Hybrid Verification of Declarative Programs with Arithmetic Non-Fail Conditions

Michael Hanus

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

Abstract. Functions containing arithmetic operations have often restrictions not expressible by standard type systems of programming languages. The division operation requires that the divisor is non-zero and the factorial function should not be applied to negative numbers. Such partial operations might lead to program crashes if they are applied to unintended arguments. Checking the arguments before each call is tedious and decreases the run-time efficiency. To avoid these disadvantages and support the safe use of partially defined operations, we present an approach to verify the correct use of operations at compile time. To simplify its use, our approach automatically infers non-fail conditions of operations from their definitions and checks whether these conditions are satisfied for all uses of the operations. Arithmetic conditions can be verified by SMT solvers, whereas conditions in operations defined on algebraic data types can be inferred and verified by appropriate type abstractions. Therefore, we present a hybrid method which is applicable to larger programs since only a few arithmetic non-fail conditions need to be checked by an external SMT solver. This approach is implemented for functional logic Curry programs so that it is also usable for purely functional or logic programs.

1 Introduction

Programs often contain partially defined operations that do not yield meaningful results for particular argument values. A typical example is the division operation which is not defined if the divisor is zero. User-defined operations might also have restrictions on argument values. For instance, consider the following definition of the factorial function in the functional language Haskell [31]:

```
fac :: Int  → Int
fac n | n == 0 = 1
      | n > 0  = n * fac (n - 1)
```

Due to the conditions "`n == 0`" and "`n > 0`", a run-time error occurs if `fac` is applied to a negative number since there is no branch for this case. Such an error might be avoided at compile time by restricting the argument type of `fac` to natural numbers and checking whether each call satisfy this restriction. Unfortunately, this restriction is not expressible in type systems of current strongly typed declarative programming languages, such as Haskell. This is also due to

the fact that a call like `fac (m-n)` must be considered as ill-typed if `m` is smaller than `n`, i.e., the correct typing depends on values available at run time.

In order to avoid program crashes due to such errors, one could transform the factorial function into a total function that returns a specific value indicating a meaningless result. In Haskell, this could be expressed by using the predefined type of partial values

```
data Maybe a = Nothing | Just a
```

`Nothing` represents "no value" and `Just` x the value x. Using this type, we could define a "totalized" version of `fac` as follows:

```
facT :: Int  → Maybe Int
facT n | n < 0  = Nothing
       | n == 0 = Just 1
       | n > 0  = case facT (n - 1) of Nothing  → Nothing
                                       Just m   → Just (n * m)
```

This total programming style yields ugly and less comprehensible code (note that also each client of `facT` has to check and transform the computed result). Moreover, the code is less efficient due to the additional case distinction in each recursive call.

In order to use the partially defined function `fac` without the risk of run-time errors, one can check the value of the argument before the actual call. For instance, the following code snippet defines an operation to read a number and, if it is non-negative, prints its factorial (`readInt` reads a string from the user input until it is an integer):

```
printFac = do putStr "Factorial computation for: "
              n <- readInt
              if n<0 then putStrLn "Negative number!" >> printFac
                     else print (fac n)
```

By checking the value of `n` before evaluating `(fac n)`, `printFac` never fails.

In this paper we present a fully automatic tool which can verify the non-failure of this program. For this purpose, our tool infers the *non-fail condition*

```
fac'nonfail :: Int  → Bool
fac'nonfail n = (n == 0) || (n > 0)
```

from the definition of `fac`. Then it checks whether this condition is satisfied at all call sites of `fac`. For instance, it is satisfied for the recursive call `fac (n - 1)` since `n > 0`. This property is automatically checked by an SMT solver [29]. The entire process is iterative since a non-fail condition for some operation f might require new non-fail conditions for operations that use f. For the operation

```
fac2 n = n * fac (n + 2)
```

the non-fail condition of `fac` requires `fac2'nonfail n = (n+2 == 0) || (n+2 > 0)`.

Non-fail conditions [16] are predicates which restrict the standard types of operations so that run-time failures are excluded. This idea is also present in *dependent types*, as in Agda [30], Coq [9], or Idris [10], or *refinement types* [32], as used in LiquidHaskell [36,37]. Since the development of programs w.r.t. such advanced type systems requires more work [34], we intend to support the tra-

ditional programming style with automated verification support where non-fail conditions are inferred. If they are not precise enough, the programmer can provide explicit non-fail conditions.

We specify and implement our approach in the declarative multi-paradigm language Curry [19] which extends Haskell by logic programming features. Thus, our approach can also be applied to purely functional or logic programs [17]. In this context, it is important to distinguish non-fail conditions and preconditions [7]: a precondition *must* be satisfied for any call of an operation, whereas a satisfied non-fail condition assures that a computation does not fail. Hence, it is an error if a precondition is not satisfied for some call, whereas in a logic computation, where one searches for values or solutions, one could also invoke operations with unsatisfied non-fail conditions. Hence, non-fail conditions as well as preconditions are meaningful concepts in functional logic programming (computing with failures is a feature enabling new design patterns [4]). Since top-level computations should not fail, subcomputations with possible failures must be encapsulated by specific search handlers. This provides also an opportunity to deal with complex non-fail conditions which cannot be inferred or checked automatically: encapsulate calls to such operations and check whether such a subcomputation has no result. This view is different from approaches where constraints or dependent types are generated [20,32,35] to ensure that no failures occur.

This paper is structured as follows. After reviewing the basics of Curry in the next section, we discuss non-fail conditions in Sect. 3. Section 4 defines our method to check and infer valid non-fail conditions for all operations in a program. Section 5 reviews call types which can be considered as abstract non-fail conditions for algebraic data types. Both kinds of non-fail conditions are combined and implemented in our hybrid verification tool which we describe and evaluate in Sect. 6 before we conclude with a discussion of related work.

2 Functional Logic Programming with Curry

We develop and implement our method in Curry so that it is also applicable to purely functional or logic programs. The declarative programming language Curry [19] amalgamates features from functional programming (demand-driven evaluation, strong typing, higher-order functions) and logic programming (computing with partial information, unification, constraints), see [6,15] for surveys. The syntax of Curry is close to Haskell[1] [31]. 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 combines lazy and non-deterministic evaluation [3].

A Curry program consists of data type definitions introducing *constructors* for data types (as shown with the type `Maybe` in Sect. 1) and *functions* or *oper-*

[1] Variables and function names start with lowercase letters and the names of type and data constructors start with an uppercase letter. The application of f to e is denoted by juxtaposition ("$f\ e$").

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

Fig. 1. Syntax of the intermediate language FlatCurry

ations on these types. As an example, we show the definition of two operations on lists: the list concatenation "`++`" and an operation `aPos` which returns some positive number occurring in a list of integers:[2]

```
(++) :: [a] → [a] → [a]      aPos :: [Int] → Int
[]     ++ ys = ys            aPos xs | xs == ys++[z]++zs && z>0
(x:xs) ++ ys = x : (xs ++ ys)          = z    where ys,z,zs free
```

The equation "`xs == ys++[z]++zs`" in the condition of `aPos` is solved by searching for appropriate lists such that they concatenate to `xs` and contain some element `z`. `aPos` is also called a *non-deterministic operation*, because it might deliver more than one value for a given argument, e.g., `aPos [0,-1,2,-3,4]` yields `2` and `4`. Such operations, interpreted as mappings from values into sets of values [14], are an important feature of contemporary functional logic languages.

To collect the results of non-deterministic operations and use them in purely deterministic computations (e.g., to print them), Curry offers *search handlers*, i.e., operations to encapsulate non-deterministic computations and return their results in some data structure (e.g., [5,11,22,23]). For instance, `allValues` returns all values of its argument expression in a list. The handler `oneValue` returns a single value in a `Maybe` structure and `Nothing` in case of a failure. These operations can be used to avoid program crashes with partially defined operations. For instance, the total operation `facT` shown in Sect. 1 can be defined as

```
facT n = oneValue (fac n)
```

Since Curry has many more features than described so far, language processing tools for Curry (compilers, analyzers,...) often use an intermediate language, called FlatCurry [1,15], where the syntactic sugar of the source language has been eliminated and the pattern matching strategy is explicit. Since our inference method is based on FlatCurry, we sketch the structure of FlatCurry programs.

The abstract syntax of FlatCurry is summarized in Fig. 1. A FlatCurry program consists of a sequence of function definitions (for the sake of simplicity,

[2] Curry requires the explicit declaration of free variables, as `ys,z,zs` in the rule of `aPos`, to ensure checkable redundancy.

data type definitions are omitted), 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 assume that FlatCurry programs satisfy the following properties:

- All variables introduced in a rule (parameters, free variables, let bindings, pattern variables) have unique identifiers.
- Let bindings are non-recursive, i.e., all recursion is introduced by functions.
- The patterns in case expressions 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, e.g., the predefined Curry operation `failed` which has no value.

For instance, the operation `fac` is transformed into the following FlatCurry program (which contains infix operators and some non-variable arguments for the sake of readability):

```
fac(n) = let n0 = n==0 in
         case n0 of { True  → 1
                    ; False → let n1 = n>0 in
                              case n1 of { True  → n * fac(n-1)
                                         ; False → failed } }
```

Note that conditional rules are translated into nested case distinctions. The final `failed` branch is included to ensure that each case expression branches over all data constructors of a type. Usually, the front end of a Curry compiler transforms source programs into such a simplified form for easier compilation [2,8].

3 Non-Fail Conditions

A *non-fail condition*, as introduced in [16], is a predicate to specify when an operation can be used without running into a failure. A satisfied non-fail condition does not ensure that some value is eventually computed—an infinite computation has no run-time failure. A *non-fail condition* of an n-ary operation f of type $\tau_1 \to \cdots \to \tau_n \to \tau$ is specified as an operation `f'nonfail` of type $\tau_1 \to \cdots \to \tau_n \to$ `Bool`. For instance, a non-fail condition for the factorial function `fac` as defined above can be specified as

```
fac'nonfail :: Int  → Bool
fac'nonfail n = n >= 0
```

As a further example, the integer division operation `div` has the non-fail condition

```
div'nonfail :: Int  → Int  → Bool
div'nonfail x y = y /= 0
```

Although non-fail conditions can be defined by the user, our objective is to provide a tool which frees the programmer from this task, i.e., non-fail conditions

are inferred for all user-defined operations. This is useful especially for local and auxiliary operations.

The *trivial non-fail condition* for an n-ary operation f has the form

```
f'nonfail :: τ1 → ··· → τn → Bool
f'nonfail x1 ... xn = True
```

and is the default if there is no explicitly defined non-fail condition. The *unsatisfiable non-fail condition*

```
f'nonfail :: τ1 → ··· → τn → Bool
f'nonfail x1 ... xn = False
```

expresses that there is no known condition under which the operation does not fail. For instance, the predefined operation `failed` mentioned above has the non-fail condition

```
failed'nonfail :: Bool
failed'nonfail = False
```

An unsatisfiable non-fail condition might be used for non-trivial logic-oriented operations, like `aPos` defined in Sect. 2:

```
aPos'nonfail :: [Int] → Bool
aPos'nonfail xs = False
```

Due to the logic-oriented non-deterministic definition of `aPos`, there are no argument values ensuring a computation without failures, e.g., the evaluation of `aPos [1]` contains, apart from the successful computation, also non-deterministic failing branches. This means that an application of `aPos` should be encapsulated by a search handler like `allValues`.

4 Checking and Inferring Non-Fail Conditions

In this section we present our first method to infer and check non-fail conditions without considering algebraic data types in a specific manner. In a first step (Sect. 4.1), we present a method to check non-fail conditions if they are provided for all operations. In a second step (Sect. 4.2), we modify this method to infer non-fail conditions such that they can be successfully checked.

4.1 Checking Non-Fail Conditions

In order to check non-fail conditions, we assume that, for each defined operation f of arity n, a non-fail condition `f'nonfail` is defined as an n-ary predicate. This predicate can be predefined by some formula, which is usually the case for externally defined operations, or it might be defined by a Boolean Curry operation.[3]

[3] The precise structure of non-fail conditions is not relevant. In the implementation it is only necessary to decide whether an implication w.r.t. a non-fail condition holds, which is done by an SMT solver by axiomatizing the semantics of operations defined in Curry, see Sect. 6.2. If it cannot be decided, e.g., due to a timeout, it is assumed that the implication does not hold.

$$\mathsf{Var}_{nf} \quad \Gamma, C \vdash x$$

$$\mathsf{Cons}_{nf} \quad \Gamma, C \vdash c(x_1, \ldots, x_n)$$

$$\mathsf{Func}_{nf} \quad \Gamma, C \vdash f(x_1, \ldots, x_n) \quad \text{if } C \text{ implies } f\texttt{'nonfail}(\sigma^\Gamma(x_1), \ldots, \sigma^\Gamma(x_n))$$

$$\mathsf{Or}_{nf} \quad \frac{\Gamma, C \vdash e_1 \quad \Gamma, C \vdash e_2}{\Gamma, C \vdash e_1 \; or \; e_2}$$

$$\mathsf{Free}_{nf} \quad \frac{\Gamma, C \vdash e}{\Gamma, C \vdash let \; x_1, \ldots, x_n \; free \; in \; e}$$

$$\mathsf{Let}_{nf} \quad \frac{\Gamma, C \vdash e \quad \Gamma[x \mapsto e], C \vdash e'}{\Gamma, C \vdash let \; x = e \; in \; e'}$$

$$\mathsf{Case}_{nf} \quad \frac{\Gamma, C \vdash x \quad \Gamma, C_1 \vdash e_1 \quad \ldots \quad \Gamma, C_n \vdash e_n}{\Gamma, C \vdash case \; x \; of \; \{p_1 \to e_1; \ldots; p_n \to e_n\}} \quad \text{where } C_i = C \wedge \sigma^\Gamma(x) = p_i$$

Fig. 2. Checking non-fail conditions

The checking of non-fail conditions is defined by the rules shown in Fig. 2. This inference system derives judgements of the form "$\Gamma, C \vdash e$" where Γ is a mapping from variables into expressions, also called *heap* in operational descriptions like [1], C is the current assertion, i.e., a Boolean expression satisfied in the current branch under consideration, and e is a (FlatCurry) expression. The heap Γ contains the bindings of variables introduced by let expressions. We denote by $\Gamma[x \mapsto e]$ the heap Γ' with $\Gamma'(x) = e$ and $\Gamma'(y) = \Gamma(y)$ for all $x \neq y$. σ^Γ denotes the substitution, i.e., a mapping from expressions into expressions, represented by the heap Γ. σ^Γ satisfies $\sigma^\Gamma(x) = x$ if $\Gamma(x) = x$ (i.e., there is no binding for x) and $\sigma^\Gamma(x) = \sigma^\Gamma(e)$ if $\Gamma(x) = e$ with $x \neq e$. This recursive definition is well defined since there are no cyclic bindings in Γ, which is ensured by our restrictions on FlatCurry programs (non-recursive let bindings). σ^Γ can also be interpreted as *dereferencing* w.r.t. Γ.

Intuitively, $\Gamma, C \vdash e$ means that, if σ is a substitution such that $\sigma(C)$ holds, the expression $\sigma(\sigma^\Gamma(e))$ evaluates without a failure, i.e., the non-fail conditions of all operations occurring during this evaluation are satisfied. To check the non-fail condition of an operation f defined by $f(x_1, \ldots, x_n) = e$, we try to derive the judgement $\{\}, f\texttt{'nonfail}(x_1, \ldots, x_n) \vdash e$. Thus, we analyze the right-hand side of the rule under the assumption that the non-fail condition is satisfied. Now we discuss the rules in Fig. 2 in more detail.

Rule Var_{nf} states that the evaluation of a variable cannot cause a failure. This is justified because the variable is either free or it is bound to some expression which cannot fail because rule Let_{nf} checks each bound expression (even if it is not required in a lazy evaluation). Similarly, the evaluation of a constructor-rooted expression cannot fail, as expressed by rule Cons_{nf}. Rule Func_{nf} requires that the current assertion C implies the non-fail condition of the operation to be evaluated. Since the argument variables $x_1, \ldots, x_n$ might be bound to expressions introduced by let expressions, they have to be dereferenced by σ^Γ.

Rule Or_{nf} states that both expressions of a choice must be evaluable without a failure. It could be relaxed by requiring that at least one of the choices is free of failures, but we put this stronger condition since completely fail-free computations could support simpler and more efficient implementations.

Rule Free_{nf} checks the expression without a binding for the free variables in the heap so that they are universally quantified when testing the implication in rule Func_{nf}. This is different in rule Let_{nf} where the binding $x \mapsto e$ is put into the heap before the main expression is checked. The bound expression is also checked but without a binding for x because we assume that bindings are non-recursive. Since the property whether x is actually evaluated in e' is undecidable in general, we safely approximate it by assuming that x might be evaluated, i.e., we require that the evaluation of e yields no failure.

Rule Case_{nf} is the most important rule to ensure that a potentially failing operation does not cause a problem if its application is wrapped by an appropriate condition. Each branch of a case expression is checked with an extended assertion which takes into account that the discriminating variable must be equal to the corresponding branch pattern.[4] This extended assertion is used to check further function calls occurring in this branch by rule Func_{nf} so that it is important to collect the condition about the discriminating variable in the current assertion. For instance, consider an extended definition of the factorial function:

```
facInt x = if x < 0 then 0 else fac x
```

This is translated into the FlatCurry definition

```
facInt(x) = let y = x < 0 in
            case y of { True →  0 ; False →  fac(x) }
```

Assume that `facInt'nonfail`$(x) = \mathit{true}$ and `fac'nonfail`$(x) = x \geq 0$. When checking the case expression of `facInt`, we have $\Gamma = \{y \mapsto x < 0\}$ and $C = \mathit{true}$. Then the expression `fac(x)` of the `False`-branch is checked with the extended assertion $C_2 = \mathit{true} \wedge (x < 0) = \mathit{false}$ which is equivalent to $x \geq 0$. Hence, C_2 implies the non-fail condition of `fac`.

Checking non-fail conditions can also be used to ensure that operations defined by rules with several guards are completely defined. For instance, consider the operation to compute the absolute value of an integer:

```
abs n | n >= 0 = n
      | n <  0 = 0 - n
```

Similarly to the FlatCurry representation of `fac` shown in Sect. 2, `abs` has the FlatCurry representation

```
abs(n) = let c1 = n>=0 in
         case c1 of { True  → n
                    ; False → let c2 = n<0 in
                              case c2 of { True  → 0 - n
                                         ; False → failed } }
```

[4] In principle, we could omit the premise $\Gamma, C \vdash x$ since it is always satisfiable by rule Var_{nf}, but we included it to make it explicit that the discriminating argument must be also non-failing.

When checking this definition with the trivial non-fail condition of `abs`, the assertion in the final `failed` branch is $(n \geq 0) = \mathit{false} \wedge (n < 0) = \mathit{false}$. Since this can be shown as unsatisfiable by an SMT solver, this branch is unreachable so that the operation `failed` will never be called, i.e., `abs` is totally defined. Hence, a compiler could use this information to transform the code into the following slightly more efficient code:

```
abs(n) = let c1 = n>=0 in case c1 of { True  → n
                                     ; False → 0 - n }
```

We can state the correctness of the inference rules in Fig. 2 as follows.

Theorem 1. *Assume that the non-fail conditions of all defined operations are successfully checked by the rules in Fig. 2. Let f be defined by $f(x_1, \ldots, x_n) = e$, $C = f\texttt{'nonfail}(x_1, \ldots, x_n)$ and $\{\}, C \vdash e$ be derivable by these inference rules, and σ be a substitution such that $\sigma(C)$ holds. If $g(t_1, \ldots, t_m)$ is reduced in any reduction of $\sigma(e)$, $g\texttt{'nonfail}(t_1, \ldots, t_m)$ holds.*

Hence, if all non-fail conditions are successfully checked and we evaluate an operation with a satisfied non-fail condition, the non-fail conditions of all operations reduced during this evaluation hold. The proof, which is omitted due to lack of space, is by induction on the number evaluation steps and a nested induction on the height of the inference tree.

4.2 Inferring Non-Fail Conditions

In order to infer non-fail conditions for user-defined operations, we start with trivial non-fail conditions for all operations, except for externally defined operations with non-trivial non-fail conditions, like division operators or `failed`. As can be seen in the rules of Fig. 2, the only situation where a rule might not be applicable is rule Func_{nf} due to its side condition. Hence, if this condition does not hold, we enforce it by adding the implication, i.e., the formula

$$\neg C \vee f\texttt{'nonfail}(\sigma^{\Gamma}(x_1), \ldots, \sigma^{\Gamma}(x_n))$$

is added as a conjunct to the non-fail condition of the operation currently checked. If the extended non-fail condition is unsatisfiable, we set it to *false*.

As an example, consider the operation `fac` where its FlatCurry definition is shown in Sect. 2. When checking the occurrence of `failed`, the current assertion is $(n = 0) = \mathit{false} \wedge (n > 0) = \mathit{false}$. Obviously, this does not imply the non-fail condition *false* of `failed`. Therefore, we add the conjunct

$$\neg\, ((n = 0) = \mathit{false} \wedge (n > 0) = \mathit{false}) \vee \mathit{false}$$

which is equivalent to $(n = 0) \vee (n > 0)$, to the existing trivial non-fail condition of `fac`. When we re-check the definition of `fac` with this modified non-fail condition, the current assertion when checking `failed` is

$$((n = 0) \vee (n > 0)) \wedge (n = 0) = \mathit{false} \wedge (n > 0) = \mathit{false}$$

This is equivalent to *false* (i.e., the current branch is not reachable) so that the non-fail condition of `failed` holds. Similarly, the new non-fail condition holds for the recursive call to `fac`. Thus, the inferred non-fail condition of `fac` is valid.

Note that the inference of non-fail conditions is an iterative process. For instance, if our program contains the definition of `fac` as well as the operation

```
facMult n = n * fac n
```

then checking `facMult` is successful w.r.t. the initial trivial non-fail conditions. After refining the non-fail condition of `fac`, we have to check `facMult` again and infer a refined non-fail condition also for `facMult`, which is identical to the non-fail condition of `fac`. Checking `facMult` w.r.t. the refined non-fail condition is successful so that no further iteration is necessary.

The iterative refinement of non-fail conditions has the risk of non-termination. For instance, consider the operation

```
infpos n | n > 0 = infpos (n - 1)
```

When checking it with the trivial non-fail condition, we obtain the refined non-fail condition $n > 0$. Using this condition in the next iteration, we obtain the refined non-fail condition $n > 0 \ \wedge \ (n - 1) > 0$ which is equivalent to $n > 1$. Further iterations yields the non-fail conditions $n > 2$, $n > 3$, and so on. To avoid such infinite loops, one has to stop the refinements at some point and set the non-fail condition to *false*. In our implementation, we stop it after the second refinement. Although this heuristic seems limited, the manual inspection of the few cases where unsatisfiable non-fail conditions are inferred in existing programs (see Sect. 6.3) showed that better results would not be computable by increasing this limit.

4.3 External Operations and User-Defined Non-Fail Conditions

The inference of non-fail conditions is based on the analysis of the rules defining operations. In particular, the inferred non-fail conditions are determined by the structure of case expressions. This has the consequence that externally defined operations, i.e., predefined operations defined by the run-time system, cannot be analyzed. Therefore, we assume that predefined operations have a trivial non-fail condition except for a few operations where a non-fail condition is explicitly defined, like the always failing operation `failed`, division operations (`mod`, `div`, "`/`",...) where the divisor must be non-zero, the square root operation where the argument must be non-negative, among others.[5]

Based on the non-fail conditions of external operations, one can infer non-fail conditions for all user-defined operations as described in the previous section. In the worst case, unsatisfiable non-fail conditions might be derived, as for the operation `infpos` shown above, but our evaluation in Sect. 6.3 shows that meaningful non-fail conditions are inferred in most practical cases. However, there are also cases where an inferred non-fail condition is not precise enough. For instance, consider the operation `nth` which selects the n-th element of a given list:

[5] Search handlers are treated differently: their arguments are not transformed into let expressions so that the arguments are not checked, because their failures are encapsulated.

```
nth :: [a]  → Int  → a
nth (x:xs) n | n == 0 = x
             | n > 0  = nth xs (n - 1)
```

The inferred non-fail condition, in a simplified form, is

```
nth'nonfail xs n = not (null xs) && n <= 0 && n >= 0
```

(the predicate `null` is satisfied if the argument is an empty list). This is equivalent to

```
nth'nonfail xs n = not (null xs) && n == 0
```

Although this condition is satisfiable and ensures that `nth` does not fail for arguments satisfying this condition, it is obviously too strong. A reasonable and more relaxed non-fail condition has to ensure that the index is not negative and the argument list has enough elements for the selection. This can be specified by the following condition:

```
nth'nonfail xs n = n >= 0 && length xs > n
```

This non-fail condition can be successfully checked with our method. The potentially failing branches for an empty list xs or a negative number n are not reachable due to the condition that n is non-negative and the length of the list is positive. Furthermore, the non-fail condition of the recursive call to `nth` holds. For this purpose, one has to verify that the current assertion at this call

$$n \geq 0 \ \wedge \ \texttt{length}(xs) > n \ \wedge \ xs = (y{:}ys) \ \wedge \ n \neq 0 \ \wedge \ n > 0$$

implies the non-fail condition

$$(n-1) \geq 0 \ \wedge \ \texttt{length}(ys) > (n-1)$$

of the recursive call. The proof of the first conjunct uses reasoning on integer arithmetic, as supported by SMT solvers, and the second conjunct can also be proved by SMT solvers when the rules of the operation `length` are axiomatized as logic formulas (see Section 6.2).

Due to these considerations, our tool takes into account explicit user-defined non-fail conditions. If they do not hold, i.e., they are refined as described in Sect. 4.2, then they are replaced by an unsatisfiable non-fail condition so that the resulting conditions are always correct.

5 Combining Non-Fail Conditions and Call Types

So far, we discussed the verification of arithmetic non-fail conditions, although our method can also be applied to conditions involving algebraic data types, as shown by the list index operation `nth` above. However, operations on algebraic data types might require more complex reasoning about input/output dependencies. For instance, consider the following operation which splits a list into sublists of ascending elements [27]:

```
risers []       = []
risers [x]      = [[x]]
risers (x:y:etc) = let (s:ss) = risers (y:etc)
                   in if x <= y then (x:s):ss else [x]:(s:ss)
```

In order to show the non-failure of this operation on all lists, one has to verify that the recursive call to `risers` always returns a non-empty list (so that the pattern matching in the let expression does not fail). Though this is not true in general, an analysis of `risers` shows that it returns an empty list for an empty input list and, for a non-empty input list, it returns a non-empty result list. Hence, the pattern matching on the result of the recursive call is non-failing.

Such a type-based reasoning is proposed in [18] where input/output type relations (also called *in/out types*) are approximated for operations. These are used to approximate *call types*, i.e., under-approximations of sets of argument values which ensure the non-failing execution of operations. For this purpose, a domain of *abstract types* is used where each element in the domain approximates a set of concrete values. For instance, the domain of top constructors (also called depth-1 abstraction in [33]) contains sets of data constructors as abstract elements or the specific constraint $\top$ denoting any value. Sets of data constructors denote terms having one of the constructors at the root. In this case, the in/out type of `risers` is

$\{\{\texttt{[]}\} \hookrightarrow \{\texttt{[]}\}, \{\texttt{:}\} \hookrightarrow \{\texttt{:}\}\}$

This can be read as: if the argument is an empty list ($\{\texttt{[]}\}$), the result is an empty list, and if the argument is a non-empty list ($\{\texttt{:}\}$), the result is a non-empty list. Hence, the in/out type of `risers` ensures that the recursive call always returns a non-empty list, which is used to verify `risers` as non-failing. Therefore, $\{\texttt{[]}, \texttt{:}\}$ is a valid call type of `risers`. This verification does not demand any user annotation. In contrast, LiquidHaskell requires an explicit specification of the input/output behavior of `risers` to verify it as a total operation [36].

As another example, consider the definition of an operation to return the last element of a list:

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

A valid call type of `last` is $\{\texttt{:}\}$ since it fails on the empty list.

If the domain of abstract types is finite (as the top constructor domain sketched above), fixpoint iterations with such types terminate (when monotonic operations are used). This can yield more precise results compared to our method which might stop iterations too early (as discussed in Sect. 4.2). On the other hand, our method can deal with more precise information on arithmetic domains but requires an external verifier to check implications. Thus, it is reasonable to use a hybrid approach where both methods are combined. An implementation of this idea is sketched in the next section.

6 Implementation and Evaluation

We implemented the hybrid verification method, based on the ideas discussed above, in a fully automatic tool written in Curry[6] and exploiting the SMT solver

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

Z3 [29]. In the following, we sketch its implementation and show the result of applying it to various libraries.

6.1 Basic Implementation Scheme

In order to enable the verification of larger programs, the verification is performed in an incremental modular manner: when a Curry module M has been checked, the analysis results of M (non-fail conditions and call types) are stored so that they are available for other modules importing M. Thus, our tool performs the following steps to check a single Curry module:

- The module is translated into a corresponding FlatCurry program by the standard front end of Curry.
- For all imported modules, their non-fail conditions and call types are loaded (after they have been checked). This is necessary since imported operations might be used in the right-hand sides of operations defined in the current module.
- For each operation defined in the current module, in/out types and call types are inferred, as described in [18]. If an inferred call type is empty, i.e., there is no condition on the arguments, expressible as a call type, to ensure that the operation does not fail, a non-fail condition is inferred and checked as described in Sect. 4.
- If a non-fail condition or call type of some operation is refined (as discussed in Sect. 4.2), the current module is checked again w.r.t. the new condition. This fixpoint computation terminates when all conditions are stabilized. For an efficient computation, a call dependency graph is computed so that one has to re-check only operations with refined conditions and the operations that use them.

Thus, non-fail conditions are not computed (i.e., they are trivial) and checked if the set of non-failing arguments can be expressed by a call type, as in the operation `last` above. This hybrid method is useful since the satisfaction of non-trivial non-fail conditions (see rule Func_{nf} in Fig. 2) is checked by invoking an SMT solver: this might be costly since the formulas to be checked may contain operations defined in Curry that have to be translated into SMT formulas, as described next.

6.2 Axiomatization of Defined Operations

As discussed for the list index operator `nth` (Section 4.3), it might be necessary to use some information about user-defined operations during verification. Thus, their semantics must be known to the verifier. For this purpose, user-defined operations are translated into SMT formulas which axiomatize their intended semantics. Each rule defining a Curry operation is translated into an SMT formula stating an equality[7] between a function call and the right-hand side. For

[7] The semantics of non-deterministic operations is axiomatized by disjunctions.

instance, consider the operation to compute the length of a list in its FlatCurry representation:

```
length(zs) = case zs of { []     → 0
                        ; (x:xs) → 1 + length(xs) }
```

This definition is transformed into an SMT formula by using a `match` binder:

```
(assert
  (forall ((x1 (List TVar)))
    (= (length x1)
       (match x1 ((nil 0)
                  ((insert x2 x3) (+ 1 (length x3))))))))
```

Our tool generates these axiomatizations by collecting all user-defined operations occurring in non-fail conditions, loading the FlatCurry code of these operations, and then translating this code into SMT formulas. If polymorphic operations occur in contexts with different type instances, their axiomatizations are duplicated for each type instance.

6.3 Evaluation

In the following we evaluate our approach by discussing some examples and applying it to various libraries.

Compared to a previous tool to verify non-failing Curry programs [16] which also used an SMT solver to verify non-fail conditions, our tool is fully automatic. The tool in [16] requires that the programmer annotates all partially defined operations with non-fail conditions. These are translated into proof obligations to be checked by an SMT solver. In contrast, our tool can be used without explicit non-fail conditions and, thanks to our hybrid approach, an SMT solver is required only in the cases where an unsatisfiable call type is inferred, e.g., when arithmetic conditions are involved. All non-fail conditions provided in [16] for the libraries shown below are automatically derived by our tool, except for the prelude operation "`!!`", which is identical to `nth` shown in Sect. 4.3, where our tool infers a less precise non-fail condition. If we explicitly define the same non-fail condition as in [16], our tool verifies it. This also demonstrates the intended use of our tool. Since it reports all inferred non-trivial non-fail conditions and call types, the user can examine them and decide to either accept them, provide other non-fail conditions, or modify the program code to handle possible failures (by using search handlers, like `oneValue`) so that trivial non-fail conditions suffice.

Table 1 contains the results of checking various Curry libraries and the module `Examples` containing most of the examples discussed in this paper. Non-fail conditions were not explicitly provided (except for external operations, see Sect. 4.3). The "operations" column contains the number of public (exported) user-defined operations and the number of all operations (defined or generated) in the module. Similarly, the following three columns show the information for public and all operations:

– *call types*: This column shows the numbers of inferred non-trivial call types. Thus, this is the number of operations which might fail but the set of argu-

Table 1. Inference of non-fail conditions for some standard libraries

Module	operations	call types	non-fail conditions	failing	checked calls	itera-tions	verify time
	pub/all	pub/all	pub/all	pub/all	all/SMT		
`Prelude`	214/1263	20/69	2/9	9/51	66/14	5	6474
`Data.Char`	9/9	0/0	0/0	0/0	0/0	1	64
`Data.Either`	7/11	2/2	0/0	0/0	0/0	1	4
`Data.List`	49/87	8/16	0/0	1/1	18/2	3	2287
`Data.Maybe`	8/9	0/0	0/0	0/0	0/0	1	3
`Numeric`	5/7	0/0	0/0	0/0	0/0	1	15
`System.IO`	23/51	0/0	0/0	0/0	0/0	1	26
`Text.Show`	4/4	0/0	0/0	0/0	0/0	1	2
`Examples`	11/15	1/3	4/4	0/0	17/13	3	382

ments to avoid a failing computation can be described by a non-empty call type (so that an SMT solver is not required to check it).
- *non-fail conditions*: These are the numbers of operations where a non-trivial but satisfiable non-fail conditions is inferred (so that an SMT solver is invoked to check their correct usage).
- *failing*: These are the numbers of operations where an unsatisfiable non-fail condition is inferred. Thus, there is no precise information about the arguments required to ensure a non-failing evaluation (e.g., `failed` or operations involving unification).

Thus, all operations not counted in these columns have trivial non-fail conditions, i.e., they do not fail when applied to any argument. The column *checked calls* contains the number of function calls in right-hand sides of program rules where the called operation has a non-trivial call type so that it needs to be checked. The first number is the total number of such calls (in all iterations) and the second number is the number of such calls where an external SMT solver is used to check the satisfaction of the non-fail condition (according to rule Func_{nf} in Fig. 2), i.e., without the SMT-component of our hybrid approach, these calls are classified as failing. The difference between these numbers is the number of calls where the consideration of call types is sufficient for the verification, thanks to our hybrid approach.

To show how many iterations are required to infer this information (this is only relevant for the inference of call types), their number is shown in the next to last column. The last column shows the verification time in milliseconds.[8]

This evaluation indicates that even quite complex modules, like the prelude, have only a few operations with non-trivial non-fail conditions or call types that need to be checked. The low numbers in the column *non-fail conditions* indicate that the standard libraries contain only a few operations with non-

[8] 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.

trivial arithmetic conditions. This might be different in application programs using more arithmetic operations. The higher numbers in the column *call types* indicate an advantage of our hybrid approach. In a purely SMT-based approach, all these operations need to be checked by SMT scripts, as in [16], and it is not obvious how to infer these conditions for polymorphic operations on recursive data structures, as relevant in the `Data.List` module.

A manual inspection of the functions appearing in the *failing* column shows that the reason is not a weakness of our method: precise non-fail conditions for these functions are demanding. For instance, non-fail conditions of prelude operations involving unification have to consider the unifiability of arguments. The single failing operation of the module `Data.List` is the matrix operation `transpose`: since the input matrix is represented by a list of lists, all input lists must have the same length to avoid a failure when transposing the matrix. Although such a non-fail condition can be expressed by some Curry code using auxiliary operations, the automatic inference of such a complex condition fails so that it is approximated by the unsatisfiable non-fail condition.

7 Related Work

We have shown in the introduction that operations which are not totally defined on their statically declared input type domain are useful, because programming a "totalized" version yields less comprehensible code. Our practical evaluation of various system libraries in the previous section indicated that partial operations are the exception. However, a single occurrence and wrong use can crash an entire application. Although this could be avoided by exception handlers, these handlers are often used to catch environment (input/output) errors rather than controlling partial operations, like division operators. Therefore, the exclusion of such run-time failures is a practically relevant but also challenging issue.

Contracts, as introduced in the context of imperative and object-oriented programming languages [24], are a method to specify intended invocations of operations. Our non-fail conditions can also be considered as contracts on input arguments to ensure the non-failing evaluation of operations. Contracts can be tested at run time to obtain better error messages. However, they can also be checked at compile time. For instance, the Eiffel compiler ensures by appropriate type declarations and static analysis that pointer dereferencing failures ("null pointer exceptions") cannot occur in a program accepted by the compiler [25]. In the following, we review approaches for functional and logic programming related to our work.

In logic programming, there is no common definition of "non-failing" due to different interpretations of non-determinism. We are interested to exclude any failure in a top-level computation, i.e., also in branches of some non-deterministic computation. Other approaches, like [12,13], consider a predicate in a logic program as non-failing if at least one answer is produced. Similarly to our approach, non-failure properties are approximated, but the concrete methods are different due to the different meaning of a non-failed computation.

Another notion of failing programs in a dynamically typed programming language is based on success types, e.g., as used in Erlang [21]. Success types over-approximate possible uses of an operation: if a success type is empty, it indicates that an operation never evaluates to a value. Success types can show definite failures, whereas 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 failures cannot occur ("well-typed programs do not go wrong" [26]). As discussed above, 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 [27,28]. Their checker extracts and solves specific constraints from pattern-based definitions. In contrast to our approach, only pattern failures are considered there. As a result, programs where the completeness depends on complete case distinctions on numbers cannot be handled by these tools.

An approach to handle more complex non-fail conditions is described in [20]. Their HMC algorithm is based on generating (arithmetic) constraints which have to be satisfied by a safe functional program, i.e., a program which does not fail, e.g., due to an incorrect array index access. These constraints are translated into an imperative program such that the constraints are satisfiable iff the translated program is safe. Similarly to our approach, HMC supports a fully automatic verification of functional programs but it is not applicable to logic-oriented subcomputations. Furthermore, it is not clear whether HMC scales for larger programs.

Another approach to ensure the absence of failures is to make the type system more expressive in order to encode non-failing conditions on the type level. For instance, dependently typed programming languages, such as Coq [9], Agda [30], or Idris [10], require that operations are total functions, i.e., they must be terminating and non-failing. These languages have termination checkers but non-fail conditions need to be explicitly encoded in the types. Therefore, each use of an operation with a non-trivial non-fail condition must be accompanied with an explicit proof for the satisfaction of the non-fail condition w.r.t. the actual arguments. Although these proofs are checked by the type checker, programming in a dependently typed language is more challenging since the programmer has to construct such non-failure proofs.

Refinement or liquid types [32], as used in LiquidHaskell [36,37], 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. In contrast to our approach, where non-fail conditions can contain arbitrary user-defined operations, refinement types use a specific set of primitive functions and predicates (arithmetic operators and comparisons, length operation, etc). This allows the inference of refinement types based on generating and solving constraints w.r.t. these functions [32] provided that the expected refinement types can be described with the given entities. The latter restriction is relaxed in [35], where dependent types are inferred by generating constraints and

solving and refining them by finding interpolants. These approaches could infer in many cases precise type refinements to verify specific properties of programs, e.g., safe array access or sorted result lists. However, if such properties cannot be inferred, the program is not valid. In contrast, our approach always infers non-fail conditions. Since we do not require a type language with fixed entities but possibly generate non-fail conditions with arbitrary user-defined predicates, the inferred non-fail conditions of some operations might not be precise enough, e.g., unsatisfiable in the worst case. However, this does not mean that we cannot use such operations. Instead, we can wrap their application with appropriate search handlers in order to control possible failures at run time. Moreover, our hybrid approach allows to verify operations defined on algebraic data types without an external (SMT) solver, as discussed in the `risers` example in Sect. 5.

As already mentioned, potentially failing operations can be encapsulated with search handlers, which is relevant to the application of logic programming techniques. This aspect is also the motivation for the non-failure checking tool proposed in [16]. As already discussed in Sect. 6.3, the advantage of our new approach is the automatic inference of non-failing conditions which supports an easier application to larger programs. Although this aspect is covered in [18], our hybrid approach extends non-failure checking to a larger class of programs, in particular, operations with arithmetic constraints.

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, involving arithmetic conditions as well as conditions on algebraic data types. In contrast to other approaches, our approach does not require the explicit specification of non-fail conditions but is able to infer them—even for larger programs in a reasonable amount of time. Since we developed our approach for Curry, it is also applicable to purely functional and logic programs. We do not intend to abandon all potentially failing operations because partially defined operations and failing evaluations are reasonable in logic-oriented subcomputations provided that they are encapsulated in order to control possible failures. This distinguishes non-fail conditions from traditional preconditions, since preconditions have to be satisfied before invoking the operation.

The inference of non-fail conditions is based on a fixpoint iteration and might yield, in the worst case, unsatisfiable non-fail conditions. However, 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 results for missing cases) or control the invocation of this operation by checking its outcome with some search handler.

Acknowledgments. The author is grateful to the anonymous reviewers for their helpful comments to improve the paper.

References

1. Albert, E., Hanus, M., Huch, F., Oliver, J., Vidal, G.: Operational semantics for declarative multi-paradigm languages. Journal of Symbolic Computation **40**(1), 795–829 (2005). https://doi.org/10.1016/j.jsc.2004.01.001
2. Antoy, S.: Constructor-based conditional narrowing. In: Proc. of the 3rd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP 2001). pp. 199–206. ACM Press (2001). https://doi.org/10.1145/773184.773205
3. Antoy, S., Echahed, R., Hanus, M.: A needed narrowing strategy. Journal of the ACM **47**(4), 776–822 (2000). https://doi.org/10.1145/347476.347484
4. Antoy, S., Hanus, M.: Functional logic design patterns. In: Proc. of the 6th International Symposium on Functional and Logic Programming (FLOPS 2002). pp. 67–87. Springer LNCS 2441 (2002). https://doi.org/10.1007/3-540-45788-7_4
5. Antoy, S., Hanus, M.: Set functions for functional logic programming. In: Proceedings of the 11th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'09). pp. 73–82. ACM Press (2009). https://doi.org/10.1145/1599410.1599420
6. Antoy, S., Hanus, M.: Functional logic programming. Communications of the ACM **53**(4), 74–85 (2010). https://doi.org/10.1145/1721654.1721675
7. Antoy, S., Hanus, M.: Contracts and specifications for functional logic programming. In: Proc. of the 14th International Symposium on Practical Aspects of Declarative Languages (PADL 2012). pp. 33–47. Springer LNCS 7149 (2012). https://doi.org/10.1007/978-3-642-27694-1_4
8. Antoy, S., Hanus, M., Jost, A., Libby, S.: ICurry. In: Declarative Programming and Knowledge Management - Conference on Declarative Programming (DECLARE 2019). pp. 286–307. Springer LNCS 12057 (2020). https://doi.org/10.1007/978-3-030-46714-2_18
9. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. An EATCS Series, Springer (2004). https://doi.org/10.1007/978-3-662-07964-5
10. Brady, E.: Idris, a general-purpose dependently typed programming language: Design and implementation. Journal of Functional Programming **23**(5), 552–593 (2013). https://doi.org/10.1017/S095679681300018X
11. Braßel, B., Hanus, M., Huch, F.: Encapsulating non-determinism in functional logic computations. Journal of Functional and Logic Programming **2004**(6) (2004)
12. Bueno, F., López-García, P., Hermenegildo, M.: Multivariant non-failure analysis via standard abstract interpretation. In: 7th International Symposium on Functional and Logic Programming (FLOPS 2004). pp. 100–116. Springer LNCS 2998 (2004). https://doi.org/10.1007/978-3-540-24754-8_9
13. Debray, S., López-García, P., Hermenegildo, M.: Non-failure analysis for logic programs. In: 14th International Conference on Logic Programming (ICLP'97). pp. 48–62. MIT Press (1997)
14. González-Moreno, J., Hortalá-González, M., López-Fraguas, F., Rodríguez-Artalejo, M.: An approach to declarative programming based on a rewriting logic. Journal of Logic Programming **40**, 47–87 (1999). https://doi.org/10.1016/S0743-1066(98)10029-8
15. Hanus, M.: Functional logic programming: From theory to Curry. In: Programming Logics - Essays in Memory of Harald Ganzinger. pp. 123–168. Springer LNCS 7797 (2013). https://doi.org/10.1007/978-3-642-37651-1_6

16. Hanus, M.: Verifying fail-free declarative programs. In: Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming (PPDP 2018). pp. 12:1–12:13. ACM Press (2018). https://doi.org/10.1145/3236950.3236957
17. Hanus, M.: From logic to functional logic programs. Theory and Practice of Logic Programming **22**(4), 538–554 (2022). https://doi.org/10.1017/S1471068422000187
18. Hanus, M.: Inferring non-failure conditions for declarative programs. In: Proc. of the 17th International Symposium on Functional and Logic Programming (FLOPS 2024). pp. 167–187. Springer LNCS 14659 (2024). https://doi.org/10.1007/978-981-97-2300-3_10
19. Hanus (ed.), M.: Curry: An integrated functional logic language (vers. 0.9.0). Available at http://www.curry-lang.org (2016)
20. Jhala, R., Majumdar, R., Rybalchenko, A.: HMC: verifying functional programs using abstract interpreters. In: 23rd International Conference on Computer Aided Verification (CAV 2011). pp. 470–485. Springer LNCS 6806 (2011). https://doi.org/10.1007/978-3-642-22110-1_38
21. Lindahl, T., Sagonas, K.: 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). pp. 167–178. ACM Press (2006). https://doi.org/10.1145/1140335.1140356
22. López-Fraguas, F., Sánchez-Hernández, J.: A proof theoretic approach to failure in functional logic programming. Theory and Practice of Logic Programming **4**(1), 41–74 (2004). https://doi.org/10.1017/S1471068403001728
23. Lux, W.: Implementing encapsulated search for a lazy functional logic language. In: Proc. 4th Fuji International Symposium on Functional and Logic Programming (FLOPS'99). pp. 100–113. Springer LNCS 1722 (1999). https://doi.org/10.1007/10705424_7
24. Meyer, B.: Object-oriented Software Construction. Prentice Hall, second edn. (1997)
25. Meyer, B.: Ending null pointer crashes. Communications of the ACM **60**(5), 8–9 (2017). https://doi.org/10.1145/3057284
26. Milner, R.: A theory of type polymorphism in programming. Journal of Computer and System Sciences **17**, 348–375 (1978)
27. Mitchell, N., Runciman, C.: A static checker for safe pattern matching in Haskell. In: Trends in Functional Programming. vol. 6, pp. 15–30. Intellect (2007)
28. Mitchell, N., Runciman, C.: 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). pp. 49–60. ACM (2008). https://doi.org/10.1145/1411286.1411293
29. de Moura, L., Bjørner, N.: 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). pp. 337–340. Springer LNCS 4963 (2008). https://doi.org/10.1007/978-3-540-78800-3
30. Norell, U.: Dependently typed programming in Agda. In: Proceedings of the 6th International School on Advanced Functional Programming (AFP'08). pp. 230–266. Springer LNCS 5832 (2008). https://doi.org/10.1007/978-3-642-04652-0_5
31. Peyton Jones, S. (ed.): Haskell 98 Language and Libraries—The Revised Report. Cambridge University Press (2003)

32. Rondon, P., Kawaguchi, M., Jhala, R.: Liquid types. In: Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI'08). pp. 159–169. ACM Press (2008). https://doi.org/10.1145/1375581.1375602
33. Sato, T., Tamaki, H.: Enumeration of success patterns in logic programs. Theoretical Computer Science **34**, 227–240 (1984). https://doi.org/10.1016/0304-3975(84)90119-1
34. Stump, A.: Verified Functional Programming in Agda. ACM and Morgan & Claypool (2016). https://doi.org/10.1145/2841316
35. Unno, H., Kobayashi, N.: Dependent type inference with interpolants. In: Proceedings of the 11th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'09). pp. 277–288. ACM Press (2009). https://doi.org/10.1145/1599410.1599445
36. Vazou, N., Seidel, E., Jhala, R.: LiquidHaskell: Experience with refinement types in the real world. In: Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell. pp. 39–51. ACM Press (2014). https://doi.org/10.1145/2633357.2633366
37. Vazou, N., Seidel, E., Jhala, R., Vytiniotis, D., Peyton Jones, S.: Refinement types for Haskell. In: Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP). pp. 269–282. ACM Press (2014). https://doi.org/10.1145/2628136.2628161