Verifying Fail-Free Declarative Programs

Michael Hanus
CAU Kiel
Institut für Informatik
Kiel, Germany
mh@informatik.uni-kiel.de

ABSTRACT

Failed computations are a frequent problem in software system development. Some failures have external reasons (e.g., missing files) that can be caught by exception handlers. Many other failures have internal reasons, such as calling a partially defined operation with unintended arguments. In order to avoid the latter kind of failures, one can try to analyze the program at compile time for potential occurrences of these failures at run time. In this paper we present an approach to verify the absence of such failures in functional logic programs. Since programming with failures is a typical technique in logic programming, we are not interested to abandon partially defined operations at all. Instead, we want to verify conditions which ensure that operations can be executed without running into a failure. For this purpose, we propose to annotate operations with non-fail conditions that are verified at compile time with an SMT solver. For successfully verified programs, it is ensured that computations never fail provided that the non-fail condition of the main operation is satisfied.

CCS CONCEPTS

• **Software and its engineering → Functional languages**; **Constraint and logic languages**; **Multiparadigm languages**; **Control structures**; *Semantics*;

KEYWORDS

Declarative programming, program failures, verification

ACM Reference Format:
Michael Hanus. 2018. Verifying Fail-Free Declarative Programs. In *The 20th International Symposium on Principles and Practice of Declarative Programming (PPDP '18), September 3–5, 2018, Frankfurt am Main, Germany.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3236950.3236957

1 INTRODUCTION

The occurrence of failures during a program execution is an annoying but frequent problem when software systems are developed. There are two main reasons for such failures. There are external reasons which are outside the control of the program, like missing files or access rights, unexpected formats of external data, etc. Such failures can be caught by exception handlers in order to avoid a crash of the entire software system. Other failures have internal reasons which are due to programming errors, like calling a partially defined operation with unintended arguments. In imperative programs a typical error of this kind is dereferencing a pointer variable whose current value is the null pointer. Although such kinds of errors cannot occur in declarative programs, there are other errors typical for declarative programming, like failures due to incomplete pattern matching. For instance, consider the following operations (shown in Haskell syntax) which compute the first element and the tail of a list:

```
head :: [a] → a
head (x:xs) = x

tail :: [a] → a
tail (x:xs) = xs
```

Since the case for empty lists is not given, the operations are partially defined so that a computation runs into a failure when these operations are applied to an empty list, as in the expression "`head (tail [1])`". Hence, a careful programmer must ensure that such calls do not occur at run time. For instance, if partially defined operations are applied to unknown data, e.g., given by the user at run time, it is good style to check the arguments before the actual application. This is done in the following code snipped which 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):

```
readCommand = do
  putStr "Input a command:"
  s ← getLine
  let ws = words s
  if null ws then readCommand
             else processCommand (head ws) (tail ws)
```

Here, the calls to `head` and `tail` cannot lead to a failure since it is ensured that the variable `ws` is bound to a non-empty list when entering the `else` branch.

The objective of this work is to provide a tool to verify a program for the absence of such run-time failures. One possibility is to enrich the type system to express constraints on applying operations like `head` and `tail`, e.g., by dependent types, as in Agda [33], Coq [8], or Idris [9], or refinement types, as in LiquidHaskell [38, 39]. This provides safety but makes program development much harder [37]. However, we do not consider only functional programs but we are interested in declarative programming which also includes logic programming. There, programming with failures is a typical programming technique so that it is acceptable that operations

PPDP '18, September 3–5, 2018, Frankfurt am Main, Germany
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-6441-6/18/09…$15.00
https://doi.org/10.1145/3236950.3236957

might fail when we search for solutions. A typical example is the unification operation which should fail if its arguments are not unifiable. Similarly, one can accept failing calls to `head` and `tail` when one searches for list structures satisfying some constraints. Therefore, we are not interested to abandon partially defined operations. Instead, we want to verify for purely functional computations, which might contain encapsulated non-deterministic subcomputations, that failing calls do not occur. We call such programs *fail free*. For instance, a call to `head` is acceptable in a purely functional computation only if the argument is a non-empty list, whereas the argument can be arbitrary in encapsulated non-deterministic subcomputations.

In order to specify when operations are fail free, we propose to annotate operations with *non-fail conditions*. For instance, a non-fail condition for `head` is

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

which specifies that the argument `xs` must be a non-empty list. The intended meaning of a non-fail condition is that the evaluation of an operation applied to arguments satisfying the non-fail condition never fails. In this paper, we present a method to verify such non-fail conditions at compile time by extracting conditions from the program that are sent to an SMT solver (e.g., Z3 [15]) for verification. The use of an SMT solver allows us also to verify non-fail conditions involving arithmetic constraints.

Note that non-fail conditions are different from *preconditions* which must be satisfied for any call of an operation. Pre- and post-conditions as programming contracts have been proposed for functional logic programming in [6]. They can be used as dynamic run-time checks or might be verified at compile time [19]. However, if we consider the above non-fail condition for `head` as a precondition, it is not allowed to use `head` in logic computations to search for non-empty lists.

In the following, we present our method for the verification of fail-free programs and present a corresponding tool for programs written in the functional logic language Curry [22]. Since Curry conceptually subsumes purely functional languages like Haskell as well as logic languages, the same ideas can also be applied to these languages. In the next section, we briefly review Curry and its semantics. The idea of non-fail conditions is discussed in Section 3. A formal model of verifying non-fail condictions is introduced in Section 4. Section 5 discusses the use of contracts and emphasizes the differences between contracts and non-fail conditions. The current implementation is presented in Section 6 where also some quite encouraging practical results are presented. Related work is discussed in Section 7 before we conclude.

2 FUNCTIONAL LOGIC PROGRAMS: SYNTAX AND SEMANTICS

The motivation for the development of functional logic languages is to combine the most important features of functional and logic programming in order to support a variety of declarative programming techniques in a single language (see [18] for a survey). As a concrete programming language, we consider the functional logic language Curry [22] in this paper. Curry conceptually extends Haskell with common features of logic programming, i.e., non-determinism, free variables, and constraint solving. We briefly review those elements of Curry that are necessary to understand our approach to verify fail-free programs presented in this paper. More details can be found in surveys on functional logic programming [18] and in the language report [22].

The syntax of Curry is close to Haskell [35]. Recent implementations of Curry also support type classes as in Haskell but do not implement all extensions to the basic type class system. Since this is not relevant for this paper, we ignore type classes here. 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 rules. In contrast to Prolog, these variables must be explicitly declared unless they are anonymous. Function calls can contain free variables, in particular, variables without a value at call time. These calls are evaluated lazily where free variables as demanded arguments are non-deterministically instantiated [2].

Example 2.1. The following simple program shows the functional and logic features of Curry. It defines an operation "++" to concatenate two lists, which is identical to the Haskell encoding. The operation `ins` inserts an element at some (unspecified) position in a list:

```
(++) :: [a] → [a] → [a]
[]     ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)

ins :: a → [a] → [a]
ins x ys     = x : ys
ins x (y:ys) = y : ins x ys
```

Note that `ins` is a *non-deterministic operation* since it might deliver more than one result for a given argument, e.g., the evaluation of `ins 0 [1,2]` yields the values `[0,1,2]`, `[1,0,2]`, and `[1,2,0]`. Non-deterministic operations, which are interpreted as mappings from values into sets of values [17], are an important feature of contemporary functional logic languages. Hence, there is also a predefined *choice* operation:

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

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

Non-deterministic operations can be used as any other operation. For instance, we can use `ins` to define an operation `perm` that returns an arbitrary permutation of a list:

```
perm :: [a] → [a]
perm []     = []
perm (x:xs) = ins x (perm xs)
```

Non-deterministic operations are quite expressive since they can be used to completely eliminate logic variables in functional logic programs. Actually, it has been shown that non-deterministic operations and logic variables have the same expressive power [4, 14]. For instance, a Boolean logic variable can be replaced by the non-deterministic *generator* operation for Booleans defined by

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

Figure 1: Syntax of the intermediate language FlatCurry

```
aBool :: Bool
aBool = False ? True
```

This equivalence can be exploited when Curry is implemented by translation into a target language without support for non-determinism and logic variables. For instance, KiCS2 [12] compiles Curry into Haskell by adding a mechanism to handle non-deterministic computations. In this paper, we exploit this fact by simply ignoring logic variables since we consider them as syntactic sugar for non-deterministic value generators.

Curry has many additional features not described here, like monadic I/O [40] for declarative input/output, set functions [5] to encapsulate non-deterministic search, functional patterns [3] and default rules [7] to specify complex transformations in a high-level manner, and a hierarchical module system together with a package manager[1] that provides access to currently more than 80 packages with several hundreds modules.

Due to the complexity of the source language, language processing tools for Curry, like compilers, analyzers, or optimization tools, 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 to specify the operational semantics of Curry programs [1]. Since we will use FlatCurry as the basis for our verification method, we sketch the structure of FlatCurry and its semantics.

The abstract syntax of FlatCurry is summarized in Fig. 1. In contrast to some other presentations (e.g., [1, 18]), we omit the difference between rigid and flexible case expressions since we do not consider residuation (which becomes less important in practice and is also omitted in newer implementations of Curry [12]). A FlatCurry program consists of a sequence of function definitions, where each function is defined by a single rule. Patterns in source programs are compiled into case expressions and overlapping rules are joined by explicit disjunctions. For instance, the non-deterministic insert operation `ins` is represented in FlatCurry as

```
ins(x,xs) =    (x:xs)
             or
               case xs of { y:ys  →  y : ins(x,ys) }
```

The semantics of FlatCurry programs is defined in [1] as an extension of Launchbury's natural semantics for lazy evaluation [24]. For this purpose, we consider only *normalized* FlatCurry programs, i.e., programs where the arguments of constructor and function calls and the discriminating argument of case expressions are always variables. Any FlatCurry program can be normalized by introducing new variables with `let` expressions [1]. For instance, the expression "`y : ins(x,ys)`" in the FlatCurry rule above is normalized into

```
let { z = ins(x,ys) } in y : z
```

In the following, we assume that all FlatCurry programs are normalized.

In order to model sharing, which is important for lazy evaluation and also semantically relevant in case of non-deterministic operations [17], variables are interpreted as references into a heap. New let bindings are stored in the heap. If a variable bound to some function call is evaluated, the binding in the heap is updated with its evaluated result. To be more precise, a *heap*, denoted by Γ, Δ, or Θ, is a partial mapping from variables to normalized expressions. The *empty heap* is denoted by $[]$. $\Gamma[x \mapsto e]$ denotes a heap Γ' with $\Gamma'(x) = e$ and $\Gamma'(y) = \Gamma(y)$ for all $x \neq y$.

Using heap structures, one can provide a high-level description of the operational behavior of FlatCurry programs in natural semantics style. The semantics uses judgements of the form "$\Gamma : e \Downarrow \Delta : v$" with the meaning that in the context of heap Γ the expression e evaluates to value (head normal form) v and produces a modified heap Δ. Figure 2 shows the rules defining this semantics w.r.t. a given normalized FlatCurry program P, where $\overline{o_k}$ denotes a sequence of objects $o_1, \ldots, o_k$.

Constructor-rooted expressions (i.e., head normal forms) are just returned by rule Val. Rule VarExp retrieves a binding for a variable from the heap and evaluates it. In order to avoid the re-evaluation of the same expression, VarExp updates the heap with the computed value, which models sharing. In contrast to the original rules [1], VarExp removes the binding from the heap. On the one hand, this allows the detection of simple loops ("black holes") as in functional programming. On the other hand, it is crucial in combination with non-determinism to avoid the binding of a variable to different values in the same derivation (see [10] for a detailed discussion on this issue). Rule Fun unfolds function calls by evaluating the right-hand side after binding the formal parameters to the actual ones. Let introduces new bindings in the heap and renames the variables in the expressions with the fresh names introduced in the heap. Or non-deterministically evaluates one of its arguments. Finally, rule Select deals with *case* expressions. When the discriminating

[1]http://curry-language.org/tools/cpm

$$
\begin{array}{lcl}
\textsf{Val} & \Gamma : v \Downarrow \Gamma : v & \text{where } v \text{ is constructor-rooted} \\[2ex]
\textsf{VarExp} & \dfrac{\Gamma : e \Downarrow \Delta : v}{\Gamma[x \mapsto e] : x \Downarrow \Delta[x \mapsto v] : v} & \\[3ex]
\textsf{Fun} & \dfrac{\Gamma : \rho(e) \Downarrow \Delta : v}{\Gamma : f(\overline{x_n}) \Downarrow \Delta : v} & \text{where } f(\overline{y_n}) = e \in P \text{ and } \rho = \{\overline{y_n \mapsto x_n}\} \\[3ex]
\textsf{Let} & \dfrac{\Gamma[\overline{y_k \mapsto \rho(e_k)}] : \rho(e) \Downarrow \Delta : v}{\Gamma : \mathit{let}\ \{\overline{x_k = e_k}\}\ \mathit{in}\ e \Downarrow \Delta : v} & \text{where } \rho = \{\overline{x_k \mapsto y_k}\} \text{ and } \overline{y_k} \text{ are fresh variables} \\[3ex]
\textsf{Or} & \dfrac{\Gamma : e_i \Downarrow \Delta : v}{\Gamma : e_1\ \mathit{or}\ e_2 \Downarrow \Delta : v} & \text{where } i \in \{1, 2\} \\[3ex]
\textsf{Select} & \dfrac{\Gamma : x \Downarrow \Delta : c(\overline{y_n}) \qquad \Delta : \rho(e_i) \Downarrow \Theta : v}{\Gamma : \mathit{case}\ x\ \mathit{of}\ \{\overline{p_k \to e_k}\} \Downarrow \Theta : v} & \text{where } p_i = c(\overline{x_n}) \text{ and } \rho = \{\overline{x_n \mapsto y_n}\}
\end{array}
$$

Figure 2: Natural semantics of normalized FlatCurry programs

argument of *case* evaluates to a constructor-rooted term, Select evaluates the corresponding branch of the *case* expression.

FlatCurry and its operational semantics has been used for various language-oriented tools, like compilers, partial evaluators, or debugging and profiling tools (see [18] for references). We use it in this paper to define our verification method.

3 NON-FAIL CONDITIONS

As discussed in the introduction, we do not want to abandon all uses of partially defined operations since they might be used in logic (search) oriented computations in a meaningful way. However, we want to verify that some uses of such operations, e.g., in purely functional computations, are not going to fail. Hence, we have to specify when operations can be used without failure. For this purpose, we allow to annotate operations with non-fail conditions as discussed in this section.

In order to avoid introducing additional syntax for non-fail conditions and allow the use of standard Curry operations in their definition, a *non-fail condition* for an operation f of type $\tau \to \tau'$[2] is an operation f`'nonfail` of type $\tau \to$ `Bool`. We have already seen a non-fail condition for the operation `head` in the introduction. As a further example, the integer division operation `div` has the following non-fail condition:

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

Intuitively, a non-fail condition specifies requirements on arguments so that the evaluation of this operation does not fail. Note that this does not mean that the evaluation always delivers a value, since infinite computations are still possible.

If f is an operation of type $\tau \to \tau'$, the *trivial non-fail condition* is

```
f'nonfail :: τ → Bool
f'nonfail x = True
```

Since the trivial non-fail condition does not specify any restriction on arguments, we assume that operations without explicitly defined non-fail conditions are implicitly annotated with trivial non-fail conditions.

On the other hand, the *unsatisfiable non-fail condition*

```
f'nonfail :: τ → Bool
f'nonfail x = False
```

expresses that there is no explicit condition under which the operation does not fail. For instance, the predefined Curry operation `failed`, whose evaluation always fails, has the non-fail condition

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

The unification operator "`=:=`" could fail on non-unifiable arguments. Since we cannot expect to verify such a condition at compile time, we define its non-fail condition as[3]

```
=:='nonfail :: a → a → Bool
=:='nonfail _ _ = False
```

Such non-fail conditions do not mean that we cannot use a potentially always failing operation in a verified program—we simply have to encapsulate their use in order to control their failures, e.g., by using set functions [5], as discussed later in Section 6.3.

Verifying a non-fail condition means to prove that a computation with arguments satisfying the non-fail condition never fails, neither by incomplete pattern matching (as in operations like `head`) nor by calling other operations that might fail. For instance, consider the operation to compute the sign of an integer:

```
sig :: Int → Int
sig x | x>0  = 1
      | x==0 = 0
      | x<0  = -1
```

[2]For the sake of simplicity, we consider only unary operations in the formal development. The extension to operations with more than one argument is straightforward.

[3]Note that the identifier of this non-fail condition is not allowed in actual Curry programs. Thus, our verification tool accepts a specific form of non-fail conditions for operators.

$$\text{CaseFailCons} \quad \frac{\Gamma : x \Downarrow \Delta : c(\overline{y_n})}{\Gamma : \mathit{case}\ x\ \mathit{of}\ \{\overline{p_k \to e_k}\} \Downarrow \Delta : \textsc{Fail}} \quad \text{if no } p_i \text{ is rooted by } c$$

$$\text{CaseFail} \quad \frac{\Gamma : x \Downarrow \Delta : \textsc{Fail}}{\Gamma : \mathit{case}\ x\ \mathit{of}\ \{\overline{p_k \to e_k}\} \Downarrow \Delta : \textsc{Fail}}$$

$$\text{ValF} \quad \Gamma : v \Downarrow \Gamma : v \quad \text{where } v \text{ is constructor-rooted or } v = \textsc{Fail}$$

$$\text{PrimFailed} \quad \Gamma : \texttt{failed} \Downarrow \Gamma : \textsc{Fail}$$

Figure 3: Extension of the natural semantics to deal with failures

Since guarded rules are an abbreviation for nested if-then-else expressions, the rule will be desugared to

```
sig x = if x>0 then 1
              else if x==0 then 0
                           else if x<0 then -1
                                       else failed
```

Since `sig` has a trivial non-fail condition (there is no explicit one), one has to show that `sig` never fails. This amounts to showing that the last alternative with the call to `failed` is not reachable. As we will see later, this can be proved with an SMT solver.

In order to specify the correctness of non-fail conditions, we have to extend the semantics of Fig. 2 in order to make failing computations explicit. The necessary extensions and changes are summarized in Fig. 3. In order to reason about failures, we introduce a new constant Fail which does not occur in existing programs, i.e., it is different from any constructor in the program. This constant is intended to indicate a failed computation. Since a failure occurs when a pattern matching is not complete, we add the rules CaseFailCons and CaseFail shown in Fig. 3. Furthermore, we have to consider a Fail value as a final result of a computation. For this purpose, we replace rule Val of Fig. 2 by rule ValF. Since failures might also occur due to the use of primitive operations, like `failed`, we introduce specific rules for them, as shown in rule PrimFailed. In the same way, we could also specify the semantics of the unification operator "`=:=`" [1].

Now we can define the correctness of non-fail conditions.

Definition 3.1 (Correctness of non-fail conditions). Let f be an operation of type $\tau \to \tau'$. The non-fail condition f`'nonfail` is correct if, for all (normalized) values v of type τ such that f`'nonfail`(v) holds and all heaps Γ with $\Gamma(x) = v$, there is no valid judgement $\Gamma : f(x) \Downarrow \Gamma' : \textsc{Fail}$.

Thus, a non-fail condition is correct if there is no failing derivation for arguments satisfying the condition. This implies that, for non-deterministic computations, all branches must be non-failing. For instance, consider the operation

```
idOrTail xs = xs ? tail xs
```

Although `idOrTail []` has value `[]`, the trivial non-fail condition is not correct since $[] :$ `idOrTail([])` $\Downarrow [] : \textsc{Fail}$ is a valid judgement.

4 VERIFICATION OF NON-FAIL CONDITIONS

In order to verify non-fail conditions, we have to analyze the pattern-matching structure of all operations in order to prove that failures cannot occur for computations where non-fail conditions are satisfied. For simple operations, like `head`, the task is easy to solve. For a call like `head xs`, one has to check that the conjunction of the condition of the missing case branch (`xs = []`) and the non-fail condition (`not (null xs)`) is not satisfiable. This can be shown by expanding the definition of the predefined operations `null` and `not`:

$$\begin{aligned}\texttt{not (null xs)} \wedge \texttt{xs = []} &\equiv \texttt{not (xs = [])} \wedge \texttt{xs = []}\\ &\equiv \neg\texttt{(xs = [])} \wedge \texttt{xs = []}\\ &\equiv \mathit{false}\end{aligned}$$

More complex operations, in particular, operations calling other operations with non-trivial non-fail conditions, require advanced reasoning methods. For instance, consider the following operation (a part of the operation `readCommand` from the introduction):

```
f ws = if null ws
         then readCommand
         else processCommand (head ws) (tail ws)
```

In order to show that the non-fail conditions of `head` and `tail` are satisfied in the calls of these operations, one has to push the information that the `if`-condition (`null xs`) is not satisfied into the `else` branch.

As a further example, consider the operation `sig` defined in Section 3. In order to show that the precondition for the operation `failed` (which is always `False`) is satisfied in the last `else` branch, one has to collect the information that the conditions of the preceding `if`-conditions are not satisfied, i.e., one has to show that

$$\neg\texttt{(x>0)} \wedge \neg\texttt{(x==0)} \wedge \neg\texttt{(x<0)}$$

is unsatisfiable. Since this can automatically be proved by an SMT solver, we will integrate SMT solvers in our verification tool.

These examples demonstrate that the verification of non-fail conditions requires to collect properties that are ensured to be valid in particular points of the right-hand sides of program rules. In the following, we call such properties *assertions*. To formalize this process, we define an *assertion-collecting semantics* for FlatCurry programs.[4] The definition follows the structure of the concrete

[4]A semantics following the same idea is defined in [19] to verify contracts. Our semantics is different from that one in order to verify non-fail conditions.

Val
$$\Gamma : C \mid z \leftarrow v \Downarrow C \wedge z = v \qquad \text{where } v \text{ is constructor-rooted, } v = \textsc{Fail}\text{, or } v \text{ is a variable not bound in } \Gamma$$

VarExp
$$\frac{\Gamma : C \mid z \leftarrow e \Downarrow D}{\Gamma[x \mapsto e] : C \mid z \leftarrow x \Downarrow D}$$

Fun
$$\Gamma : C \mid z \leftarrow f(\overline{x_n}) \Downarrow C \wedge (f\texttt{'nonfail}(\overline{x_n}) \vee z = \textsc{Fail})$$

Let
$$\frac{\Gamma[\overline{y_k \mapsto \rho(e_k)}] : C \mid z \leftarrow \rho(e) \Downarrow D}{\Gamma : C \mid z \leftarrow let\ \{\overline{x_k = e_k}\}\ in\ e \Downarrow D} \qquad \text{where } \rho = \{\overline{x_k \mapsto y_k}\} \text{ and } \overline{y_k} \text{ are fresh variables}$$

Or
$$\frac{\Gamma : C \mid z \leftarrow e_1 \Downarrow D_1 \qquad \Gamma : C \mid z \leftarrow e_2 \Downarrow D_2}{\Gamma : C \mid z \leftarrow e_1\ or\ e_2 \Downarrow D_1 \vee D_2}$$

Select
$$\frac{\Gamma : C \mid x \leftarrow x \Downarrow D \quad \Gamma : D_1 \mid z \leftarrow e_1 \Downarrow E_1 \ \ldots\ \Gamma : D_k \mid z \leftarrow e_k \Downarrow E_k}{\Gamma : C \mid z \leftarrow case\ x\ of\ \{\overline{p_k \to e_k}\} \Downarrow E_1 \vee \ldots \vee E_k} \qquad \text{where } D_i = D \wedge x = p_i\ (i = 1, \ldots, k)$$

PrimFailed
$$\Gamma : C \mid z \leftarrow \texttt{failed} \Downarrow C \wedge z = \textsc{Fail}$$

Figure 4: Assertion-collecting semantics

semantics shown in Fig. 2 and 3 but differs from the concrete semantics in the following points:

(1) The assertion-collecting semantics computes with symbolic values.
(2) It collects assertions and passes them from outer to inner positions.
(3) Functions are not evaluated (in order to ensure finiteness of derivations) but assertions about their non-fail conditions are collected.

This semantics uses judgements of the form "$\Gamma : C \mid z \leftarrow e \Downarrow D$" where Γ is a heap, z is a (result) variable, e is an expression, and C and D are assertions, i.e., Boolean formulas over the program signature. The meaning of such a judgement is: if e is evaluated to z in the context Γ where C holds, then D holds after the evaluation.

Figure 4 shows the rules defining the assertion-collecting semantics. Here, we assume that all case branches are complete, i.e., branches missing for some constructors are supplemented with failed. For instance, the FlatCurry representation of head is

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

Rule *Val* immediately returns the assertion about the computed result. Since the assertion-collecting semantics computes with symbolic values, there might be variables without a binding to a concrete value. Hence, *Val* also returns such unbound variables. Rule *VarExp* behaves similarly to rule VarExp of the concrete semantics and returns the assertions collected during the evaluation of the expression. Since the assertion-collecting semantics should always return a result, it should be possible to derive a judgement for any expression. Therefore, rule *Fun* does not invoke the function by evaluating its right-hand side in order to collect their assertions. Instead, we add a disjunction of the non-fail condition and the condition that the result is a failure. The notation $f\texttt{'nonfail}(\overline{x_n})$ in the assertion denotes the logical formula corresponding to the non-fail condition. If the accumulated assertions imply that the non-fail condition $f\texttt{'nonfail}(\overline{x_n})$ is satisfied, the disjunction $f\texttt{'nonfail}(\overline{x_n}) \vee z = \textsc{Fail}$ is also satisfied so that the part $z = \textsc{Fail}$ is irrelevant for the collected assertions. Otherwise, i.e., when one cannot show that the non-fail condition $f\texttt{'nonfail}(\overline{x_n})$ is satisfied w.r.t. the accumulated assertions, the part $z = \textsc{Fail}$ becomes relevant which means that we might deduce that this call fails. Thus, the assertion-collecting semantics computes an over-approximation of the concrete evaluation, since a concrete function application might not fail even if the non-fail condition is not satisfied.

Rule *Let* adds the let bindings to the heap, similarly to the concrete semantics, before evaluating the argument expression. Rules *Or* and *Select* collect all information derived from alternative computations, instead of the non-deterministic concrete semantics. This is intended since we want to approximate *all* potential failures occurring in derivations. Rule *Select* also collects inside each branch the condition that must hold in the selected branch, which is important to get precise information for successful verification. Note that branches that cannot be selected if the non-fail condition holds do not add information to the collected assertions. For instance, consider the operation head above. If the non-fail condition (see Section 1) holds for zs, then the conjunction $not(null(zs)) \wedge zs = \texttt{[]}$ is equivalent to *false* so that the assertion of this branch is irrelevant for the disjunction obtained from the case expression.

To avoid the renaming of local variables in different branches, we implicitly assume that all local variables are unique in a normalized function definition.

In contrast to the concrete semantics, the assertion-collecting semantics is deterministic:

Proposition 4.1. *Let Γ be a heap, C an assertion, z a variable, and e an expression. Then there is a unique (up to variable renamings in let bindings) proof tree and assertion D so that the judgement "$\Gamma : C \mid z \leftarrow e \Downarrow D$" is derivable.*

Proof. The claim follows from the fact that the rules of the assertion-collecting semantics cover all kinds of expressions, do not overlap, and reduce the size of the expressions in the premises. ◻

The assertion-collecting semantics is intended to verify the correctness of non-fail conditions. If we denote by $\widehat{\Gamma}$ the representation of heap information as a logic formula, i.e.,

$$\widehat{\Gamma} = \bigwedge\{x = e \mid x \mapsto e \in \Gamma,\ e \text{ constructor-rooted or a variable}\}$$

we can state the correct approximation of failing computations as follows:

Theorem 4.2. *Let Γ be a heap, C an assertion such that $\widehat{\Gamma} \Rightarrow C$, e an expression where all operations in e have unsatisfiable non-fail conditions, z a fresh variable not occurring in Γ, C, or e. Assume that there is a valid judgement $\Gamma : C \mid z \leftarrow e \Downarrow D$ so that $D \wedge z = \text{FAIL}$ is unsatisfiable. If there is a derivation of $\Gamma : e \Downarrow \Gamma' : v$, then $v \neq \text{FAIL}$.*

The proof is by induction on the height of the proof tree of the natural semantics and requires some technical lemmas which we omit here due to lack of space.

We can use this result to verify the correctness of non-fail conditions. Consider an operation f of type $\tau \rightarrow \tau'$ with a non-fail condition f'nonfail. Assume that we derive the judgement $[] : f\texttt{'nonfail}(x) \mid z \leftarrow f(x) \Downarrow D$ and show that $D \wedge z = \text{FAIL}$ is unsatisfiable. If v is a (normalized) value of type τ such that $f\texttt{'nonfail}(v)$ holds, and Γ a heap with $\Gamma(x) = v$, then we can also derive the judgement $\Gamma : f\texttt{'nonfail}(x) \mid z \leftarrow f(x) \Downarrow D'$ where D' differs from D by replacing x with v so that $D' \wedge z = \text{FAIL}$ is also unsatisfiable. By the theorem above, the evaluation of $f(v)$ (which is identical to $f(x)$ w.r.t. heap Γ) never fails so that the non-fail condition is correct. To prove the theorem also for expressions containing operations with arbitrary non-fail conditions, one has to use an induction on the number of applied operations under the assumption that the non-fail conditions of them are correct.

Note that the result FAIL is introduced only in rules *Select* (in branches for missing constructors), *Fun* (when the non-fail condition does not hold), and *PrimFailed*. Hence, we can prove that the result of the assertion-collecting semantics is always different from FAIL by showing that, when one of these rules is used, the assertions collected at these points are unsatisfiable. For instance, the assertion collected at the FAIL branch of operation head above is not $(\texttt{null}(\texttt{xs})) \wedge \texttt{xs} = \texttt{[]}$ and, thus, unsatisfiable. The operation sig defined in Section 3 has the following definition in FlatCurry:

```
sig(x) = case (x>0) of
           True   → 1
           False  → case (x==0) of
                      True   → 0
                      False  → case (x<0) of
                                 True   → -1
                                 False  → failed
```

Hence, the assertion collected at the call to failed is

$$(\texttt{x>0}) = \mathit{false} \ \wedge\ (\texttt{x==0}) = \mathit{false} \ \wedge\ (\texttt{x<0}) = \mathit{false}$$

This formula is unsatisfiable so that the result FAIL is impossible here.

The unsatisfiabilty of such formulas can be checked by SMT solvers like Z3 [15], which can reason over a number of built-in theories, like integers in our case. Often one needs also a combination of reasoning over integers and data structures. For instance, consider the list index operator which selects the *n*th element of a given list:

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

A non-fail condition has to ensure that the index is not negative and the list must have enough elements for the selection. This can be specified by the following condition:

```
nth'nonfail xs n =
  n >= 0 && length (take (n+1) xs) == n+1
```

In order to allow the application of nth to infinite lists, we do not compute the length of the entire list but ensure by the operation take (which returns a list prefix with at most the given number of elements) that the given list has at least n+1 elements. In order to verify that the non-fail condition for nth is correct, one has to prove three properties:

(1) The first argument is always non-empty. This follows from the non-fail condition.
(2) The guards cover all possible cases. This is a consequence of the part n>=0 of the non-fail condition which shows that the uncovered case n<0 is not reachable. It can be proved by reasoning on integer arithmetic, as in the operation sig above.
(3) The non-fail condition of the recursive call holds. For this purpose, one has to verify that the collected assertions

$$n \geq 0 \wedge \texttt{length}(\texttt{take}(n+1, xs)) = n+1$$
$$\wedge\ xs = (y{:}ys) \wedge n \neq 0 \wedge n > 0$$

imply the non-fail condition

$$(n-1) \geq 0\ \wedge\ \texttt{length}(\texttt{take}((n-1)+1, ys)) = (n-1)+1$$

of the recursive call. The proof of the first conjunct uses reasoning on integer arithmetic, as above, and the second conjunct can also be proved by SMT solvers when the rules of the operations length and take are axiomatized as logic formulas (see Section 6.5).

Using our tool described below, this non-fail condition is proved in a fully automatic manner.

5 USING CONTRACTS

Contracts have been introduced in the context of imperative and object-oriented programming languages [28] to improve the quality of software. Typically, a contract consists of both a pre- and a postcondition. The *precondition* is an obligation for the arguments of an operation application. The *postcondition* is an obligation for both the arguments of an operation application and the result of the operation application to those arguments. The use of contracts for functional logic programming have been proposed in [6] where it has been shown that the features of functional logic programming supports writing specifications, contracts, and implementations in the same language. Thus, contracts and specifications for some operation are operations with the same name and a specific suffix. If f is an operation of type $\tau \rightarrow \tau'$, then a *specification* for f is an operation f'spec of type $\tau \rightarrow \tau'$, a *precondition* for f is an operation f'pre of type $\tau \rightarrow$ Bool (stating requirements on arguments), and a *postcondition* for f is an operation f`'post` of type $\tau \rightarrow \tau' \rightarrow$ `Bool` (stating required relations between argument and result values).

Pre- and postconditions can be used to check the intended use and results of operations at run time (dynamic contract checking). One can also try to verify some contracts at compile so that they do not need to be checked dynamically (static contract checking), as shown in [19] for Curry. Thus, contracts, in particular, preconditions are related to non-fail conditions. However, there is an essential difference in their semantics. Preconditions specify the intended use of an operation—if the precondition is not satisfied for a given argument, the operation should not be executed. In contrast, non-fail conditions are sufficient conditions to ensure that the execution of an operation never fails in any branch of a non-deterministic operation. However, in a logic-oriented computation, one can still invoke an operation where the non-fail condition is not satisfied. For instance, if the operation `head` has the precondtion

```
head'pre xs = not (null xs)
```

then it would not be allowed to apply `head` to an empty list or to a free variable which might be instantiated to an empty list. However, the non-fail condition of `head`, as defined in Section 1, allows the use of `head` with free variables, but then it is not ensured that every computation does not fail.

Albeit from these differences, contracts can be exploited for the verification of non-fail conditions. If we assume that contracts hold during a computation (by static or dynamic contract checking), we can use their information when collecting assertions for verifying non-fail conditions. On the one hand, we can assume the validity of the precondition when verifying non-failures. For instance, consider the definition of the factorial function

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

and its precondition to avoid the unintended application of `fac` to negative numbers:

```
fac'pre n = n >= 0
```

If we assume the validity of this precondition, we can verify the non-failure of `fac` without any additional non-fail condition. Thus, preconditions can be combined with non-fail conditions in order to make them stronger.

On the other hand, postconditions can be useful to verify the non-fail condition of operations called by some operation. For instance, consider the operation `split` (from Curry's standard library `List`) which splits a list into components delimited by separators, where the separator elements are characterized by a given predicate:

```
split :: (a → Bool) → [a] → [[a]]
split _ []    = [[]]
split p (x:xs)
  | p x       = [] : split p xs
  | otherwise = let sp = split p xs
                in (x : head sp) : tail sp
```

In the `let` expression of the last line, the result of `split p xs` must be a non-empty list, otherwise the definition cannot be verified as fail free. This property can be stated as a postcondition:

```
split'post p xs ys = not (null ys)
```

Actually, this postcondition can be verified by the techniques presented in [19]. If we know that this postcondition always holds, we can deduce that the assertion `not (null sp)` holds so that the calls to `head` and `tail` are fail free. Thus, the definition of `split` is verified w.r.t. the trivial non-fail condition.

Another example for using postconditions is taken from [31]:

```
average :: [Int] → Int
average xs = if null xs then 0
                        else sum xs `div` length xs
```

Note that the condition `null xs` is necessary to avoid the potential division-by-zero failure of `div`. The postcondition for the operation `length`

```
length'post xs n = (null xs && n==0) ||
                   (not (null xs) && n>0)
```

can be verified by the tool presented in [19]. If we use this postcondition in the verification of `average`, we can deduce that the result of the call `length xs` is always greater than zero so that the call to `div` never fails.

Since contracts yield useful information to verify non-fail conditions, our tool (see next section) has an option to include contract information in the verification process. It is an option and not the default case since one has to keep in mind that, when contract information is used, the verification results are valid only if contract checking is activated. An interesting option for future work is to integrate static contract checking into our tool so that statically verified contracts are automatically used for the verification of non-fail conditions.

6 IMPLEMENTATION AND BENCHMARKS

In order to test our verification method, we have implemented a fully automatic tool to verify non-fail conditions. In this section, we discuss some aspects of the implementation.

6.1 Basic Implementation Scheme

When verifying the non-fail conditions of a given Curry module, the tool performs the following steps:

- The module is translated into a corresponding FlatCurry program by the standard front end of Curry. The normalization of this FlatCurry program, as required in the rules of Fig. 4, is done on the fly when verifying each operation, in order to avoid an additional transformation phase.
- For all imported modules, the non-fail conditions are loaded. This is necessary to check the validity of non-fail conditions of operations occurring in right-hand sides of defined functions (compare rule *Fun*, Fig. 4).
- For each operation f defined in the given FlatCurry program, its non-fail condition is verified:
  - The given non-fail condition f`'nonfail` (or the trivial non-fail condition if f has no explicit condition) is used as the initial assertion when processing the right-hand side of f's rule.
  - If there is a `case` expression for variable x with a branch missing for some constructor c, then, according to rule

Select of Fig. 4, the collected assertions together with the condition $x = c$ are translated into an SMT formula. This formula is checked by the SMT solver Z3 [15]. If it is unsatisfiable, the missing branch cannot cause a failure. Otherwise, a warning with the missing constructor is issued.

– If there is a call to some operation g in the right-hand side of f's rule, its non-fail condition g'`nonfail` is considered. If the condition is trivial, no further action is taken. Otherwise, the assertions collected at this point according to Fig. 4 are translated into an SMT formula. Then it is checked, by using the SMT solver, whether this formula implies the non-fail condition g'`nonfail` applied to the current arguments of g. If this is true, no failure can occur due to this call of g provided that the non-fail condition of g is also verified.
  Note that a specific handling of the operation `failed` is not necessary, since this operation has the non-fail condition `False`. Thus, a call to `failed` will not cause a failure if the SMT solver proves that the assertions collected at this call are unsatisfiable.

If all potential failure situations are excluded in this way, the non-fail condition of f is verified. Otherwise, the tool reports the potential failure situations so that the programmer can make the non-fail conditions stronger. In the worst case, the programmer can define `False` as a non-fail condition, which is always verifiable.

If one can verify the non-fail conditions of all operations defined in the given module, the module is verified. Otherwise, all potential failures are reported. For instance, verifying `head` without its non-fail condition yields the message

```
POSSIBLY FAILING OPERATIONS:
head: maybe not defined on constructor 'Prelude.[]'
```

If a non-fail condition of some operation in the right-hand in a rule of some operation cannot be verified, the corresponding call is reported, e.g., the verification of the operation `idOrTail` (Section 3, where the non-fail condition of `tail` is identical to that of `head`) yields:

```
POSSIBLY FAILING OPERATIONS:
idOrTail (due to call 'Prelude.tail v1')
```

This basic implementation scheme is successful only for simple programs, since many programs contain additional features not covered by the rules shown in Fig. 4. Therefore, we briefly discuss how the basic scheme can be extended.

6.2 Higher-Order Operations

An important feature of declarative languages are higher-order operations. One can use non-fail conditions also for the definition of higher-order operations. For instance, the accumulator operation for non-empty lists is defined in the standard prelude of Curry as

```
foldr1 :: (a → a → a) → [a] → a
foldr1 f [x]      = x
foldr1 f (x:y:ys) = f x (foldr1 f (y:ys))
```

Since this operation fails on empty lists, we add the following non-fail condition:

```
foldr1'nonfail f xs = not (null xs)
```

The question is how to verify the call of the unknown operation `f` in the definition of `foldr1`. For this purpose, we take a conservative approach. We assume that functional arguments always have a trivial non-fail condition. Thus, it is not necessary to verify these calls. As a consequence, we have to ensure that all actual operations passed as functional arguments have trivial non-fail conditions. Fortunately, this is relatively easy to ensure. Since such functional arguments are partial applications in the FlatCurry representation (i.e., function calls where some arguments are missing), we simply check whether the operation in a partial application has a trivial non-fail condition, otherwise a potential failure is reported.

Surprisingly, this simple solution works well in practice. As an example, consider the definition of the operation `unwords`, defined in the standard prelude of Curry, which concatenates a list of strings with a blank between two strings:

```
unwords    :: [String] → String
unwords ws =
  if ws==[] then []
            else foldr1 (\w s → w ++ ' ':s) ws
```

Since the `if`-condition ensures that `ws` is a non-empty list in the `else` branch and the lambda abstraction has a trivial non-fail condition, we can verify that the trivial non-fail condition is correct for `unwords`.

Of course, one can refine this approach either by specializing higher-order calls into first-order calls of specialized function definitions, or by extending our framework so that non-fail conditions are attached to functional arguments. The preferred solution might depend on the practical evaluation of our approach.

6.3 Encapsulated Search

If a possible failure is present in some operation, this does not mean that we have to give up to construct a fail-free application. As already mentioned, programming with failures is standard in logic-oriented computations. However, in top-level computations (e.g., the user interface with a GUI or web interface), such failures should definitely not occur. For this purpose, functional logic languages offer features for encapsulating non-deterministic search (e.g., [5, 11, 21, 26, 27]) so that non-determinism and failures of subcomputations do not influence the main deterministic computation. The idea of encapsulated search is to collect all results of a non-deterministic computation in some data structure (e.g., list or set). Thus, one can also encapsulate failing computations which do not contribute a result.

Although primitives for this purpose have been introduced into logic programming long time ago [32], the combination of lazy evaluation and non-deterministic search is more subtle [11]. Therefore, *set functions* [5] were proposed for Curry as an evaluation-independent method for encapsulated search. For every function f, the set function f_S yields the set of all results of f applied to some value. Thus, set functions encapsulate only the non-determinism introduced by the definition of f but not the non-determinism of the arguments. For instance, `head`$_S$ `([] ? [1,2])` evaluates to (a representation of) the sets `{}` and `{1}`.

Hence, if the evaluation of some function might fail, we can use it in a fail-free computation by evaluating its set function and analyzing the resulting set. If we use this programming technique, we must also extend our tool to deal with set functions. Due to the fact that set functions encapsulate the non-determinism and failures of the corresponding defined function, the evaluation of a set function itself never fails. Thus, if f is a unary function, the non-fail condition of its set function is

f_S`'nonfail x = True`

With such non-fail conditions for set functions, one can show that the evaluation of the expression `head`$_S$ `([] ? [1,2])` is fail-free.

6.4 Run-time Errors

An interesting question is whether run-time errors should be considered as failed computations. For instance, consider the following (Haskell-style) definition of `head`:

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

Now, the trivial non-fail condition for `head` is correct if `error` is considered as non-failing. This might be a reasonable view since the evaluation does not fail but shows us a result—the error message. On the other hand, we might be interested to avoid failures as well as run-time errors, which is reasonable in a safety-critical application that should not terminate with an error message. This can be obtained by specifying the following non-fail condition for `error`:

```
error'nonfail s = False
```

In this case, the trivial non-fail condition for `head` is not correct. Moreover, if a complete application program is fail free, i.e., the trivial non-fail condition for the main operation is correct, then it is ensured that no run-time error occurs. Exception handlers can be treated similarly to encapsulated search operators as described above.

Since there is no clear preferred interpretation of run-time errors, our tool offers both views. In the default case, the operation `error` is not failing. By setting an option, one can also perform the verification under the assumption that `error` always fails.

6.5 Axiomatization of Defined Operations

As discussed for the list index operator `nth` (Section 4), it might be necessary to use some information about user-defined operations during verification. In particular, if non-fail conditions involve user-defined operations, their semantics must be known to the verifier.

This requirement is implemented in our verification tool by translating user-defined functions into SMT formulas which axiomatize their intended semantics. For this purpose, the defining rules of a Curry function are translated into an SMT formula stating an equality between a function call and the right-hand side. For instance, consider the computation of the length of a list:

```
length :: [a] → Int
length []     = 0
length (x:xs) = 1 + length xs
```

Table 1: Verifying various system modules

Module	Operations	non-fail conditions	pattern tests	call tests
`Prelude`	187	11	7	13
`AnsiCodes`	41	0	0	0
`Array`	23	0	0	9
`Char`	18	0	0	8
`Combinatorial`	11	2	0	0
`Either`	11	2	2	0
`ErrorState`	22	0	0	0
`Integer`	22	9	2	33
`List`	81	8	15	9
`Maybe`	15	1	0	0
`Nat`	6	2	0	2
`Socket`	11	0	0	0
`ShowS`	11	0	0	3
`State`	21	0	0	0

The FlatCurry representation of this operation is:

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

Since SMT solvers do not support polymorphic types, we introduce a sort in SMT to represent polymorphic arguments:

```
(declare-sort TVar)
```

Then we can define the signature of `length` in SMT:

```
(declare-fun length ((List TVar)) Int)
```

Finally, we translate the FlatCurry definition of `length` by using appropriate test and selector functions:

```
(assert
  (forall ((x1 (List TVar)))
    (= (length x1)
       (ite (= x1 nil)
            0
            (let ((x2 (head x1))
                  (x3 (tail x1)))
              (+ 1 (length x3)))))))
```

This purely schematic translation can be improved but it is sufficient for our first experiments. Our tool generates these axiomatizations by collecting all user-defined operations occurring in non-fail conditions, loading the FlatCurry code of these operations (which might be stored in imported modules), and then translating this code into SMT formulas.

6.6 Results

Although our tool is a prototype[5] which must be extended in various ways, we applied it to a number of programs in order to evaluate our approach.

[5]The implementation is available as Curry package "`failfree`" which can be installed with Curry's package manager CPM.

Concerning smaller examples, our tool could verify all non-fail conditions shown in this paper. To test our tool on larger examples, we verified some standard modules of the Curry systems KiCS2 [12] and PAKCS [20]. In order to enable a successful verification, we had to add some non-fail conditions in these modules.

Table 1 shows the results of these tests. The columns are: the name of the module, the number of tested operations, the number of non-trivial non-fail conditions that have been added to user-defined operations, and the number of tests (performed by the SMT solver Z3) for missing constructor patterns and calls to operations with non-trivial non-fail conditions.

It should be mentioned that all these modules have been successfully verified after adding appropriate non-fail conditions. For the module `List`, it was necessary to add four postconditions, similarly to operation `split` (Section 5), in order to verify all operations. Furthermore, the operation `List.transpose` requires a complex non-fail condition: since this operation transposes the rows and columns of a matrix represented by a list of lists, all input lists must have the same length, otherwise the evaluation fails. Although one can easily express this requirement in Curry, the SMT solver is not able to verify it for the recursive calls of `transpose`. Therefore, we use the non-fail condition `False` which is trivially verifiable. As a consequence, in a verified fail-free program, clients of `transpose` have to encapsulate the call and check whether a failure occurred. This solution is applicable in general when the non-fail conditions are too complex for automatic verification.

These initial results are quite encouraging. They indicate that there are usually only a few operations which require non-trivial non-fail conditions. Furthermore, most operations have `case` expressions with complete pattern matching and call only a few operations with non-trivial non-fail conditions. Hence, the verification effort is limited.

7 RELATED WORK

Avoiding run-time failures caused by incomplete definitions of operations have a long tradition. Contracts, as introduced in the context of imperative and object-oriented programming languages [28], are a method to specify intended invocations of operations. They 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 dereference failures ("null pointer exceptions") cannot occur in a program accepted by the compiler [29].

An approach to analyze failures in logic programs is presented in [16]. There, information about call modes of predicates is used to distinguish between test predicates and other predicates called in the body of a clause. Information about regular types is used to compute cover information for predicates. Then, the proposed analysis can show that predicates are non-failing, i.e., produce at least one answer, by examining the call graph of predicates and showing that the argument types are covered by each predicate. This approach is further developed in [13] by extending it to a multi-variant analysis yielding more precise results. The tests used in the bodies of the predicates can be unification of Herbrand terms as well as linear arithmetic constraints. Hence, they are more restricted than our non-fail conditions which can also include user-defined functions, as shown in the example of the list index operator `nth` (Section 4). Moreover, the meaning of non-failing predicates differs from our framework. In [13, 16], non-failing means that there is at least one non-determinstic branch that does not fail, whereas we require that all branches do not fail. Thus, we can also use our method to verify that run-time errors do not occur, as discussed in Section 6.4.

An approach to detect failures in dynamically typed programming languages are success types, e.g., as used to detect failures in Erlang programs [25]. Success types are an over-approximation of possible uses of an operation. If a success type is empty for a given operation, then one knows for sure that this operation can never be evaluated to some value. This is in contrast to our intention where we want to detect all possible failures in a computation. Success types are computed by deriving subtype constraints and then solving them. Since this approach does not require any type annotations, the inferred success types can be much weaker than expected by the programmer.

The logic language Mercury [36] allow determinism annotations for predicates to exclude situations where a predicate might fail. The compiler checks these annotations and exploits them to generate efficient target code. Since the compiler does not use an external verifier, the checks are limited, in particular, for predicates using constraints on numbers. Furthermore, it is not possible to constrain determinism annotations on particular kinds of arguments, as with our non-fail conditions.

A technique to check a Haskell program for the absence of pattern-match errors due to functions with incomplete patterns in their definitions is presented in [30]. The authors propose a static checker that extracts constraints from pattern-based definitions and tries to solve them by simplification and fixpoint iteration. The technique is improved in [31] by adding multipattern constraints that scale better for larger programs. Since only pattern failures are considered in these tools, programs where the completeness arise from case distinctions on numbers cannot be handled in contrast to our approach. For instance, [31] mentions that their Catch tool can prove the safety of the list indexing operator `nth` (Section 4) only for infinite lists.

Another approach to prove the absence of failures at compile time is static contract checking, which has also been explored in purely functional languages. For instance, [41, 42] present methods to verify contracts by a symbolic execution procedure that is applied at verification time. Since an external verifier, like an SMT solver, is not used when programs are symbolically executed, the approach is more limited.

Dependently typed programming languages, such as Coq [8], Agda [33], or Idris [9], support the development of programs with strong correctness properties which are verified by an expressive type system [37]. There, functions must be totally defined, i.e., terminating and non-failing. This can be achieved by encoding requirements, like non-fail conditions, in the type level. For instance, if we define the operation `head` in Agda [33], we exclude the failing application of `head` to an empty list by requiring, as an additional argument, a proof that the argument list is not empty. Thus, the type signature of `head` could be in Agda as follows:

```
head : {A : Set} → (xs : List A)
```

```
    →  is-empty xs == ff  →  A
```

Thus, each client of head must provide an explicit proof for the non-emptiness of the argument list xs. On the one hand, type-checked Agda programs do not contain run-time errors that we try to avoid with our approach. On the other hand, programming in a dependently typed language is more challenging since the programmer has to provide non-failure proofs.

Another approach to encode contracts on the type-level are refinement types, as used in LiquidHaskell [38, 39]. Refinement types are standard types extended by a predicate that restricts the set of allowed values. For instance, one can exclude applications of head to the empty list by the following refinement type [38]:

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

Then applications of head are allowed only if the refinement type checker can verify (possibly with exploiting other refinement types) that the actual argument is always a non-empty list. Since refinement types are checked by an SMT solver, there are similaries to our approach. However, refinement types are stronger than non-fail conditions since refinement types must be verified in valid programs whereas non-fail conditions are only a sufficient condition to avoid failures so that operations with non-satisfied non-fail conditions can still be used in encapsulated subcomputations.

Refinement types are more related to contracts and the approach presented in [19] to check contracts for Curry operations statically with an SMT solver. The general technique used there (collecting assertions in a FlatCurry program and sending them to an SMT solver) is similar to our approach. Howevever, the semantics of non-fail conditions differs from contracts so that the actual framework and tool is different from a static contract checker. The latter should eliminate dynamic contracts from a program whereas we are interested to derive specific failing information about the run-time behavior of a program.

Compared to these related works, our approach is the only one which can ensure top-level non-failing computations together with the use of failing computations which are encapsulated in logic-oriented subcomputations.

Although pattern matching is mainly related to declarative programming languages, the advantages of pattern matching has also been recognized in object-oriented programming, e.g., with Scala's case classes [34]. If such features are used, it is also useful for the programming if a tool points to incomplete patterns in case expressions. For instance, [23] proposes a method to check such properties. The method is compatible with data abstraction in object-oriented programming. Similarly to our non-fail conditions, "matching preconditions" are proposed in [23] to specify sufficient conditions for successful pattern matchings, and SMT solvers are used to verify pattern matching w.r.t. such preconditions. On ther other hand, their verification can be unsound due to the imperative features of the underlying programming language.

8 CONCLUSIONS

In this paper we proposed a new technique and a tool to verify declarative programs for the absence of failing computations caused by partially defined operations. Our approach is based on the idea to specify non-fail conditions for operations. If the actual argument of an operation satisfies the non-fail condition, the evaluation should never fail. This correctness requirement can be verified at compile time by proving specific assertions derived from an assertion-collecting semantics as unsatisfiable. Our tool uses an SMT solver to prove these assertions in a fully automatic manner.

The advantage of our approach is its full automation. If the verification tool cannot prove the correctness of all non-fail conditions, it points to the potential failure situations. These can either be corrected (e.g., by adding missing case branches or calling an operation only after additional checks) or, if it is intended, by strengthening the non-fail conditions. After these changes, one can run the verifier again on the modified program.

Since we developed our framework for functional logic programs, our objective is not to abandon all operations with potential failures from programs. Instead, we support the use of partially defined operations and failing evaluations in logic-oriented subcomputations provided that they are encapsulated in order to control possible failures. On the other hand, top-level deterministic computations should never fail. Thus, one has to ensure that all operations called there must satisfy the non-fail conditions. Of course, the same reasoning is also possible for purely functional programs, i.e., we can apply our verification method also to Haskell programs.

Initial experiments with some standard modules show that our approach can successfully be applied by adding a few non-fail conditions to a module. Although programming with verified non-fail conditions requires some effort to define appropriate conditions, it has the advantage that an important class of run-time errors can be excluded at compile time.

For future work, we will improve our tool in order to test the effectiveness of our approach on more programs and evaluate the necessary efforts to verify existing Curry applications. Since the use of contracts might be required for this task, the combination of non-fail conditions and contracts should be further improved, e.g., by combining static contract verification methods with the methods developed in this paper. Finally, it would also be interesting to develop methods to extract appropriate non-fail conditions from a given program which could not be verified.

ACKNOWLEDGMENTS

The author is grateful to the anonymous referees for their constructive remarks. The research described in this paper has been partially supported by the German Federal Ministry of Education and Research (BMBF) under Grant No. 01IH15006B.

REFERENCES

[1] E. Albert, M. Hanus, F. Huch, J. Oliver, and G. Vidal. 2005. Operational Semantics for Declarative Multi-Paradigm Languages. *Journal of Symbolic Computation* 40, 1 (2005), 795–829.
[2] S. Antoy, R. Echahed, and M. Hanus. 2000. A Needed Narrowing Strategy. *J. ACM* 47, 4 (2000), 776–822. https://doi.org/10.1145/347476.347484
[3] S. Antoy and M. Hanus. 2005. Declarative Programming with Function Patterns. In *Proceedings of the International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'05)*. Springer LNCS 3901, 6–22.
[4] S. Antoy and M. Hanus. 2006. Overlapping Rules and Logic Variables in Functional Logic Programs. In *Proceedings of the 22nd International Conference on Logic Programming (ICLP 2006)*. Springer LNCS 4079, 87–101.
[5] S. Antoy and M. Hanus. 2009. Set Functions for Functional Logic Programming. In *Proceedings of the 11th ACM SIGPLAN International Conference on Principles*

and Practice of Declarative Programming (PPDP'09). ACM Press, 73–82. https://doi.org/10.1145/1599410.1599420

[6] S. Antoy and M. Hanus. 2012. Contracts and Specifications for Functional Logic Programming. In *Proc. of the 14th International Symposium on Practical Aspects of Declarative Languages (PADL 2012)*. Springer LNCS 7149, 33–47. https://doi.org/10.1007/978-3-642-27694-1_4

[7] S. Antoy and M. Hanus. 2017. Default Rules for Curry. *Theory and Practice of Logic Programming* 17, 2 (2017), 121–147. https://doi.org/10.1017/S1471068416000168

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

[9] E. Brady. 2013. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming* 23, 5 (2013), 552–593. https://doi.org/10.1017/S095679681300018X

[10] B. Braßel. 2011. *Implementing Functional Logic Programs by Translation into Purely Functional Programs*. Ph.D. Dissertation. Christian-Albrechts-Universität zu Kiel.

[11] B. Braßel, M. Hanus, and F. Huch. 2004. Encapsulating Non-Determinism in Functional Logic Computations. *Journal of Functional and Logic Programming* 2004, 6 (2004).

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

[13] F. Bueno, P. López-García, and M.V. Hermenegildo. 2004. Multivariant Non-failure Analysis via Standard Abstract Interpretation. In *7th International Symposium on Functional and Logic Programming (FLOPS 2004)*. Springer LNCS 2998, 100–116. https://doi.org/10.1007/978-3-540-24754-8_9

[14] J. de Dios Castro and F.J. López-Fraguas. 2007. Extra variables can be eliminated from functional logic programs. *Electronic Notes in Theoretical Computer Science* 188 (2007), 3–19.

[15] L. de Moura and N. Bjørner. 2008. 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)*. Springer LNCS 4963, 337–340. https://doi.org/10.1007/978-3-540-78800-3

[16] S. Debray, P. López-García, and M.V. Hermenegildo. 1997. Non-failure Analysis for Logic Programs. In *14th International Conference on Logic Programming (ICLP'97)*. MIT Press, 48–62.

[17] J.C. González-Moreno, M.T. Hortalá-González, F.J. López-Fraguas, and M. Rodríguez-Artalejo. 1999. An approach to declarative programming based on a rewriting logic. *Journal of Logic Programming* 40 (1999), 47–87.

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

[19] M. Hanus. 2017. Combining Static and Dynamic Contract Checking for Curry. In *Proceedings of the 27th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2017)*. Spriner LNCS 10855, 323–340. https://doi.org/10.1007/978-3-319-94460-9_19

[20] M. Hanus, S. Antoy, B. Braßel, M. Engelke, K. Höppner, J. Koj, P. Niederau, R. Sadre, and F. Steiner. 2017. PAKCS: The Portland Aachen Kiel Curry System. Available at http://www.informatik.uni-kiel.de/~pakcs/.

[21] M. Hanus and F. Steiner. 1998. Controlling Search in Declarative Programs. In *Principles of Declarative Programming (Proc. Joint International Symposium PLILP/ALP'98)*. Springer LNCS 1490, 374–390.

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

[23] C. Isradisaikul and A.C. Myers. 2013. Reconciling exhaustive pattern matching with objects. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, 343–354. https://doi.org/10.1145/2462156.2462194

[24] J. Launchbury. 1993. A Natural Semantics for Lazy Evaluation. In *Proc. 20th ACM Symposium on Principles of Programming Languages (POPL'93)*. ACM Press, 144–154.

[25] T. Lindahl and K. Sagonas. 2006. 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)*. ACM Press, 167–178. https://doi.org/10.1145/1140335.1140356

[26] F.J. López-Fraguas and J. Sánchez-Hernández. 2004. A Proof Theoretic Approach to Failure in Functional Logic Programming. *Theory and Practice of Logic Programming* 4, 1 (2004), 41–74.

[27] W. Lux. 1999. Implementing Encapsulated Search for a Lazy Functional Logic Language. In *Proc. 4th Fuji International Symposium on Functional and Logic Programming (FLOPS'99)*. Springer LNCS 1722, 100–113.

[28] B. Meyer. 1997. *Object-oriented Software Construction* (second ed.). Prentice Hall.

[29] B. Meyer. 2017. Ending null pointer crashes. *Commun. ACM* 60, 5 (2017), 8–9. https://doi.org/10.1145/3057284

[30] N. Mitchell and C. Runciman. 2007. A Static Checker for Safe Pattern Matching in Haskell. In *Trends in Functional Programming*, Vol. 6. Intellect, 15–30.

[31] N. Mitchell and C. Runciman. 2008. 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)*. ACM, 49–60. https://doi.org/10.1145/1411286.1411293

[32] L. Naish. 1985. All Solutions Predicates in Prolog. In *Proc. IEEE Internat. Symposium on Logic Programming*. IEEE-CS, Boston, 73–77.

[33] U. Norell. 2008. Dependently Typed Programming in Agda. In *Proceedings of the 6th International School on Advanced Functional Programming (AFP'08)*. Springer LNCS 5832, 230–266. https://doi.org/10.1007/978-3-642-04652-0_5

[34] M. Odersky and T. Rompf. 2014. Unifying Functional and Object-Oriented Programming with Scala. *Commun. ACM* 57, 4 (2014), 76–86. https://doi.org/10.1145/2591013

[35] S. Peyton Jones (Ed.). 2003. *Haskell 98 Language and Libraries—The Revised Report*. Cambridge University Press.

[36] Z. Somogyi, F. Henderson, and T. Conway. 1996. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming* 29, 1-3 (1996), 17–64.

[37] A. Stump. 2016. *Verified Functional Programming in Agda*. ACM and Morgan & Claypool. https://doi.org/10.1145/2841316

[38] N. Vazou, E.L. Seidel, and R. Jhala. 2014. LiquidHaskell: Experience with Refinement Types in the Real World. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*. ACM Press, 39–51. https://doi.org/10.1145/2633357.2633366

[39] N. Vazou, E.L. Seidel, R. Jhala, and S. Peyton Jones. 2014. Refinement Types for Haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. ACM Press, 269–282. https://doi.org/10.1145/2628136.2628161

[40] P. Wadler. 1997. How to Declare an Imperative. *Comput. Surveys* 29, 3 (1997), 240–263.

[41] D.N. Xu. 2006. Extended static checking for Haskell. In *Proc. of the 36th ACM SIGPLAN Workshop on Haskell (Haskell 2006)*. 48–59. https://doi.org/10.1145/1159842.1159849

[42] D.N. Xu, S.L. Peyton Jones, and K. Claessen. 2009. Static contract checking for Haskell. In *Proc. of the 36th ACM Symposium on Principles of Programming Languages (POPL 2009)*. 41–52. https://doi.org/10.1145/1480881.1480889