Handbook*. CRC Press, in cooperation with ACM, 1997.

[3] K. Claessen, T. Vullinghs, and E. Meijer. Structuring graphical paradigms in TkGofer. In *Proc. of ICFP'97*, pp. 251–262. ACM SIGPLAN Notices Vol. 32, No. 8, 1997.

[4] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proc. 9th Annual Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[5] S. K. Debray and D. S. Warren. Detection and optimization of functional computations in Prolog. In *Proc. of ICLP'86*, pages 490–504. Springer LNCS, 1986.

[6] B. Demoen, M. García de la Banda, W. Harvey, K. Marriott, and P. Stuckey. Herbrand constraint solving in HAL. In *Proc. of ICLP'99*, pages 260–274. MIT Press, 1999.

[7] Y.-C. Fuh and P. Mishra. Type Inference with Subtypes. *Theoretical Computer Science*, 73:155–175, 1990.

[8] J.C. Gonzáles-Moreno, M.T. Hortalá-Gonzáles, F.J. López-Fraguas, and M. Rodríguez-Artalejo. A rewriting logic for declarative programming. In *Proc. ESOP'96*, pages 156–172. Springer LNCS 1058, 1996.

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

[10] M. Hanus. A unified computation model for functional and logic programming. In *Proc. of the 24th ACM Symposium on Principles of Programming Languages (Paris)*, pages 80–93, 1997.

[11] M. Hanus. Distributed programming in a multi-paradigm declarative language. In *Proc. of PPDP'99*, pages 376–395. Springer LNCS 1702, 1999.

[12] M. Hanus. A Functional Logic Programming Approach to Graphical User Interfaces. In *Proc. of PADL'00*, pp. 47–62. Springer LNCS 1753, 2000.

[13] M. Hanus and C. Prehofer. Higher-Order Narrowing with Definitional Trees. In *Journal of Functional Programming*, 9(1):33–75, 1999.

[14] M. Hanus and R. Sadre. An Abstract Machine for Curry and its Concurrent Implementation in Java. *Journal of Functional and Logic Programming*, 1999(6).

[15] M. Hanus and F. Steiner. Controlling search in declarative programs. In *Proc. of Joint International Symposium PLILP/ALP'98*, pages 374–390. Springer LNCS 1490, 1998.

[16] M. Hanus (ed.). Curry: An integrated functional logic language. Available at `http://www.informatik.uni-kiel.de/~curry/`, 2000.

[17] M. Hanus , S. Antoy, J. Koj, R. Sadre, and F. Steiner. PAKCS: The Portland Aachen Kiel Curry System. Available at: `http://www.informatik.uni-kiel.de/~pakcs/`, 2000.

[18] F. Henderson and T. Somogyi, Z. Conway. Determinism analysis in the Mercury compiler. In *Proc. of the Nineteenth Australasian Computer Science Conference*, pages 337–346, 1996.

[19] T. Johnsson. Lambda Lifting: Transforming Programs to Recursive Functions. In *Functional Programming Languages and Computer Architecture*, pp. 190–203. Springer LNCS 201, 1985.

[20] F. Liu. Towards lazy evaluation, sharing and non-determinism in resolution based functional logic languages. In *Proc. of FPCA'93*, pages 201–209, New York, NY, USA, 1993. ACM Press.

[21] R. Loogen and S. Winkler. Dynamic detection of determinism in functional logic languages. *Theoretical Computer Science*, 142(1):59–87, 1995.

[22] J.M. Lucassen and D.K. Gifford. Polymorphic Effect Systems. In *Proc. of the 15th ACM Symposium on Principles of Programming Languages*, pages 47–57, 1988.

[23] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1999.

[24] H. R. Nielson and F. Nielson. Communication analysis for Concurrent ML. In *ML with Concurrency*, Monographs in Computer Science, pages 185–235. Springer-Verlag, 1997.

[25] U. Nilsson. Towards a Framework for the Abstract Interpretation of Logic Programs. In *Proc. of PLILP'88*, pp. 68–82, Orléans, 1988. Springer LNCS 348.

[26] F. Pessaux and X. Leroy. Type-based analysis of uncaught exceptions. In *Proc. of POPL'99*, pages 276–290. ACM Press, 1999.

[27] J. Peterson et al. Haskell: A Non-strict, Purely Functional Language (Version 1.4). Technical Report, Yale University, 1997.

[28] C. Schulte and G. Smolka. Encapsulated search for higher-order concurrent constraint programming. In *Proc. of ILPS'94*, pages 505–520. MIT Press, 1994.

[29] P. Van Roy, B. Demoen, and Y.D. Willems. Improving the execution speed of compiled Prolog with modes, clause selection, and determinism. In *Proc. of the TAPSOFT '87*, pages 111–125. Springer LNCS 250, 1987.

[30] P. Wadler. How to declare an imperative. *ACM Computing Surveys*, 29(3)240–263, 1997.

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

[32] E. Yardeni and E. Shapiro. A Type System for Logic Programs. *Journal of Logic Programming*, 10:125–153, 1991.