Automated Verification of Fail-Free Declarative Programs

– Extended Abstract –

Michael Hanus

Institut für Informatik, CAU Kiel, 24098 Kiel, Germany
mh@informatik.uni-kiel.de

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

1 Introduction

The occurrence of failures during a program execution is painful but still frequent when developing software systems. The two 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 dollare mistake"[1]).

Although null pointer failures cannot occur in declarative programs, such programs might contain other typical programming errors, like failures due to

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

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
```

In a correct program, it must be ensured that `head` and `tail` are not evaluated on empty lists. If we are not sure about the data provided at run time, we can check the arguments of partial operations before the application. For instance, the following code snipped 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 `processCommand` with the input data:

```
readCommand = do
  putStr "Input a command:"
  s <- getLine
  let ws = words s
  case null ws of True  → readCommand
                  False → processCommand (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 work 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, with *dependent types*, as in Agda [8], Coq [1], or Idris [2], or *refinement types*, as in LiquidHaskell [10,11], one can express restrictions on arguments of operations. Since one has to prove that these restrictions hold during the construction of programs, the development of such programs becomes harder [9]. Another alternative, proposed in [4], 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 [3]. 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 [5].

The main idea of this work is to *infer* the non-fail conditions of operations. Since the inference of precise conditions is undecidable in general, we approximate them by *abstract types*, i.e., finite representations of sets of values. In particular, our methods performs the following steps:

1. We define a *call type* for each operation. If the actual arguments belong to the call type, the operation is reducible with some rule.
2. We define *in/out types* for each operation to approximate the input/output behavior of the operation.
3. For each call to an operation g occurring in a rule defining operation f, we check, by considering the call structure and in/out types, whether the call

type of g is satisfied. If this is not the case, the call type of f is refined and we repeat the checks with the refined call type.

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

To sketch an example application of our method, consider the example above. Since in most cases declarative programs are defined by case distinctions on data constructors, we use as abstract types the set of top-level data constructors, where $\top$ denotes the set of all constructors. This domain is finite and can be order by set inclusion. For instance, the abstract type $\{$`:`$\}$ denotes all terms having the list constructor "`:`" at the top, i.e., all non-empty lists. Therefore, the call types of the operations `head` and `tail` can be characterized by the abstract type $\{$`:`$\}$. This call type is easy to derive from the left-hand sides of the rules defining `head` and `tail`. The call type states that, if the argument to `head` and `tail` is a non-empty list, the application is reducible.

Next we approximate the input/output behavior of operations by in/out types. These are basically disjunctions of abstract types for the input and the associated output. For instance, the operation `null` is defined by

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

so that the in/out type of `null` is

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

(where the disjunction is represented as a set). The in/out types can also be computed from the structure of the program with a fixpoint computation for recursive operations.

Now we want to verify the non-failure of `readCommand`. Since its definition contains calls to the partially defined operations `head` and `tail`, we have to show that the call types of these operations are satisfied at their call sites. This can be deduced by analyzing the case expression

```
case null ws of True  → readCommand
                False → processCommand (head ws) (tail ws)
```

In the branch containing the calls to `head` and `tail`, we know that the result of `null ws` is `False`. From the in/out type of `null`, we can infer that this is only the case of the argument `ws` is a non-empty list. Thus, the call types of `head` and `tail` are satisfied by the argument `ws` at the call site.

In order to make our approach accessible to various declarative languages, we formulate and implement it in the declarative multi-paradigm language Curry [7]. Since Curry extends Haskell by logic programming features and there are also methods to transform logic programs into Curry programs [6], 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, i.e., 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.

References

1. 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.
2. E. Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23(5):552–593, 2013.
3. 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.
4. 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.
5. M. Hanus. Combining static and dynamic contract checking for Curry. *Fundamenta Informaticae*, 173(4):285–314, 2020.
6. M. Hanus. From logic to functional logic programs. *Theory and Practice of Logic Programming*, 22(4):538–554, 2022.
7. M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.9.0). Available at http://www.curry-lang.org, 2016.
8. 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.
9. A. Stump. *Verified Functional Programming in Agda*. ACM and Morgan & Claypool, 2016.
10. 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.
11. 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.