Determinism Types for Functional Logic Programming

Michael Hanus
Kiel University
Kiel, Germany
mh@informatik.uni-kiel.de

Kai-Oliver Prott
Kiel University
Kiel, Germany
kpr@informatik.uni-kiel.de

Abstract

Functional logic programming languages, such as Curry, integrate features of functional and logic paradigms, in particular, demand-driven deterministic evaluation from functional programming with non-deterministic search from logic programming. Though useful for programming, this combination can lead to unintended results and subtle bugs. To support programming with this powerful computation model, this paper proposes a method to detect unintended non-determinism at compile time. For this purpose, we propose determinism types to approximate the determinism behavior of functions and expressions. In contrast to standard types in strongly typed languages, determinism types do not restrict the set of admissible programs but support the programmer and programming tools in reasoning about functional logic programs, e.g., to enforce determinism in top-level I/O operations. We present the motivation behind this approach, discuss core concepts of functional logic programming and Curry, and outline methods to check for determinism through type-based analysis.

CCS Concepts

• **Software and its engineering** → **Compilers**; **Functional languages**; **Semantics**; *Multiparadigm languages*; • **Theory of computation** → *Logic and verification.*

Keywords

functional programming, logic programming, functional logic programming, determinism, type systems, Curry, static analysis, non-determinism

ACM Reference Format:
Michael Hanus and Kai-Oliver Prott. 2025. Determinism Types for Functional Logic Programming. In *Proceedings of the 27th International Symposium on Principles and Practice of Declarative Programming*. ACM, New York, NY, USA, 13 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Functional logic programming [10, 21] combines the demand-driven and, for particular classes of programs, optimal [29] evaluation model from functional programming with the expressiveness and flexibility of logic programming. This synergy, exemplified by the language Curry [27], introduces powerful programming abstractions, including higher-order functions, unification with evaluable functions, and non-deterministic computations. However, these strengths also bring challenges in reasoning about program behavior, especially concerning determinism.

Determinism refers to the property that a program or operation always produces the same output for the same input, without involving any implicit choices or ambiguity. In functional logic languages, non-determinism is a first-class feature, but many programs are intended to be deterministic. Accidental non-determinism can lead to subtle bugs, unpredictable results, or even run-time errors, especially when interacting with I/O.

For instance, consider the following definition from the Haskell library `Data.Maybe`:

```
isNothing :: Maybe a → Bool
isNothing Nothing = True
isNothing _       = False
```

Although the last equation defining `isNothing` does not seem meaningful on its own, it is reasonable in Haskell due to its sequential pattern matching from top to bottom. This is different in Curry where *all* rules defining an operation are taken into account in order to search for values or answers, as in logic programming. Consequently, `isNothing Nothing` returns both `True` and `False` in Curry, where the order is unspecified. Hence, the top-level evaluation of

```
print (isNothing Nothing)
```

leads to a run-time error since one cannot duplicate the world to enable non-deterministic printing.

To avoid such errors and to express determinism behavior in programs, we propose to add *determinism type signatures* as an optional language extension to Curry. These signatures allow developers to annotate functions with their expected determinism behavior which is statically checked by the compiler. For example, in the following Curry code snippet, the operation `append` is annotated to indicate that it is deterministic:

```
append :? Det → Det → Det
append :: [a] → [a] → [a]
append [] ys = ys
append (x:xs) ys = x : append xs ys
```

Since the result of a call to `append` contains the values of both arguments, the result of `append` is deterministic only if the input lists are deterministic as well. That is why the determinism signature (annotated by "`:?`") contains a `Det` for both arguments, which are separated by the arrow →. The similarity to the standard type signature is intentional and helps programmers reason about both the shape and the determinism of their functions in a uniform way. When `append` is called with free variables, Curry's demand-driven evaluation strategy instantiates these variables only as much as required for the pattern matching on the arguments. Although this leads to more than one result, we consider free variables as (potentially) non-deterministic. Thus, the signature of append still holds.

For some functions, the determinism status of the result does not depend on the arguments. For example, the operation const is deterministic even if its second argument is non-deterministic:

```
const :? Det → Any → Det
const :: a   → b   → a
const x _ = x
```

Here, the second argument can be non-deterministic (Any), but since it is not used in the result, the operation as a whole remains deterministic. In order to avoid the definition of multiple determinism types for a single operation, we assume that Det is a subtype of Any so that we can also apply const to a second argument of type Det.

The disuse of arguments is not the only possibility to deal with non-deterministic arguments in deterministic operations. An important feature of logic-oriented languages is the encapsulation of non-deterministic computations. Similarly to Prolog's findall predicate, Curry supports an operation allValues [5] which returns the list of all values of the argument. With our proposal, the signature of allValues is

```
allValues :? Any → Det
allValues :: a   → [a]
```

Similarly, one can also specify the determinism behavior of set functions [9] which provide an alternative method to encapsulate search.

Another use of determinism types is to enforce determinism in instances of type classes [46]. For example, it is reasonable to require that the function show that converts a value to a string is deterministic, since data is usually shown in I/O operations. This might be expressed by the type class definition

```
class Show a where
  show :? Det → Det
  show :: a → String
```

Now, providing an instance of Show with a non-deterministic implementation like the following would be rejected by the compiler:

```
instance Show Bool where
  show True = "True"
  show _    = "False"
```

This instance is non-deterministic since it returns "True" and "False" for the input True due to the overlapping patterns, similarly to the operation isNothing discussed above. This is a common mistake in Curry by Haskell programmers, where one intends to define a deterministic function but accidentally created a non-deterministic one by using the underscore pattern as a catch-all case.

Non-determinism is essential for problems involving search, inference, and symbolic computation so that we do not wish to restrict its use. However, in practice, many operations are intended to be deterministic—even in languages that support non-determinism. Detecting when operations inadvertently exhibit non-deterministic behavior is valuable both for correctness and efficiency. This is our main motivation to add determinism types to functional logic programs.

Adding determinism information to the type system offers multiple benefits:

- **Intentional clarity**: Determinism type annotations serve as documentation of intent, signaling to other developers and tools that an operation should behave deterministically.
- **Static guarantees**: The determinism checker can detect violations early, reducing the need for dynamic debugging.
- **Optimization opportunities**: Knowing an operation is deterministic allows compilers to generate more efficient code and eliminate some run-time checks.
- **Safe I/O**: Ensuring determinism in I/O contexts prevents run-time crashes due to undefined behavior. We can achieve this by emitting a warning or error when the program entry point (e.g., the main function) does not have type Det.

Ultimately, determinism types and their analysis aim to provide a compositional and reliable way to reason about program behavior without sacrificing the flexibility of non-deterministic programming. Because our proposal is lightweight, compositional, and compatible with existing Curry implementations, it is suitable for practical use in both interactive and compiled environments.

In the next section, we review the core concepts of functional logic programming and Curry. Section 3 introduces a typed kernel language which is used in this paper. The syntax of determinism types and their meaning is shown in Section 4. Section 5 presents the rules for determinism typing. Some formal results about determinism typing and properties of programs with determinism types are given in Section 6. We also provide a full formalization of these results in the *Rocq* proof assistant [13]. Section 7 surveys potential applications of determinism types. Section 8 presents some extensions of the current framework. An implementation of determinism types is sketched in Section 9 before we conclude with a survey of related work in Section 10.

2 Functional Logic Programming and Curry

Functional logic programming languages unify the most important principles from functional programming (demand-driven evaluation, strong typing with parametric polymorphism, higher-order functions) and logic programming (non-determinism, computing with partial information, constraints). The functional logic language Curry[1] has a Haskell-like syntax, while its operational semantics [1] incorporates ideas from logic programming such as non-determinism and handling of free variables. The combination of reduction and instantiation of free variables is called narrowing [36, 40]. Curry is based on the needed narrowing strategy [6] which is optimal w.r.t. number of evaluation steps and computed solutions.

A distinguishing feature of Curry, different from Haskell, is the selection of rules. Whereas Haskell applies the (from top to bottom) first matching rule to reduce a subexpression, Curry uses all applicable rules. Thus, the choice operator "?" is predefined by

```
(?) :: a → a → a
x ? y = x
x ? y = y
```

[1]www.curry-lang.org

so that the values of "`True ? False`" are `False` and `True`.

In order to encapsulate non-deterministic computations and to return all values of an expression involving non-determinism in a single data structure, there is an operation `allValues` already introduced in Section 1. Since the result of this operation depend on the concrete evaluation strategy [15], Curry provides *set functions* [9] to encapsulate search in an evaluation-independent manner. For each defined operation f, f_S denotes its corresponding set function. f_S encapsulates the non-determinism caused by evaluating f except for the non-determinism caused by evaluating the arguments to which f is applied. For instance, consider the operation

```
hamlet b = b ? not b
```

Then the expression `hamlet True` non-deterministically yields both values `True` and `False`, whereas the expression $\texttt{hamlet}_S$ `True` evaluates to (an abstract representation of) the set `{True,False}`. Since the non-determinism of arguments is not encapsulated by a set function, the expression $\texttt{hamlet}_S$ `(hamlet True)` evaluates to the equivalent sets `{True,False}` and `{False,True}`.

To support the implementation of larger applications, Curry has many additional features not described here, like modules, which are similar to Haskell, and monadic I/O [45] for declarative input/output. The latter is based on the idea that operations performing I/O are considered as functions manipulating the state of the external world (or environment). Since a non-deterministic choice duplicates the state of a computation but the "world" (terminal, file system, etc.) cannot be duplicated, non-determinism and I/O are incompatible. Thus, Curry implementations emit a run-time error when non-deterministic I/O operations are applied. The exact mechanism for detecting this error is implementation-specific and often involves checking for multiple results during evaluation. Such errors can be avoided by encapsulating non-deterministic expressions.

Example 1 (Non-determinism and I/O). Consider the following definitions:

```
coin :: Int
coin = 0 ? 1

f :: Int → Int
f x = x + coin

main = print (f coin)
```

Since `f coin` has more than one value (0, 1, or 2), it is unclear which of the values should be printed. Therefore, as discussed above, the execution of `main` leads to a run-time error. To avoid this, one has to encapsulate the argument of `print`, where it must be decided by the programmer which of the values (e.g., all, one, minimum) should be printed. For instance,

```
main = print (allValues (f coin))
```

prints the list of all values. Note that the use of the set function of `f`, as in

```
main = print (f_S coin)
```

is not sufficient, since the non-determinism of the argument `coin` is not encapsulated.

This example shows that it would be helpful to have a tool which helps to encapsulate the right expressions. This is the motivation for adding determinism types to Curry.

3 A Typed Kernel Functional Logic Language

Curry is a powerful language with a lot of syntactic sugar — influenced by Haskell but with additional features, like functional patterns [7] and set functions as mentioned above. Moreover, Curry's type system is inspired by Haskell so that it offers parametric polymorphism and type (constructor) classes. In order to reduce the features covered by determinism typing, we consider a restricted kernel language similarly to operational descriptions [1], implementations [24], and analyses [25] of functional logic programs. This kernel language, called FlatCurry, consists of top-level operations without patterns, i.e., patterns are translated into case expressions and choice operators. Although we defined "?" by two rules with overlapping left-hand sides in the previous section, one can also consider "?" as a primitive choice operator and translate overlapping rules into non-overlapping rules with explicit choice operators "?" in the right-hand sides. For instance, overlapping rules like

```
f True = True
f _    = False
```

can be translated to the single non-overlapping rule

```
f x = case x of True  → True
                False → True ? False
```

Since any functional logic program can be transformed into this form [3], FlatCurry can be and is often used as an intermediate language to compile Curry programs. The syntax of FlatCurry as used in this paper is summarized in Figure 1. It is a standard higher-order language with case expressions for pattern matching and choices and introduction of free variables for logic programming.

For instance, the operation `append` shown in Section 1 can be represented in FlatCurry as

```
append = λxs →
           λys → case xs of
                   { []   → ys
                   ; z:zs → z : append zs ys }
```

An intricate point with determinism types is the handling of functional values. For instance, consider the following operations which use the identity operation `id` and the Boolean negation `not` in two different ways.

```
idNot1 x = id x ? not x

idNot2 = id ? not
```

When these operations are applied to a Boolean value, both values `True` and `False` are non-deterministically returned. What should be the determinism type of these operations? The type of `idNot1` is clearly

```
idNot1 :? Det → Any
```

The determinism type of `idNot2` is not obvious. It could be

```
idNot2 :? Det → Any
```

(since it is a functional value returning different values when it is applied) or also

$$\begin{array}{lcll}
P & ::= & F_1 \ldots F_m & \text{(program)} \\
F & ::= & f = e & \text{(function definition)} \\
e & ::= & x & \text{(variable)} \\
 & | & C & \text{(data constructor)} \\
 & | & e_1\ e_2 & \text{(application)} \\
 & | & \lambda x \to e & \text{(abstraction)} \\
 & | & e_1\ \mathit{or}\ e_2 & \text{(choice)} \\
 & | & \mathit{let}\ x\ \mathit{free}\ \mathit{in}\ e & \text{(free variable)} \\
 & | & \mathit{case}\ e\ \mathit{of}\ \{p_1 \to e_1; \ldots; p_k \to e_k\} & \text{(case expression, } k > 0) \\
p & ::= & C\ x_1\ \ldots\ x_n & \text{(pattern, } n \geq 0)
\end{array}$$

Figure 1: Syntax of the kernel language FlatCurry

```
idNot2 :? Any
```

(since it returns non-deterministically two functional values). Since the operational behavior of both definition is the same, one could consider the determinism types `Det → Any` and `Any` as equivalent.

However, one can distinguish the two different non-deterministic definitions by encapsulation. The expression `allValues idNot1` returns a list with the single element `idNot1` (since the argument cannot be further evaluated), whereas `allValues idNot2` encapsulates the non-deterministic choice of `idNot2` so that a list with two elements (`id` and `not`) is returned. Hence, we get different results when we apply these two expressions to the list `[True]` with the operator `<*>` of the class `Applicative` [33]:

```
allValues idNot1 <*> [True]  ⇝  [True] ? [False]
allValues idNot2 <*> [True]  ⇝  [True,False]
```

This is also one of the reasons why `allValues` is considered "unsafe" from a declarative point of view, whereas set functions, which do not encapsulate higher-order values, are a declarative method to encapsulate non-deterministic computations [9].

These complications can be avoided by using determinism types only for values of first-order data types. Interestingly, this restriction fits to the `Data` proposal for Curry [26]: the type of free variables must satisfy the `Data` class constraint which expresses that such variables do not contain functional values. This ensures a constructive method to non-deterministically enumerate values for free variables.

In the following, we consider a typed intermediate language which distinguishes between first-order and higher-order values. For the sake of simplicity, we do not consider polymorphic types but refer to Section 8.2 for a discussion about polymorphism. *First-order types* are Booleans and lists:

$$D \quad ::= \quad \texttt{Bool} \mid \texttt{List}\, D$$

These can be easily extended to other types, like integers, pairs, etc. Note that types in D are types without functional components, i.e., in Curry these are the types which are instances of class `Data` [26] (see also Section 8.1). For some of our examples, we assume an extension of the type system with some additional first-order types.

General *types* are first-order types combined with functional types:

$$T \quad ::= \quad D \mid T \to T$$

Since we are not interested in type inference, we assume that variables introduced in FlatCurry programs (as parameters in abstractions or free variables) are annotated with a type. The typing rules for this language are shown in Figure 2. There, we denote by Γ a *type environment* which is a (partial) mapping from identifiers (taken from a set of names X) to types:

$$\Gamma : X \to T$$

As usual, we denote by $\Gamma[x \mapsto \tau]$ the type environment Γ' with $\Gamma'(x) = \tau$ and $\Gamma'(y) = \Gamma(y)$ for all $x \neq y$.

Note that Figure 2 contains separate case typing rules for each first-order type. Although Figure 2 defines the type of a case expression with a complete list of branches, one can also omit some branches if they are undefined (as in the case of partially defined operations). Note that our system can be easily extended to support more data types.

A program $P = f_1 = e_1; \ldots; f_k = e_k$ is *type correct* if there is a type environment

$$\Gamma = \{f_1 \mapsto \tau_1, \ldots, f_k \mapsto \tau_k\}$$

such that $\Gamma \vdash e_i :: \tau_i$ $(i = 1, \ldots, k)$ is derivable by the typing rules in Figure 2. In the following, we consider only type-correct programs.

4 Determinism Types

Similarly to standard types in strongly typed programming languages, expressions and operations can be annotated with determinism types. Whereas standard types approximate the set of possible values to which an expression evaluates, a determinism type approximates the determinism behavior. For instance, if an expression is correctly annotated as deterministic (*Det*), its evaluation will never yield a choice between two expressions. Otherwise, the expression is annotated with *Any* meaning that it is potentially non-deterministic. The determinism type system propagates these annotations through applications and abstractions in order to annotate every expression and operation with a determinism type.

In order to associate meaningful determinism types to operations, we allow also function arrows in determinism types. Thus, the *syntax of determinism types* is:

$$\Delta \quad ::= \quad \mathit{Det} \mid \mathit{Any} \mid \Delta \to \Delta$$

Note that the type $\mathit{Any} \to \mathit{Any}$ is formally allowed but has the same information as $\mathit{Det} \to \mathit{Any}$ (see below). Hence it will never

Var	$\Gamma \vdash x :: \Gamma(x)$	
True	$\Gamma \vdash \texttt{True} :: \texttt{Bool}$	
False	$\Gamma \vdash \texttt{False} :: \texttt{Bool}$	
Nil	$\Gamma \vdash \texttt{[]} :: \texttt{List}\ \tau$	$\tau \in D$
Cons	$\Gamma \vdash \texttt{(:)} :: \tau \longrightarrow \texttt{List}\ \tau \longrightarrow \texttt{List}\ \tau$	$\tau \in D$
App	$\dfrac{\Gamma \vdash e_1 :: \tau_1 \longrightarrow \tau_2 \quad \Gamma \vdash e_2 :: \tau_1}{\Gamma \vdash e_1\ e_2 :: \tau_2}$	
Abs	$\dfrac{\Gamma[x \mapsto \tau_1] \vdash e :: \tau_2}{\Gamma \vdash \lambda x : \tau_1 \longrightarrow e :: \tau_1 \longrightarrow \tau_2}$	
Choice	$\dfrac{\Gamma \vdash e_1 :: \tau \quad \Gamma \vdash e_2 :: \tau}{\Gamma \vdash e_1\ ?\ e_2 :: \tau}$	
Free	$\dfrac{\Gamma[x \mapsto \tau] \vdash e :: \tau'}{\Gamma \vdash \mathit{let}\ x : \tau\ \mathit{free\ in}\ e :: \tau'}$	$\tau \in D$
CaseBool	$\dfrac{\Gamma \vdash e :: \texttt{Bool} \quad \Gamma \vdash e_1 :: \tau \quad \Gamma \vdash e_2 :: \tau}{\Gamma : \mathit{case}\ e\ \mathit{of}\ \{\texttt{True} \longrightarrow e_1; \texttt{False} \longrightarrow e_2\} :: \tau}$	
CaseList	$\dfrac{\Gamma \vdash e :: \texttt{List}\ \tau \quad \Gamma \vdash e_1 :: \tau' \quad \Gamma[x_1 \mapsto \tau, x_2 \mapsto \texttt{List}\ \tau] \vdash e_2 :: \tau'}{\Gamma : \mathit{case}\ e\ \mathit{of}\ \{\texttt{[]} \longrightarrow e_1; \texttt{x1:x2} \longrightarrow e_2\} :: \tau'}$	$\tau \in D$

Figure 2: Type system of FlatCurry

be inferred by our inference system. The intended meaning of these types is:

*Det***:** The expression evaluation (w.r.t. standard Curry semantics, see Section 6) is deterministic, i.e., it never yields a choice between two values.

Any $\longrightarrow$ *Det***:** If a function of this type is applied to some argument, its evaluation is deterministic, either because the argument is not used or all non-determinism is encapsulated.

Det $\longrightarrow$ *Det***:** If a function of this type is applied to a *Det* argument, its evaluation is deterministic, otherwise the evaluation is arbitrary

Det $\longrightarrow$ *Any***:** If a function of this type is applied to some argument, its evaluation is arbitrary.

Any $\longrightarrow$ *Any***:** As stated before, this type behaves like the previous *Det* $\longrightarrow$ *Any*.

*Any***:** The expression evaluates (w.r.t. standard Curry semantics) in a possibly non-deterministic manner, i.e., it possibly yields choices between values. It might also be a function that behaves in such a way.

As discussed above, we do not associate the determinism type *Det* to a functional value, i.e., an expression of type $\tau_1 \longrightarrow \tau_2$ has determinism type *Any* or is of the form $\delta_1 \longrightarrow \delta_2$. We say a determinism type δ is *compatible* to a type $\tau \in T$ if δ has the same shape as τ (except for *Any*) but first-order types are replaced by determinism types:

- For a type $\tau \in D$, *Det* or *Any* is compatible to τ.
- *Any* is compatible to type $\tau_1 \longrightarrow \tau_2$.
- The determinism type $\delta_1 \longrightarrow \delta_2$ is compatible to $\tau_1 \longrightarrow \tau_2 \in T$ if δ_i is compatible to τ_i ($i = 1, 2$).

Note that if one were to extend the type system with more first-order types, the compatibility relation needs to be extended accordingly.

The motivation for allowing compatibility between *Any* and types of the form $\tau_1 \longrightarrow \tau_2$ is that we want the determinism type of `idNot2` to be *Any*. Consequently, *Any* must be considered as compatible to its type `Bool` $\longrightarrow$ `Bool`. This design choice has the advantage of keeping the determinism typing of a choice simple: we always assign the determinism type *Any* to a choice, regardless of whether the result is a function or a first-order value. If, instead, we require `idNot2` to have the determinism type *Det* $\longrightarrow$ *Any*, the determinism typing of choices become unnecessarily complex due to the consideration of functional types. As we will see, we consider in the determinism typing of applications that the function to be applied might have type *Any*.

Here are some examples for determinism types. Consider the identity operation `id` defined by

```
id x = x
```

with type $\tau \longrightarrow \tau$ for some type $\tau \in D$.

```
id :? Det → Det
```

is a possible determinism type for `id`. The determinism type

```
id :? Det → Any
```

is also possible but less precise.

The choice operation returns one of its arguments so that its evaluation is always non-deterministic:

```
(?) :? Det → Det → Any
```

$$\delta_1 \sqsubseteq \delta_2 \quad := \begin{cases} \mathit{true} & \text{if } \delta_2 = \mathit{Any} \vee \delta_1 = \delta_2 = \mathit{Det} \\ \delta_{21} \sqsubseteq \delta_{11} \wedge \delta_{12} \sqsubseteq \delta_{22} & \text{if } \delta_1 = \delta_{11} \to \delta_{12} \wedge \delta_2 = \delta_{21} \to \delta_{22} \\ \mathit{false} & \text{otherwise} \end{cases} \qquad \delta_1 \sqcup \delta_2 \quad := \begin{cases} \delta_2 & \text{if } \delta_1 \sqsubseteq \delta_2 \\ \delta_1 & \text{if } \delta_2 \sqsubseteq \delta_1 \\ \mathit{Any} & \text{otherwise} \end{cases}$$

Figure 3: Subtyping relation and supremum of determinism types

Note that this is the type if the choice operation has first-order types as arguments, otherwise the *Det* must be replaced by a compatible determinism type.

As discussed above, the operation `allValues` encapsulates all non-determinism of its argument, but we have to be careful with functional values. Hence, we allow only the encapsulation of first-order values, i.e., `allValues` has the type

```
allValues :: τ → τ
```

for some $\tau \in D$ (we could also consider a family of operations $\texttt{allValues}_\tau$ for each $\tau \in D$). Then its determinism type is

```
allValues :? Any → Det
```

Consider a function f with type $\tau_1 \to \cdots \to \tau_n \to \tau$ and $\tau_i \in D$ ($i = 1, \ldots, n$). Then its set function $\texttt{f}_S$ encapsulates the non-determinism caused by `f` but not the non-determinism occurring in the arguments. Hence, its determinism type is

$\texttt{f}_S$ `:?` $\mathit{Det} \to \cdots \to \mathit{Det} \to \mathit{Det}$

independent of the determinism type of `f` (which has usually the target type *Any*).

These examples illustrate how determinism types can precisely capture the intended behavior of functions, and how they interact with higher-order functions and encapsulation.

5 Determinism Typing

Determinism typing is defined w.r.t. an environment which associates determinism types to identifiers, e.g., bound variables or operations defined in the program. For this purpose, we denote by a *determinism type environment* Δ a (partial) mapping from identifiers (taken from a set of names X) to determinism types:

$$\Delta : X \to \Delta$$

We assume that the determinism types associated to operations defined in a program are compatible to the types defined in Section 3, i.e., if $\Gamma(f) = \tau$, for some $\tau \in T$, then $\Delta(f)$ is compatible to τ. With respect to such a determinism type environment, the determinism typing rules (Figure 4) derive judgements of the form

$$\Delta \vdash e : \delta$$

stating that the expression e has determinism type δ with respect to the determinism type environment Δ.

For the application of an operation to an argument, it must be checked whether the argument's determinism type satisfies the requirements of the operation on its argument. For this purpose, we define a *subtyping relation* on determinism types in Figure 3. For example, if a non-deterministic operation of type $\mathit{Det} \to \mathit{Any}$ is required, we can also use an operation of type $\mathit{Det} \to \mathit{Det}$ or $\mathit{Any} \to \mathit{Det}$. However, we cannot use a non-deterministic operation of type $\mathit{Det} \to \mathit{Any}$ where a deterministic operation of type $\mathit{Det} \to \mathit{Det}$ is expected. Note that any operation can be used where *Any* is required. This simplifies the typing of the choice operator as shown below.

We explain the intended meaning of the various rules for determinism typing shown in Figure 4. To keep the rules compact, we sometimes denote by $\overline{o_n}$ a sequence of objects $o_1, \ldots, o_n$.

Rule Var is obvious. The requirement that x is in the domain of Δ is to ensure that every variable needs to have a corresponding binding site. This is always the case for correct (FlatCurry) programs.

Constructors (rule Cons) are deterministic but might become non-deterministic when applied to some non-deterministic argument, see the subsequent rules for application.

Rule AppAny covers the case when there is no precise information about the operation, in particular, if the expression e_1 involves a choice between two operations. App is the usual application rule where determinism subtyping on arguments is required for the reasons discussed above. For instance, the well-known operation `map` takes a function as a first argument and applies it to all elements of the list provided as the second argument. The FlatCurry definition typed for Boolean lists is

```
map :: (Bool → Bool) → List Bool  → List Bool
map =
  λf : Bool → Bool  →
    λxs : List Bool  →
      case xs of { []    → []
                 ; y:ys  → f y : map f ys }
```

If $\Delta(\texttt{map}) = (\mathit{Det} \to \mathit{Det}) \to \mathit{Det} \to \mathit{Det}$ and e is the right-hand side of the definition of `map`, then we derive with the rules in Figure 4 that $\Delta \vdash e :? \Delta(\texttt{map})$. We denote this judgement by the determinism type

```
map :? (Det  → Det)  → Det  → Det
```

Similarly, we derive that `map not [True]` has determinism type *Det*. The operation `hamlet`, as defined in Section 2, has determinism type

```
hamlet :? Det  → Any
```

Since $\mathit{Det} \to \mathit{Any}$ is not a subtype of $\mathit{Det} \to \mathit{Det}$ required by `map`, rule App infers the determinism type *Any* for the expression `map hamlet [True]`.

Rule Choice simply infers the determinism type *Any*. It could be made more precise by considering different kinds of function types, but such a more complex technical treatment does not pay off in realistic examples. Rule Free considers the case of a free (logic) variable. Since such a variable evaluates or can be bound to different values, it is typed as non-deterministic. One could make it more precise for the special case that the type τ contains only a

$$\textsc{Var} \qquad \Delta \vdash x :?\, \delta \qquad \text{if } x \in \mathit{Dom}(\Delta) \text{ and } \delta = \Delta(x)$$

$$\textsc{Cons} \qquad \Delta \vdash C :?\, \mathit{Det} \to \cdots \to \mathit{Det} \to \mathit{Det} \qquad \text{if } C \text{ is an } n\text{-ary constructor}$$

$$\textsc{AppAny} \qquad \frac{\Delta \vdash e_1 :?\, \mathit{Any} \quad \Delta \vdash e_2 :?\, \delta}{\Delta \vdash e_1\ e_2 :?\, \mathit{Any}}$$

$$\textsc{App} \qquad \frac{\Delta \vdash e_1 :?\, \delta_{11} \to \delta_{12} \quad \Delta \vdash e_2 :?\, \delta_2}{\Delta \vdash e_1\ e_2 :?\, \delta_3} \qquad \delta_3 = \begin{cases} \delta_{12} & \text{if } \delta_2 \sqsubseteq \delta_{11} \\ \mathit{Any} & \text{otherwise} \end{cases}$$

$$\textsc{Abs} \qquad \frac{\Delta[x \mapsto \delta_1] \vdash e :?\, \delta_2}{\Delta \vdash \lambda x : \tau \to e :?\, \delta_1 \to \delta_2} \qquad \delta_1 \text{ is compatible to } \tau$$

$$\textsc{Choice} \qquad \frac{\Delta \vdash e_1 :?\, \delta_1 \quad \Delta \vdash e_2 :?\, \delta_2}{\Delta \vdash e_1 \ ?\ e_2 :?\, \mathit{Any}}$$

$$\textsc{Free} \qquad \frac{\Delta[x \mapsto \mathit{Any}] \vdash e :?\, \delta}{\Delta \vdash \mathit{let}\ x : \tau\ \mathit{free\ in}\ e :?\, \delta}$$

$$\textsc{Case} \qquad \frac{\Delta \vdash e :?\, \delta \quad \Delta[\overline{x_{n_i} \mapsto \delta}] \vdash e_i :?\, \delta_i \ (i = 1, \ldots, k)}{\Delta \vdash \mathit{case}\ e\ \mathit{of}\ \{\overline{p_k \to e_k}\} :?\, \delta \sqcup \delta_1 \sqcup \ldots \sqcup \delta_k} \qquad \text{where } p_i = C\ \overline{x_{n_i}} \text{ and } \delta \in \{\mathit{Det}, \mathit{Any}\}$$

Figure 4: Determinism typing rules

single value. Since such a use of free variables does not occur in real programs, we omit this slight improvement.

To understand rule Case, consider the determinism type δ of the discriminating argument. Since we assume that the source program is well-typed, case distinctions over functional values are not allowed. Under the assumption that determinism types are compatible to standard types (see Section 6), a functional determinism type cannot occur for the discriminating argument e. Thus, the side condition $\delta \in \{\mathit{Det}, \mathit{Any}\}$ is not a real restriction but only written for clarity.

Concerning the determinism types for pattern variables, note that lists (type `List`) contain only first-order values as elements. Even if we extend the simple type system to allow general data structures with functional values as components (as in Curry source programs), any constructor applied to a function will still be assigned *Any* (see rules Cons and the subtyping in App). Therefore, the pattern variables in a case expressions will always have determinism types *Det* or *Any*, i.e., there is no need to consider other determinism types for them in rule Case. Therefore, rule Case checks the determinism type of each branch under the assumption that the pattern variables have the same determinism type (*Det* or *Any*) as the discriminating argument e. Under these assumptions, the determinism types of e and the right-hand sides e_i of all branches are combined by the least upper bound operation w.r.t. the subtyping relation. In particular, if the determinism type of e is *Any*, the entire case expression must have type *Any* as well. Thus, one could add a simpler inference rule for this specific case, but we put the general rule Case since this is advantageous to establish the formal results about determinism typing presented in Section 6.

Given a program $P = f_1 = e_1; \ldots; f_k = e_k$, we say $f_i :? \delta_i$ $(i = 1, \ldots, k)$ are *correct determinism types* if $\Delta \vdash e_i :? \delta_i$ $(i = 1, \ldots, k)$ can be inferred by the rules of Figure 4 w.r.t. the determinism type environment

$$\Delta = \{f_1 \mapsto \delta_1, \ldots, f_k \mapsto \delta_k\}$$

As we will see later, there always exists a correct determinism typing for a well-typed program. Thus, determinism typing is optional and not a restriction on admissible programs—in contrast to standard typing.

We discuss the determinism typing of a few more examples. We have already shown the determinism typing of `map` above. Consider the expression

```
map (id ? not) [True]
```

By rule Choice, `(id ? not)` has type *Any*. Since *Any* is not a subtype of the type $\mathit{Det} \to \mathit{Det}$ required for the first argument of `map`, rule App infers the type *Any* for `map (id ? not)`. Now rule AppAny infers the type *Any* for the entire expression. This shows how we can apply operations with a deterministic type also on non-deterministic operations and obtain the type *Any*.

As a final example, consider the expression

```
map not (allValues (hamlet True))
```

Due to the determinism type of `hamlet` shown above, the subterm `(hamlet True)` has type *Any*. Since this is a subtype of the argument type required by `allValues`, the subterm `allValues (hamlet True)` has type *Det* by rule App. Due to the deterministic types of `map` and `not`, the entire expression has type *Det*. Actually, this expression evaluates to the single result `[False,True]`.

6 Correctness of Determinism Typing

The inference system for determinism typing possesses several desirable properties, ensuring that it behaves predictably and is suitable for compile-time program analysis. As mentioned above, we consider in the following programs and expressions which are correctly typed w.r.t. the rules in Figure 2 under a type environment Γ. In the formal statements, we need to extend the notion of type compatibility to type contexts. Therefore, we define a determinism type context Δ to be compatible to a type context Γ if, for all variable

$$\textsc{AppAbs}\ \frac{}{(\lambda x : \tau \to e)\ v \Rightarrow e[x \mapsto v]} \qquad \textsc{AppOr}\ \frac{}{(e_1\ ?\ e_2)\ e_3 \Rightarrow (e_1\ e_3)\ ?\ (e_2\ e_3)} \qquad \textsc{AppStep}\ \frac{e_1 \Rightarrow e_1'}{e_1\ e_2 \Rightarrow e_1'\ e_2}$$

$$\textsc{CaseOr}\ \frac{}{\mathit{case}\ (e_1\ ?\ e_2)\ \mathit{of}\ br \Rightarrow (\mathit{case}\ e_1\ \mathit{of}\ br)\ ?\ (\mathit{case}\ e_2\ \mathit{of}\ br)} \qquad \textsc{CaseNil}\ \frac{}{\mathit{case}\ \texttt{[]}\ \mathit{of}\ \texttt{[]} \to e_1\texttt{;}\ \texttt{x:xs} \to e_2 \Rightarrow e_1}$$

$$\textsc{CaseCons}\ \frac{}{\mathit{case}\ \texttt{(y:ys)}\ \mathit{of}\ \texttt{[]} \to e_1\texttt{;}\ \texttt{x:xs} \to e_2 \Rightarrow e_2[\texttt{x} \mapsto \texttt{y}, \texttt{xs} \mapsto \texttt{ys}]} \qquad \textsc{CaseTrue}\ \frac{}{\mathit{case}\ \texttt{True}\ \mathit{of}\ \texttt{True} \to e_1\texttt{;}\ \texttt{False} \to e_2 \Rightarrow e_1}$$

$$\textsc{CaseFalse}\ \frac{}{\mathit{case}\ \texttt{False}\ \mathit{of}\ \texttt{True} \to e_1\texttt{;}\ \texttt{False} \to e_2 \Rightarrow e_2} \qquad \textsc{CaseStep}\ \frac{e \Rightarrow e'}{\mathit{case}\ e\ \mathit{of}\ br \Rightarrow \mathit{case}\ e'\ \mathit{of}\ br} \qquad \textsc{OrStepL}\ \frac{e_1 \Rightarrow e_1'}{e_1\ ?\ e_2 \Rightarrow e_1'\ ?\ e_2}$$

$$\textsc{OrStepR}\ \frac{e_2 \Rightarrow e_2'}{e_1\ ?\ e_2 \Rightarrow e_1\ ?\ e_2'} \qquad \textsc{FreeStep}\ \frac{}{\mathit{let}\ x : \tau\ \mathit{free}\ \mathit{in}\ e \Rightarrow e[x \mapsto \mathit{gen}_\tau]}$$

Figure 5: Small-step evaluation rules for expressions

names v where $\Gamma(v)$ is defined, $\Delta(v)$ is compatible to $\Gamma(v)$. That is, the two functions Δ and Γ need to be pointwise compatible.

The first important property is the completeness of determinism typing. This property ensures that the inference system can assign a determinism type to every valid expression in the language. This is crucial since our extension is not meant to restrict the set of admissible programs but rather provides a way to annotate and analyze expressions. Note that this property is not entirely trivial since not every expression can be assigned type *Any*.

Theorem 2 (Typing Completeness). *For all contexts Δ that are compatible to Γ, and all expressions e that are well-typed under Γ, there exists a determinism type δ such that $\Delta \vdash e\ :?\ \delta$ is derivable w.r.t. the determinism typing rules.*

The next interesting property is the preservation of type annotations. The preservation property guarantees that type annotations are stable under evaluation, i.e., evaluation does not introduce any non-determinism not captured by determinism types. The formal statement of this property requires the notion of an evaluation step from an expression e to e', denoted $e \Rightarrow e'$. The formal definition of this relation is shown in Figure 5.

The functional core of evaluation steps is straightforward. The rules AppAbs, CaseCons, and FreeStep use substitutions on expressions. Substitution proceeds structurally through the expression but does not replace occurrences of variables that are bound by inner declarations. For example, in $(\lambda x \to x)[x \mapsto e]$, the x inside the abstraction is not replaced by e, since substitution stops at the binding site of x. Note that there are no unbound variables since free variables are explicitly introduced.

For the evaluation of logical aspects, it should be noted that evaluation is deterministic, i.e., non-deterministic choices are represented by choice structures headed by the choice operator "?" instead of evaluating it to one of its arguments, as in [1]. Thus, non-deterministic choices are kept in a data structure, similarly to pull-tabbing [2, 4] or the Verse calculus [12]. If an expression headed by a choice operator has to be evaluated, as in the rules AppOr and CaseOr, the choice is moved from the argument position to the top. Basically, this is the idea of pull-tabbing [2, 4] used in implementations of Curry which support flexible search strategies [14, 16]. In functional logic languages based on the *call-time choice* semantics [30], such as Curry, it is necessary to annotate choice nodes with tags in order to restrict the set of sensible values, as discussed in [4]. For the sake of simplicity, we omit this additional machinery so that our semantics implements the run-time choice semantics [30]. This causes no problem in our case, since run-time choice always computes the same or more values than call-time choice. Hence, if an evaluation is deterministic w.r.t. run-time choice, it is also deterministic w.r.t. call-time choice.

Another point to mention is the handling of free variables. As shown in [8, 19], free variables are conceptually equivalent to non-deterministic operations that evaluate to all values of the type of the free variable. For instance, a free Boolean variable can be replaced by the operation

```
gen_Bool = True ? False
```

This property is exploited in rule FreeStep, where a free variable of type τ is replaced by the operation gen_τ which generates all values of type τ. Here it is essential that τ is a first-order type so that gen_τ can be defined constructively.

Based on this operational semantics, we can formally state the preservation property of determinism typing.

Theorem 3 (Preservation). *Let Δ be a determinism type environment compatible to Γ, e, e' expressions with $e \Rightarrow e'$ and e well-typed under Γ. Furthermore, let δ be a determinism type such that $\Delta \vdash e\ :?\ \delta$ is derivable by the determinism typing rules. Then there exists a determinism type δ' such that $\delta' \sqsubseteq \delta$ and $\Delta \vdash e'\ :?\ \delta'$.*

Finally, we state the main property of our determinism typing. The soundness property ensures that expressions typed as deterministic never evaluate to expressions headed by choices, thereby guaranteeing run-time determinism when promised.

Theorem 4 (Soundness). *Let Δ be a determinism type environment, e, e' expressions with $e \Rightarrow^* e'$, and $\Delta \vdash e\ :?\ \mathit{Det}$ be derivable by the determinism typing rules. Then e' does not contain a choice at the root.*

If a choice node is evaluated, it is either at the root and its arguments are evaluated, or it will be moved to the top by iterated applications of rules AppOr and CaseOr. Thus, the soundness theorem ensures that a *Det*-typed expression will never evaluate a choice. Of course, choice nodes might occur in unevaluated subexpressions, for example in unapplied lambda abstractions. However, these will not be pulled to the top by the evaluation strategy and are thus not relevant for the soundness property.

Altogether, these properties demonstrate that determinism typing is a sound and practical foundation for program verification and optimization. A complete formalization of the determinism typing rules, along with machine-checked proofs of the properties discussed above, is available at https://github.com/cau-placc/rocq-ndtypes. This formalization is carried out in the *Rocq* proof assistant [13] (formerly known as Coq). In the formal development, we explicitly track well-typedness, carefully manage variable binding and name clashes during substitution, and establish a range of auxiliary lemmas, such as the preservation of determinism types under substitution.

7 Applications

Determinism types have various applications ranging from compile-time debugging to performance enhancement. In the following we discuss some of the potential applications.

Safe I/O execution. As discussed in the introduction and Example 1, the combination of non-determinism and I/O is unsafe since it is not clear which of the non-deterministic I/O actions should be applied to some state of the world. Since non-determinism of expressions is a dynamic property, this problem is not detected at compile time so that it leads to a run-time error in most Curry systems. If one requires that expressions of type `IO` have the determinism type *Det*, we avoid run-time crashes and unexpected behavior. For instance, one can execute an interactive I/O operation `main` only if it has determinism type `main :? Det`. As an example, consider the definition

```
main = print coin
```

Since `coin`, as defined in Example 1, is non-deterministic, the determinism type system infers the judgement `main :? Any`. Thus, the compiler can reject this program or issue a warning, preventing a run-time error. To resolve this issue, the programmer can explicitly encapsulate the non-determinism and specify how to handle multiple results. For example, they could print only the first value.

```
main = print (head (allValues coin))
```

If an expression is incorrectly inferred by the compiler as potentially non-deterministic, encapsulation can also be used to suppress the warning or error.

Interactive programming environments. Curry systems have an interactive environment to execute expressions, known as a REPL (Read-Eval-Print-Loop). The REPL can restrict the evaluation of I/O expressions based on their determinism type so that run-time errors cannot occur and the user does not experience unexpected side effects as in Prolog systems.

More precise warnings. In order to avoid unintended uses of non-determinism, the front end of Curry systems issue a warning if an operation is defined by overlapping rules. For instance, when processing the definition of `isNothing` as defined in Section 1, the Curry front end shows

```
Warning: Function `isNothing' is potentially
non-deterministic due to overlapping rules
```

To avoid such warnings in intended uses of overlapping rules (as in the definition of "?"), one can turn off the warnings (globally or per module) but then one might miss warnings about unintended uses of overlapping rules. An elegant resort of these conflicting goals are determinism types. If the programmer explicitly annotates an operation with a determinism type, it is checked whether its definition satisfies this determinism type according to the rules in Figure 4. Moreover, no overlapping warning is issued when the target determinism type is *Any*. Thus, the definition

```
(?) :? Det → Det → Any
(?) :: a → a → a
x ? y = x
x ? y = y
```

is accepted without any warning.

Compiler optimizations. Curry implementations that compile to Haskell like KiCS2 [16] and KMCC [24] use determinism information to optimize its code generation. Purely functional (hence, deterministic) parts of a program can be compiled into more efficient code. For instance, the correct handling of non-determinism when combined with lazy evaluation requires the threading of choice identifiers through function calls [4]. This can be avoided if expressions are known to be deterministic.

Program analysis tools. Determinism types can be used in static analysis tools to check program invariants or identify unexpected sources of non-determinism. Determinism analysis is often performed with simple domains for first-order programs, as in [11, 25]. Our determinism types provide a more refined abstraction of determinism information which can be extended in various ways, as discussed in the following section.

8 Extensions

This section describes two different extensions of our basic proposal to add determinism types to source programs.

8.1 Determinism Types in Type Classes

As discussed in Section 1, an extension of the determinism type system is to allow the user to specify that a method of a given type class is only allowed to be deterministic, enforcing instances of that class to adhere to this restriction. This is reasonable for the method `show` of the type class `Show` due to its intended use in I/O operations. Another interesting example is discussed in the following.

In [26] the type class `Data` is proposed with the following definition:

```
class Data a where
  (===) :: a → a → Bool
  aValue :: a
```

Intuitively, the method "===" denotes syntactic equality between values of type `a`, and `aValue` non-deterministically enumerates all values of type `a`. Instances of `Data` are automatically derived for all first-order types, i.e., types without functional components, which correspond to D as defined in Section 3. The motivation of this proposal is to provide syntactic unification and value bindings for free variables only for reasonable types. Since enumerating functional values is useless (and difficult in a strongly typed language), free variables, unification, and search operators have a `Data` context on their types, i.e., they are not parametric polymorphic but overloaded entities.

Although this extension, available in recent Curry systems, is useful to provide better compile-time checks for reasonable programs, it has also a disadvantage for efficient implementations of Curry. To see this, consider the following example where the deterministic operation `isZero` uses the equality operator "===":

```
data Peano = Zero | Succ Peano

isZero :: Peano → Bool
isZero p = p === Zero
```

Both the operation `isZero` and the operator "===" are deterministic. However, the type class `Data` also contains the non-deterministic operation `aValue`. Since compilers for Curry use dictionary passing to implement type classes [34], the compiled function for `isZero` has an additional argument: a `Data` dictionary with a non-deterministic implementation of `aValue`.

Curry compilers intended to produce efficient code for purely functional computations, like KiCS2 [16] and KMCC [24], use a determinism analysis on the intermediate language in order to compile deterministic operations quite similar to purely functional languages. In the example of `isZero`, the code is not considered as deterministic since it contains as a parameter the `Data` dictionary with the non-deterministic operation `aValue` even though it is not used. Although this seems to be a tiny problem in this example, the use of `Data` contexts in encapsulated subcomputations results in serious memory problems in larger Curry applications where the entire application is compiled to potentially non-deterministic code, although non-determinism is not used or only used in small subcomputations (see also [24] for benchmarks comparing compiled code with and without determinism optimizations).

By adding determinism type annotations to the type class `Data`, we can ensure that the compiler can infer the correct determinism annotation for `isZero`. The new definition of `Data` with determinism annotations would look like this:

```
class Data a where
  (===) :? Det → Det → Det
  (===) :: a → a → Bool

  aValue :? Any
  aValue :: a
```

This extension allows the determinism checker to distinguish between deterministic and non-deterministic operations within the same type class, even without knowing the code of the instances. Since this is correct only if the compiler checks all `Data` instances, this requires an extensions of the compiler based on our determinism types.

8.2 Polymorphic Determinism Types

A downside of a monomorphic determinism type system is that it does not always lead to the desired result for some higher-order functions. For example, consider the following function to flip the order of arguments of a binary operation:

```
flip :? (Det → Det → Det) → Det → Det → Det
flip :: (a   → b   → c)   → b   → a   → c
flip f x y = f y x
```

The annotation of `flip` means that the function is deterministic if the function and other arguments passed to it are deterministic. Now consider the application `flip const`. The determinism type of `const` is $Det \rightarrow Any \rightarrow Det$ which is actually $Det \rightarrow (Any \rightarrow Det)$ since "→" associates to the right. Since this is a subtype of the type $Det \rightarrow Det \rightarrow Det$ required for the first argument of `flip`, rule App infers that `flip const` has determinism type $Det \rightarrow Det \rightarrow Det$. Hence, we have a loss of precision since the information about the possibility to pass a first non-deterministic argument to `flip const` and still obtain a deterministic result if the last argument is deterministic is lost.

To solve this problem, we can extend the determinism type system to allow for polymorphic types. That way, the determinism annotation of `flip` would be the same as its type signature and the partially applied `flip const` would be inferred as `Any → Det → Det`. By introducing determinism type variables (analogous to type variables), we can express more general determinism signatures:

```
flip :? (a → b → c) → b → a → c
```

This allows the determinism of `flip f x y` to depend precisely on the determinism of `f`, `x`, and `y`, enabling more accurate inference and fewer false positives.

The introduction of determinism type variables comes with the price of a more complex type system and type inference algorithms. Apart from this specific example, it is not clear whether there are more realistic situations where this extension yields more precision. Therefore, we leave this extension for future work.

9 Implementation

We have implemented a prototype of the determinism type system as an extension of the Curry front end used by several Curry systems.[2] The implementation already includes the type class extension discussed in Section 8.1. Since type inference with subtyping is quite hard to implement, even for decidable type systems, our implementation uses a best-effort approach (note that one can always infer a determinism type, though it might lack precision). Our type system does not guarantee principality of types, i.e., it is possible that a program has multiple determinism types where one is not more specific than the other.

Type inference proceeds similar to type inference in a Hindley-Milner type system with type variables to be unified during the inference process. Since subtyping is relevant only in rule App, it is sufficient to check the subtyping relation for function applications. If during type inference, the type of the applied function is a metavariable (i.e., a type that is yet to be determined), we delay

[2]This implementation is available at https://github.com/cau-placc/curry-frontend/tree/det_types

the subtyping check. In the end, each type gets fully instantiated to obtain a monomorphic type.

Our type inference is trivially complete, because we can assign the type *Any* (or a function type with *Any*) whenever inference fails. However, since we have not yet proven that our inference algorithm always infers valid types, we currently plan to check the result using the determinism typing rules in Figure 4 for verification.

The implementation in the existing Curry front end was minimally invasive, as we only needed to add some syntax, a separate new type checking phase, and some small changes to the interface files required for modular compilation.

We leave a full exploration of type inference for our determinism types as future work.

10 Related Work

Our approach relates to various areas: program analysis, since it is intended to analyze the run-time behavior of programs at compile time; type systems, since we add non-standard types to a strongly typed language; gradual typing, since we use a type-based approach which should not restrict the number of admissible programs but ensure run-time conditions for particular types; effect systems, since the non-deterministic behavior of computations can be considered as computations with effects. In the following, we briefly relate our approach to these areas.

10.1 Program Analysis

Abstract interpretation is a program-analysis technique based on computing with abstract domains which classify sets of concrete values [18]. In this sense, our type *Det* corresponds to a set of values without a choice at the root, see the evaluation rules in Figure 5, whereas *Any* just denote all values. Abstract interpretation has a long tradition in logic programming to optimize their execution by approximating information about modes, types, and sharing at compile time [17]. Information about modes can be used to derive information about determinism behavior, since a predicate like append for list concatenation is deterministic if the first argument is ground. In this case, more efficient code can be generated.

Modes are exploited in [20] to check and infer functional computations in Prolog which are a generalization of deterministic computations. There, modes are used to ensure the mutual exclusion of rules of a predicate if a call to this predicate satisfies the given mode. This information is propagated through a call graph to infer functional properties of other predicates. The language Mercury [41] combine modes with a strong type system and determinism annotations to produce highly efficient code. Mercury classifies predicates as deterministic, semi-deterministic, or non-deterministic, which parallels our *Det* and *Any* classifications but with finer granularity. A key difference to our approach is that mode annotations, which are essential to analyze determinism properties of Mercury programs [28], put strong restrictions on the set of admissible programs and require additional efforts when programming with Mercury. Our approach does not require modes and is intended to infer determinism types in general so that the programmer provides determinism type annotations only in certain cases, e.g., to suppress non-determinism warnings (as discussed in Section 7) or put restrictions on instances of type classes, see Section 8.1. Moreover, Mercury is a strict language, while Curry uses lazy evaluation and, thus, needs to account for potential non-determinism in function arguments in a more nuanced way.

Similarly to logic programming, abstract interpretation techniques have been also applied to functional logic programming and Curry. We already mentioned the relevance of information about deterministic subcomputations to produce efficient code so that compilers from Curry into deterministic target languages, like KiCS2 [16] and KMCC [24], integrate a determinism analysis. CASS [25] is a generic analysis system for Curry and provides more than 30 different kinds of program analyses, including a determinism analysis. Since CASS is based on a first-order intermediate language, where partial applications are represented as data constructors by defunctionalization [37], the determinism information for higher-order arguments is less precise in CASS. Moreover, CASS is intended to analyze complete programs whereas determinism types can be integrated in the standard type inference of the front end. For instance, determinism types can be used in IDEs to immediately point to problematic uses of non-determinism in I/O operations.

10.2 Extended Type Systems

The idea to encode determinism behavior in a strongly typed language with an extended type system is not new. For instance, Steimann [42] proposed an extension of the lambda calculus, called simply numbered lambda calculus (SNLC), to include strings of objects to represent multiple values. In order to distinguish parameter-passing mechanism for multiple values (comparable to the difference between call-time choice and run-time choice mentioned above), parameters in lambda abstractions are annotated with number specifiers. The basic number specifiers are ! and *, which correspond to our determinism types *Det* and *Any*, and their functional combination, called mapping constraints, to cover higher-order functions. In contrast to our approach, the SNLC requires precise numbering annotations for higher-order arguments where a method to infer them is not mentioned (and probably not intended since the numbering annotations are semantically relevant). This is different from our approach where we allow more flexibility by making *Any* a supertype of any function (see Figure 3). This is important to avoid restrictions by adding determinism types and it paves the way towards the determinism typing of polymorphic operations, as discussed in Section 8. Moreover, we consider a non-strict operational semantics which is necessary to use determinism typing for Curry.

A proposal to add determinism information to standard types can be found in [11] where an operation is considered as deterministic if all computed results are identical. This information is used to improve the efficiency of functional logic programs by avoiding irrelevant computations. In contrast to our approach, these determinism annotations are based on trust since methods to check them are not provided.

10.3 Gradual Typing

The requirement that any valid expression can be assigned a determinism type makes our approach similar to gradually typed languages, where types can be added incrementally. For example, *Type Script* has the type any for cases when one does not know what type a value might be [35]. In contrast, our determinism types are not a fallback type but rather an annotation that can be used to ensure that a function is deterministic. Gradual typing was formally introduced by Siek and Taha [39] for functional languages, allowing programs to evolve from dynamic to static typing incrementally. One of the most mature implementations of this approach is Typed Scheme (now Typed Racket) [43], which demonstrates how untyped Scheme programs can be gradually converted to statically typed programs with minimal disruption. Similar to our approach with determinism types, Typed Scheme maintains backward compatibility while providing stronger guarantees through optional annotations, requiring no changes to the underlying run-time system.

Particularly relevant to our work is research on gradual typing for logic programming languages. For example, Schrijvers et al. [38] developed a gradual typing system for Prolog. Their system allows programmers to specify varying levels of type information, from fully dynamic to fully static typing. While we share the goal of being able to assign a type to any expression, our approach introduces an additional type system, orthogonal to the standard Curry type system.

10.4 Effect Systems

The integration of type systems with effect and determinism tracking is a well-established area. Effect systems in functional programming languages, such as those proposed for ML or Haskell (e.g., monads), encode side effects in types to control and reason about program behavior.

Lucassen and Gifford's foundational work on polymorphic effect systems [32] introduced the concept of annotating types with effects to track and control computational behaviors. This approach enabled precise reasoning about which effects an expression might produce. Similarly, our determinism types can be viewed as a specialized effect system focused on non-deterministic choice effects.

Monads, popularized by Wadler [44], offer another mechanism to encapsulate and reason about effects in functional languages. While monads in Haskell can represent non-determinism (e.g., via list monads), this approach differs from ours in that it requires explicit lifting of operations into the monadic context, whereas our system naturally extends the existing typing framework of Curry where non-determinism is implicit in the standard type of a function.

More recent work on effect systems, such as Leijen's Koka language with row polymorphic effect types [31], provides a framework for tracking diverse computational effects. Koka can express determinism as one of many possible effects but lacks specialized support for the specific semantics of functional logic programming where non-determinism is a first-class feature rather than an optional effect.

While effect systems track side effects such as state or exceptions, determinism typing focuses specifically on the presence or absence of non-deterministic choices. This makes it a lightweight and focused tool for reasoning about program behavior in languages where non-determinism is a core feature.

11 Conclusion

In this paper we presented a type-based approach for analyzing and controlling the determinism behavior of Curry programs. Determinism types can be considered as non-standard types which are attached to defined operations. With determinism typing, we gain a compositional, static method to ensure the safe execution of I/O operations so that run-time errors often present in Curry programs can be caught at compile time.

Our system is lightweight, formally grounded, and compatible with existing Curry implementations so that it does not restrict the set of admissible programs. It improves program reliability by catching an important class of run-time errors, supports optimization, and enhances the developer experience in both interactive and compiled environments.

For future work, it might be interesting to extend determinism types to approximate other behaviors, like partially defined operations. Although there exist methods to infer sufficient conditions so that operations do not fail in Curry [22, 23], explicit annotations might allow the programmer to express intended partiality so that the compiler can suppress standard warnings in this case, similarly to the non-determinism warnings discussed in Section 7. We could even consider using intervals of determinism types to express the number of results computed by an operation.

Another direction to be explored is to capture more nuanced non-deterministic behavior. For instance, we consider the expression `(failed ? True)` as non-deterministic, although it has only one result. There are also situations where more than one result is computed but all of them are identical, as considered in [11]. A challenge of these extensions is to find techniques which ensure the correctness of such a different view of determinism.

References

[1] Elvira Albert, Michael Hanus, Frank Huch, Javier Oliver, and Germán Vidal. 2005. Operational Semantics for Declarative Multi-Paradigm Languages. *Journal of Symbolic Computation* 40, 1 (2005), 795–829. https://doi.org/10.1016/j.jsc.2004.01.001
[2] Abdulla Alqaddoumi, Sergio Antoy, Sebastian Fischer, and Fabian Reck. 2010. The Pull-Tab Transformation. In *Proc. of the Third International Workshop on Graph Computation Models*. Enschede, The Netherlands, 127–132. Available at http://gcm2010.imag.fr/pages/gcm2010-preproceedings.pdf.
[3] Sergio Antoy. 2001. Constructor-based Conditional Narrowing. In *Proc. of the 3rd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP 2001)*. ACM Press, 199–206. https://doi.org/10.1145/773184.773205
[4] Sergio Antoy. 2011. On the Correctness of Pull-Tabbing. *Theory and Practice of Logic Programming* 11, 4-5 (2011), 713–730. https://doi.org/10.1017/S1471068411000263
[5] Sergio Antoy and Bernd Braßel. 2007. Computing with Subspaces. In *Proceedings of the 9th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'07)*. ACM Press, 121–130.
[6] Sergio Antoy, Rachid Echahed, and Michael Hanus. 2000. A Needed Narrowing Strategy. *J. ACM* 47, 4 (2000), 776–822. https://doi.org/10.1145/347476.347484
[7] Sergio Antoy and Michael Hanus. 2005. Declarative Programming with Function Patterns. In *Proceedings of the International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'05)*. Springer LNCS 3901, 6–22. https://doi.org/10.1007/11680093_2
[8] Sergio Antoy and Michael Hanus. 2006. Overlapping Rules and Logic Variables in Functional Logic Programs. In *Proceedings of the 22nd International Conference on Logic Programming (ICLP 2006)*. Springer LNCS 4079, 87–101. https://doi.org/10.1007/11799573_9
[9] Sergio Antoy and Michael Hanus. 2009. Set Functions for Functional Logic Programming. In *Proceedings of the 11th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'09)*. ACM Press, 73–82. https://doi.org/10.1145/1599410.1599420

[10] Sergio Antoy and Michael Hanus. 2010. Functional Logic Programming. *Commun. ACM* 53, 4 (2010), 74–85. https://doi.org/10.1145/1721654.1721675

[11] Sergio Antoy and Michael Hanus. 2017. Eliminating Irrelevant Non-determinism in Functional Logic Programs. In *Proc. of the 19th International Symposium on Practical Aspects of Declarative Languages (PADL 2017)*. Springer LNCS 10137, 1–18. https://doi.org/10.1007/978-3-319-51676-9_1

[12] Lennart Augustsson, Joachim Breitner, Koen Claessen, Ranjit Jhala, Simon Peyton Jones, Olin Shivers, Guy L. Steele, and Tim Sweeney. 2023. The Verse Calculus: A Core Calculus for Deterministic Functional Logic Programming. In *Proc. ACM International Conference on Functional Programming (ICFP 2023)*. 203:1–203:31. https://doi.org/10.1145/3607845

[13] Yves Bertot and Pierre Castéran. 2004. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Springer. https://doi.org/10.1007/978-3-662-07964-5

[14] Jonas Böhm, Michael Hanus, and Finn Teegen. 2021. From Non-determinism to Goroutines: A Fair Implementation of Curry in Go. In *Proc. of the 23rd International Symposium on Principles and Practice of Declarative Programming (PPDP 2021)*. ACM Press, 16:1–16:15. https://doi.org/10.1145/3479394.3479411

[15] Bernd Braßel, Michael Hanus, and Frank Huch. 2004. Encapsulating Non-Determinism in Functional Logic Computations. *Journal of Functional and Logic Programming* 2004, 6 (2004).

[16] Bernd Braßel, Michael Hanus, Björn Peemöller, and Fabian Reck. 2011. KiCS2: A New Compiler from Curry to Haskell. In *Proc. of the 20th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2011)*. Springer LNCS 6816, 1–18. https://doi.org/10.1007/978-3-642-22531-4_1

[17] Maurice Bruynooghe, Gerda Janssens, Alain Callebaut, and Bart Demoen. 1987. Abstract interpretation: towards the global optimization of Prolog programs. In *Proc. 4th IEEE Internat. Symposium on Logic Programming*. San Francisco, 192–204.

[18] Patrick Cousot and Rhadia Cousot. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *Proc. of the 4th ACM Symposium on Principles of Programming Languages*. 238–252. https://doi.org/10.1145/512950.512973

[19] Javier de Dios Castro and Francisco J. López-Fraguas. 2007. Extra variables can be eliminated from functional logic programs. *Electronic Notes in Theoretical Computer Science* 188 (2007), 3–19. https://doi.org/10.1016/j.entcs.2006.05.049

[20] Saumya K. Debray and David S. Warren. 1989. Functional Computations in Logic Programs. *ACM Trans. Program. Lang. Syst.* 11, 3 (1989), 451–481. https://doi.org/10.1145/65979.65984

[21] Michael Hanus. 2013. Functional Logic Programming: From Theory to Curry. In *Programming Logics - Essays in Memory of Harald Ganzinger*. Springer LNCS 7797, 123–168. https://doi.org/10.1007/978-3-642-37651-1_6

[22] Michael Hanus. 2024. Hybrid Verification of Declarative Programs with Arithmetic Non-fail Conditions. In *Proc. of the 22nd Asian Symposium on Programming Languages and Systems (APLAS 2024)*. Springer LNCS 15194, 109–129. https://doi.org/10.1007/978-981-97-8943-6_6

[23] Michael Hanus. 2024. Inferring Non-Failure Conditions for Declarative Programs. In *Proc. of the 17th International Symposium on Functional and Logic Programming (FLOPS 2024)*. Springer LNCS 14659, 167–187. https://doi.org/10.1007/978-981-97-2300-3_10

[24] Michael Hanus, Kai-Oliver Prott, and Finn Teegen. 2022. A Monadic Implementation of Functional Logic Programs. In *Proc. of the 24th International Symposium on Principles and Practice of Declarative Programming (PPDP 2022)*. ACM Press, 1:1–1:15. https://doi.org/10.1145/3551357.3551370

[25] Michael Hanus and Fabian Skrlac. 2014. A Modular and Generic Analysis Server System for Functional Logic Programs. In *Proc. of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation (PEPM'14)*. ACM Press, 181–188. https://doi.org/10.1145/2543728.2543744

[26] Michael Hanus and Finn Teegen. 2020. Adding `Data` to Curry. In *Declarative Programming and Knowledge Management - Conference on Declarative Programming (DECLARE 2019)*. Springer LNCS 12057, 230–246. https://doi.org/10.1007/978-3-030-46714-2_15

[27] Michael Hanus (ed.). 2016. Curry: An Integrated Functional Logic Language (Vers. 0.9.0). Available at http://www.curry-lang.org.

[28] Fergus Henderson, Zoltan Somogyi, and Thomas Conway. 1996. Determinism analysis in the Mercury compiler. In *Proc. of the Nineteenth Australian Computer Science Conference*. 337–346.

[29] Gérard P. Huet and Jean-Jaques Lévy. 1991. Computations in Orthogonal Rewriting Systems. In *Computational Logic: Essays in Honor of Alan Robinson*, J.-L. Lassez and G. Plotkin (Eds.). MIT Press, 395–443.

[30] Heinrich Hussmann. 1992. Nondeterministic Algebraic Specifications and Nonconfluent Term Rewriting. *Journal of Logic Programming* 12 (1992), 237–255. https://doi.org/10.1016/0743-1066(92)90026-Y

[31] Daan Leijen. 2014. Koka: Programming with Row Polymorphic Effect Types. In *Proceedings 5th Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2014, Grenoble, France, 12 April 2014 (EPTCS, Vol. 153)*, Paul Blain Levy and Neel Krishnaswami (Eds.). 100–126. https://doi.org/10.4204/EPTCS.153.8

[32] John M. Lucassen and David K. Gifford. 1988. Polymorphic effect systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '88)*. Association for Computing Machinery, New York, NY, USA, 47–57. https://doi.org/10.1145/73560.73564

[33] Connor McBride and Ross Paterson. 2008. Applicative programming with effects. *Journal of Functional Programming* 18, 1 (2008), 1–13. https://doi.org/10.1017/S0956796807006326

[34] John Peterson and Mark P. Jones. 1993. Implementing Type Classes. In *Proc. of ACM SIGPLAN SYmposium on Programming Language Design and Implementation (PLDI'93)*. ACM SIGPLAN Notices Vol. 28, No. 6, 227–236. https://doi.org/10.1145/155090.155112

[35] Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin Bierman, and Panagiotis Vekris. 2015. Safe & Efficient Gradual Typing for TypeScript. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Mumbai, India and New York, NY, USA) *(Popl '15)*. Association for Computing Machinery, 167–180. https://doi.org/10.1145/2676726.2676971

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

[37] John C. Reynolds. 1972. Definitional Interpreters for Higher-Order Programming Languages. In *Proceedings of the ACM Annual Conference*. ACM Press, 717–740. https://doi.org/10.1145/800194.805852

[38] Tom Schrijvers, Vítor Santos Costa, Jan Wielemaker, and Bart Demoen. 2008. Towards Typed Prolog. In *Proceedings of the 24th International Conference on Logic Programming* (Berlin, Heidelberg) *(ICLP '08)*. Springer-Verlag, 693–697. https://doi.org/10.1007/978-3-540-89982-2_59

[39] Jeremy G. Siek and Walid Taha. 2006. Gradual Typing for Functional Languages. In *Scheme and Functional Programming Workshop* (Portland, Oregon, USA). 81–92. http://scheme2006.cs.uchicago.edu/13-siek.pdf

[40] James R. Slagle. 1974. Automated Theorem-Proving for Theories with Simplifiers, Commutativity, and Associativity. *J. ACM* 21, 4 (1974), 622–642. https://doi.org/10.1145/321850.321859

[41] Zoltan Somogyi, Fergus Henderson, and Thomas Conway. 1996. The Execution Algorithm of Mercury, an Efficient Purely Declarative Logic Programming Language. *The Journal of Logic Programming* 29, 1-3 (1996), 17–64. https://doi.org/10.1016/S0743-1066(96)00068-4

[42] Friedrich Steimann. 2023. A Simply Numbered Lambda Calculus. In *Eelco Visser Commemorative Symposium, EVCS 2023 (OASIcs, Vol. 109)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 24:1–24:12. https://doi.org/10.4230/OASICS.EVCS.2023.24

[43] Sam Tobin-Hochstadt and Matthias Felleisen. 2008. The design and implementation of Typed Scheme. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '08)*. Association for Computing Machinery, New York, NY, USA, 395–406. https://doi.org/10.1145/1328438.1328486

[44] Philip Wadler. 1992. The essence of functional programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Albuquerque, New Mexico, USA) *(POPL '92)*. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/143165.143169

[45] Philip Wadler. 1997. How to Declare an Imperative. *Comput. Surveys* 29, 3 (1997), 240–263. https://doi.org/10.1145/262009.262011

[46] Philip Wadler and Stephen Blott. 1989. How to make ad-hoc polymorphism less ad hoc. In *Proc. of the 16th ACM Symposium on Principles of Programming Languages (POPL'89)*. 60–76. https://doi.org/10.1145/75277.75283