Inferring Non-Failure Conditions for Declarative Programs

Michael Hanus

Institut für Informatik, Kiel University, Germany
`mh@informatik.uni-kiel.de`

Abstract. Unintended failures during a computation are painful but frequent during software development. Failures due to external reasons (e.g., missing files, no permissions) can be caught by exception handlers. Programming failures, such as calling a partially defined operation with unintended arguments, are often not caught due to the assumption that the software is correct. This paper presents an approach to verify such assumptions. For this purpose, non-failure conditions for operations are inferred and then checked in all uses of partially defined operations. In the positive case, the absence of such failures is ensured. In the negative case, the programmer could adapt the program to handle possibly failing situations and check the program again. Our method is fully automatic and can be applied to larger declarative programs. The results of an implementation for functional logic Curry programs are presented.

1 Introduction

The occurrence of failures during a program execution is painful but still frequent when developing software systems. The main reasons for such failures are

– external, i.e., outside the control of the program, like missing files or access rights, unexpected formats of external data, etc.
– internal, i.e., programming errors like calling a partially defined operation with unintended arguments.

External failures can be caught by exception handlers to avoid a crash of the entire software system. Internal failures are often not caught since they should not occur in a correct software system. In practice, however, they occur during software development and even in deployed systems which results in expensive debugging tasks. For instance, a typical internal failure in imperative programs is dereferencing a pointer variable whose current value is the null pointer (due to this often occurring failure, Tony Hoare called the introduction of null pointers his "billion dollar mistake"[1]).

Although null pointer failures cannot occur in declarative programs, such programs might contain other typical programming errors, like failures due to incomplete pattern matching. For instance, consider the following operations (shown in Haskell syntax):

[1] `http://qconlondon.com/london-2009/speaker/Tony+Hoare`

```
head :: [a]  →  a                tail :: [a]  →  [a]
head (x:xs) = x                  tail (x:xs) = xs
```

In a correct program, it must be ensured that `head` and `tail` are not evaluated on empty lists. If we are not sure about the data provided at run time, we can check the arguments of partial operations before the application. For instance, the following code snippet defines an operation to read a command together with some arguments from standard input (the operation `words` breaks a string into a list of words separated by white spaces) and calls an operation `processCmd` with the input data:

```
readCmd = do putStr "Input a command:"
             s <- getLine
             let ws = words s
             case null ws of True  → readCmd
                             False → processCmd (head ws) (tail ws)
```

By using the predicate `null` to check the emptiness of a list, it is ensured that `head` and `tail` are not applied to an empty list in the `False` branch of the case expression.

In this paper we present a fully automatic tool which can verify the non-failure of this program. Our technique is based on analyzing the types of arguments and results of operations in order to ensure that partially defined operations are called with arguments of appropriate types. The principle idea to use type information for this purpose is not new. For instance, one can express restrictions on arguments of operations with *dependent types*, as in Agda [35], Coq [10], or Idris [11], or *refinement types*, as in LiquidHaskell [39,40]. Since one has to prove that these restrictions hold during the construction of programs, the development of such programs becomes harder [38]. Another alternative, proposed in [21], is to annotate operations with *non-fail conditions* and verify that these conditions hold at each call site by an external tool, e.g., an SMT solver [16]. In this way, the verification is fully automatic but requires user-defined annotations and, in some cases, also the verification of post-conditions or contracts to state properties about result values of operations [22].

The main idea of this work is to *infer* non-fail conditions of operations. Since the inference of precise conditions is undecidable in general, we approximate them by *abstract types*, e.g., finite representations of sets of values. Hence, our contributions are:

1. We define a *call type* for each operation. If the actual arguments belong to the call type, the operation is reducible with some rule.
2. For each operation, we define *in/out types* to approximate its input/output behavior.
3. For each call to an operation g occurring in a rule defining f, we check, by considering the call structure and in/out types, whether the call type of g is satisfied. If this is not the case, the call type of f is refined and we repeat the checks with the refined call type.

At the end of this process, each operation has some correct call type which ensures that it does not fail on arguments belonging to its call type. Note that the call type might be empty on always failing operations. To avoid empty call types, one can modify the program code so that a different branch is taken in case of a failure.

In order to make our approach accessible to various declarative languages, we formulate and implement it in the declarative multi-paradigm language Curry [26]. Since Curry extends Haskell by logic programming features and there are also methods to transform logic programs into Curry programs [23], our approach can also be applied to purely functional or logic programs. A consequence of using Curry is the fact that programs might compute with failures, e.g., it is not an immediate programming error to apply `head` and `tail` to possibly empty lists. However, subcomputations involving such possibly failing calls must be encapsulated so that it can be checked whether such a computation has no result (this corresponds to exception handling in deterministic languages). If this is done, one can ensure that the overall computation does not fail even in the presence of encapsulated logic (non-deterministic) subcomputations.

The paper is structured as follows. After sketching the basics of Curry in the next section, we introduce call types and their abstraction in Sect. 3. Section 4 defines in/out types and methods to approximate them. The main section 5 presents our method to infer and check call types for all operations in a program. We evaluate our approach in Sect. 6 before we conclude with a discussion of related work. More details as well as correctness results and their proofs can be found in [24].

2 Functional Logic Programming and Curry

The declarative language Curry [26] amalgamates features from functional programming (demand-driven evaluation, strong typing, higher-order functions) and logic programming (computing with partial information, unification, constraints), see [6,20] for surveys. The syntax of Curry is close to Haskell [36]. In addition to Haskell, Curry applies rules with overlapping left-hand sides in a (don't know) non-deterministic manner (where Haskell always selects the first matching rule) and allows *free* (*logic*) *variables* in conditions and right-hand sides of defining rules. The operational semantics is based on an optimal lazy evaluation strategy [4].

Curry is strongly typed so that a Curry program consists of data type definitions (introducing *constructors* for data types) and *functions* or *operations* on these types. As an example, we show the definition of two operations: the list concatenation "`++`" and an operation `dup` which returns some number having at least two occurrences in a list:[2]

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

[2] Note that Curry requires the explicit declaration of free variables, as `x` in the rule of `dup`, to ensure checkable redundancy, except for anonymous variables, denoted by an underscore.

$$
\begin{array}{lcll}
P & ::= & D_1 \dots D_m & \text{(program)} \\
D & ::= & f(x_1, \dots, x_n) = e & \text{(function definition)} \\
e & ::= & x & \text{(variable)} \\
 & \mid & c(x_1, \dots, x_n) & \text{(constructor application)} \\
 & \mid & f(x_1, \dots, x_n) & \text{(function call)} \\
 & \mid & e_1 \; \mathit{or} \; e_2 & \text{(disjunction)} \\
 & \mid & \mathit{let} \; x_1, \dots, x_n \; \mathit{free} \; \mathit{in} \; e & \text{(free variables)} \\
 & \mid & \mathit{let} \; x = e \; \mathit{in} \; e' & \text{(let binding)} \\
 & \mid & \mathit{case} \; x \; \mathit{of} \; \{p_1 \to e_1; \dots; p_n \to e_n\} & \text{(case expression)} \\
p & ::= & c(x_1, \dots, x_n) & \text{(pattern)}
\end{array}
$$

Fig. 1. Syntax of the intermediate language FlatCurry

Since `dup` might deliver more than one result for an argument, e.g., `dup [1,2,2,1]` yields `1` and `2`, it is also called a *non-deterministic operation*. Such operations, which are interpreted as mappings from values into sets of values [19], are an important feature of contemporary functional logic languages. To express failing computations, there is also a predefined operation `failed` which has no value.

Curry has more features than described so far.[3] Due to these numerous features, language processing tools for Curry (compilers, analyzers,. . .) often use an intermediate language where the syntactic sugar of the source language has been eliminated and the pattern matching strategy is explicit. This intermediate language, called FlatCurry, has also been used, apart from compilers, to specify the operational semantics of Curry programs [1] or to implement a modular framework for the analysis of Curry programs [25]. Since we will use FlatCurry to describe and implement our inference method, we sketch the structure of FlatCurry programs.

Figure 1 summarizes the abstract syntax of FlatCurry. A FlatCurry program consists of a sequence of function definitions (we omit data type definitions here), where each function is defined by a single rule. Patterns in source programs are compiled into case expressions, overlapping rules are joined by explicit disjunctions, and arguments of constructor and function calls are variables (introduced in left-hand sides, let expressions, or patterns). We will write $\mathcal{F}$ for the set of defined operations and $\mathcal{C}$ for the set of constructors of a program. In order to provide a simple definition of our inference method, we assume that FlatCurry programs satisfy the following properties:

- All variables introduced in a rule (parameters, free variables, let bindings, pattern variables) have unique identifiers.
- For the sake of simplicity, let bindings are non-recursive, i.e., all recursion is introduced by functions (although our implemented tool supports recursive bindings).

[3] Conceptually, Curry is intended as an extension of Haskell although not all extensions of Haskell are actually supported.

– The patterns in each case expression are non-overlapping and cover all data constructors of the type of the discriminating variable. Hence, if this type contains n constructors, there are n branches without overlapping patterns. This can be ensured by adding missing branches with failure expressions (`failed`).

Usually, the front end of a Curry compiler transforms source programs into such a form [3,7]. For instance, the operation `head` is transformed into the FlatCurry definition

```
head(zs) = case zs of { x:xs  →  x ; []  →  failed }
```

3 Call Types and Abstract Types

We consider a computation as *non-failing* if it does not stop due to a pattern mismatch or a call to `failed`. In order to infer conditions on arguments of operations so that the evaluation of an operation does not fail, we will analyze the rules of each operation.[4] For instance, the operation `head` is not defined on empty lists so that the condition for a non-failing evaluation of `head` is the non-emptiness of the argument list. Sometimes the exact condition requires more advanced descriptions. Consider the operation

```
lastTrue [True]   = True
lastTrue (x:y:ys) = lastTrue (y:ys)
```

The evaluation of a call `lastTrue` l does not fail if the argument list l ends with `True`. Although such lists could be finitely described using regular types [15], such a description is impossible for arbitrary operations. For instance, if some branch in a condition of an operation causes a failure but the condition of the branch contains a function call, the failure is only relevant if the function call terminates. Due to the undecidability of the halting problem, we cannot hope to infer exact non-failure conditions.

Due to this general problem, we *approximate* non-failure conditions so that the evaluation of a call with arguments satisfying the non-failure condition is non-failing. However, there might be successfully evaluable calls which do not satisfy the inferred non-failure condition.

In order to support different structures to approximate non-failure conditions, we do not fix a language for call types but assume that there is a domain $\mathcal{A}$ of *abstract types*. Elements of this domain describe sets of concrete *data terms*, i.e., terms consisting of data constructors only. There are various options for such abstract types, like depth-k abstractions [37] or regular types [15]. The latter have been used to infer success types to analyze logic programs [18], whereas depth-k abstractions were used in the abstract diagnosis of functional programs [2] or in the abstraction of term rewriting systems [8,9]. Since regular types are more complex and computationally more expensive, we use depth-k abstractions in our examples. In this domain, denoted by A_k, subterms exceeding the given

[4] Note that we do not consider external failures of operations, like file access errors, since they need to be handled differently.

depth k are replaced by a specific constant ($\top$) that represents any term. Since the size of this domain is quickly growing for $k > 1$, we use $k = 1$ in examples, i.e., terms are approximated by their top-level constructors. As we will see, this is often sufficient in practice to obtain reasonable results. Nevertheless, our technique and implementation is parametric over the abstract type domain.

If $\mathcal{C}$ is the set of data constructors, *depth-1 types* can be simply described by the set

$$\mathcal{A}_1 = \{D \subseteq \mathcal{C} \mid \text{all constructors of } D \text{ belong to the same type}\} \cup \{\top\}$$

Hence, each element of $\mathcal{A}_1$ is either a set of data constructors of the same type or $\top$. The latter denotes the set of all data terms when no type information is available.

Following the framework of abstract interpretation [14], the meaning of abstract values is specified by a concretization function γ. For $\mathcal{A}_1$, γ is defined by

$$\begin{array}{rcl} \gamma(\top) & = & \{t \mid t \text{ is a data term}\} \\ \gamma(D) & = & \{t \mid t = c(t_1, \ldots, t_n) \text{ is a data term with } c \in D\} \end{array}$$

Thus, $\varnothing$ is the bottom element of this domain w.r.t. the standard ordering defined by $a \sqsubseteq \top$ for any a, and $a_1 \sqsubseteq a_2$ if $a_1 \subseteq a_2$.

In the following, we present a framework for the inference of call types which is parametric over the abstract domain $\mathcal{A}$. Thus, we assume that $\mathcal{A}$ is a lattice with an ordering $\sqsubseteq$, greatest lower bound ($\sqcap$) and least upper bound ($\sqcup$) operations, a least or bottom element $\bot$, and a greatest or top element $\top$. Furthermore, for each n-ary data constructor c, there is an *abstract constructor application* c^α which maps abstract values $a_1, \ldots, a_n$ into an abstract value a such that $c(t_1, \ldots, t_n) \in \gamma(a)$ for all $t_1 \in \gamma(a_1), \ldots, t_n \in \gamma(a_n)$. For the domain $\mathcal{A}_1$, this can be defined by $c^\alpha(x_1, \ldots, x_n) = \{c\}$ (it could also be defined by $c^\alpha(x_1, \ldots, x_n) = \top$ but this yields less precise approximations).

We use $\mathcal{A}$ to specify *call types* or *non-failure conditions* for operations. Let f be a unary operation (the extension to more than one argument is straightforward). A call type $C \in \mathcal{A}$ is *correct* for f if the evaluation of $f(t)$ is non-failing for any $t \in \gamma(C)$. For instance, the depth-1 type $\{:\}$ is correct for the operations `head` or `tail` defined above.

In order to verify the correctness of call types for a program, we have to check whether each call of an operation satisfies its call type. Since this requires the analysis of conditions and other operations (see the operation `readCmd` defined in Sect. 1), we will approximate the input/output behavior of operations, as described next.

4 In/Out Types

To provide a fully automatic inference method for call types, we need some knowledge about the behavior of auxiliary operations. For instance, consider the operation

```
null []     = True
null (x:xs) = False
```

This operation is used in the definition of `readCmd` (see Sect. 1) to ensure that `head` and `tail` are applied to non-empty lists. In order to verify this property, we have to infer that, if "`null ws`" evaluates to `False`, the argument is a non-empty list.

For this purpose, we associate an in/out type to each operation. An *in/out type io* for an n-ary operation f is a set of elements containing a sequence of $n+1$ abstract types:

$$io \subseteq \{a_1 \cdots a_n \hookrightarrow a \mid a_1, \ldots, a_n, a \in \mathcal{A}\}$$

The first n components of each element approximate input values (where we write ε if $n = 0$) and the last component approximate output values associated to the inputs. An in/out type io is *correct* for f if, for each value t' of $f(t_1, \ldots, t_n)$, there is some $a_1 \cdots a_n \hookrightarrow a \in io$ such that $t_i \in \gamma(a_i)$ $(i = 1, \ldots, n)$ and $t' \in \gamma(a)$.

In/out types are disjunctions of possible input/output behaviors of an operation. For instance, a correct in/out type of `null` is $\{\{\texttt{[]}\} \hookrightarrow \{\texttt{True}\}, \{\texttt{:}\} \hookrightarrow \{\texttt{False}\}\}$ (w.r.t. $\mathcal{A}_1$). Another trivial and less precise in/out type is $\{\top \hookrightarrow \top\}$.

In/out types allow also to express non-terminating operations. For instance, a correct in/out type for the operation `loop` defined by

```
loop = loop
```

is $\{\varepsilon \hookrightarrow \varnothing\}$. The empty type in the result indicates that this operation does not yield any value.

Similarly to call types, we approximate in/out types since the inference of precise in/out types is intractable in general. For this purpose, we analyze the definition of each operation and associate patterns to result values. Result values are based on general information about the abstract result types of operations. Therefore, we assume that there is a mapping $R : \mathcal{F} \to \mathcal{A}$ which associates to each defined function $f \in \mathcal{F}$ an abstract type $R(f) \in \mathcal{A}$ approximating the possible values to which f (applied to some arguments) can be evaluated. For instance, $R(\texttt{loop}) = \varnothing$, $R(\texttt{null}) = \{\texttt{False}, \texttt{True}\}$, and $R(\texttt{head}) = \top$ (w.r.t. the domain $\mathcal{A}_1$). Approximations for R can be computed in a straightforward way by a fixpoint computation. Using the Curry analysis framework CASS [25], this program analysis can be defined in 20 lines of code—basically a case distinction on the structure of FlatCurry operations.

Our actual approximation of in/out types is defined by the rules in Fig. 2. A sequence $o_1, \ldots, o_n$ of objects is abbreviated by $\overline{o_n}$. We use a *type environment* Γ which maps variables into abstract types. We denote by $\Gamma[x \mapsto e]$ the environment Γ' with $\Gamma'(x) = e$ and $\Gamma'(y) = \Gamma(y)$ for all $x \neq y$. The judgement $\Gamma \vdash e : \{\overline{\Gamma_k \hookrightarrow a_k}\}$ is interpreted as "the evaluation of the expression e in the context Γ yields a new context Γ_i and result value of abstract type a_i, for some $i \in \{1, \ldots, k\}$." To infer an in/out type io of an operation f defined by $f(x_1, \ldots, x_n) = e$, we derive the judgement $\{\overline{x_n \mapsto \top}\} \vdash e : \{\overline{\Gamma_k \hookrightarrow a_k}\}$ and return the in/out type

$$io = \{\Gamma_i(x_1) \cdots \Gamma_i(x_n) \hookrightarrow a_i \mid i = 1, \ldots, k\}$$

$$\begin{array}{rcl}
\mathsf{Var} & \Gamma \vdash x : \{\Gamma \hookrightarrow \Gamma(x)\} & (x \text{ variable}) \\
\mathsf{Cons} & \Gamma \vdash c(x_1,\ldots,x_n) : \{\Gamma \hookrightarrow c^{\alpha}(\Gamma(x_1),\ldots,\Gamma(x_n))\} & (c \text{ constructor}) \\
\mathsf{Func} & \Gamma \vdash f(x_1,\ldots,x_n) : \{\Gamma \hookrightarrow R(f)\} & (f \text{ operation}) \\
\mathsf{Or} & \dfrac{\Gamma \vdash e_1 : io_1 \quad \Gamma \vdash e_2 : io_2}{\Gamma \vdash e_1 \ or\ e_2 : io_1 \cup io_2} & \\
\mathsf{Free} & \dfrac{\Gamma[\overline{x_n \mapsto \top}] \vdash e : io}{\Gamma \vdash let\ x_1,\ldots,x_n\ free\ in\ e : io} & \\
\mathsf{Let} & \dfrac{\Gamma[x \mapsto \top] \vdash e' : io}{\Gamma \vdash let\ x = e\ in\ e' : io} & \\
\mathsf{Case} & \dfrac{\Gamma_1 \vdash e_1 : io_1 \quad \ldots \quad \Gamma_n \vdash e_n : io_n}{\Gamma \vdash case\ x\ of\ \{p_1 \to e_1; \ldots ; p_n \to e_n\} : io_1 \cup \ldots \cup io_n} & \\
 & \text{where } p_i = c_i(\overline{x_{n_i}}) \text{ and } \Gamma_i = \Gamma[x \mapsto c_i^{\alpha}(\overline{\top}), \overline{x_{n_i} \mapsto \top}] &
\end{array}$$

Fig. 2. Approximation of in/out types

Thus, we derive an in/out type without any restriction on the arguments.

Let us consider the inference rules in more detail. In the case of variables or applications, the type environment is not changed and the approximated result is returned, e.g., the abstract type of the variable (rule Var), the abstract representation of the constructor (rule Cons), or the approximated result value of the operation (rule Func). Rule Or combines the results of the different branches. Rules Free and Let add the new variables to the type environment with most general types. Although one could refine these types, we try to keep the analysis simple since this seems to be sufficient in practice.

The most interesting rule is Case. The results from the different branches are combined, but inside each branch, the type of the discriminating variable x is refined to the constructor of the branch. For instance, consider the operation

```
null(zs) = case zs of { []  →  True ; (x:xs)  →  False }
```

If we analyze the in/out type with our rules, we start with the type environment $\Gamma_0 = \{\texttt{zs} \mapsto \top\}$. Inside the branch, Γ_0 is refined to $\Gamma_1 = \{\texttt{zs} \mapsto \{\texttt{[]}\}\}$ and $\Gamma_2 = \{\texttt{zs} \mapsto \{\texttt{:}\}, \texttt{x} \mapsto \top, \texttt{xs} \mapsto \top\}$, respectively, so that the in/out type (w.r.t. $\mathcal{A}_1$) derived for `null` is $\{\{\texttt{[]}\} \hookrightarrow \{\texttt{True}\}, \{\texttt{:}\} \hookrightarrow \{\texttt{False}\}\}$.

In our implementation, we keep in/out types in a normalized form where different pairs with identical input types are joined by the least upper bound of their output types. Moreover, the in/out types of failed branches are omitted so that we obtain

$\texttt{head} : \{\{\texttt{:}\} \hookrightarrow \top\}$
$\texttt{tail} : \{\{\texttt{:}\} \hookrightarrow \top\}$

5 Inference and Checking of Call Types

Based on the pieces introduced in the previous sections, we can present our method to infer and verify call types for all operations in a given program. Basically, our method performs the following steps:

1. The in/out types for all operations are computed (see Sect. 4).
2. Initial call types for all operations are computed by considering the left-hand sides or case structure of their defining rules.
3. These call types are abstracted w.r.t. the abstract type domain.
4. For each call to an operation g occurring in a rule defining operation f, we check, by considering the call structure and in/out types, whether the call type of g is satisfied.
5. If some operation cannot be verified due to unsatisfied call type restrictions, its call type is refined by considering the additional call-type constraints due to operations called in its right-hand side, and start again with step 4.

This fixpoint computation terminates if the abstract type domain is finite (which is the case for depth-k types) or it is ensured that there are only finitely many refinements for each call type in step 5 (which could be ensured by widening steps in infinite abstract domains [14]). In the worst case, an empty call type might be inferred for some operation. This does not mean that this operation is not useful but one has to encapsulate its use with some safeness check.

In the following, we describe these steps in more detail.

5.1 Initial Call Types

Concrete call types are easy to derive by considering the structure of `case` expressions in the transformed FlatCurry program. If all constructors of some data type are covered in non-failed branches of some `case` construct, there is no call type restriction due to this pattern matching. Otherwise, the call type restriction consists of those constructors occurring in non-failed branches. For instance, the operation `null` has no call type restriction, whereas the operations `head` and `tail` have failed branches for the empty list so that the call type restriction could be expressed by the set of terms

$$\{t_1 : t_2 \mid t_1, t_2 \text{ are arbitrary terms}\}$$

As already discussed, we map such sets into a finite representation by using abstract types. Hence, the *abstract call type* of an n-ary operation is a sequence of elements of $\mathcal{A}$ of length n. We say that such a type is *trivial* if all elements in this sequence are $\top$. In case of the abstract type domain $\mathcal{A}_1$, the set above is abstracted to $\{:\}$, thus, it is non-trivial. Since the derivation of concrete call types and their abstraction is straightforward, we omit further details here.

5.2 Call Type Checking

We assume that two kinds of information are given for each operation f:

$$\mathsf{Var}_{nf} \qquad \Delta, z = x \vdash \{(z, \{\varepsilon \mapsto \Delta(x)\}, \varepsilon)\}$$

$$\mathsf{Cons}_{nf} \qquad \Delta, z = c(x_1, \ldots, x_n) \vdash \{(z, \{\top^n \hookrightarrow c^\alpha(\overline{\Delta(x_n)})\}, x_1 \ldots x_n)\}$$

$$\mathsf{Func}_{nf} \qquad \frac{CT(f) = a_1 \ldots a_n \quad \Delta(x_i) \sqsubseteq a_i \ (i = 1, \ldots, n)}{\Delta, z = f(x_1, \ldots, x_n) \vdash \{(z, IO(f), x_1 \ldots x_n)\}}$$

$$\mathsf{Or}_{nf} \qquad \frac{\Delta, z = e_1 \vdash \Delta_1 \quad \Delta, z = e_2 \vdash \Delta_2}{\Delta, z = e_1 \ or \ e_2 \vdash \Delta_1 \cup \Delta_2}$$

$$\mathsf{Free}_{nf} \qquad \frac{\Delta \cup \{x_1 :: \top, \ldots, x_n :: \top\}, z = e \vdash \Delta'}{\Delta, z = let \ x_1, \ldots, x_n \ free \ in \ e \vdash \Delta'}$$

$$\mathsf{Let}_{nf} \qquad \frac{\Delta, x = e \vdash \Delta' \quad \Delta \cup \Delta', z = e' \vdash \Delta''}{\Delta, z = let \ x = e \ in \ e' \vdash \Delta''}$$

$$\mathsf{Case}_{nf} \qquad \frac{\Delta_{r_1}, z = e_{r_1} \vdash \Delta'_{r_1} \quad \ldots \quad \Delta_{r_k}, z = e_{r_k} \vdash \Delta'_{r_k}}{\Delta, z = case \ x \ of \ \{p_1 \to e_1; \ldots; p_n \to e_n\} \vdash \Delta'_{r_1} \cup \ldots \cup \Delta'_{r_k}}$$

where $p_i = c_i(\overline{x_{n_i}})$, $\Delta_i = (\Delta \wedge [x \mapsto c_i]) \cup \{x_1 :: \top, \ldots, x_{n_i} :: \top\}$,
and $r_1, \ldots, r_k$ are the reachable branches (i.e., $\Delta_{r_j}(x) \neq \bot$)

Fig. 3. Call type checking

- An in/out type $IO(f)$ approximating the input/output behavior of f.
- An abstract call type $CT(f)$ specifying the requirements to evaluate f without failure.

$IO(f)$ can be computed as shown in Sect. 4. $CT(f)$ can be approximated as discussed above, but we have to show that all calls to f actually satisfy these requirements. This is the purpose of the inference system shown in Fig. 3.

As discussed in Sect. 4, it is important to have information about the input/output behavior of operations. Therefore, we introduced the notion of in-/out types. Now we use this information to approximate values of variables occurring in program rules and pass this information through the rules during checking time. For this purpose, we use *variable types* which are triples of the form $(z, io, x_1 \ldots x_n)$ where $z, x_1, \ldots, x_n$ are program variables and io is an in-/out type for an n-ary operation. This is interpreted as: z might have some value of the result type a for some $a_1 \ldots a_n \hookrightarrow a \in io$ and, in this case, $x_1, \ldots, x_n$ have values of type $a_1, \ldots, a_n$, respectively. For instance, the variable type

$(z,\ \{\{$`[]`$\} \hookrightarrow \{$`True`$\}, \{$`:`$\} \hookrightarrow \{$`False`$\}\},\ xs)$

expresses that z might have value `True` and xs is an empty list, or z has value `False` and xs is a non-empty list. Since we approximate values, we abstract a set of variable environments with concrete values for variables to a set of variable types. If such a set contains only one triple for some variable and the io component is a one-element set, we can use it for definite reasoning. To have a more compact notation for the abstract type of a program variable, we denote by $x :: a$ the triple $(x, \{\varepsilon \mapsto a\}, \varepsilon)$.

Now we have a closer look at the rules of Fig. 3. This inference system derives judgements of the form $\Delta, z = e \vdash \Delta'$ containing sets of variable types Δ, Δ', a variable z, and an expression e. This is interpreted as "if Δ holds, then the expression e evaluates without a failure and, if z is bound to the result of this evaluation, Δ' holds." To check the call type $a_1 \ldots a_n$ of an operation f defined by $f(x_1, \ldots, x_n) = e$, we try to derive the judgement

$$\{x_1 :: a_1, \ldots, x_n :: a_n\}, z = e \vdash \Delta$$

for some fresh variable z. Thus, we assign the call types as initial values of the parameters and analyze the right-hand side of the operation.

Keeping the interpretation of variable types in mind, the inference rules are not difficult to understand. $\Delta(x)$ denotes the least upper bound of all abstract type information about variable x available in Δ, which is defined by

$$\Delta(x) = \bigsqcup \{a \mid (x, \{\ldots, a_1 \ldots a_n \hookrightarrow a, \ldots\}, \ldots) \in \Delta\}$$

Rule Var_{nf} is immediate since the evaluation of a value cannot fail so that we set the result z to the abstract type of x. Rule Cons_{nf} adds the simple condition that z is bound to the constructor c after the evaluation ($\top^n = \top \ldots \top$ is a sequence of n $\top$ elements). Rule Func_{nf} is the first interesting rule. The condition states that the abstract arguments of the function must be smaller than the required call type so that the concrete values are in a subset relationship. If the requirements on call types hold, the operation is evaluable and we connect the results and the arguments with the in/out type of the operation. The rules for disjunctions and free variable introduction are straightforward. In rule Let_{nf}, the result of analyzing the local binding is used to analyze the expression. We finally discuss the most important rule for case selections.

In rule Case_{nf}, $\Delta \wedge [x \mapsto c_i]$ denotes the set of variable types Δ modified by the definite binding of x to the constructor c_i. This means that, if Δ contains a triple (x, io, xs), all result values in io which are incompatible to c_i are removed. After this modification of Δ, it may happen that $\Delta(x)$ is the empty type, i.e., there is no concrete value which x can have so that this branch is *unreachable*. Therefore, the right-hand side of this branch need not be analyzed so that rule Case_{nf} does not consider them. For the remaining reachable branches, the right-hand side is analyzed with the modified set of variable types so that the value in the specific branch value is considered.

As an example, we check the simple operation

```
f(x) = let y = null(x) in case y of True   → True
                                    False  → head(x)
```

For the abstract type domain $\mathcal{A}_1$, the in/out type of `null` is

$IO(\texttt{null}) = \{\{\texttt{[]}\} \hookrightarrow \{\texttt{True}\}, \{\texttt{:}\} \hookrightarrow \{\texttt{False}\}\}$

and the abstract call type of `head` is $\{\texttt{:}\}$. When we check the right-hand side of the definition of `f`, we start the checking of the `case` (after having checked the `let` binding) with the set of variable types

$\Delta_1 = \{(\texttt{x}, \{\varepsilon \hookrightarrow \top\}, \varepsilon),\ \ (\texttt{y}, \{\{\texttt{[]}\} \hookrightarrow \{\texttt{True}\}, \{\texttt{:}\} \hookrightarrow \{\texttt{False}\}\}, \texttt{x})\}$

The check of the first case branch is immediate. For the second case branch, we modify the previous set of variable types to $\Delta_2 = \Delta_1 \wedge [\mathtt{y} \mapsto \mathtt{False}]$ so that we have

$\Delta_2 = \{(\mathtt{x}, \{\varepsilon \hookrightarrow \top\}, \varepsilon),\ (\mathtt{y}, \{\{\mathtt{:}\} \hookrightarrow \{\mathtt{False}\}\}, \mathtt{x})\}$

The definite binding for `y` implies a definite binding for `x` so that Δ_2 is equivalent to

$\Delta_3 = \{(\mathtt{x}, \{\varepsilon \hookrightarrow \{\mathtt{:}\}\}, \varepsilon),\ (\mathtt{y}, \{\{\mathtt{:}\} \hookrightarrow \{\mathtt{False}\}\}, \mathtt{x})\}$

Hence, if we check the call "`head(x)`" w.r.t. Δ_3, the abstract argument type is $\Delta_3(\mathtt{x}) = \{\mathtt{:}\}$ so that the call type of `head` is satisfied.

As we have seen in this example, sets of variable types should be kept in a simplified form in order to deduce most precise type information. For instance, the definite bindings of variables, like $(\mathtt{y}, \{\{\mathtt{:}\} \hookrightarrow \{\mathtt{False}\}\}, \mathtt{x})$, should be propagated to get a definitive binding for `x`. Although this is not explicitly stated in the inference rules, we assume that it is always done whenever sets of variable types are modified.

5.3 Iterated Call Type Checking

Consider the operation

```
hd(x) = head(x)
```

Applying the inference rules of Fig. 3 is not successful: the initial abstract call type for `hd` is $\top$ so that the call type requirement for `head` is not satisfied.

In order to compute call types for all operations, we try to refine the call type of `hd`. For this purpose, we collect the requirements on variables for unsatisfied call types during the check of an operation. If such a required type is on some variable occurring in the left-hand side of an operation, the call type of the operation is restricted and the operation is checked again. In case of the operation `hd`, the failure in the call `head(x)` leads to the requirement that `x` must have the abstract type $\{\mathtt{:}\}$ so that we check `hd` again but with this new call type—which is now successful.

There are also cases where such a refinement is not possible. For instance, consider the slightly modified example

```
hdfree(x) = let y free in head(y)
```

Since the type restriction $\{\mathtt{:}\}$ on variable `y` can not be obtained by restricting the call type of `hdfree`, we assume the most restricted call type $CT(\mathtt{hdfree}) = \{\}$. This means that any call to `hdfree` might fail so that one has to encapsulate calls to `hdfree` with some safeness check.

This strategy leads to an iterated analysis of call types. In each iteration, either all call types can be verified or the call type of some operation becomes more restricted. This iteration always terminates if one can ensure finitely many refinements of call types (which is the case for depth-k types).

For an efficient computation of this fixpoint computation, it is reasonable to use call dependencies of operations so that one has to re-check only the more restricted operations and the operations that use them. We have implemented this strategy in our tool and obtained a good improvement compared to the initial naive fixpoint computation. For instance, the prelude of Curry (the base module containing a lot of basic definitions for arithmetic, lists, type classes, etc) contains 1262 operations (public and also auxiliary operations). After the first iteration, the call types of 14 operations are refined so that 17 operations are reanalyzed in the next iteration. Altogether, the check of the prelude requires five iterations.

5.4 Extensions

Up to now, we presented the analysis of a kernel language. Since application programs use more features, we discuss in the following how to cover all features occurring in Curry programs.

Literals Programs might contain numbers or characters which are not introduced by explicit data definitions. Although there are conceptually infinitely many literals, their handling is straightforward. A literal can be treated as a 0-ary constructor. Since there are only finitely many literals in each program, the abstract types for a given program are also finite. For instance, consider the operation

```
k 0 = 'a'
k 1 = 'b'
```

The call type of `k` inferred w.r.t. domain $\mathcal{A}_1$ is $CT(\texttt{k}) = \{\texttt{0}, \texttt{1}\}$. Similarly, the in/out type of `k` is $IO(\texttt{k}) = \{\{\texttt{0}\} \hookrightarrow \{\texttt{'a'}\}, \{\texttt{1}\} \hookrightarrow \{\texttt{'b'}\}\}$.

External operations Usually, externally defined primitive operations do not fail so that they have trivial call types. There are a few exceptions which are handled by explicitly defined call types, like the always failing operation `failed`, or arithmetic operations like division.

Higher-order operations Since it is seldom that generic higher-order operations have functional parameters with non-trivial call types, we take a simple approach to check higher-order operations. We assume that higher-order arguments have trivial call types and check this property for each call to a higher-order operation. Thus, a call like "`map head [[1,2],[3,4]]`" is considered as potentially failing. Our practical evaluation shows this assumption provides reasonable results in practice.

Encapsulation Failures might occur during run time, either due to operations with complex non-failure conditions (e.g., arithmetic) or due to the use of logic programming techniques with search and failures. In order to ensure an overall non-failing application in the presence of possibly failing subcomputations, the programmer has to encapsulate such subcomputations and then analyze its outcome, e.g., branching on the result of the encapsulation. For this purpose, one can use an exception handler (which represents a failing computation as an error value) or some method to encapsulate non-deterministic search (e.g., [5,12,29,30]). For instance, the primitive operation `allValues` returns all the values of its argument expression in a list so that a failure corresponds to an empty list. In order to include such a primitive in our framework, we simply skip the analysis of its arguments. For instance, a source expression like `allValues (head ys)` is not transformed into `let x = head(y) in allValues(x)` (where `x` is fresh), but it is kept as it is. Furthermore, rule Func_{nf} is specialized for `allValues` so that the condition on the arguments w.r.t. the call type is omitted and the in/out type is trivial, i.e., $IO(\texttt{allValues}) = \{\top \hookrightarrow \top\}$. In a similar way, other methods to encapsulate possibly non-deterministic and failing operations, like *set functions* [5], can be handled.

Errors as Failures The operation `error` is an external operation to emit an error message and terminate the program (if it does not occur inside an exception handler). Since we are mainly interested to avoid internal programming errors, `error` is not considered as a failing operation in the default mode. Thus, if we change the definition of `head` into (as in the prelude of Haskell)

```
head :: [a]  → a
head []     = error "head: empty list"
head (x:xs) = x
```

the inferred call type is $\top$ so that the call "`head []`" is not considered as failing. From some point of view, this is reasonable since the evaluation does not fail but shows a result—the error message.

However, in safety-critical applications we want to be sure that all errors are caught. In this case, we can still use our framework and define the call type of `error` as $\bot$ so that any call to `error` is considered as failing. Moreover, exception handlers can be treated similarly to encapsulated search operators as described above. In order to be flexible with the interpretation of `error`, our tool (see below) provides an option to set one of these two views of `error`.

6 Evaluation

We have implemented the methods described above in a tool[5] written in Curry. In the following we evaluate it by discussing some examples and applying it to various libraries.

First, we compare our approach to a previous tool to verify non-failing Curry programs [21]. In that tool the programmer has to annotate partially defined operations with *non-fail conditions.* Based on these conditions, the tool extracts proof obligations from a program which are sent to an SMT solver. For instance, consider the operation to compute the last element of a non-empty list:

```
last [x]      = x
last (_:x:xs) = last (x:xs)
```

[5] Available as package `https://cpm.curry-lang.org/pkgs/verify-non-fail-1.0.0.html`

The condition to express the non-failure of this expression must be explicitly defined as a predicate on the argument:

```
last'nonfail xs = not (null xs)
```

This predicate together with the definition of the involved operations are translated to SMT formulas and then checked by an SMT solver, e.g., Z3 [16]. Using our approach, the abstract call type $CT(\texttt{last}) = \{\texttt{:}\}$ is automatically inferred and the definition of `last` is successfully checked. Actually, we tested our tool on various libraries and could deduce almost all manually written non-fail conditions of [21]. Only in four prelude operations, our tool could not infer these non-fail conditions since they contain arithmetic conditions on integers. We leave it for future work to combine our approach with an SMT solver to enable also successful checks in these cases.

Another interesting example is the operation `split` from the library `Data.List`. This operation takes a predicate and a list as arguments and splits this list into sublists at positions where the predicate holds. It is defined in Curry as

```
split :: (a  →  Bool)  →  [a]  →  [[a]]
split _ []                  = [[]]
split p (x:xs) | p x        = [] : split p xs
               | otherwise = let (ys:yss) = split p xs
                              in (x:ys):yss
```

Since the pattern in the let expression is translated into partially defined selector functions in FlatCurry, this definition cannot be directly verified by [21] due to missing non-fail conditions for these auxiliary operations. Furthermore, a post-condition on `split` must be stated and proved. Our method infers all these conditions and verifies the non-failure of `split`.

If our tool is applied to a Curry module, it infers the in/out types and the call types of all operations defined in this module and then checks all branches and calls whether they might be failing. If this is the case, the call types are refined and the problematic ones are reported to the user. Then the user can decide to either accept the refined call types or modify the program code to handle possible failures so that the call type does not need a refinement.

Table 1 contains the results of checking various Curry libraries with our tool. The "operations" column contains the number of public (exported) operations and the number of all operations defined in the module. Similarly, the following three columns shows the information for public and all operations. The "in/out types" column shows the numbers of non-trivial in/out types. The initial and final call types are the number of non-trivial call types computed at the beginning and obtained after some iterations (the number of iterations is shown in the next to last column). The "final failing" column contains the number of operations where an empty call type is inferred, i.e., there is no precise information about the required call types. The last column shows the verification time in milliseconds.[6]

As one can see from this table, even quite complex modules, like the prelude, have only a few operations with non-trivial call types that need to be checked.

[6] We measured the verification time on a Linux machine running Ubuntu 22.04 with an Intel Core i7-1165G7 (2.80GHz) processor with eight cores.

Module	operations	in/out types	initial call types	final call types	final failing	itera-tions	verify time
`Prelude`	862/1262	605/857	24/32	63/71	45/53	5	969
`Data.Char`	9/9	0/0	0/0	0/0	0/0	1	272
`Data.Either`	7/11	5/9	2/2	2/2	0/0	1	113
`Data.List`	49/87	39/73	7/15	8/16	1/1	2	290
`Data.Maybe`	8/9	7/8	0/0	0/0	0/0	1	113
`Numeric`	5/7	0/2	0/0	0/0	0/0	1	273
`System.Console.GetOpt`	6/47	5/41	0/0	0/0	0/0	1	287
`System.IO`	32/51	10/12	0/0	0/0	0/0	1	115
`Text.Show`	4/4	4/4	0/0	0/0	0/0	1	110

Table 1. Inference of call types for some standard libraries

Therefore, the effort to infer and check modules is limited. The higher number of failing operations in the prelude are the various arithmetic division operators and enumeration and parsing operations where a precise call type cannot be inferred.

7 Related Work

The exclusion of run-time failures at compile time is a practically relevant but also challenging issue. Therefore, there are many approaches targeting it so that we can only discuss a few of them. We concentrate on approaches for functional and logic programming, although there are also many in the imperative world. As mentioned in the introduction, the exclusion of dereferencing null pointers is quite relevant there. As an example from object-oriented programming, the Eiffel compiler uses appropriate type declarations and static analysis to ensure that pointer dereference failures cannot occur in accepted programs [31].

In logic programming, there is no common definition of “non-failing” due to different interpretations of non-determinism. Whereas we are interested to exclude any failure in a top-level computation, other approaches, like [13,17], consider a predicate in a logic program as non-failing if at least one answer is produced. Similarly to our approach, type abstractions are used to approximate non-failure properties, but the concrete methods are different.

Another notion of failing programs in a dynamically typed programming language is based on success types, e.g., as used in Erlang [28]. Success types over-approximate possible uses of an operation so that an empty success type indicates an operation that never evaluates to some value. Thus, success types can show definite failures, whether we are interested in definite non-failures.

Strongly typed programming languages are a reasonable basis to check run-time failures at compile time, since the type system already ensures that some kind of failures cannot occur (“well-typed programs do not go wrong” [32]). However, failures due to definitions with partial patterns are not covered by a standard type system. Therefore, Mitchell and Runciman developed a checker for Haskell to verify the absence of pattern-match errors due to incomplete patterns [33,34]. Their checker extracts and solves specific constraints from pattern-based definitions. Although these constraints have similarities to the abstract type domain $\mathcal{A}_1$, our approach is generic w.r.t. the abstract type domain so that it can also deal with more powerful abstract type domains.

An approach to handle failures caused by restrictions on number arguments is described in [27]. It is based on generating (arithmetic) constraints which are translated into an imperative program such that the constraints are satisfiable iff the translated program is safe. This enables the inference of complex conditions on numbers, but pattern matching with algebraic data types and logic-oriented subcomputations are not supported.

Another approach to ensure the absence of failures is to make the type system stronger or more expressive in order to encode non-failing conditions in the types. For instance, operations in dependently typed programming languages, such as Coq [10], Agda [35], or Idris [11], must be totally defined, i.e., terminating and non-failing. Such languages have termination checkers but non-fail conditions need to be explicitly encoded in the types. For instance, the definition of the operation `head` in Agda [35] requires, as an additional argument, a proof that the argument list is not empty. Thus, `head` could have the type signature

```
head : {A : Set} → (xs : List A) → is-empty xs == ff → A
```

Therefore, each use of `head` must provide, as an additional argument, an explicit proof for the non-emptiness of the argument list `xs`. Type-checked Agda programs do not contain run-time failures but programming in a dependently typed language is more challenging since the programmer has to construct non-failure proofs.

Refinement types, as used in LiquidHaskell [39,40], are another approach to encode non-failing conditions or more general contracts on the type level. Refinement types extend standard types by a predicate that restricts the set of allowed values. For instance, the applications of `head` to the empty list can be excluded by the following refinement type [39]:

```
head :: {xs : [a] | 0 < len xs} → a
```

The correctness of refinement types is checked by an SMT solver so that they are more expressive than our non-failure conditions. On the other hand, refinement types must be explicitly added by the programmer whereas our goal is to infer non-failure conditions from a standard program. This allows the use of potentially failing operations in encapsulated subcomputations, which is relevant to use logic programming techniques. This aspect is also the motivation for the non-failure checking tool proposed in [21]. As already discussed in Sect. 6, the advantage of our tool is the automatic inference of non-failing conditions which supports an easier application to larger programs.

8 Conclusions

In this paper we proposed a new technique and a fully automatic tool to check declarative programs for the absence of failing computations. In contrast to other approaches, our approach does not require the explicit specification of non-fail conditions but is able to infer them. In order to provide flexibility with the structure of non-fail conditions, our approach is generic w.r.t. a domain of abstract types to describe non-fail conditions. Since we developed our approach for Curry, it is also applicable to purely functional or logic programs. Due to the use of Curry, we do not need to abandon all potentially failing operations. Partially defined operations and failing evaluations are still allowed in logic-oriented subcomputations provided that they are encapsulated in order to control possible failures.

Although the inference of non-fail conditions is based on a fixpoint iteration and might yield, in the worst case, an empty (i.e., always failing) condition, our practical evaluation showed that even larger programs contain only a few operations with non-trivial non-fail conditions which are inferred after a small number of iterations. When a non-trivial non-fail condition is inferred for some operation, the programmer can either modify the definition of this operation (e.g., by adding missing case branches) or control the invocation of this operation by checking its outcome with some control operator.

For future work, we plan to extend our approach to built-in types, like integers, and infer non-failure conditions on such types, like non-negative or positive numbers, and check them using SMT solvers. Furthermore, it is interesting to see whether other abstract domains, e.g., regular types, yield more precise results in application programs.

References

1. E. Albert, M. Hanus, F. Huch, J. Oliver, and G. Vidal. Operational semantics for declarative multi-paradigm languages. *Journal of Symbolic Computation*, 40(1):795–829, 2005.
2. M. Alpuente, M. Comini, S. Escobar, M. Falaschi, and S. Lucas. Abstract diagnosis of functional programs. In *Proc. of the 12th Int'l Workshop on Logic-Based Program Synthesis and Transformation (LOPSTR 2002)*, pages 1–16. Springer LNCS 2664, 2002.
3. S. Antoy. Constructor-based conditional narrowing. In *Proc. of the 3rd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP 2001)*, pages 199–206. ACM Press, 2001.
4. S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. *Journal of the ACM*, 47(4):776–822, 2000.
5. S. Antoy and M. Hanus. Set functions for functional logic programming. In *Proceedings of the 11th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'09)*, pages 73–82. ACM Press, 2009.
6. S. Antoy and M. Hanus. Functional logic programming. *Communications of the ACM*, 53(4):74–85, 2010.

7. S. Antoy, M. Hanus, A. Jost, and S. Libby. ICurry. In *Declarative Programming and Knowledge Management - Conference on Declarative Programming (DECLARE 2019)*, pages 286–307. Springer LNCS 12057, 2020.
8. D. Bert and R. Echahed. Abstraction of conditional term rewriting systems. In *Proc. of the 1995 International Logic Programming Symposium*, pages 147–161. MIT Press, 1995.
9. D. Bert, R. Echahed, and M. Østvold. Abstract rewriting. In *Proc. Third International Workshop on Static Analysis*, pages 178–192. Springer LNCS 724, 1993.
10. Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.
11. E. Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23(5):552–593, 2013.
12. B. Braßel, M. Hanus, and F. Huch. Encapsulating non-determinism in functional logic computations. *Journal of Functional and Logic Programming*, 2004(6), 2004.
13. F. Bueno, P. López-García, and M.V. Hermenegildo. Multivariant non-failure analysis via standard abstract interpretation. In *7th International Symposium on Functional and Logic Programming (FLOPS 2004)*, pages 100–116. Springer LNCS 2998, 2004.
14. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *Proc. of the 4th ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.
15. P.W. Dart and J. Zobel. A regular type language for logic programs. In F. Pfenning, editor, *Types in Logic Programming*, pages 157–187. MIT Press, 1992.
16. L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proc. of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008)*, pages 337–340. Springer LNCS 4963, 2008.
17. S. Debray, P. López-García, and M.V. Hermenegildo. Non-failure analysis for logic programs. In *14th International Conference on Logic Programming (ICLP'97)*, pages 48–62. MIT Press, 1997.
18. J.P. Gallagher and K.S. Henriksen. Abstract domains based on regular types. In *20th International Conference on Logic Programming (ICLP 2004)*, pages 27–42. Springer LNCS 3132, 2004.
19. J.C. González-Moreno, M.T. Hortalá-González, F.J. López-Fraguas, and M. Rodríguez-Artalejo. An approach to declarative programming based on a rewriting logic. *Journal of Logic Programming*, 40:47–87, 1999.
20. M. Hanus. Functional logic programming: From theory to Curry. In *Programming Logics - Essays in Memory of Harald Ganzinger*, pages 123–168. Springer LNCS 7797, 2013.
21. M. Hanus. Verifying fail-free declarative programs. In *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming(PPDP 2018)*, pages 12:1–12:13. ACM Press, 2018.
22. M. Hanus. Combining static and dynamic contract checking for Curry. *Fundamenta Informaticae*, 173(4):285–314, 2020.
23. M. Hanus. From logic to functional logic programs. *Theory and Practice of Logic Programming*, 22(4):538–554, 2022.
24. M. Hanus. Inferring non-failure conditions for declarative programs. *CoRR*, abs/2402.12960, 2024.

25. M. Hanus and F. Skrlac. 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)*, pages 181–188. ACM Press, 2014.
26. M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.9.0). Available at `http://www.curry-lang.org`, 2016.
27. R. Jhala, R. Majumdar, and A. Rybalchenko. HMC: verifying functional programs using abstract interpreters. In *23rd International Conference on Computer Aided Verification (CAV 2011)*, pages 470–485. Springer LNCS 6806, 2011.
28. T. Lindahl and K. Sagonas. Practical type inference based on success typings. In *Proceedings of the 8th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP 2006)*, pages 167–178. ACM Press, 2006.
29. F.J. López-Fraguas and J. Sánchez-Hernández. A proof theoretic approach to failure in functional logic programming. *Theory and Practice of Logic Programming*, 4(1):41–74, 2004.
30. W. Lux. Implementing encapsulated search for a lazy functional logic language. In *Proc. 4th Fuji International Symposium on Functional and Logic Programming (FLOPS'99)*, pages 100–113. Springer LNCS 1722, 1999.
31. B. Meyer. Ending null pointer crashes. *Communications of the ACM*, 60(5):8–9, 2017.
32. R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.
33. N. Mitchell and C. Runciman. A static checker for safe pattern matching in Haskell. In *Trends in Functional Programming*, volume 6, pages 15–30. Intellect, 2007.
34. N. Mitchell and C. Runciman. Not all patterns, but enough: an automatic verifier for partial but sufficient pattern matching. In *Proc. of the 1st ACM SIGPLAN Symposium on Haskell (Haskell 2008)*, pages 49–60. ACM, 2008.
35. U. Norell. Dependently typed programming in Agda. In *Proceedings of the 6th International School on Advanced Functional Programming (AFP'08)*, pages 230–266. Springer LNCS 5832, 2008.
36. S. Peyton Jones, editor. *Haskell 98 Language and Libraries—The Revised Report.* Cambridge University Press, 2003.
37. T. Sato and H. Tamaki. Enumeration of success patterns in logic programs. *Theoretical Computer Science*, 34:227–240, 1984.
38. A. Stump. *Verified Functional Programming in Agda.* ACM and Morgan & Claypool, 2016.
39. N. Vazou, E.L. Seidel, and R. Jhala. LiquidHaskell: Experience with refinement types in the real world. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*, pages 39–51. ACM Press, 2014.
40. N. Vazou, E.L. Seidel, R. Jhala, D. Vytiniotis, and S. Peyton Jones. Refinement types for Haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 269–282. ACM Press, 2014.