Inferring Non-Failure Conditions for Declarative Programs*

Michael Hanus

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

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

1 Introduction

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

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

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

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

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

* This paper is a revised and extended version of [27].

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

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

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

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

In this paper we present a fully automatic tool which can verify the non-failure of this program. Our technique is based on analyzing the types of arguments and results of operations in order to ensure that partially defined operations are called with arguments of appropriate types. The principle idea to use type information for this purpose is not new. For instance, one can express restrictions on arguments of operations with *dependent types*, as in Agda [39], Coq [11], or Idris [12], or *refinement types*, as in LiquidHaskell [46,47]. Since one has to prove that these restrictions hold during the construction of programs, the development of such programs becomes harder [44]. Another alternative, proposed in [23], is to annotate operations with *non-fail conditions* and verify that these conditions hold at each call site by an external tool, e.g., an SMT solver [17]. In this way, the verification is fully automatic but requires user-defined annotations and, in some cases, also the verification of post-conditions or contracts to state properties about result values of operations [24].

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

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

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

In order to make our approach accessible to various declarative languages, we formulate and implement it in the declarative multi-paradigm language Curry [30]. Since Curry extends Haskell by logic programming features and there are also methods to transform logic programs into Curry programs [25], our approach can also be applied

to purely functional or logic programs. A consequence of using Curry is the fact that programs might compute with failures, e.g., it is not an immediate programming error to apply `head` and `tail` to possibly empty lists. However, subcomputations involving such possibly failing calls must be encapsulated so that it can be checked whether such a computation has no result (this corresponds to exception handling in deterministic languages). If this is done, one can ensure that the overall computation does not fail even in the presence of encapsulated logic (non-deterministic) subcomputations.

This paper is structured as follows. After sketching the basics of Curry in the next section, we introduce call types and their abstraction in Section 3. Section 4 defines in/out types and methods to approximate them. The main Section 5 presents our method to check and infer call types for all operations in a program. Section 6 presents an extension of our method to include also externally defined arithmetic operations. We evaluate our approach in Section 7 before we conclude with a discussion of related work.

2 Functional Logic Programming and Curry

As mentioned above, we develop and implement our method in Curry so that it is also available for purely functional or logic programs. This section sketches the basics of Curry and the intermediate language FlatCurry used to define and implement our framework to verify non-failing programs.

2.1 Curry

The declarative language Curry [30] amalgamates features from functional programming (demand-driven evaluation, strong typing, higher-order functions) and logic programming (computing with partial information, unification, constraints), see [7,21] for surveys. The syntax of Curry is close to Haskell[2] [40]. In addition to Haskell, Curry applies rules with overlapping left-hand sides in a (don't know) non-deterministic manner (where Haskell always selects the first matching rule) and allows *free* (*logic*) *variables* in conditions and right-hand sides of defining rules. The operational semantics is based on an optimal evaluation strategy [5]—a conservative extension of lazy functional programming and logic programming.

Curry is strongly typed so that a Curry program consists of data type definitions (introducing *constructors* for data types) and *functions* or *operations* on these types. As an example, we show the definition of two operations: the list concatenation "`++`" and an operation `dup` which returns some number having at least two occurrences in a list:[3]

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

```
dup :: [Int] → Int
dup xs | xs == _ ++ [x] ++ _ ++ [x] ++ _
       = x    where x free
```

[2] Variables and function names usually 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 ("$f\ e$").

[3] Note that Curry requires the explicit declaration of free variables, as `x` in the rule of `dup`, to ensure checkable redundancy. Anonymous variables, denoted by an underscore, need not be declared.

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

Fig. 1. Syntax of the intermediate language FlatCurry

Since `dup` might deliver more than one result for an argument, e.g., `dup [1,2,2,1]` yields `1` and `2`, it is also called a *non-deterministic operation*. Such operations, which are interpreted as mappings from values into sets of values [20], are an important feature of contemporary functional logic languages. Therefore, there is also a predefined choice operation "`?`" which non-deterministically returns one of its two arguments, e.g., the expression `1 ? 2` evaluates to `1` and `2`. To express failing computations, there is a predefined operation `failed` which has no value.

To control failing and non-deterministic operations, Curry supports *encapsulated search operators* which return some or all values of an expression in a data structure. Similarly to Prolog's `findall` predicate, Curry has an operation `allValues` [4] which returns the list of all values of the argument, or `oneValue` which returns `Nothing` if the argument has no value, otherwise `Just` some value.[4]

```
allValues :: a  →  [a]
oneValue  :: a  →  Maybe a
```

For instance, `allValues (dup [1,2,2,1])` yields the list `[1,2]`.

Curry has more features than described so far.[5] Due to these numerous features, language processing tools for Curry (compilers, analyzers, etc.) often use an intermediate language where the syntactic sugar of the source language has been eliminated and the pattern matching strategy is explicit. This intermediate language, called FlatCurry, has also been used, apart from compilers, to specify the operational semantics of Curry programs [1] or to implement a modular framework for the analysis of Curry programs [29]. Since we will use FlatCurry to describe and implement our inference method, we sketch the structure of FlatCurry programs in the next section.

2.2 FlatCurry

Figure 1 summarizes the abstract syntax of FlatCurry. A FlatCurry program consists of a sequence of function definitions (we omit data type definitions here), where each

[4] Since the precise order of returned values depends on the implementation, these encapsulation operators are considered unsafe. An alternative to encapsulate search with a declarative semantics are set functions [6].

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

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). Applications to empty argument lists, like $c()$, are simply written as c. Due to the important role of failure in this paper, *failed* belongs to the syntax of FlatCurry to express a failing computation (whereas it is treated as a predefined external operation in Curry).

To distinguish the syntax of FlatCurry from the source language Curry, FlatCurry denotes applications with parentheses and comma-separated arguments.[6] We will write $\mathcal{F}$ for the set of defined operations and $\mathcal{C}$ for the set of constructors of a program. A *value* or *data term* is an expression containing only constructor applications. We write $\mathcal{D}$ for the set of all *data terms*. We denote by $Var(e)$ the set of all *unbound variables* of an expression e, i.e., all variables occurring in e which are not introduced in patterns or let expressions.

In order to provide a simple definition of the subsequent methods to infer call types, we assume that FlatCurry programs are transformed into a standard form satisfying the following properties (which is often used by Curry compilers, see below):

- Function definitions do not contain unbound variables, i.e., all variables occurring in the right-hand side are either parameters or introduced by *let* bindings or *case* patterns.
- All variables introduced in a rule (parameters, free variables, let bindings, pattern variables) have unique identifiers.
- For the sake of simplicity, let bindings are non-recursive, i.e., all recursion is introduced by functions (although our implemented tool supports recursive bindings).
- The patterns in each case expression are non-overlapping and cover all data constructors of the type of the discriminating variable. Hence, if this type contains n constructors, there are n branches without overlapping patterns. This can be ensured by adding missing branches with *failed* expressions.

Any Curry source program can be translated into the format required by FlatCurry [3]. Actually, the front end of most Curry compilers transforms source programs into such a form for easier compilation [8]. For instance, the operation `head` is transformed into the FlatCurry definition (where we use the standard notation for lists)

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

As an example for the translation of operations with overlapping rules, consider the non-deterministic operation `insert` which inserts an element at an arbitrary position into a list:

```
insert x ys     = x : ys
insert x (y:ys) = y : insert x ys
```

This definition can be transformed into the FlatCurry definition

```
insert(x,xs) =    x : xs
               or case xs of { y:ys → let zs = insert(x,ys)
                                   in y : zs
                             ; [] → failed }
```

[6] FlatCurry uses a first-order syntax since all higher-order features of source programs are implemented by defunctionalization [41].

Conditional rules can be translated into FlatCurry by using a "conditional" operator `cond` which is predefined as

```
cond True x = x
```

Thus, `cond` fails if applied to an argument not reducible to `True`. Then the operation `dup` is transformed into the FlatCurry definition (where we use infix notation and some non-variable arguments for the sake of readability)

```
dup(xs) = let x1,x2,x3,x free
          in cond(xs == x1 ++ [x] ++ x2 ++ [x] ++ x3, x)
```

2.3 Evaluation of FlatCurry Programs

When we later state the correctness of inferred call types, we will show that these types correctly approximate possible evaluations of functional logic programs, in particular, that there are no failing evaluations. Therefore, we have to specify evaluations of FlatCurry programs.

Expressions to be evaluated are FlatCurry expressions as introduced above with the generalization that the arguments of constructor and function applications as well as the discriminating argument of a case expression are also allowed to be expressions (and not only variables). This generalization allows to define evaluations by rewriting expressions instead of handling a separate heap structure for variable bindings [1]. Moreover, expressions under evaluation do not contain unbound variables. The goal of a (successful) evaluation is the computation of a value of a given expression.

Since functional logic programs are non-deterministically evaluated, we define a non-deterministic rewriting semantics. Our semantics might allow more derivations than a particular implementation of Curry using a specific strategy [3,5]. However, this is not a problem since we will show that correct call types ensure fail-free evaluations so that the results also hold for a concrete semantics having less derivations.

The rewriting semantics is defined by the rules in Figure 2. A sequence $o_1, \ldots, o_n$ of objects is abbreviated by $\overline{o_n}$. If some constructor argument is evaluable, it will be evaluated (rule Cons). Function calls (rule Func) are replaced by their body expressions where the parameters are replaced by the actual arguments (we use the usual term rewriting notation for substitutions). A disjunction e_1 *or* e_2 is non-deterministically evaluated, i.e., reduced to either e_1 or e_2 (rules OrL and OrR). The introduction of free variables is evaluated by guessing arbitrary values for the free variables and proceeding with the instantiated expression (rule Free). Actual implementations defer this guessing to the point where a value of a variable is needed, which is called narrowing. A let binding is evaluated by evaluating the bound expression to a value[7] t and evaluating the main expression where the bound variable is replaced (rules Let and Bind). Finally, a case expression is evaluated by evaluating the discriminating argument (rule Case) and matching it against some pattern in a branch (rule Match). Note that the case expression covers all constructors of a type so that a reduction step is always possible if the discriminating argument is headed by a constructor. Since *failed* cannot be evaluated to a value, there is no evaluation rule for it.

[7] In order to model non-strict evaluation, one could add partial terms, i.e., an undefined value, and the possibility to reduce a term to an undefined value, as in the rewriting logic CRWL [20]. For the sake of simplicity, we omit the introduction of partial terms.

$$\textsc{Cons}\ \frac{e_i \Rightarrow e_i'}{c(e_1,\ldots,e_i,\ldots,e_n) \Rightarrow c(e_1,\ldots,e_i',\ldots,e_n)}$$

$$\textsc{Func}\ \frac{}{f(e_1,\ldots,e_n) \Rightarrow \sigma(e)}\ \text{where } f(x_1,\ldots,x_n) = e, \sigma = \{\overline{x_n \mapsto e_n}\}$$

$$\textsc{OrL}\ \frac{}{e_1\ \mathit{or}\ e_2 \Rightarrow e_1} \qquad \textsc{OrR}\ \frac{}{e_1\ \mathit{or}\ e_2 \Rightarrow e_2}$$

$$\textsc{Free}\ \frac{}{\mathit{let}\ x_1,\ldots,x_n\ \mathit{free}\ \mathit{in}\ e \Rightarrow \sigma(e)}\ \text{where } \sigma = \{\overline{x_n \mapsto t_n}\} \text{ with } t_1,\ldots,t_n \in \mathcal{D}$$

$$\textsc{Let}\ \frac{e_1 \Rightarrow e_1'}{\mathit{let}\ x = e_1\ \mathit{in}\ e_2 \Rightarrow \mathit{let}\ x = e_1'\ \mathit{in}\ e_2}$$

$$\textsc{Bind}\ \frac{}{\mathit{let}\ x = e_1\ \mathit{in}\ e_2 \Rightarrow \sigma(e_2)}\ \text{where } e_1 \text{ is a value and } \sigma = \{x \mapsto e_1\}$$

$$\textsc{Case}\ \frac{e \Rightarrow e'}{\mathit{case}\ e\ \mathit{of}\ \{\overline{p_k \to e_k}\} \Rightarrow \mathit{case}\ e'\ \mathit{of}\ \{\overline{p_k \to e_k}\}}$$

$$\textsc{Match}\ \frac{}{\mathit{case}\ c_i(\overline{t_n})\ \mathit{of}\ \{\ldots, c_i(\overline{x_n}) \to e_i \ldots\} \Rightarrow \sigma(e_i)}\ \text{where } \sigma = \{\overline{x_n \mapsto t_n}\}$$

Fig. 2. Rewrite rules to evaluate expressions

We denote by $\Rightarrow^*$ the reflexive and transitive closure of $\Rightarrow$. We call t *a value of* the expression e, denoted by $e \rightsquigarrow t$, if $t \in \mathcal{D}$ is a data term with $e \Rightarrow^* t$.

Types in FlatCurry. As mentioned above, Curry is a strongly typed language but we omit explicit type information in FlatCurry for the sake of simplicity. Instead, we assume that, for each n-ary constructor or function f, $\mathit{Type}(f) \subseteq \mathcal{D}^n$ defines the *argument type* of f, i.e., the set of n-tuples which are allowed for a well-typed call to f. For instance, the condition operator `cond` defined above has argument type

$$\mathit{Type}(\texttt{cond}) = \{\texttt{False}, \texttt{True}\} \times \mathcal{D}$$

In this paper, type information is relevant only in case expressions where the patterns in the branches cover all data constructors of some type. For instance, the list concatenation "`++`" defined in Section 2.1 is non-failing since the patterns cover all data constructors, but it might fail if it is called with `True` as a first argument. This kind of failure is avoided if "`++`" is called with type-correct arguments. Therefore, we assume in this paper that (FlatCurry) expressions are well typed, i.e., all applications are performed with type-correct arguments and the values guessed for free variables are well-typed according to their usage in the expression.

3 Call Types and Abstract Types

We consider a computation as *non-failing* if it does not stop due to reaching *failed*. This ensures that there is no pattern mismatch in the source program since missing

pattern cases are completed with *failed* branches in FlatCurry.[8] Note that a non-failing computation might not end in a value since a computation could be infinite.

Although it is desirable that all function calls are non-failing, partially defined functions are reasonable in practice, i.e., there are functions which are non-failing only if applied to a particular arguments. For such functions, one has to ensure that each call is invoked only with these arguments, as shown in the example operation `readCmd` defined in Section 1. Therefore, one needs conditions on arguments of operations so that their evaluation does not fail. For this purpose, we will analyze the rules of each operation. For instance, consider again the operation `head` which is defined in Curry by the single rule

```
head (x:xs) = x
```

Since there is no rule covering the empty list, the condition for a non-failing evaluation of `head` is the non-emptiness of the argument list. Thus, conditions for non-failing computations of operations can be expressed as subsets of argument types.

Definition 1 (Call types and non-failure conditions). *Let f be an n-ary operation. Then a* call type *of f is a set of n-tuples $CT \subseteq Type(f)$. A call type CT is* correct *for f if the evaluation of the expression $f(t_1, \ldots, t_n)$ is non-failing for all $(t_1, \ldots, t_n) \in CT$. A correct call type is also called a* non-failure condition.

For instance, the set $\{t_1{:}t_2 \mid t_1, t_2 \in \mathcal{D}\} \cap Type(\texttt{head})$ is a correct call type of `head`, i.e., a non-failure condition for `head` that can be described by considering the top-level constructor only. In other cases, non-failure conditions require more advanced descriptions. For instance, consider the operation `lastTrue` defined by

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

The evaluation of a call `lastTrue` l does not fail if the argument list l ends with `True`. Although such lists could be finitely described using regular types [16], finite and decidable descriptions might be impossible for arbitrary operations. For instance, if some branch in a condition of an operation causes a failure but the condition of the branch contains a function call, the failure is only relevant if the function call terminates. Due to the undecidability of the halting problem, we cannot hope to describe every non-failure condition in a simple (finite, decidable) manner. Of course, the empty set is always a correct but useless call type.

In order to get reasonable descriptions of correct call types, we *approximate* them by finite and decidable descriptions of sets of concrete data terms so that the evaluation of a call with approximated arguments is non-failing. As we will see, this is an under-approximation, i.e., there might be non-failing calls which do not satisfy the approximated non-failure condition.

In order to support different structures to approximate non-failure conditions, we do not fix a language for approximated call types but assume some domain $\mathcal{A}$ of *abstract types*. Elements of this domain describe sets of concrete data terms. There are various options for such abstract types, like depth-k abstractions [43] or regular types [16]. The latter have been used to infer success types to analyze logic programs [19], whereas depth-k abstractions were used in the abstract diagnosis of functional programs [2] or in the abstraction of term rewriting systems [9,10]. Since regular types

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

are more complex and computationally more expensive, we use depth-k abstractions in our examples. In this domain, denoted by $\mathcal{A}_k$, subterms exceeding the given depth k are replaced by a specific constant (e.g., "_") that represents any term. Since the size of this domain is quickly growing for $k > 1$, we use $k = 1$ in examples, i.e., terms are approximated by their top-level constructors. As we will see, this is often sufficient in practice to obtain reasonable results. Nevertheless, our technique and implementation is parametric over the abstract type domain (practical results w.r.t. different domains are shown in Section 7).

In the following, we present a framework for the inference of call types which is parametric over an abstract domain $\mathcal{A}$, which we call abstract type. It is a specialization of the general framework of abstract interpretation [15] to the concrete domain of data terms.

Definition 2 (Abstract type). *An* abstract type $\mathcal{A}$ *is a lattice with an ordering* $\sqsubseteq$*, greatest lower bound* ($\sqcap$) *and least upper bound* ($\sqcup$) *operations, a least or bottom element* $\bot$*, and a greatest or top element* $\top$*. Furthermore, there is a* concretization function $\gamma : \mathcal{A} \to 2^{\mathcal{D}}$ *which maps each element of the abstract domain into a set of data terms with the properties*

$$\begin{array}{l} \gamma(\bot) = \varnothing \\ \gamma(\top) = \mathcal{D} \\ \gamma(a_1) \subseteq \gamma(a_2) \quad \textit{if } a_1 \sqsubseteq a_2 \end{array}$$

For each n*-ary data constructor* c*, there is also an* abstract constructor application c^α *which maps abstract values* $a_1, \ldots, a_n$ *into an abstract value* a *such that* $c(t_1, \ldots, t_n) \in \gamma(a)$ *for all* $t_1 \in \gamma(a_1), \ldots, t_n \in \gamma(a_n)$ *with* $(t_1, \ldots, t_n) \in \mathit{Type}(c)$*.*

As an example, consider the domain $\mathcal{A}_1$ of *depth-1 types*, which we use in concrete examples. If $\mathcal{C}$ is the set of data constructors, $\mathcal{A}_1$ is indeed the set

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

Hence, each element of $\mathcal{A}_1$ is either a set of data constructors of the same type or $\top$ (any data term). The standard ordering of this lattice is defined by

$$\begin{array}{rl} a \sqsubseteq \top & \text{for any } a \\ a_1 \sqsubseteq a_2 & \text{if } a_1 \subseteq a_2 \end{array}$$

so that $\varnothing$ is the bottom element and union and intersection are the least upper bound and greatest lower bound operations, respectively. The concretization function can be defined by

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

The abstract constructor application can be defined by $c^\alpha(x_1, \ldots, x_n) = \{c\}$ (it could also be defined by $c^\alpha(x_1, \ldots, x_n) = \top$ but this yields less precise approximations).

Abstract types are used to specify abstractions of (concrete) call types as follows, where they are interpreted w.r.t. type-correct arguments.

Definition 3 (Abstract call types). *Let f be an n-ary operation and $\mathcal{A}$ the domain of some abstract type with its concretization function γ. An* abstract call type *of f is a sequence of $a_1, \ldots, a_n$ with $a_i \in \mathcal{A}$ $(i = 1, \ldots, n)$. It denotes the (concrete) call type*

$$\{(t_1, \ldots, t_n) \in \mathit{Type}(f) \mid t_i \in \gamma(a_i), 1 \leq i \leq n\}$$

The abstract call type is correct for *f if the corresponding (concrete) call type is correct for f.*

For instance, the depth-1 type $\{\texttt{:}\}$ is correct for the operations `head` and `tail` defined above.

The abstract type $\mathcal{A}_1$ defined above contains a redundancy when it is used to describe abstract call types. For instance, the list concatenation "`++`" is totally defined so that a non-failure condition for the first argument could be approximated by $\{\texttt{[]}, \texttt{:}\}$ as well as $\top$. This redundancy can be avoided by identifying sets containing all constructors with $\top$. We call a set $\delta \subseteq \mathcal{C}$ *complete* if it contains all constructors of some data type. Now we modify the definition of $\mathcal{A}_1$ as follows:

$$\mathcal{A}_1 = \{\delta \subseteq \mathcal{C} \mid \text{all } c \in \delta \text{ belong to the same type but } \delta \text{ is not complete}\} \cup \{\top\}$$

The ordering is as before, but the least upper bound operation is now defined by

$$a_1 \sqcup a_2 = \begin{cases} \top & \text{if } a_1 = \top \text{ or } a_2 = \top \text{ or } a_1 \cup a_2 \text{ is complete} \\ a_1 \cup a_2 & \text{otherwise} \end{cases}$$

(the greatest lower bound operation must be similarly adapted). Since this modified definition of $\mathcal{A}_1$ provides more precise results, we use it in the examples in this paper.

Our inference of non-failure conditions is parametric over a domain $\mathcal{A}$ of abstract call types. Hence, we assume in the subsequent formal development that a domain $\mathcal{A}$ of abstract call types is fixed. In concrete examples, we use the depth-1 domain $\mathcal{A}_1$.

4 In/Out Types

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

For instance, consider the operation `null` defined by

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

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

Since the inference of the precise input/output behavior of operations is intractable and also undecidable in general, we approximate it with abstract types. For this purpose, we associate an in/out type to each operation.

Definition 4 (In/out types). *Let f be an n-ary operation. An* in/out type *io for f is a set of elements containing a sequence of $n+1$ abstract types:*

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

The first n components of each element approximate input values (where we write ε if $n = 0$) and the last component approximates output values associated to the inputs.

Intuitively, a correct in/out type should approximate all possible evaluations of a call to f to some value, i.e., for each evaluation there is at least one element in the in/out type set which covers the arguments and computed result. In order to formally define the notion of correct in/out types, we assume, according to Definition 2, a given abstract type domain $\mathcal{A}$ together with a concretization function γ. Then the following definition makes the intuition more precise.

Definition 5 (Correct in/out types). *Let f be an n-ary operation and*

$$\{a_{i1} \cdots a_{in} \hookrightarrow a_i \mid i = 1, \ldots, k\}$$

be an in/out type for f. It is called a correct in/out type *for f if, for all $(t_1, \ldots, t_n) \in Type(f)$ and $t \in \mathcal{D}$ with $f(t_1, \ldots, t_n) \rightsquigarrow t$, there is some $i \in \{1, \ldots, k\}$ with $t_j \in \gamma(a_{ij})$ $(j = 1, \ldots, n)$ and $t \in \gamma(a_i)$.*

Thus, in/out types can be interpreted as disjunctions of possible input/output behaviors of an operation. For instance, a correct in/out type of `null` w.r.t. $\mathcal{A}_1$ is

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

Another trivial and less precise in/out type is $\{\top \hookrightarrow \top\}$.

Correct and precise in/out types can also characterize non-terminating operations. For instance, a correct in/out type for the operation `loop` defined by

```
loop = loop
```

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

4.1 Inferring In/Out Types

To approximate the input/output behavior of operations with in/out types, we analyze the definition of each operation and associate abstract result values to abstract patterns. For instance, for the operation `null` and the domain $\mathcal{A}_1$, we associate the value $\{\texttt{True}\}$ to the abstract pattern $\{\texttt{[]}\}$ and the value $\{\texttt{False}\}$ to the abstract pattern $\{\texttt{:}\}$. This is obvious if the right-hand sides are constructor terms, as in the definition of `null`. However, what should be associated if the right-hand side is headed by a function? This is the purpose of a result value approximation.

Definition 6 (Result value approximation). *A* result value approximation *is a mapping $R : \mathcal{F} \to \mathcal{A}$ which associates to each operation $f \in \mathcal{F}$ an abstract type $R(f) \in \mathcal{A}$. A result value approximation R is* correct *if $t \in \gamma(R(f))$ holds for all $(t_1, \ldots, t_n) \in Type(f)$ with $f(t_1, \ldots, t_n) \rightsquigarrow t$.*

$$\Gamma \vdash x : \Gamma(x) \qquad (x \text{ variable})$$

$$\Gamma \vdash c(x_1, \ldots, x_n) : c^\alpha(\Gamma(x_1), \ldots, \Gamma(x_n)) \qquad (c \text{ constructor})$$

$$\Gamma \vdash f(x_1, \ldots, x_n) : R(f) \qquad (f \text{ operation})$$

$$\frac{\Gamma \vdash e_1 : a_1 \quad \Gamma \vdash e_2 : a_2}{\Gamma \vdash e_1 \; or \; e_2 : a_1 \sqcup a_2}$$

$$\Gamma \vdash \mathit{failed} : \bot$$

$$\frac{\Gamma[\overline{x_n \mapsto \top}] \vdash e : a}{\Gamma \vdash \mathit{let}\ x_1, \ldots, x_n\ \mathit{free}\ \mathit{in}\ e : a}$$

$$\frac{\Gamma[x \mapsto \top] \vdash e' : a}{\Gamma \vdash \mathit{let}\ x = e\ \mathit{in}\ e' : a}$$

$$\frac{\Gamma_1 \vdash e_1 : a_1 \quad \ldots \quad \Gamma_n \vdash e_n : a_n}{\Gamma \vdash \mathit{case}\ x\ \mathit{of}\ \{p_1 \to e_1; \ldots; p_n \to e_n\} : a_1 \sqcup \ldots \sqcup a_n}$$

where $p_i = c_i(\overline{x_{n_i}})$ and $\Gamma_i = \Gamma[x \mapsto c_i^\alpha(\overline{\top}), \overline{x_{n_i} \mapsto \top}]$

Fig. 3. Approximation of result values of expressions

A simple result value approximation is $R(f) = \top$ for all operations f. A more precise approximation can be obtained by an abstract interpretation of expressions, as supported by the Curry analysis framework CASS [29]. To show an example of an analysis, we denote by Γ a *type environment* which maps variables into abstract types. $\Gamma[x \mapsto a]$ denotes the type environment Γ' with $\Gamma'(x) = a$ and $\Gamma'(y) = \Gamma(y)$ for all $x \neq y$. If the operation f is defined by $f(x_1, \ldots, x_n) = e$, one can define $R(f) = a$ if the judgement $\{\overline{x_n \mapsto \top}\} \vdash e : a$ is derivable with the rules of Figure 3. As usual in abstract interpretation [15], the recursive structure of this definition can be solved by a fixpoint computation. We start with the initial assumption $R(f) = \bot$ for all operations f and apply the rules of Figure 3 to compute new abstract results for all operations until a fixpoint of R is reached (termination can be ensured by a finite abstract domain $\mathcal{A}$).

For instance, this method computes (for the domain $\mathcal{A}_1$) the result values $R(\texttt{loop}) = \varnothing$, $R(\texttt{head}) = \top$, and $R(\texttt{null}) = \top$ (since $\{\texttt{False}\} \sqcup \{\texttt{True}\} = \top$). Since the framework CASS supports the implementation of such fixpoint computations, this program analysis can be implemented in 20 lines of Curry code—basically a case distinction on the structure of FlatCurry programs.[9]

We do not further discuss or prove the soundness of this specific approximation method, since there are various choices for a result value approximation R. For instance, one can improve the precision of the approximation of a *let* binding by taking the approximated value of the bound expression into account (instead of approximating it as $\top$, as in Figure 3). Our implementation (evaluated in Section 7) uses the one presented in Figure 3. However, the definition $R(f) = \top$ (for all operations f) is also sufficient for the simple examples shown in this paper.

Our actual approximation of in/out types is defined by the rules in Figure 4 where we assume that R is a correct result value approximation. As introduced above, Γ

[9] See module `Analysis.Values` of the Curry package `https://cpm.curry-lang.org/pkgs/cass-analysis.html`

$$\mathsf{Var}_{io} \quad \Gamma \vdash x : \{\Gamma \hookrightarrow \Gamma(x)\} \qquad (x \text{ variable})$$

$$\mathsf{Cons}_{io} \quad \Gamma \vdash c(x_1, \ldots, x_n) : \{\Gamma \hookrightarrow c^{\alpha}(\Gamma(x_1), \ldots, \Gamma(x_n))\} \qquad (c \text{ constructor})$$

$$\mathsf{Func}_{io} \quad \Gamma \vdash f(x_1, \ldots, x_n) : \{\Gamma \hookrightarrow R(f)\} \qquad (f \text{ operation})$$

$$\mathsf{Or}_{io} \quad \frac{\Gamma \vdash e_1 : \Theta_1 \quad \Gamma \vdash e_2 : \Theta_2}{\Gamma \vdash e_1 \; or \; e_2 : \Theta_1 \cup \Theta_2}$$

$$\mathsf{Failed}_{io} \quad \Gamma \vdash \mathit{failed} : \{\Gamma \hookrightarrow \bot\}$$

$$\mathsf{Free}_{io} \quad \frac{\Gamma[\overline{x_n \mapsto \top}] \vdash e : \Theta}{\Gamma \vdash \mathit{let}\ x_1, \ldots, x_n\ \mathit{free}\ \mathit{in}\ e : \Theta}$$

$$\mathsf{Let}_{io} \quad \frac{\Gamma[x \mapsto \top] \vdash e' : \Theta}{\Gamma \vdash \mathit{let}\ x = e\ \mathit{in}\ e' : \Theta}$$

$$\mathsf{Case}_{io} \quad \frac{\Gamma_1 \vdash e_1 : \Theta_1 \quad \ldots \quad \Gamma_n \vdash e_n : \Theta_n}{\Gamma \vdash \mathit{case}\ x\ \mathit{of}\ \{p_1 \to e_1; \ldots; p_n \to e_n\} : \Theta_1 \cup \ldots \cup \Theta_n}$$

where $p_i = c_i(\overline{x_{n_i}})$ and $\Gamma_i = \Gamma[x \mapsto c_i^{\alpha}(\overline{\top}), \overline{x_{n_i} \mapsto \top}]$

Fig. 4. Approximation of in/out types

denotes a type environment, i.e., a mapping from variables to abstract types. The inference system derives judgements of the form

$$\Gamma \vdash e : \{\overline{\Gamma_k \hookrightarrow a_k}\}$$

Intuitively, such a judgement is interpreted as "the evaluation of the expression e in the context Γ yields a new context Γ_i and result value of abstract type a_i, for some $i \in \{1, \ldots, k\}$." The alternative context/value pairs occurring in the right-hand side of the judgement are due to the fact that case distinctions and non-deterministic choices contribute to different contexts and result values. Joining them together would result in a loss of precision in the later phase of inferring non-failure conditions.

Let us consider the inference rules in more detail. In the case of variables or applications, the type environment is not changed and the approximated result is returned, e.g., the abstract type of the variable (rule Var_{io}), the abstract representation of the constructor (rule Cons_{io}), and the approximated result value of the operation (rule Func_{io}), respectively. Rule Or_{io} combines the results of the different branches. Rule Failed_{io} returns the least element of the abstract type domain as a result, since *failed* cannot be evaluated to a value. Rules Free_{io} and Let_{io} add the new variables to the type environment with most general types. Although one could refine these types, we try to keep the analysis simple since this seems to be sufficient in practice.

The most interesting rule is Case_{io}. The results from the different branches are combined, but inside each branch, the type of the discriminating variable x is refined to the constructor of the branch. Then the in/out types of the expressions in the branches are inferred w.r.t. this refined environment.

Before we show an example inference, we define how to obtain the in/out type of an operation from the inference of the body's expression.

Definition 7 (Inferred in/out type of an operation). *Let f be an operation defined by $f(x_1, \ldots, x_n) = e$. If the judgement*

$$\{\overline{x_n \mapsto \top}\} \vdash e : \{\overline{\Gamma_k \hookrightarrow a_k}\}$$

can be derived for the body e of operation f, then the in/out type

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

is an inferred in/out type *of f.*

Thus, an in/out type of an operation is derived without any restriction on the arguments. Since the rules in Figure 4 are non-overlapping and cover all expressions, it is always possible to derive a unique in/out type of an operation.

As a simple example, consider the operation

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

We infer an in/out type of `null` with our rules and abstract domain $\mathcal{A}_1$. According to Definition 7, we start with the general type environment $\Gamma_0 = \{\texttt{zs} \mapsto \top\}$ and apply rule Case_{io} to infer an in/out type for the body of the function, i.e., the entire case expression. Inside the branches of the case expression, Γ_0 is refined to $\Gamma_1 = \{\texttt{zs} \mapsto \{\texttt{[]}\}\}$ and $\Gamma_2 = \{\texttt{zs} \mapsto \{\texttt{:}\}, \texttt{x} \mapsto \top, \texttt{xs} \mapsto \top\}$, respectively. Since the expressions in both branches are constructors, we can apply axiom Cons_{io} in both branches so that we obtain the following derivation tree:

$$\frac{\Gamma_1 \vdash \texttt{True} : \{\Gamma_1 \hookrightarrow \{\texttt{True}\}\} \qquad \Gamma_2 \vdash \texttt{False} : \{\Gamma_2 \hookrightarrow \{\texttt{False}\}\}}{\Gamma_0 \vdash \textit{case}\ \texttt{zs}\ \textit{of}\ \{\texttt{[]} \to \texttt{True}\ ;\ \texttt{(x:xs)} \to \texttt{False}\} : \{\Gamma_1 \hookrightarrow \{\texttt{True}\}, \Gamma_2 \hookrightarrow \{\texttt{False}\}\}}$$

According to Definition 7, the in/out type of `null` is obtained by applying the "in" type environments of the right-hand side of the judgement to the argument variable `zs`. Thus, we infer the in/out type

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

To make in/out types more readable, one can simplify them if their correctness according to Definition 5 does not change, like joining pairs with identical input types by the least upper bound of their output types, or omitting elements with output type $\bot$ (failed branches). For instance, we infer the following simplified in/out types for the operations `head` and `tail`:

```
head : {{:} ↪ ⊤}
tail : {{:} ↪ ⊤}
```

4.2 Correctness of In/Out Types

To prove the correctness of our in/out inference, we show in a first step that the inference rules in Figure 4 correctly approximate the input-output behavior of evaluating expressions. Recall that a type environment Γ is a mapping from variables into abstract types. $Dom(\Gamma)$ denotes the domain of Γ. We state the correctness of the inference rules as follows.

Theorem 1. *Let R be a correct result value approximation, Γ be a type environment, e be an expression with $Var(e) \subseteq Dom(\Gamma)$, $\Gamma \vdash e : \{\overline{\Gamma_k \hookrightarrow a_k}\}$ be derivable with the inference rules in Figure 4, and σ be a substitution with $\sigma(x) \in \gamma(\Gamma(x))$ for all $x \in Var(e)$. If $\sigma(e) \rightsquigarrow t$, then there is some $i \in \{1, \ldots, k\}$ with $t \in \gamma(a_i)$ and $\sigma(x) \in \gamma(\Gamma_i(x))$ for all $x \in Var(e)$, and, for any $y \notin Var(e)$ with $y \in Dom(\Gamma)$, $\Gamma(y) = \Gamma_j(y)$ for all $j \in \{1, \ldots, k\}$.*

Proof. By induction on the height of the proof tree to derive $\Gamma \vdash e : \Theta$. We assume that the theorem's preconditions hold, i.e., $Var(e) \subseteq Dom(\Gamma)$, $\Gamma \vdash e : \Theta$ with $\Theta = \{\overline{\Gamma_k \hookrightarrow a_k}\}$ is derivable with a proof tree of height h, σ is a substitution with $\sigma(x) \in \gamma(\Gamma(x))$ for all $x \in Var(e)$, and $\sigma(e)$ is reducible to some data term t. We have to show that there is some $i \in \{1, \ldots, k\}$ with

(1) $t \in \gamma(a_i)$ and
(2) $\sigma(x) \in \gamma(\Gamma_i(x))$ for all $x \in Var(e)$,

and, for any $y \notin Var(e)$ with $y \in Dom(\Gamma)$,

(3) $\Gamma(y) = \Gamma_j(y)$ for all $j \in \{1, \ldots, k\}$.

Induction base $h = 1$: We distinguish the different axioms, i.e., the rules Failed_{io}, Var_{io}, Cons_{io}, and Func_{io}. Note that (3) holds for these rules since the input parts of the derived in/out types are not modified.

Rule Failed_{io} ***is applied:*** Since $e = \mathit{failed}$ is not reducible, the claim vacuously holds.

Rule Var_{io} ***is applied:*** Then $e = x$, $k = 1$, $\Gamma_1 = \Gamma$, $a_1 = \Gamma(x)$, and $t = \sigma(x)$ since $\sigma(x)$ is a data term. Since $\sigma(x) \in \gamma(\Gamma(x))$, properties (1) and (2) hold.

Rule Cons_{io} ***is applied:*** In this case we have $e = c(x_1, \ldots, x_n)$, $k = 1$, $\Gamma_1 = \Gamma$, and $a_1 = c^\alpha(\Gamma(x_1), \ldots, \Gamma(x_n))$. $\sigma(x_i) \in \gamma(\Gamma(x_i)) = \gamma(\Gamma_1(x_i))$ $(i = 1, \ldots, n)$ so that (2) holds. Since $\sigma(e)$ is a data term, $t = \sigma(e) = c(\sigma(x_1), \ldots, \sigma(x_n))$. This implies property (1) since $\sigma(x_i) \in \gamma(\Gamma(x_i))$ $(i = 1, \ldots, n)$ and $t = c(\sigma(x_1), \ldots, \sigma(x_n)) \in \gamma(c^\alpha(\Gamma(x_1), \ldots, \Gamma(x_n)))$ by definition of the abstract constructor application (see Definition 2).

Rule Func_{io} ***is applied:*** Then $e = f(x_1, \ldots, x_n)$, $k = 1$, $\Gamma_1 = \Gamma$, and $a_1 = R(f)$. Hence $\sigma(x_i) \in \gamma(\Gamma(x_i)) = \gamma(\Gamma_1(x_i))$ $(i = 1, \ldots, n)$ so that (2) holds. Furthermore, $\sigma(e) = f(t_1, \ldots, t_n)$ for some terms $t_1, \ldots, t_n$. Since R is a correct result value approximation, $t \in \gamma(R(f))$ by Definition 6.

Induction step, i.e., $h > 1$: In this case, one of the rules Or_{io}, Free_{io}, Let_{io}, or Case_{io} is applied:

Rule Or_{io} ***is applied:*** Then $e = e_1$ *or* e_2 so that either $\sigma(e_1)$ or $\sigma(e_2)$ is reducible to t. Consider the case that $\sigma(e_1)$ is reducible to t (the other case can be similarly proved). Since $\Gamma \vdash e_1 : \Theta_1$ with $\Theta_1 = \{\overline{\Gamma'_{k'} \hookrightarrow a'_{k'}}\}$ is derivable and $\sigma(e_1)$ is reducible to t, by induction hypothesis, there is some $j \in \{1, \ldots, k'\}$ with $\sigma(x) \in \gamma(\Gamma'_j(x))$ for all $x \in Var(e_1)$ and $t \in \gamma(a'_j)$. If there is some variable $y \in Var(e_2)$ and $y \notin Var(e_1)$, then $y \in Dom(\Gamma)$ so that $\Gamma'_j(y) = \Gamma(y)$ holds by property (3) of the induction hypothesis. Hence $\sigma(y) \in \gamma(\Gamma(y)) = \gamma(\Gamma'_j(y))$. Since $\Theta_1 \subseteq \Theta$, properties (1) and (2) hold.
If $y \in Dom(\Gamma)$ but $y \notin Var(e)$, then $y \notin Var(e_1)$ and $y \notin Var(e_2)$. Hence, by the induction hypothesis, property (3) holds.

Rule Free_{io} ***is applied:*** An evaluation of the expression $e = \mathit{let}\ x_1, \ldots, x_n\ \mathit{free\ in}\ e'$ guesses values for the free variables $x_1, \ldots, x_n$ in order to evaluate e'. Since $\sigma(e)$ reduces to t, there must be a substitution $\rho = \{\overline{x_n \mapsto t_n}\}$ for the free variables such that $\sigma(e)$ is reduced to $\sigma(\rho(e'))$ which is reducible to t. $\mathit{Var}(e') \subseteq \mathit{Dom}(\Gamma[\overline{x_n \mapsto \top}])$ since $\mathit{Var}(e) \subseteq \mathit{Dom}(\Gamma)$. Hence, by the induction hypothesis, there is some $i \in \{1, \ldots, k\}$ with $\sigma(\rho(y)) \in \gamma(\Gamma_i(y))$ for all $y \in \mathit{Var}(e')$ and $t \in \gamma(a_i)$. Thus, property (1) holds. Since ρ is the identity on variables different from $x_1, \ldots, x_n$, $\sigma(y) \in \gamma(\Gamma_i(y))$ for all $y \in \mathit{Var}(e)$ so that property (2) also holds.
If $y \notin \mathit{Var}(e)$ but $y \in \mathit{Dom}(\Gamma)$, then $y \notin \mathit{Var}(e')$ so that (3) holds by the induction hypothesis (y is different from all $x_1, \ldots, x_n$ due to the freshness condition on variables in FlatCurry expressions).

Rule Let_{io} ***is applied:*** This is similar to rule Free_{io} since this rule introduces the bound variable without any restriction on its value so that we can ignore the evaluation of the bound expression.

Rule Case_{io} ***is applied:*** Then $e = \mathit{case}\ x\ \mathit{of}\ \{p_1 \to e_1; \ldots; p_n \to e_n\}$. Since $\sigma(e)$ is reducible to t, there is some branch $i \in \{1 \ldots, n\}$ such that $\sigma(\rho(e_i))$ is reducible to t, where ρ is a matching substitution on the variables of pattern $p_i = c_i(\overline{x_{n_i}})$. Let $\Gamma_i = \Gamma[x \mapsto c_i^\alpha(\overline{\top}), \overline{x_{n_i} \mapsto \top}]$. Then $\mathit{Var}(e_i) \subseteq \mathit{Dom}(\Gamma_i)$, $\Gamma_i \vdash e_i : \Theta_i$ is derivable where the height of the proof tree is smaller than h, and $\Theta_i = \{\overline{\Gamma'_{k'} \hookrightarrow a'_{k'}}\} \subseteq \Theta$. By the induction hypothesis, there is some $j \in \{1, \ldots, k'\}$ with $\sigma(y) \in \gamma(\Gamma'_j(y))$ for all $y \in \mathit{Var}(e_i)$ and $t \in \gamma(a'_j)$. This immediately implies property (1). To show property (2), consider the discriminating variable x of the case expression, which might not occur in $\mathit{Var}(e_i)$. In this case, $\Gamma'_j(x) = \Gamma_i(x)$ by property (3) of the induction hypothesis. Since branch i has been selected for the reduction of $\sigma(e)$, $\sigma(x)$ must match the pattern $c_i(\overline{x_{n_i}})$ so that $\sigma(x) \in \gamma(c_i^\alpha(\overline{\top})) = \gamma(\Gamma_i(x))$. Finally, if there is some variable $y \in \mathit{Var}(e)$ with $y \notin \mathit{Var}(e_i)$, then $\sigma(y) \in \gamma(\Gamma(y)) = \gamma(\Gamma'_j(y))$ by property (3) of the induction hypothesis. Therefore, $\sigma(y) \in \gamma(\Gamma'_j(y))$ holds for all $y \in \mathit{Var}(e)$, which proves property (2).
If $y \notin \mathit{Var}(e)$ but $y \in \mathit{Dom}(\Gamma)$, then $y \notin \mathit{Var}(e_i)$ (for all $i \in \{1, \ldots, n\}$) so that (3) holds by the induction hypothesis (y does not occur in the patterns $p_1, \ldots, p_n$ due to the freshness condition on variables in FlatCurry expressions). □

According to Definition 7, if f is an operation defined by $f(x_1, \ldots, x_n) = e$, we infer the in/out type

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

for f if the judgement

$$\{\overline{x_n \mapsto \top}\} \vdash e : \{\overline{\Gamma_k \hookrightarrow a_k}\}$$

can be derived. Hence, Theorem 1 implies the following correctness property of inferred in/out types.

Corollary 1 (Correctness of inferred in/out types). *Inferred in/out types are correct, i.e., if*

$$\{a_{i1} \cdots a_{in} \hookrightarrow a_i \mid i = 1, \ldots, k\}$$

is an inferred in/out type for an n-ary operation f and $(t_1, \ldots, t_n) \in \mathit{Type}(f)$ such that $f(t_1, \ldots, t_n) \rightsquigarrow t$, then there is some $i \in \{1, \ldots, k\}$ with $t_j \in \gamma(a_{ij})$ $(j = 1, \ldots, n)$ and $t \in \gamma(a_i)$.

5 Checking and Inferring Abstract Call Types

In this section we present our method to check and infer abstract call types. In contrast to the inference of in/out types presented in the previous section, correct abstract call types cannot be inferred in one pass over the structure of all functions and their bodies. For instance, consider the operations

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

If we start with the most general abstract call types $\top$ for both operations, the checking of the abstract call type $\top$ (which will be described in Section 5.1) of `hd` is successful, but the abstract call type $\top$ of `head` cannot be correct since `head` fails on the argument `[]`. Thus, one has to refine the abstract call type of `head` to `{:}` and check the definitions again. When `hd` is checked again, the abstract call type $\top$ cannot be correct since `head` requires the abstract call type `{:}`. Thus, one has to refine the abstract call type of `hd` to `{:}` and check the definitions again—which is now successful.

This simple example shows that there might be operations where the abstract call type of one operation depends on the abstract call type of another operation. Due to arbitrary dependencies between operations, such an inference might need several iterations on a program. Therefore, we develop our method in the following steps:

1. We present a method to *check* abstract call types associated to all operations by considering the call structure and in/out types. We prove that, if this check is successful for all operations, then all abstract call types are correct.
2. We propose a method to *refine* a given abstract call type if the check of this type fails. This is based on making abstract call types stronger, i.e., some abstract type a is replaced by a new one a' with $a' \sqsubset a$. Then the refined abstract call types are checked again. This iteration terminates if it is ensured that there are only finitely many refinements for each abstract call type.
3. To reduce the number of iterations, one can compute more precise initial abstract call types for all operations by considering the case structure of their defining rules.
4. Finally, we discuss some extensions of the call type analysis of the kernel language, like higher-order operations and search operators.

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

5.1 Checking Abstract Call Types

To check abstract call types, we assume that two kinds of information are given for each operation f:

- A correct in/out type $IO(f)$ approximating the input/output behavior of f (as described in Section 4).
- An abstract call type $CT(f)$ specifying the requirements to evaluate f without failure.

At the moment, we assume that abstract call types are given and present a method to check them for correctness. Basically, this is done by traversing the body of each operation and check whether the function applications satisfy the abstract call types

required by the called functions. The precise definition of this method will be presented in this section.

As discussed in Section 4, it is important to have information about the input/output behavior of operations. Therefore, we introduced the notion of in/out types. Now we use this information to approximate values of variables occurring in program rules and pass this information through the inference rules. For this purpose, we use variable types (we omit "abstract" since this is clear from the context) which relate variables and in/out types.

Definition 8 (Variable types). *A* variable type *is a triple of the form*

$$(z, io, x_1 \dots x_n)$$

where $z, x_1, \dots, x_n$ are program variables and io is an in/out type for an n-ary operation. We denote by $x :: a$ the basic variable type $(x, \{\varepsilon \mapsto a\}, \varepsilon)$.

Let Δ be a set of variable types. The domain *of Δ is defined by*

$$Dom(\Delta) = \{x \mid (x, io, x_1 \dots x_n) \in \Delta\}$$

$\Delta(x)$ denotes the least upper bound of all abstract type information about variable x available in Δ, which is defined by

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

Furthermore, $\Delta|_{x=a}$ denotes the set of variable types Δ restricted to the case where x has abstract value $a \in \mathcal{A}$. This is defined by

$$\Delta|_{x=a} = \{(x, io|_a, xs) \mid (x, io, xs) \in \Delta\} \cup \{(y, io, ys) \in \Delta \mid x \neq y\}$$

The restriction of an in/out type *io to some abstract result a is defined by*

$$io|_a = \{a_1, \dots, a_n \hookrightarrow (a_0 \sqcap a) \mid a_1, \dots, a_n \hookrightarrow a_0 \in io\}$$

A variable type $(z, io, x_1 \dots x_n)$ is interpreted as: z might have some value of the result type a for some $a_1 \dots a_n \hookrightarrow a \in io$ and, in this case, $x_1, \dots, x_n$ have values of type $a_1, \dots, a_n$, respectively. For instance, the set of variable types

$\Delta = \{(\mathtt{y}, \{\{\mathtt{[]}\} \hookrightarrow \{\mathtt{True}\}, \{\mathtt{:}\} \hookrightarrow \{\mathtt{False}\}\}, \mathtt{x})\}$

expresses that `y` has value `True` and `x` is the empty list, or `y` has value `False` and `x` is a non-empty list.

Restrictions of set of variables types are used when some (abstract) value of a variable is known. For instance, the restriction of Δ defined above to the case where `y` has the abstract value {`True`} is

$\Delta|_{\mathtt{y}=\{\mathtt{True}\}} = (\mathtt{y}, \{\{\mathtt{[]}\} \hookrightarrow \{\mathtt{True}\}, \{\mathtt{:}\} \hookrightarrow \varnothing\}, \mathtt{x})$

Since the element $\{\mathtt{:}\} \hookrightarrow \varnothing$ contains no information according to the interpretation of variable types (see Proposition 1 below), we can omit it so that we write the restriction in the simpler form

$\Delta|_{\mathtt{y}=\{\mathtt{True}\}} = (\mathtt{y}, \{\{\mathtt{[]}\} \hookrightarrow \{\mathtt{True}\}\}, \mathtt{x})$

Sets of variable types are used to approximate concrete substitutions occurring during the evaluation of expressions. For this purpose, we define when this approximation is correct for a substitution.

Definition 9 (Correct set of variable types). *Let Δ be a set of variable types and σ be a substitution, i.e., a mapping from variables to data terms. Δ is* correct *for σ if, for all variables $x \in Dom(\Delta)$, there is some $(x, io, x_1 \dots x_n) \in \Delta$ and some $a_1 \dots a_n \hookrightarrow a \in io$ such that $\sigma(x) \in \gamma(a)$ and $\sigma(x_i) \in \gamma(a_i)$ for $i = 1, \dots, n$.*

Thus, a set of variable types is correct for a substitution if there is *some* in/out type conform to the substitution. This weakness is a consequence of approximating non-deterministic computations where various result values might be computed for an expression.

Since in/out types with empty results are not relevant for correct sets of variable types, these elements can be removed (as shown in the previous example). The following proposition justifies it.

Proposition 1. *Let $\Delta = \Delta' \cup \{(x, io \cup \{a_1, \dots, a_n \hookrightarrow \bot\}, y_1 \dots y_n)\}$ be a set of variable types. If Δ is correct for a substitution σ, then $\Delta' \cup \{(x, io, y_1 \dots y_n\}$ is also correct for σ.*

Proof. By Definition 9, the in/out element $a_1, \dots, a_n \hookrightarrow \bot$ is not relevant for the correctness, since $\gamma(\bot) = \varnothing$ by Definition 2. □

If variables types have definite information, i.e., a single in/out element, we can use it for definite reasoning by propagating single abstract values. This is justified by the following proposition. For the sake of simplicity, we state it only for unary operations but it is not difficult to extend it to operations with more than one argument.

Proposition 2. *Let $\Delta = \Delta' \cup \{(x, \{a \hookrightarrow a'\}, y)\}$ be a set of variable types with $x \notin Dom(\Delta')$. If Δ is correct for a substitution σ, then $\Delta'|_{y=a} \cup \{(x, \{a \hookrightarrow a'\}, y)\}$ is also correct for σ.*

Proof. Let Δ be defined as above and σ be a substitution so that Δ is correct for σ. By Definition 9, $\sigma(x) \in \gamma(a')$ and $\sigma(y) \in \gamma(a)$. Again by Definition 9, there is some $(y, io, y_1 \dots y_n) \in \Delta'$ and some $a_1 \dots a_n \hookrightarrow a_0 \in io$ such that $\sigma(y) \in \gamma(a_0)$ and $\sigma(y_i) \in \gamma(a_i)$, for $i = 1, \dots, n$. By Definition 8, $a_1 \dots a_n \hookrightarrow (a_0 \sqcap a) \in io|_a$ and $(y, io|_a, y_1 \dots y_n) \in \Delta'|_{y=a}$. Since $\sigma(y) \in \gamma(a_0)$ and $\sigma(y) \in \gamma(a)$, $\sigma(y) \in \gamma(a_0 \sqcap a)$ (since $(a_0 \sqcap a) \sqsubseteq a_0$, $(a_0 \sqcap a) \sqsubseteq a$, and Definition 2). Thus, $\Delta'|_{y=a}$ is correct for σ. □

Our method to check the correctness of abstract call types given for all functions is specified by the inference rules of Figure 5. This inference system derives judgements of the form

$$\Delta, z = e \vdash \Delta'$$

containing sets of variable types Δ, Δ', a variable z, and an expression e. This is interpreted as (a precise statement is made in Theorem 2 below):

> If there is a substitution σ so that Δ is correct for σ, then $\sigma(e)$ evaluates without a failure and, if z is bound to the result of this evaluation, Δ' is correct for σ.

To check the abstract call type $a_1 \dots a_n$ of an operation f defined by $f(x_1, \dots, x_n) = e$, we try to derive the judgement

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

$$\mathsf{Var}_{nf} \quad \Delta, z = x \vdash \{z :: \Delta(x)\}$$

$$\mathsf{Cons}_{nf} \quad \Delta, z = c(x_1, \ldots, x_n) \vdash \{z :: c^{\alpha}(\overline{\Delta(x_n)})\}$$

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

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

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

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

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

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

Fig. 5. Checking abstract call types

for some fresh variable z, i.e., z is different from every x_i $(i = 1, \ldots, n)$ and does not occur in expression e. Thus, we assign the abstract call type of f as initial values of the parameters and analyze the right-hand side of the operation.

Keeping the interpretation of variable types in mind, the inference rules are not difficult to understand. Rule Var_{nf} is immediate: the evaluation of a value cannot fail so that we set the result z to the abstract type of x. Rule Cons_{nf} adds the abstract condition that z is bound to the constructor c after the evaluation. Rule Func_{nf} is the first interesting rule. The condition states that the abstract arguments of the function must be smaller than the required abstract call type so that the concrete values are in a subset relationship. If this requirement on abstract call types holds, the operation is evaluable and we connect the results and the arguments with the in/out type of the operation. The rules for disjunctions and free variable introduction are straightforward. There is no rule for *failed* since its evaluation is always failing. In rule Let_{nf}, the result of analyzing the local binding is used to analyze the expression. We finally discuss the most important rule for case selection.

In rule Case_{nf}, the set of variable types Δ is restricted in each branch to the abstract value of the pattern in this branch. It may happen that this restriction yields the empty (bottom) type for the discriminating variable of the case expression. This means that there is no concrete value for the variable in this branch so that we call such a branch *unreachable*. Therefore, the right-hand side of an unreachable branch need not be analyzed so that rule Case_{nf} does not consider unreachable branches. For the remaining reachable branches, each right-hand side is analyzed with the restricted set of variable types so that the value of the discriminating variable in the specific branch is considered when checking the right-hand side of the branch.

As a simple example for checking abstract call types, consider the operation

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

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

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

The abstract call type of `head` is assumed to be $\{\texttt{:}\}$. When we check the right-hand side of the definition of `f`, we use the initial set of variable types

$\Delta_0 = \{(\texttt{x}, \{\varepsilon \hookrightarrow \top\}, \varepsilon)\}$

The check of the *let* binding `y = null(x)` adds the variable type

$\Delta_1 = \{(\texttt{y}, IO(\texttt{null}), \texttt{x})\}$

so that we check the `case` expression with the set of variable types

$\Delta_2 = \{(\texttt{x}, \{\varepsilon \hookrightarrow \top\}, \varepsilon),\ (\texttt{y}, \{\{\texttt{[]}\} \hookrightarrow \{\texttt{True}\}, \{\texttt{:}\} \hookrightarrow \{\texttt{False}\}\}, \texttt{x})\}$

The check of the first case branch is immediate: we use the restricted set of variable types

$\Delta_3 = \Delta_2|_{\texttt{y} = \{\texttt{True}\}}$

to check the constructor `True` and obtain the set of variable types

$\Delta_4 = \{(\texttt{z}, \{\texttt{True}\}, \varepsilon)\}$

for this branch.

For the second case branch, we restrict Δ_2 to

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

The definite binding for `y` implies a definite binding for `x` (see Proposition 2) so that we can simplify it to

$\Delta_5 = \{(\texttt{x}, \{\varepsilon \hookrightarrow \{\texttt{:}\}\}, \varepsilon),\ (\texttt{y}, \{\{\texttt{:}\} \hookrightarrow \{\texttt{False}\}\}, \texttt{x})\}$

to check the right-hand side of this branch. Hence, if we check the call "`head(x)`" w.r.t. Δ_5, the abstract argument type is $\Delta_5(\texttt{x}) = \{\texttt{:}\}$ so that the call type of `head` is satisfied.

Altogether, the proof tree to check the call type of `f` is as follows:

$$\dfrac{\Delta_0, \texttt{y} = \texttt{null(x)} \vdash \Delta_1 \qquad \dfrac{\Delta_3, \texttt{z} = \texttt{True} \vdash \Delta_4 \qquad \Delta_5, \texttt{z} = \texttt{head(x)} \vdash \{(\texttt{z}, IO(\texttt{head}), \texttt{x})\}}{\Delta_2, \texttt{z} = \textit{case}\ \texttt{y}\ \textit{of}\ \ldots \vdash \{(\texttt{z}, \{\texttt{True}\}, \varepsilon), (\texttt{z}, IO(\texttt{head}), \texttt{x})\}}}{\Delta_0, \texttt{z} = \textit{let}\ \texttt{y = null(x)}\ \textit{in case}\ \texttt{y}\ \textit{of}\ \ldots \vdash \{(\texttt{z}, \{\texttt{True}\}, \varepsilon), (\texttt{z}, IO(\texttt{head}), \texttt{x})\}}$$

5.2 Correctness of Abstract Call Types Checking

In the following we show that the inference rules in Figure 5 correctly approximate the non-failure property of operations. As we will see, this is the case if the abstract call types of all functions are "successfully checked" as defined in the following definition.

Definition 10 (Verification of Abstract Call Types). *Let CT be a mapping of functions to abstract call types and f be an operation defined by $f(x_1, \ldots, x_n) = e$ and $CT(f) = a_1 \ldots a_n$. If z is a fresh variable not occurring in the definition of f and the judgement*

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

is derivable with the rules in Figure 5, we say that $CT(f)$ is verified.

A (variable-free) expression e is called *non-failing* if all finite reductions of e end in a data term. In the following we show that function calls are non-failing for arguments satisfying CT if the abstract call types of all functions specified by CT are verified.

We state two properties of correct sets of variables which are used in the correctness proof of abstract call type checking.

Lemma 1. *If a set of variable types Δ is correct for a substitution σ, then $\sigma(x) \in \gamma(\Delta(x))$ for all variables $x \in Dom(\Delta)$.*

Proof. If Δ is correct for σ and $x \in Dom(\Delta)$, then there is some $(x, io, x_1 \dots x_n) \in \Delta$ and some $a_1 \dots a_n \hookrightarrow a \in io$ with $\sigma(x) \in \gamma(a)$. Hence, by definition of $\Delta(x)$, $a \sqsubseteq \Delta(x)$ so that $\sigma(x) \in \gamma(a) \subseteq \gamma(\Delta(x))$. □

Lemma 2. *If $\Delta = \{(x, \{\varepsilon \mapsto a\}, \varepsilon)\}$ and $\sigma(x) \in \gamma(a)$, then Δ is correct for σ.*

Proof. This is an immediate consequence of Definition 9. □

Now we can state the correctness of the inference rules in Figure 5 as follows.

Theorem 2. *Assume that the call types of all functions specified by CT are verified. Let $Var(e) \subseteq Dom(\Delta)$, $z \notin Var(e)$, $\Delta, z = e \vdash \Delta'$ be derivable with the inference rules in Figure 5, and σ be a substitution such that Δ is correct for σ. Then all finite reductions of $\sigma(e)$ end in some data term t with $t \in \gamma(\Delta'(z))$ and Δ' is correct for σ.*

Proof. We assume that the preconditions of the theorem hold, i.e., $\Delta, z = e \vdash \Delta'$ is derivable with a proof tree of height h and σ is a substitution such that Δ is correct for σ. By Lemma 1, $\sigma(x) \in \gamma(\Delta(x))$ for all $x \in Var(e)$.

We prove the claim by induction on the number of steps of finite evaluations of $\sigma(e)$ and a nested induction on the height of the proof tree. We distinguish the kind of expression e (which cannot be *failed* since there is no derivation rule for it):

Variable: If $e = x$, where x is a variable, then $\sigma(x) \in \gamma(\Delta(x))$ is a data term so that the evaluation of $\sigma(e)$ ends in $\sigma(x)$ (base case with 0 derivation steps) and $\sigma(x) \in \gamma(\Delta(x)) = \gamma(\Delta'(z))$ (by rule Var_{nf}). By Lemma 2, the claim holds.

Constructor: If $e = c(x_1, \dots, x_n)$, where x is a variable, then $\sigma(x_i) \in \gamma(\Delta(x_i))$ are data terms ($i = 1, \dots, n$) so that $\sigma(e)$ is also a data term. Hence, the evaluation of $\sigma(e)$ ends in the data term $t = \sigma(e)$ (base case with 0 derivation steps). Since $\Delta' = \{(z, \{\varepsilon \mapsto c^{\alpha}(\overline{\Delta(x_n)})\}, \varepsilon)\}$ (by rule Cons_{nf}), $t = \sigma(e) \in \gamma(\Delta'(z))$ so that the claim holds by Lemma 2.

Function: Let $e = f(x_1, \dots, x_n)$, f defined by $f(y_1, \dots, y_n) = e'$, and $CT(f) = a_1 \dots a_n$. Since Func_{nf} is applicable, $\sigma(x_i) \in \gamma(\Delta(x_i)) \subseteq \gamma(a_i)$ ($i = 1, \dots, n$). Let $\sigma' = \{\overline{y_n \mapsto \sigma(x_n)}\}$. Then $\sigma(e)$ is reducible to $\sigma'(e')$ and, for all finite derivations of $\sigma(e)$, the same result can be derived from $\sigma'(e')$ with a smaller number of derivations steps so that we can apply the induction hypothesis to $\sigma'(e')$. Let

$$\Delta'' = \{(y_1, \{\varepsilon \mapsto a_1\}, \varepsilon), \dots, (y_n, \{\varepsilon \mapsto a_n\}, \varepsilon)\}$$

Since $\sigma'(y_i) = \sigma(x_i) \in \gamma(a_i)$ for $i = 1, \dots, n$, by Lemma 2, Δ'' is correct for σ'. Since the call type of f is verified, $\Delta'', z' = e' \vdash \Delta'''$ is derivable for some fresh variable z' and some set Δ''' of variable types. By the induction hypothesis, $\sigma'(e')$ is non-failing. Let t be some value of an evaluation of $\sigma'(e')$. The claim holds by Corollary 1: In particular, there is some $a_1 \cdots a_n \hookrightarrow a \in IO(f)$ with $t \in \gamma(a)$. Since $a \sqsubseteq \Delta'(z)$ (by rule Func_{nf} and definition of $\Delta'(z)$), $t \in \gamma(\Delta'(z))$ so that the claim holds.

Or expression: If $e = e_1$ *or* e_2, then the first step of evaluating $\sigma(e)$ reduces it to $\sigma(e_1)$ or $\sigma(e_2)$. Assume the case $\sigma(e_1)$ (the other is symmetric). Since $\Delta, z = e_1 \vdash \Delta_1$ is derivable with a proof tree of height smaller than h, the induction hypothesis implies that $\sigma(e_1)$ is non-failing, Δ_1 is correct for σ, and if t is some value of $\sigma(e_1)$, then $t \in \gamma(\Delta_1(z))$. Since $\Delta \subseteq \Delta'$, $\Delta_1(z) \sqsubseteq \Delta'(z)$ and the claim holds (note that Δ_2 is also correct for σ by the induction hypothesis applied to the proof tree of $\Delta, z = e_2 \vdash \Delta_2$).

Free variables: If $e = \mathit{let}\ x_1, \ldots, x_n\ \mathit{free\ in}\ e'$, then the first step of evaluating $\sigma(e)$ reduces it to $\sigma(\rho(e'))$ for some substitution $\rho = \{\overline{x_n \mapsto t_n}\}$ for the free variables. By rule Free_{nf}, $\Delta \cup \{(x_i, \{\varepsilon \hookrightarrow \top\}, \varepsilon) \mid i \in \{1, \ldots, n\}\}, z = e \vdash \Delta'$ is derivable with a proof tree of height smaller than h. By induction hypothesis (the extended set of variable types is trivially correct for $\sigma \circ \rho$), $\sigma(\rho(e'))$ is non-failing, Δ' is correct for $\sigma \circ \rho$, and if t is some value of $\sigma(\rho(e'))$, then $t \in \gamma(\Delta'(z))$. This proves the claim.

Let binding: Let $e = \mathit{let}\ x = e'\ \mathit{in}\ e''$. First, consider the evaluation of the bound expression $\sigma(e')$. By rule Let_{nf}, $\Delta, x = e' \vdash \Delta'$ is derivable with a proof tree of height smaller than h. Thus, by the induction hypothesis, $\sigma(e')$ is non-failing and Δ' is correct for σ. Consider some value t' of $\sigma(e')$ used to evaluate $\sigma(e)$ to t. Then $t' \in \gamma(\Delta'(x))$ by the induction hypothesis. Let $\sigma' = \{x \mapsto t'\} \circ \sigma$. Since $\Delta \cup \Delta', z = e'' \vdash \Delta''$ is derivable with a proof tree of height smaller than h, by the induction hypothesis (note that $\sigma'(y) \in \gamma((\Delta \cup \Delta')(y)$ for all $y \in \mathit{Var}(e'')$), $\sigma'(e'')$ is also non-failing, Δ'' is correct for σ', and, if t'' is some value of $\sigma'(e'')$, $t'' \in \gamma(\Delta''(z))$. Thus, the claim holds.

Case expression: If $e = \mathit{case}\ x\ \mathit{of}\ \{p_1 \to e_1; \ldots; p_n \to e_n\}$, $\sigma(e)$ is evaluated by selecting some matching branch (note that there is no evaluation step in the discriminating argument $\sigma(x)$ since $\sigma(x) \in \gamma(\Delta(x))$). Such a selection step is possible since $\sigma(x) = c_i(y_1, \ldots, y_m)$ for some $i \in \{1, \ldots, n\}$ (since the case expression covers all constructors). By rule Case_{nf}, $\Delta_i, z = e_i \vdash \Delta_i'$ is derivable with a proof tree of height smaller than h, where $p_i = c_i(x_1, \ldots, x_m)$ and $\Delta_i = \Delta|_{x=c_i^\alpha(\overline{\top})} \cup \{x_1 :: \top, \ldots, x_m :: \top\}$. Since $\Delta|_{x=c_i^\alpha(\overline{\top})}$ constrains the abstract value of x to the abstraction of the actual value $\sigma(x)$, Δ_i is correct for $\sigma \circ \rho$. Thus, by the induction hypothesis, $\sigma(\rho(e_i))$ is non-failing, where ρ is the matching substitution on the variables of pattern p_i, and Δ_i' is correct for $\sigma \circ \rho$. If t is some value of $\sigma(\rho(e_i))$, then $t \in \gamma(\Delta_i'(z))$ so that the claim holds (since $\Delta_i' \subseteq \Delta'$). □

The previous theorem implies the following property of successfully checked call types.

Corollary 2 (Correctness of abstract call type checking). *If, for all operations f defined by $f(x_1, \ldots, x_n) = e$ and $CT(f) = a_1 \ldots a_n$, the judgement*

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

is derivable (for some fresh variable z), then the abstract call type $CT(f)$ is correct, i.e., $f(t_1, \ldots, t_n)$ is non-failing if the arguments satisfy the call types, i.e., $(t_1, \ldots, t_n) \in \mathit{Type}(f)$ and $t_i \in \gamma(a_i)$ $(i = 1, \ldots, n)$.

Proof. Let

$$\Delta_0 = \{(x_1, \{\varepsilon \mapsto a_1\}, \varepsilon), \ldots, (x_n, \{\varepsilon \mapsto a_n\}, \varepsilon)\}$$

and $\sigma = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ where $(t_1, \ldots, t_n) \in \mathit{Type}(f)$ and $t_i \in \gamma(a_i)$ $(i = 1, \ldots, n)$. By Lemma 2, Δ_0 is correct for σ. By Theorem 2, all finite reductions of $\sigma(e)$

end in some data term so that $f(t_1, \ldots, t_n)$ is non-failing. By Definition 3, $CT(f)$ is correct. □

Since the correctness proof is based on the fact that a set of variables types is correct for some substitution, one can modify sets of variable types during an inference as long as these modification do not influence correctness of substitutions. Therefore, it is reasonable to keep sets of variable types in a simplified form, according to Proposition 1 and Proposition 2, whenever they are constructed in a proof tree w.r.t. Figure 5.

5.3 Inferring Abstract Call Types

So far we have seen that we can ensure non-failing computations if we guess abstract call types so that all operations can be successfully checked with the inference rules of Figure 5. A simple but useless guess are always failing call types, i.e., $CT(f) = \bot \ldots \bot$ for all operations f. In order to provide more appropriate abstract call types, we propose a heuristic to infer them by refining abstract call types if they cannot be successfully checked. In the worst case, an always failing call type is inferred, but our evaluation in Section 7 will show that our heuristic produces reasonable results.

To understand our method, we review the situations where a derivation with the inference rules of Figure 5 is not successful. For this purpose, consider that we check an operation f with abstract call type $CT(f) = a_1 \ldots a_n$ with these inference rules, as described in Corollary 2. There are only two possible situations where this check might fail:

1. Rule Func_{nf} is not applicable since there is a function call $g(y_1, \ldots, y_m)$ where the abstract argument types are not smaller than the abstract call type of the function. Hence, $CT(g) = a'_1 \ldots a'_n$ and there is some $i \in \{1, \ldots, m\}$ where $\Delta(y_i) \sqsubseteq a'_i$ does not hold. Then we can distinguish the following cases:
   (a) If y_i is some parameter $j \in \{1, \ldots, n\}$ of f, then we can avoid this failure by making the abstract call type of f stronger, i.e., we set the abstract call type of f to
   $$CT(f) = a_1 \ldots a_{j-1} \, (a_j \sqcap a'_i) \, a_{j+1} \ldots a_n$$
   and check the abstract call types again. Of course, it is not sufficient to check only the definition of f again but also all operations where f is directly or indirectly used.
   (b) If y_i is different from all parameters of f, we set the abstract call type of f to $CT(f) = \bot \ldots \bot$ and check all abstract call types again.

   We can improve the second worst case by checking whether y_i is some subterm of an argument of f, i.e., contained in some constructor pattern argument of f and refine only this part of the abstract call type of f. This could avoid the worst case if the abstract domain can represent this refinement (which is not the case for $\mathcal{A}_1$). However, our evaluation in Section 7 shows that this improvement makes no difference in practice.
2. We try to check an occurrence of *failed* where no inference rule exists. Instead of switching to the worst case, i.e., setting $CT(f) = \bot \ldots \bot$, we can consider whether we are in a branch of some case expression. In this case, we can try to refine the abstract call type of the discriminating variable x of the case expression so that this branch becomes unreachable. For instance, if the pattern of this branch is headed

by the constructor c and C are all constructors of the type containing c, i.e., $c \in C$, then x must have the abstract type

$$\bigsqcup\{d^{\alpha}(\overline{\top}) \mid d \in C, d \neq c\}$$

in order to make this branch unreachable. Now we continue as in the first case, i.e., we check whether x is a parameter of f so that we make the abstract call type of f stronger.

We show a few examples for these refinements of abstract call types. First, consider the already known operation

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

If we check the definition of `head` with the general abstract call type $CT(\texttt{head}) = \top$, we fail due to the second situation described above. Since *failed* occurs in a branch of a case expression, we refine the abstract type of `zs` to $\{\texttt{:}\}$ so that the branch for the empty list becomes unreachable. Since `zs` is also the parameter of `head`, we change the abstract call type of `head` to $CT(\texttt{head}) = \{\texttt{:}\}$ and check `head` again—which succeeds.

As another example, consider the operation

```
hd(ys) = head(ys)
```

already discussed at the beginning of this section. Checking this definition with the correct abstract call type $CT(\texttt{head}) = \{\texttt{:}\}$ and the general abstract call type $CT(\texttt{hd}) = \top$ is not successful: we fail due to the first situation described above since $\Delta(\texttt{ys}) = \top \not\sqsubseteq \{\texttt{:}\}$. Since `ys` is the parameter of `hd`, we refine the abstract call type of `hd` to

$$CT(\texttt{hd}) = (\top \sqcap \{\texttt{:}\}) = \{\texttt{:}\}$$

Now this can be checked as a correct abstract call type.

In order to show an example where such a precise refinement is not possible, consider the operation

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

The abstract type $\{\texttt{:}\}$ for variable `y`, which is demanded by the application `head(y)`, cannot be obtained by restricting the call type of `hdfree` since `y` is not a parameter of `hdfree`. Therefore, the abstract call type of `hdfree` is set to the $CT(\texttt{hdfree}) = \{\}$. This means that a call to `hdfree` might fail. Actually, the evaluation of `hdfree []` returns some list when the local variable is bound to a non-empty list, but it can also fail when the local variable `y` is bound to an empty list (see the definition of `head` shown above). To control this potential failure, one can encapsulate calls to `hdfree` by some search operator, like `allValues` or `oneValue`, and check the result of this encapsulation.

Our heuristic to refine abstract call types leads to an iterated inference. Initially, we start with most general abstract call types for all operations. In each iteration, either all abstract call types can be successfully checked or the abstract call type of some operation becomes stronger, i.e., more restricted. Then we check all operations again with this refined abstract call type. Since in each iteration some abstract call type is replaced by a smaller one, the iteration terminates if it is ensured that there are only finitely many refinements for each abstract call type. For instance, this is the case if all descending chains w.r.t. the ordering $\sqsubseteq$ of the abstract type domain $\mathcal{A}$ are finite. This property is satisfied for the domain of depth-k types.

In the worst case, the abstract call type $\bot$ might be inferred for some operation. Such a result does not mean that this operation is not useful but one has to encapsulate

its use with some safeness check. For instance, if f is an operation with abstract call type $\bot$, one can safely use f by replacing a call $f\ t$ by `oneValue (`$f\ t$`)`, test whether the result is a value, and branch to an error handling case when the result is `Nothing`. Of course, this is not a universal recipe since the exact code changes depend on the application program.

For an efficient implementation of this fixpoint computation, it is reasonable to use call dependencies of operations so that one has to re-check only the operations with refined abstract call types and the operations that use them. We have implemented this strategy in our tool and obtained a good improvement compared to the initial naive fixpoint computation. For instance, the prelude of Curry (the base module containing a lot of basic definitions for arithmetic, lists, type classes, etc) contains 1265 operations (public and also auxiliary operations). After the first iteration, the call types of 20 operations are refined so that 24 operations are reanalyzed in the next iteration. The second iteration refines the call types of 9 operations so that 12 operations have to be reanalyzed. Altogether, the check of the prelude requires five iterations.

5.4 Initial Abtract Call Types

In the iterative inference described above, we start with most general abstract call types for each operation. For instance, we start with $CT(\texttt{head}) = \top$ which is refined to $CT(\texttt{head}) = \{\texttt{:}\}$ by one iteration.

In order to avoid some iterations and speed up the overall inference, we can try to compute more precise initial abstract call types by considering the structure of `case` expressions in the transformed FlatCurry program. If there is a parameter used as a discriminating argument in some case expression but no branch contains *failed*, there is no need to make the abstract call type for this parameter stronger. However, if some branches contain *failed*, we make them unreachable by the method described in Section 5.3. With this method, abstract call types of `head` or `tail` are directly initialized to $\{\texttt{:}\}$ so that an iteration when checking their definition is avoided.

5.5 Extensions

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

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

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

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

External operations Usually, externally defined primitive operations do not fail so that they have trivial abstract call types. There are a few exceptions which are handled by explicitly defined abstract call types, like the always failing operation `failed`, or arithmetic operations like division (see Section 6 for an improved treatment of arithmetic operations).

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

Encapsulation Failures might occur during run time, either due to operations with complex non-failure conditions or due to the use of logic programming techniques with search and failures. In order to ensure an overall non-failing application even in the presence of possibly failing subcomputations, the programmer has to encapsulate such subcomputations and then analyze its outcome, e.g., branching on the result of the encapsulation. For this purpose, one can use an exception handler (which represents a failing computation as an error value) or some method to encapsulate non-deterministic search (e.g., [6,13,33,34]). For instance, the primitive operation `allValues` returns all the values of its argument expression in a list so that a failure corresponds to an empty list. In order to include such a primitive in our framework, we simply skip the analysis of its argument. For instance, a source expression like `allValues (head ys)` is not transformed into `let x = head(ys) in allValues(x)` (where `x` is fresh), but it is kept as it is. Furthermore, rule Func_{nf} is specialized for `allValues` so that the condition on the arguments w.r.t. the call type is omitted and the in/out type is trivial, i.e., $IO(\texttt{allValues}) = \{\top \hookrightarrow \top\}$. In a similar way, other methods to encapsulate possibly non-deterministic and failing operations, like *set functions* [6], can be handled.

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

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

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

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

6 Arithmetic Non-Fail Conditions

6.1 Non-Fail Conditions

Up to now we have approximated possible failures due to incomplete pattern matching. Since patterns are based on constructors, we approximated non-failing argument sets by abstract types, i.e., abstractions of sets of constructor terms. Although this is reasonable for declarative programs where all operations are explicitly defined, it is sometimes too weak for programs with external operations, like arithmetic functions and relations. For instance, the integer division operation `div` fails if the second argument is zero. Since this property cannot be expressed by an abstract call type used so far, the abstract call type of `div` is simply $\bot$ so that all operations using `div` are inferred as failing.

As another example, consider the definition of the factorial function:

```
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. Actually, this definition 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 } }
```

Since the *failed* branch cannot be enforced to be unreachable by restricting the abstract call type of `fac` (as described in Section 5.3), the empty call type is inferred for `fac`. This is unsatisfying for operations which call `fac` only with non-negative numbers. For instance, consider the following code snippet which 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, but our method to infer abstract call types computes $\bot$ for `printFac`.

In order to improve our method so that, for instance, `printFac` can be verified as non-failing, one has to take the semantics of arithmetic operations into account. For instance, consider the point in the definition of `fac` where *failed* is reached. Combining the expressions and the constructors in the selected branches, we know that the following condition holds at this branch:

$$\texttt{(n == 0)} = \texttt{False} \wedge \texttt{(n > 0)} = \texttt{False}$$

Now, if `fac` is called with a non-negative argument, we also know that $\texttt{(n} \geq \texttt{0)}$ holds. Since the combined condition

$$\texttt{(n} \geq \texttt{0)} \wedge \texttt{(n == 0)} = \texttt{False} \wedge \texttt{(n > 0)} = \texttt{False}$$

is unsatisfiable, the last *failed* branch is not reachable. Thus, the condition $\texttt{(n} \geq \texttt{0)}$ is sufficient to ensure that `fac` does not fail.

This shows that it is useful to consider, beyond call types, also other Boolean expressions as non-fail conditions. To obtain a fully automatic method, one needs a tool to reason about the (un)satisfiability of such conditions. In the case of arithmetic operations, an SMT solver [17] is a good choice, e.g., the unsatisfiability of the condition above is automatically verified by an SMT solver like Z3. This extension is developed in detail in [26]. In the following, we sketch its basic ideas and integration into our method.

If the method described in Section 5 infers an abstract call type different from $\bot$ for some operation, one can check applications of this operation with this abstract call type. However, if the inferred call type is $\bot$, i.e., always failing, we have no useful condition to check non-failing applications. In this case, it is reasonable to consider Boolean expressions as non-fail conditions. For instance, a *non-fail condition* for `fac` is

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

Actually, the method described in the following can infer this condition.

6.2 Checking Non-Fail Conditions

Since non-fail conditions are only relevant if an inferred abstract call type is $\bot$, we use a separate inference system for non-fail condition checking. However, in the implemented tool (see Section 7), both abstract call types as well as non-fail conditions are inferred in parallel.

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.[10]

The checking of non-fail conditions is defined by the rules shown in Figure 6. 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.

[10] The precise structure of non-fail conditions is not relevant. It is only necessary to decide whether an implication w.r.t. a non-fail condition holds, which is done by an SMT solver. If it cannot be decided, e.g., due to a timeout, it is assumed that the implication does not hold.

$$\mathsf{Var}_{nfc} \qquad \Gamma, C \vdash x$$

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

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

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

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

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

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

Fig. 6. Checking non-fail conditions

Rule Var_{nfc} 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_{nfc} 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_{nfc}. Rule Func_{nfc} 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_{nfc} 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 (similarly to Section 5) since completely fail-free computations could support simpler and more efficient implementations.

Rule Free_{nfc} 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_{nfc}. This is different in rule Let_{nfc} where the binding $x \mapsto e$ is added to 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_{nfc} 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, as in the example `printFac` shown above. 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.[11] This extended assertion is used

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

to check further function calls occurring in this branch by rule Func_{nfc} 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`.

6.3 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`. Their non-fail conditions can be specified as

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

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

As visible in the rules of Figure 6, the only situation where a rule might not be applicable is rule Func_{nfc} due to its side condition (or an occurrence of *failed* which can be handled as a call to `failed`). Hence, if this condition does not hold, we enforce it by adding the implication, i.e., the formula

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

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 Section 6.1. 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.

Similarly to the inference of abstract call types, 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.

Since the domain of Boolean expressions is not finite, the iterative refinement of non-fail conditions might not terminate. To ensure a finite inference, our implementation simply stops the refinement of non-fail conditions after the second refinement, i.e., the non-fail condition is then set to `False`. Although this heuristic seems limited, the manual inspection of the few cases where unsatisfiable non-fail conditions are inferred showed that better results would not be computable by increasing this limit.

7 Evaluation

We have implemented the methods described above in a tool[12] written in Curry. In this section we evaluate it by discussing some examples and applying it to various libraries.

In general, we have not uncovered unknown bugs in existing libraries and programs. Our method and tool is intended to help programmers to develop more reliable programs—either by checking already developed programs (as shown below) or by taking into account non-fail conditions during program development. To support the latter aspect, our inference tool has been integrated into CurryInfo [28], a tool to manage information about program entities defined in modules of Curry packages. The information managed by CurryInfo can be accessed by a Curry REPL or IDE. For instance, the Curry Language Server[13] can be configured so that information from CurryInfo is shown when hovering over a program entity. With this configuration, a programmer using Visual Studio Code can immediately see the non-fail condition of an imported entity when hovering over its name. In this way, specific non-trivial non-fail conditions can be considered during program development (see [28] for more details). To implement this development support, it is important to have a fully automatic method to infer non-fail conditions, since the existing Curry packages[14] contain more than seven hundred Curry modules.

7.1 Examples

In the first part of the evaluation, we compare our approach to a previous tool to verify non-failing Curry programs [23]. In that tool the programmer has to annotate partially defined operations with *non-fail conditions*. Based on these conditions, the tool extracts proof obligations from a program which are sent to an SMT solver. For instance, consider the operation to compute the last element of a non-empty list:

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

[12] The implementation is available as Curry package `verify-non-fail`, see `https://cpm.curry-lang.org/pkgs/verify-non-fail.html`. There is also a web interface to the inference system, accessible at `https://cpm.curry-lang.org/webapps/failfree/`.

[13] `https://github.com/fwcd/curry-language-server`

[14] `https://cpm.curry-lang.org/`

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

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

This predicate together with the definition of the involved operations are translated to SMT formulas and then checked by an SMT solver, e.g., Z3 [17]. Using the new approach described in this paper, the abstract call type $CT(\texttt{last}) = \{:\}$ is automatically inferred and the definition of `last` is successfully checked.

Actually, we tested our new tool on various libraries and could deduce all manually written non-fail conditions of [23] except for the prelude operation "`!!`" where the call `(xs!!n)` returns the `n`-th element of the list `xs`. A reasonable non-fail condition, as shown in [23], requires that the length of the input list `xs` is greater than `n` (note that elements are indexed from 0). If this condition is explicitly stated, as in [23], our tool can verify it. However, the non-fail condition inferred by our tool requires that the list is non-empty and `n` has value 0. Although this is a correct non-fail condition, it is too restrictive.

Another interesting example discussed in [23] is the operation `split` from the library `Data.List`. This operation takes a predicate and a list as arguments and splits this list into sublists at positions where the predicate holds. It is defined in Curry as

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

The interesting point in this example is the pattern matching in the right-hand side "`let (ys:yss) =` $\cdots$". In order to implement this pattern matching in a lazy manner, specific selector operations are generated when this definition is transformed into a kernel language like FlatCurry, since FlatCurry allows only variable bindings but not constructor patterns in `let` expressions. Thus, the FlatCurry code generated for this definition adds two selector operations (named `split_ys` and `split_yss` below) to implement the lazy pattern matching in the `let` expression. The standard front end of Curry implementations translates the definition above into the following FlatCurry code (which is bit more relaxed than required in Figure 1):

```
split :: (a → Bool) → [a] → [[a]]
split(p,zs) = case zs of
  []      → [] : []
  x : xs  → let px = apply(p,x)
            in case px of
                True  → [] : split(p,xs)
                False → let o = otherwise
                        in case o of
                            True  → let ts = split(p,xs)
                                    in let ys = split_ys(ts)
                                       in let yss = split_yss(ts)
                                          in (x : ys) : yss
                            False → failed

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

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

`apply` is a predefined primitive operation to implement higher-order application. The predefined operation `otherwise` is equivalent to `True` so that the occurrence of *failed* in `split` is not reachable.

Note that the selector operations `split_ys` and `split_yss` are partially defined (they correspond to `head` and `tail`). Since they are generated during the compilation process, a non-fail condition cannot be explicitly defined in the source program so that the tool described in [23] could not verify this definition of `split`. As mentioned in [23], it is necessary to include explicit calls to `head` and `tail` instead of the pattern matching of `let`. Moreover, the post-condition

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

had to be added and proved by a separate contract checker [24] so that this information is used by the verifier to ensure that the recursive call to `split` always returns a non-empty list.

With the method described in this paper, such manual additions are not required since the call types of the generated selector operations are automatically inferred together with the in/out type

$IO(\texttt{split}) = \{\top \cdot \{\texttt{[]}\} \hookrightarrow \{\texttt{:}\},\ \top \cdot \{\texttt{:}\} \hookrightarrow \{\texttt{:}\}\}$

Thanks to this in/out type, the abstract result of the recursive call to `split` is $\{\texttt{:}\}$ which matches the call types required for the selector operations. Thus, in contrast to [23], no manual annotations or code modifications are necessary to check the non-failure of `split`.

Although the examples discussed so far are purely functional, our method is also applicable to logic-oriented programs. Since an operation is non-failing if *all* finite derivations result in a data term, the definition of the non-deterministic list insertion operation `insert`, shown in Section 2.2, is considered as failing. However, the slightly modified definition

```
insert x []     = [x]
insert x (y:ys) = (x : y : ys) ? (y : insert x ys)
```

is non-failing. If this definition is used to define a permutation operation

```
perm []     = []
perm (x:xs) = insert x (perm xs)
```

our method infers the abstract call type $\top$ for `perm`, i.e., it verifies that `perm` never results in a failure. As a further logic-oriented example, the prelude of Curry defines an operation `anyOf` which non-deterministically returns any element of a list by

```
anyOf :: [a] → a
anyOf xs = foldr1 (?) xs
```

Since `foldr1` accumulates non-empty lists, our method yields the abstract call type $\{\texttt{:}\}$ for `anyOf`, i.e., `anyOf` never results in a failure if it is applied to a non-empty list of values.

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`	214/1265	20/69	2/9	9/51	66/14	5	5375
`Data.Char`	9/9	0/0	0/0	0/0	0/0	1	6
`Data.Either`	7/11	2/2	0/0	0/0	0/0	1	0
`Data.List`	49/87	8/16	0/0	1/1	18/2	3	2208
`Data.Maybe`	8/9	0/0	0/0	0/0	0/0	1	0
`Numeric`	5/7	0/0	0/0	0/0	0/0	1	1
`System.IO`	23/51	0/0	0/0	0/0	0/0	1	2
`Text.Show`	4/4	0/0	0/0	0/0	0/0	1	0
`Examples`	11/15	1/3	4/4	0/0	17/13	3	347

7.2 Benchmarks

After the discussion of individual examples, we show some data obtained by applying the inference tool to various libraries implemented in Curry.

If our tool is applied to a Curry module, it infers in/out types and initial abstract call types of all operations defined in this module and then checks all branches and calls whether they might be failing. In the case of failures, some abstract call types are refined until the abstract call types of all operations can be verified. At the end, non-trivial abstract call types are reported so that the programmer can decide to either accept the refined abstract call types or modify the program code to handle possible failures so that the abstract call type does not need a refinement.

Table 1 contains the results of checking various Curry libraries and the module `Examples` containing various smaller examples discussed in this paper. Non-fail conditions were not explicitly provided (except for external operations, see Section 6.3). 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 abstract call types. Thus, this is the number of operations which might fail but the set of arguments to avoid a failing computation can be described by a non-empty abstract 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., like the prelude operation `failed` or operations involving predefined unification operators where failures depend on the actual arguments at run time).

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 abstract 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_{nfc} in Figure 6), 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 abstract 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 abstract call types), their number is shown in the next to last column. The last column shows the verification time in milliseconds.[15]

This evaluation indicates that even quite complex modules, like the prelude, have only a few operations with non-trivial non-fail conditions or abstract 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-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 [23], 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.

Our inference of abstract call types is parametric w.r.t. the domain of abstract types. In the examples of the paper, we used a simple domain, where data terms are abstracted to their top-level constructors. The actual implementation supports also depth-k abstractions [43] with $k > 1$. In principle, deeper abstractions could provide more precision. For instance, consider the following operations:

```
falseTail (False:xs) = xs

ft = falseTail [False,True]
```

With the abstract type domain $\mathcal{A}_1$, the abstract call type $\bot$ is inferred for both operations, since the partial pattern matching below a constructor cannot be represented in $\mathcal{A}_1$. However, with the abstract type domain $\mathcal{A}_2$, the abstract call type $\top$ is inferred for `ft`, since the abstract call type of `falseTail` can be represented by the $\mathcal{A}_2$-element

$\{\{\texttt{False}\} : \top\}$

Thus, we can verify that `ft` is non-failing with the domain $\mathcal{A}_2$ but not with $\mathcal{A}_1$. In practice, however, these differences do not seem relevant. At least, this is indicated by the analysis of standard libraries. Table 2 shows the results for $k = 1$, $k = 2$, and $k = 5$. The columns are as in Table 1. Since the use of an SMT solver is not relevant

[15] 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.

Table 2. Inference of non-fail conditions w.r.t. different domains

Inference of abstract call types with the depth-1 domain

Module	operations	in/out	initial	final	failing	iter.	time
`Prelude`	214/1265	76/858	19/40	22/78	11/60	5	594
`Data.Char`	9/9	0/0	0/0	0/0	0/0	1	61
`Data.Either`	7/11	5/9	2/2	2/2	0/0	1	33
`Data.List`	49/87	39/73	7/15	8/16	1/1	2	72
`Data.Maybe`	8/9	7/8	0/0	0/0	0/0	1	32
`Numeric`	5/7	0/2	0/0	0/0	0/0	1	57
`System.IO`	23/51	2/12	0/0	0/0	0/0	1	36
`Text.Show`	4/4	4/4	0/0	0/0	0/0	1	32

Inference of abstract call types with the depth-2 domain

Module	operations	in/out	initial	final	failing	iter.	time
`Prelude`	214/1265	76/858	19/40	22/78	11/60	5	592
`Data.Char`	9/9	0/0	0/0	0/0	0/0	1	60
`Data.Either`	7/11	5/9	2/2	2/2	0/0	1	32
`Data.List`	49/87	39/73	7/15	8/16	1/1	2	72
`Data.Maybe`	8/9	7/8	0/0	0/0	0/0	1	31
`Numeric`	5/7	0/2	0/0	0/0	0/0	1	56
`System.IO`	23/51	2/12	0/0	0/0	0/0	1	35
`Text.Show`	4/4	4/4	0/0	0/0	0/0	1	32

Inference of abstract call types with the depth-5 domain

Module	operations	in/out	initial	final	failing	iter.	time
`Prelude`	214/1265	76/858	19/40	22/78	11/60	5	588
`Data.Char`	9/9	0/0	0/0	0/0	0/0	1	61
`Data.Either`	7/11	5/9	2/2	2/2	0/0	1	32
`Data.List`	49/87	39/73	7/15	8/16	1/1	2	73
`Data.Maybe`	8/9	7/8	0/0	0/0	0/0	1	33
`Numeric`	5/7	0/2	0/0	0/0	0/0	1	56
`System.IO`	23/51	2/12	0/0	0/0	0/0	1	34
`Text.Show`	4/4	4/4	0/0	0/0	0/0	1	32

for this comparison, it has been turned off (which explains the faster execution times compared to Table 1). Apart from the execution times, the results are identical.

8 Related Work

The exclusion of run-time failures at compile time is a practically relevant but also challenging issue. Therefore, there are many approaches targeting it so that we can only discuss a few of them. We concentrate on approaches for functional and logic programming, although there are also many in the imperative world. As mentioned in the introduction, the exclusion of dereferencing null pointers is quite relevant there. As an example from object-oriented programming, the Eiffel compiler uses appropriate type declarations and static analysis to ensure that pointer dereference failures cannot occur in accepted programs [35].

In logic programming, there is no common definition of "non-failing" due to different interpretations of non-determinism. Whereas we are interested to exclude any failure in a top-level computation, other approaches, like [14,18], consider a predicate in a logic program as non-failing if at least one answer is produced. Similarly to our approach, type abstractions are used to approximate non-failure properties, but the concrete methods are different.

Another notion of failing programs in a dynamically typed programming language is based on success types, e.g., as used in Erlang [32]. Success types over-approximate possible uses of an operation so that an empty success type indicates an operation that never evaluates to some value. Thus, success types can show definite failures, whether we are interested in definite non-failures.

Strongly typed programming languages are a reasonable basis to check run-time failures at compile time, since the type system already ensures that some kind of failures cannot occur ("well-typed programs do not go wrong" [36]). However, failures due to definitions with partial patterns are not covered by a standard type system. Therefore, Mitchell and Runciman developed a checker for Haskell to verify the absence of pattern-match errors due to incomplete patterns [37,38]. Their checker extracts and solves specific constraints from pattern-based definitions. Although these constraints have similarities to the abstract type domain $\mathcal{A}_1$, our approach is generic w.r.t. the abstract type domain so that it can also deal with more powerful abstract type domains.

An approach to handle more complex non-fail conditions is described in [31]. 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 stronger or more expressive in order to encode non-failing conditions in the types. For instance, operations in dependently typed programming languages, such as Coq [11], Agda [39], or Idris [12], must be totally defined, i.e., terminating and non-failing. Such languages have termination checkers but non-fail conditions need to be explicitly encoded in the types. For instance, the definition of the operation `head` in Agda [39] requires, as an additional argument, a proof that the argument list is not empty. Thus, `head` could have the type signature

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

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

Refinement or liquid types [42], as used in LiquidHaskell [46,47], are another approach to encode non-failing conditions or more general contracts on the type level. Refinement types extend standard types by a predicate that restricts the set of allowed values. For instance, the applications of `head` to the empty list can be excluded by the following refinement type [46]:

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

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 [42] provided that the expected refinement types can be described with the given entities. The latter restriction is relaxed in [45], 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 approach allows to verify operations defined on algebraic data types without an external (SMT) solver.

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 [23]. As already discussed in Section 7, the advantage of our new approach is the automatic inference of non-failing conditions which supports an easier application to larger programs.

9 Conclusions

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

Although the inference of non-fail conditions is based on a fixpoint iteration and might yield, in the worst case, an unsatisfiable non-fail condition, our practical evaluation showed that even larger modules 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.

In order to use our method during the development of real applications, more engineering work is necessary. Real applications implemented in Curry, like the Curry

package manager[16] [22] or the curricula and module information system of our department[17], consists of more than one hundred modules organized in dozens of software packages. As discussed at the beginning of Section 7, CurryInfo [28] already provides an infrastructure to organize analysis and verification results of Curry software packages. This is helpful to provide information about non-fail conditions of imported program entities, but the immediate verification along with the development of programs is a non-trivial task and requires more work.

Acknowledgments. The authors is grateful to the anonymous reviewers for their helpful and constructive comments to improve this paper.

References

1. E. Albert, M. Hanus, F. Huch, J. Oliver, and G. Vidal. Operational semantics for declarative multi-paradigm languages. *Journal of Symbolic Computation*, 40(1):795–829, 2005.
2. M. Alpuente, M. Comini, S. Escobar, M. Falaschi, and S. Lucas. Abstract diagnosis of functional programs. In *Proc. of the 12th Int'l Workshop on Logic-Based Program Synthesis and Transformation (LOPSTR 2002)*, pages 1–16. Springer LNCS 2664, 2002.
3. S. Antoy. Constructor-based conditional narrowing. In *Proc. of the 3rd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP 2001)*, pages 199–206. ACM Press, 2001.
4. S. Antoy and B. Braßel. Computing with subspaces. In *Proceedings of the 9th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'07)*, pages 121–130. ACM Press, 2007.
5. S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. *Journal of the ACM*, 47(4):776–822, 2000.
6. S. Antoy and M. Hanus. Set functions for functional logic programming. In *Proceedings of the 11th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'09)*, pages 73–82. ACM Press, 2009.
7. S. Antoy and M. Hanus. Functional logic programming. *Communications of the ACM*, 53(4):74–85, 2010.
8. S. Antoy, M. Hanus, A. Jost, and S. Libby. ICurry. In *Declarative Programming and Knowledge Management - Conference on Declarative Programming (DECLARE 2019)*, pages 286–307. Springer LNCS 12057, 2020.
9. D. Bert and R. Echahed. Abstraction of conditional term rewriting systems. In *Proc. of the 1995 International Logic Programming Symposium*, pages 147–161. MIT Press, 1995.
10. D. Bert, R. Echahed, and M. Østvold. Abstract rewriting. In *Proc. Third International Workshop on Static Analysis*, pages 178–192. Springer LNCS 724, 1993.
11. Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.
12. E. Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23(5):552–593, 2013.
13. B. Braßel, M. Hanus, and F. Huch. Encapsulating non-determinism in functional logic computations. *Journal of Functional and Logic Programming*, 2004(6), 2004.
14. F. Bueno, P. López-García, and M.V. Hermenegildo. Multivariant non-failure analysis via standard abstract interpretation. In *7th International Symposium on Functional and Logic Programming (FLOPS 2004)*, pages 100–116. Springer LNCS 2998, 2004.

[16] `https://curry-lang.org/tools/cpm/`
[17] `https://moduldb.informatik.uni-kiel.de/`

15. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *Proc. of the 4th ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.
16. P.W. Dart and J. Zobel. A regular type language for logic programs. In F. Pfenning, editor, *Types in Logic Programming*, pages 157–187. MIT Press, 1992.
17. L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proc. of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008)*, pages 337–340. Springer LNCS 4963, 2008.
18. S. Debray, P. López-García, and M.V. Hermenegildo. Non-failure analysis for logic programs. In *14th International Conference on Logic Programming (ICLP'97)*, pages 48–62. MIT Press, 1997.
19. J.P. Gallagher and K.S. Henriksen. Abstract domains based on regular types. In *20th International Conference on Logic Programming (ICLP 2004)*, pages 27–42. Springer LNCS 3132, 2004.
20. J.C. González-Moreno, M.T. Hortalá-González, F.J. López-Fraguas, and M. Rodríguez-Artalejo. An approach to declarative programming based on a rewriting logic. *Journal of Logic Programming*, 40:47–87, 1999.
21. M. Hanus. Functional logic programming: From theory to Curry. In *Programming Logics - Essays in Memory of Harald Ganzinger*, pages 123–168. Springer LNCS 7797, 2013.
22. M. Hanus. Semantic versioning checking in a declarative package manager. In *Technical Communications of the 33rd International Conference on Logic Programming (ICLP 2017)*, OpenAccess Series in Informatics (OASIcs), pages 6:1–6:16. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017.
23. M. Hanus. Verifying fail-free declarative programs. In *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming (PPDP 2018)*, pages 12:1–12:13. ACM Press, 2018.
24. M. Hanus. Combining static and dynamic contract checking for Curry. *Fundamenta Informaticae*, 173(4):285–314, 2020.
25. M. Hanus. From logic to functional logic programs. *Theory and Practice of Logic Programming*, 22(4):538–554, 2022.
26. M. Hanus. Hybrid verification of declarative programs with arithmetic non-fail conditions. In *Proc. of the 22nd Asian Symposium on Programming Languages and Systems (APLAS 2024)*, pages 109–129. Springer LNCS 15194, 2024.
27. M. Hanus. Inferring non-failure conditions for declarative programs. In *Proc. of the 17th International Symposium on Functional and Logic Programming (FLOPS 2024)*, pages 167–187. Springer LNCS 14659, 2024.
28. M. Hanus. CurryInfo: Managing analysis and verification information about Curry packages. In *Proceedings of the 35th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2025)*, pages 123–134. Springer LNCS 16117, 2025.
29. M. Hanus and F. Skrlac. A modular and generic analysis server system for functional logic programs. In *Proc. of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation (PEPM'14)*, pages 181–188. ACM Press, 2014.
30. M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.9.0). Available at `http://www.curry-lang.org`, 2016.
31. R. Jhala, R. Majumdar, and A. Rybalchenko. HMC: verifying functional programs using abstract interpreters. In *23rd International Conference on Computer Aided Verification (CAV 2011)*, pages 470–485. Springer LNCS 6806, 2011.
32. T. Lindahl and K. Sagonas. Practical type inference based on success typings. In *Proceedings of the 8th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP 2006)*, pages 167–178. ACM Press, 2006.
33. F.J. López-Fraguas and J. Sánchez-Hernández. A proof theoretic approach to failure in functional logic programming. *Theory and Practice of Logic Programming*, 4(1):41–74, 2004.

34. W. Lux. Implementing encapsulated search for a lazy functional logic language. In *Proc. 4th Fuji International Symposium on Functional and Logic Programming (FLOPS'99)*, pages 100–113. Springer LNCS 1722, 1999.
35. B. Meyer. Ending null pointer crashes. *Communications of the ACM*, 60(5):8–9, 2017.
36. R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.
37. N. Mitchell and C. Runciman. A static checker for safe pattern matching in Haskell. In *Trends in Functional Programming*, volume 6, pages 15–30. Intellect, 2007.
38. N. Mitchell and C. Runciman. Not all patterns, but enough: an automatic verifier for partial but sufficient pattern matching. In *Proc. of the 1st ACM SIGPLAN Symposium on Haskell (Haskell 2008)*, pages 49–60. ACM, 2008.
39. U. Norell. Dependently typed programming in Agda. In *Proceedings of the 6th International School on Advanced Functional Programming (AFP'08)*, pages 230–266. Springer LNCS 5832, 2008.
40. S. Peyton Jones, editor. *Haskell 98 Language and Libraries—The Revised Report*. Cambridge University Press, 2003.
41. J.C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM Annual Conference*, pages 717–740. ACM Press, 1972.
42. P.M. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI'08)*, pages 159–169. ACM Press, 2008.
43. T. Sato and H. Tamaki. Enumeration of success patterns in logic programs. *Theoretical Computer Science*, 34:227–240, 1984.
44. A. Stump. *Verified Functional Programming in Agda*. ACM and Morgan & Claypool, 2016.
45. H. Unno and N. Kobayashi. Dependent type inference with interpolants. In *Proceedings of the 11th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'09)*, pages 277–288. ACM Press, 2009.
46. N. Vazou, E.L. Seidel, and R. Jhala. LiquidHaskell: Experience with refinement types in the real world. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*, pages 39–51. ACM Press, 2014.
47. N. Vazou, E.L. Seidel, R. Jhala, D. Vytiniotis, and S. Peyton Jones. Refinement types for Haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 269–282. ACM Press, 2014.