Fundamenta Informaticae XX (2019) 1–30
DOI 10.3233/FI-2016-0000
IOS Press

Combining Static and Dynamic Contract Checking for Curry

Michael Hanus[*]
Institut für Informatik
CAU Kiel, Germany

Abstract. Static type systems are usually not sufficient to express all requirements on function calls. Hence, contracts with pre- and postconditions can be used to express more complex constraints on operations. Contracts can be checked at run time to ensure that operations are only invoked with reasonable arguments and return intended results. Although such dynamic contract checking provides more reliable program execution, it requires execution time and could lead to program crashes that might be detected with more advanced methods at compile time. To improve this situation for declarative languages, we present an approach to combine static and dynamic contract checking for the functional logic language Curry. Based on a formal model of contract checking for functional logic programming, we propose an automatic method to verify contracts at compile time. If a contract is successfully verified, it can be omitted from dynamic checking. This method decreases execution time without degrading reliable program execution. In the best case, when all contracts are statically verified, it provides trust in the software since crashes due to contract violations cannot occur during program execution.

Keywords: Declarative programming, contracts, verification

1. Introduction

Static types, provided by the programmer or inferred by the compiler, are useful to detect specific classes of run-time errors at compile time. This is expressed by Milner [1] as “well-typed expressions

Address for correspondence: M. Hanus, Institut für Informatik, CAU Kiel, D-24098 Kiel, Germany
[*]The research described in this paper has been partially supported by the German Federal Ministry of Education and Research (BMBF) under Grant No. 01IH15006B.

do not go wrong." However, not all requirements on operations can be expressed by standard static type systems. Hence, one can either refine the type system, e.g., use a dependently typed programming language and a more sophisticated programming discipline [2], or add contracts with pre- and postconditions to operations. Stronger type systems are quite expressive, but programming in a language with dependent types can be challenging since one has to construct proofs expressed as types before running the program. The use of contracts does not provide strong compile-time guarantees, but contracts are easier to use since they can be automatically checked at run time. Hence, one can write programs without additional efforts to prove the correctness of contracts.

Since contracts can easily be added to any programming language and do not require changes in the traditional way of software development, we consider them in this paper. As a motivating example, consider the well-known factorial function:

```
fac n = if n==0 then 1
                else n * fac (n-1)
```

Although `fac` is intended to work on non-negative natural numbers, standard static type systems cannot express this constraint so that

```
fac :: Int → Int
```

is provided or inferred as the static type of `fac`.[1] Although this type avoids the application of `fac` on characters or strings, it allows to apply `fac` on negative numbers which results in an infinite loop.

A *precondition* is a Boolean expression that restricts the applicability of an operation. Following the notation proposed in [3], a precondition for an operation f is a Boolean operation with name f`'pre`. For instance, a precondition for `fac` is

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

To use this precondition for checking invocations of `fac` at run time, a preprocessor could transform each call to `fac` by attaching an additional test whether the precondition is satisfied (see [3]). After this transformation, an application to `fac` to a negative number results in a run-time error (contract violation) instead of an infinite loop.

Unfortunately, run-time contract checking requires additional execution time so that it is often turned off, in particular, in production systems. To improve this situation for declarative languages, we propose to reduce the number of contract checks by (automatically) verifying them at compile time. Since we do not expect to verify all of them at compile time, our approach can be seen as a compromise between full static verification, e.g., with proof assistants like Agda, Coq, or Isabelle, which is time-consuming and difficult, and full dynamic checking, which might be inefficient.

For instance, one can verify (e.g., with an SMT solver [4]) that the precondition for the recursive call of `fac` is always satisfied provided that `fac` is called with a satisfied precondition. Hence, we can omit the precondition checking for recursive calls so that $n-1$ precondition checks are avoided when we evaluate `fac` n.

In the following, we make this idea more precise for the functional logic language Curry [5],

[1]The inferred type depends on the underlying static type system. For instance, Haskell (and also implementations of Curry supporting type classes) infers the more general overloaded type `Num a => a → a`.

briefly reviewed in the next section, so that the same ideas can also be applied to purely functional as well as logic languages. After discussing contracts for Curry in Sect. 3, we define a formal model of contract checking for Curry in Sect. 4. This is the basis to extract proof obligations for contracts at compile time. If these proof obligations can be verified, the corresponding dynamic checks can be omitted. Some examples for contract verification are shown in Sect. 5. The current implementation is sketched in Sect. 6 and benchmark results are presented in Sect. 7. Section 8 discusses other areas where the results of static contract checking can be applied. Finally, we discuss in Sect. 9 related work before we conclude.

2. Functional Logic Programming and Curry

Functional logic languages combine the most important features of functional and logic programming in a single language (see [6] for a recent survey). In particular, the language Curry [5] conceptually extends Haskell with common features of logic programming, i.e., non-determinism, free variables, and constraint solving. Since we discuss our methods in the context of functional logic programming, we briefly review those elements of functional logic languages, such as Curry, that are necessary to understand the contents of this paper. More details can be found in surveys on functional logic programming [6] and in the language report [5].

The syntax of Curry is close to that of Haskell [7]. In addition to Haskell, Curry applies rules with overlapping left-hand sides in a (don't know) non-deterministic manner (where Haskell always selects the first matching rule) and allows *free* (*logic*) *variables* in conditions and right-hand sides of rules. These variables must be explicitly declared (e.g., by `let...free`) unless they are anonymous. Function calls can contain free variables, in particular, variables without a value at call time. These calls are evaluated lazily where free variables as demanded arguments are non-deterministically instantiated [8].

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

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

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

Note that `ins` is a *non-deterministic operation* since it might deliver more than one result for a given argument, e.g., the evaluation of `ins 0 [1,2]` yields the values `[0,1,2]`, `[1,0,2]`, and `[1,2,0]`. Non-deterministic operations, which are interpreted as mappings from values into sets of values [9], are an important feature of contemporary functional logic languages. Hence, there is also a predefined *choice* operation:

```
x ? _  =  x
_ ? y  =  y
```

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

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

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

Non-deterministic operations are quite expressive since they can be used to completely eliminate logic variables in functional logic programs. For instance, consider the definition of the Boolean conjunction operator "`&&`" and the operator `ifThen` to represent conditional expressions:[2]

```
(&&) :: Bool → Bool → Bool           ifThen :: Bool → a → a
True  && x      = x                  ifThen True x = x
False && _      = False
```

Exploiting these definitions, the expression

```
let x free in ifThen (x && x) x
```

evaluates to all values of the logic (free) variable `x` such that the expression "`x && x`" evaluates to `True`. Traditionally, this is done by narrowing [8]. As an alternative, one can replace the Boolean logic variable by a non-deterministic *generator* operation for Booleans defined by

```
aBool = False ? True
```

so that the expression above can be transformed into

```
let x = aBool in ifThen (x && x) x
```

The equivalence of logic variables and non-deterministic value generators [11, 12] can be exploited when Curry is implemented by translation into a target language without support for non-determinism and logic variables. For instance, KiCS2 [13] compiles Curry into Haskell by adding a mechanism to handle non-deterministic computations. In our case, we exploit this fact by simply ignoring logic variables since they are considered as syntactic sugar for non-deterministic value generators.

Curry has many additional features not described here, like monadic I/O [14] for declarative input/output, set functions [15] to encapsulate non-deterministic search, functional patterns [16] and default rules [17] to specify complex transformations in a high-level manner, and a hierarchical module system together with a package manager[3] that provides access to currently more than one hundred packages with several hundred modules.

Due to the complexity of the source language, compilers or analysis and optimization tools often use an intermediate language where the syntactic sugar of the source language has been eliminated and the pattern matching strategy is explicit. This intermediate language, called FlatCurry, has also

[2]`ifThen` can be used to transform conditional into unconditional rules [10] so that we consider only unconditional rules in our intermediate language presented in Fig. 1.

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

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

Figure 1. Syntax of the intermediate language FlatCurry

been used to specify the operational semantics of Curry programs [18]. Since we will use FlatCurry as the basis for verifying contracts, we sketch the structure of FlatCurry and its semantics.

The abstract syntax of FlatCurry is summarized in Fig. 1. In contrast to some other presentations (e.g., [18, 6]), we omit the difference between rigid and flexible case expressions since we do not consider residuation (which becomes less important in practice and is also omitted in newer implementations of Curry [13]). A FlatCurry program consists of a sequence of function definitions, where each function is defined by a single rule. Patterns in source programs are compiled into case expressions and overlapping rules are joined by explicit disjunctions. For instance, the non-deterministic insert operation `ins` is represented in FlatCurry as

$$\texttt{ins}(x, xs) \;=\; (x : xs)\ \mathit{or}\ (\mathit{case}\ xs\ \mathit{of}\ \{y : ys \;\to\; y : \texttt{ins}(x, ys)\}$$

The semantics of FlatCurry programs is defined in [18] as an extension of Launchbury's natural semantics for lazy evaluation [19]. For this purpose, we consider only *normalized* FlatCurry programs, i.e., programs where the arguments of constructor and function calls and the discriminating argument of case expressions are always variables. Any FlatCurry program can be normalized by introducing new variables by let expressions [18]. For instance, the expression "$y : \texttt{ins}(x, ys)$" is normalized into "$\mathit{let}\ \{z = \texttt{ins}(x, ys)\}\ \mathit{in}\ y : z$." In the following, we assume that all FlatCurry programs are normalized.

In order to model sharing, which is important for lazy evaluation and also semantically relevant in case of non-deterministic operations [9], variables are interpreted as references into a heap where new let bindings are stored and function calls are updated with their evaluated results. To be more precise, a *heap*, denoted by Γ, Δ, or Θ, is a partial mapping from variables to expressions. The *domain* of a heap Γ is the set of variables bound in the heap, i.e., $\mathcal{D}om(\Gamma) = \{x \mid \Gamma(x) \text{ is defined}\}$. The *empty heap* is denoted by $[]$. $\Gamma[x \mapsto e]$ denotes a heap Γ' with $\Gamma'(x) = e$ and $\Gamma'(y) = \Gamma(y)$ for all $y \in \mathcal{D}om(\Gamma)$ with $x \neq y$.

Using heap structures, one can provide a high-level description of the operational behavior of FlatCurry programs in natural semantics style. The semantics uses judgements of the form "$\Gamma : e \Downarrow \Delta : v$" with the meaning that in the context of heap Γ the expression e evaluates to value (head

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

Figure 2. Natural semantics of normalized FlatCurry programs

normal form) v and produces a modified heap Δ. Figure 2 shows the rules defining this semantics w.r.t. a given normalized FlatCurry program P, i.e., a set of function definitions according to Fig. 1. We use the notation $\overline{o_k}$ to refer to a sequence of objects $o_1, \ldots, o_k$.

Constructor-rooted expressions (i.e., head normal forms) are just returned by rule Val. Rule VarExp retrieves a binding for a variable from the heap and evaluates it. In order to avoid the re-evaluation of the same expression, VarExp updates the heap with the computed value, which models sharing. In contrast to the original rules [18], VarExp removes the binding from the heap. On the one hand, this allows the detection of simple loops ("black holes") as in functional programming. On the other hand, it is crucial in combination with non-determinism to avoid the binding of a variable to different values in the same derivation (see [20] for a detailed discussion on this issue). Rule Fun unfolds function calls by evaluating the right-hand side after binding the formal parameters to the actual ones via the renaming substitution ρ. Let introduces new bindings in the heap and renames the variables in the expressions with the fresh names introduced in the heap. Or non-deterministically evaluates one of its arguments. Finally, rule Select deals with *case* expressions. When the discriminating argument of *case* evaluates to a constructor-rooted term, Select evaluates the corresponding branch of the *case* expression.

The FlatCurry representation of Curry programs and its operational semantics has been used for various language-oriented tools, like compilers, partial evaluators, or debugging and profiling tools (see [6] for references). We use it in this paper to define a formal model of contract checking and extract proof obligations for contracts from programs.

3. Contracts

The use of contracts in declarative programming languages has been motivated in Sect. 1. Contracts in the form of pre- and postconditions as well as specifications have been introduced into functional logic programming in [3]. Contracts and specifications for some operation are operations with the same name and a specific suffix. If f is an operation of type $\tau \to \tau'$, then a *specification* for f is an operation f`'spec` of type $\tau \to \tau'$, a *precondition* for f is an operation f`'pre` of type $\tau \to$ `Bool`, and a *postcondition* for f is an operation f`'post` of type $\tau \to \tau' \to$ `Bool`.

Intuitively, an operation and its specification should be equivalent operations. For instance, a specification of non-deterministic list insertion could be stated with a single rule containing a functional pattern [16] as follows:

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

A precondition should be satisfied if an operation is invoked, and a postcondition is a relation between input and output values which should be satisfied when an operation yields some result. We have already seen a precondition for the factorial function in Sect. 1. A postcondition for the same operation could state that the result is always positive:

```
fac'post n f = f > 0
```

This postcondition ensures that the precondition of nested `fac` applications always holds, like in the expression `fac (fac 3)`. If an operation has no postcondition but a specification, the latter can be used as a postcondition. For instance, a postcondition derived from the specification for `ins` is

```
ins'post :: a  →  [a]  →  [a]  →  Bool
ins'post x ys zs = zs `valueOf` ins'specS x ys
```

This postcondition states that the value `zs` computed by `ins` is in the set of all values computed by `ins'spec` (where f_S denotes the set function of f, see [15]).

Antoy and Hanus [3] describe a tool that transforms a program containing contracts and specifications into a program where these contracts and specifications are dynamically checked. This tool is available as Curry package `dsdcurry` for various Curry implementations. It acts as a preprocessor so that the transformation can be automatically performed when Curry programs are compiled. Furthermore, the property-based testing tool CurryCheck [21] automatically tests contracts and specifications with generated input data.

Although these dynamic and static testing tools provide some confidence in the software under development, static *verification* of contracts is preferable since it holds for all input values, i.e., it is ensured that violations of verified contracts cannot occur at run time so that their run-time tests can be omitted. As a first step towards this objective, we specify the operational meaning of contract checking by extending the semantics of Fig. 2. Since pre- and postconditions are checked before and after a function invocation, respectively, it is sufficient to extend rule Fun. Assume that function f has a precondition f`'pre` and a postcondition f`'post` (if one of them is not present, we assume that they are defined as predicates that always return `True`). Then we replace rule Fun by the extended rule

FunCheck:

$$\frac{\Gamma : f\texttt{'pre}(\overline{x_n}) \Downarrow \Gamma' : \texttt{True} \quad \Gamma' : \rho(e) \Downarrow \Delta' : v \quad \Delta' : f\texttt{'post}(\overline{x_n}, v) \Downarrow \Delta : \texttt{True}}{\Gamma : f(\overline{x_n}) \Downarrow \Delta : v}$$

where $f(\overline{y_n}) = e \in P$ and $\rho = \{\overline{y_n \mapsto x_n}\}$. For the sake of readability, we omit the normalization of the postcondition in the premise, which can be added by an introduction of a let binding for v. The reporting of contract violations can be specified by the following rules:

$$\frac{\Gamma : f\texttt{'pre}(\overline{x_n}) \Downarrow \Gamma' : \texttt{False}}{\Gamma : f(\overline{x_n}) \Downarrow \texttt{"Error: precondition of } f \texttt{ violated"}}$$

$$\frac{\Gamma : f\texttt{'pre}(\overline{x_n}) \Downarrow \Gamma' : \texttt{True} \quad \Gamma' : \rho(e) \Downarrow \Delta' : v \quad \Delta' : f\texttt{'post}(\overline{x_n}, v) \Downarrow \Delta : \texttt{False}}{\Gamma : f(\overline{x_n}) \Downarrow \texttt{"Error: postcondition of } f \texttt{ violated"}}$$

These rules are intended to specify when an error is reported but not used as part of a normal evaluation. This means that if $\Gamma : e \Downarrow \Delta : v$ is a valid judgement (where rule FunCheck is used instead of rule Fun), all pre- and postconditions are satisfied during the evaluation of e to v.

Note that rule FunCheck specifies *eager* contract checking, i.e., pre- and postconditions are immediately and completely evaluated. Although this is often intended, there are cases where eager contract checking might influence the execution behavior of a program, e.g., if the evaluation of a pre- or postcondition requires to evaluate more than demanded by the original program. To avoid this problem, Chitil et al. [22] proposed *lazy* contract checking where contract arguments are not evaluated but the checks are performed when the demanded arguments become evaluated by the application program. Lazy contract checking could have the problem that the occurrence of contract violations depend on the demand of evaluation so that they are detected "too late." Since there seems to be no ideal solution to this problem [23] and lazy contract checking is operationally more complex, we simply stick to eager contract checking.

4. Contract Verification

In order to statically verify contracts, we have to extract some proof obligation from the program and contracts. For instance, consider the factorial function and its precondition, as shown in Sect. 1. The normalized FlatCurry representation of the factorial function is

```
fac(n) = let { x = 0 ; y = n==x }
         in case y of True   → 1
                      False  → let { n1 = n - 1 ; f = fac(n1) }
                               in n * f
```

Now consider the call fac(n). Since we assume that the precondition holds when an operation is invoked, we know that $n \geq 0$ holds before the case expression is evaluated. If the False branch of the case expression is selected, we know that $n = 0$ has the value False. Altogether, we know that

$$n \geq 0 \wedge \neg(n = 0)$$

Val $\Gamma : C \mid z \leftarrow v \Downarrow C \wedge z = v$ where v is constructor-rooted or v is a variable not bound in Γ

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

Fun $\Gamma : C \mid z \leftarrow f(\overline{x_n}) \Downarrow C \wedge f\texttt{'pre}(\overline{x_n}) \wedge f\texttt{'post}(\overline{x_n}, z)$

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

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

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

Figure 3. Assertion-collecting semantics

holds when the right-hand side of the `False` branch is evaluated. Since this implies that $n > 0$ and, thus, $(n-1) \geq 0$ holds (in integer arithmetic), we know that the precondition of the recursive call to `fac` always holds. Hence, its check can be omitted at run time.

This example shows that we have to collect properties that are ensured to be valid when we reach particular points in the rules' right-hand sides. For this purpose, we define an *assertion-collecting semantics*. It is oriented towards the concrete semantics shown before but has the following differences:

1. We compute with symbolic values instead of concrete ones.

2. We collect properties that are known to be valid (also called *assertions* in the following).

3. Instead of evaluating functions, we collect their pre- and postconditions.

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

Figure 3 shows the rules defining the assertion-collecting semantics. In order to emphasize the relation between the concrete semantics (Fig. 2) and this assertion-collecting semantics, we use the same names but different fonts for the inference rules. Rule *Val* immediately returns the collected assertions. Since this semantics is intended to compute with symbolic values, there might be variables without a binding to a concrete value. Hence, *Val* also returns such unbound variables. Rule

VarExp behaves similarly to rule VarExp of the concrete semantics and returns the assertions collected during the evaluation of the expression. Note that the assertion-collecting semantics does not really evaluate expressions since it should always return the collected assertions in a finite amount of time. For the same reason, rule *Fun* does not invoke the function in order to evaluate its right-hand side. Instead, the pre- and postcondition information is added to the collected assertions since they must hold if the function returns some value. The notation $f\texttt{'pre}(\overline{x_n})$ and $f\texttt{'post}(\overline{x_n}, z)$ in the assertion means that the logical formulas corresponding to the pre- and postcondition are added as an assertion. These formulas might be simplified by replacing occurrences of operations defined in the program by their definitions. Rule *Let* adds the let bindings to the heap, similarly to the concrete semantics, before evaluating the argument expression. Rules *Or* and *Select* collect all information derived from alternative computations, instead of the non-deterministic concrete semantics. Rule *Select* also collects inside each branch the condition that must hold in the selected branch, which is important to get precise proof obligations. To avoid the renaming of local variables in different branches, we implicitly assume that all local variables are unique in a normalized function definition.

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

Proposition 4.1. (Uniqueness)
Let Γ be a heap, C an assertion, z a variable, and e an expression. Then there is a unique (up to variable renamings in let bindings) proof tree and assertion D so that the judgement

$$\Gamma : C \mid z \leftarrow e \Downarrow D$$

is derivable.

Proof:
First of all, note that for any heap Γ, assertion C, variable z, and expression e, there is one and only one (up to the names of fresh variables chosen in rule *Let*) applicable inference rule for a judgement of the form $\Gamma : C \mid z \leftarrow e \Downarrow D$ for some assertion D. Thus, in order to prove the proposition, it is sufficient to show that the premises are always smaller than the conclusion w.r.t. some size measure. For this purpose, we define the size of an expression as the number of all symbols occurring in it. The size of a heap is the sum of the sizes of all bound expressions. The size of a judgement $\Gamma : C \mid z \leftarrow e \Downarrow D$ is the sum of the size of the heap Γ and the size of e. We show that the premises have smaller sizes than the conclusion by a case distinction on the rules having premises:

- Rule *VarExp*: Since the bound expression e is removed from the heap and the variable x is replaced by e, the size of the premise is decreased by one symbol.
- Rule *Let*: Since ρ is a variable renaming, the sum of the sizes of the heap and expressions in the premise is smaller than the size of the conclusion.
- Rules *Or* and *Select*: In each premise, the size of the expression is decreased and the heap in the premises remains identical.

□

The assertion-collecting semantics allows to extract proof obligations to verify contracts. For instance, to verify that a postcondition f'`post` for some function f defined by $f(\overline{x_n}) = e$ holds, one derives a judgement (where z is a new variable)

$$[] : f\texttt{'pre}(\overline{x_n}) \mid z \leftarrow e \Downarrow C$$

and proves that C implies $f\texttt{'post}(\overline{x_n}, z)$.

As an example, consider the non-deterministic operation

```
coin = 1 or 2
```

and its postcondition

```
coin'post z = z > 0
```

(the precondition is simply `True`). We construct for the right-hand side of `coin` the following proof tree:

$$\mathit{Or}\ \dfrac{\mathit{Val}\ \dfrac{}{[] : \mathit{true} \mid z \leftarrow 1 \Downarrow z = 1} \qquad \dfrac{}{[] : \mathit{true} \mid z \leftarrow 2 \Downarrow z = 2}\ \mathit{Val}}{[] : \mathit{true} \mid z \leftarrow 1\ \mathit{or}\ 2 \Downarrow z = 1 \vee z = 2}$$

Since $z = 1 \vee z = 2$ implies $z > 0$, the postcondition of `coin` is always satisfied.

If we construct the proof tree for the right-hand side e of the factorial function, we derive the following judgement:

$$[] : n \geq 0 \mid z \leftarrow e \Downarrow (n \geq 0 \wedge y = \mathit{true} \wedge z = 1) \vee (n \geq 0 \wedge y = \mathit{false})$$

Since there is no condition on the result variable z in the right part of the disjunction, this assertion does not imply the postcondition $z > 0$. The reason is that the recursive call to `fac` is not considered in the proof tree since it does not occur at the top level. Note that rule *Fun* only adds the contract information of top-level operations but no contracts of operations occurring in arguments. Due to the lazy evaluation strategy, one does not know at compile time whether some argument expression is evaluated. Hence, it would not be correct to add the contract information of nested arguments. For instance, consider the operations

```
const x y = y

f x | x > 0 = 0
f'post x z = x > 0

g x = const (f x) 42
```

If e denotes the right-hand side of `g` (in normalized FlatCurry form), then we can derive with the inference rules of Fig. 3 the judgement

$$[] : \mathit{true} \mid z \leftarrow e \Downarrow \mathit{true}$$

If we change rule *Fun* so that the contracts of argument calls are also added to the returned assertion, then we could derive

$$[] : \mathit{true} \mid z \leftarrow e \Downarrow x > 0$$

This postcondition is clearly wrong since `(g 0)` successfully evaluates to `42`.

Nevertheless, we can improve our semantics in cases where it is ensured that arguments are evaluated. For instance, primitive operations, like `+`, `*`, or `==`, evaluate their arguments before the operation is applied. This can be specified in the concrete semantics of Fig. 2 by adding the following inference rule (where the left occurrence of the primitive operation $\oplus$ in the conclusion denotes the syntax of the primitive operation, whereas the right occurrence of the same symbol denotes the semantics, i.e., the mathematical function denoted by this symbol):

$$\mathsf{PrimOp}\quad \frac{\Gamma : x \Downarrow \Delta : v_x \qquad \Delta : y \Downarrow \Theta : v_y}{\Gamma :\ x \,\oplus\, y \Downarrow \Theta : v_x \,\oplus\, v_y} \quad \text{where } \oplus \in \{\texttt{==}, \texttt{+}, \texttt{-}, \texttt{*}, \ldots\}$$

In order to collect appropriate assertions for primitive operations, we add the following rule to the assertion-collecting semantics (and restrict rule *Fun* to exclude these operations):

$$\textit{PrimOp}\quad \frac{\Gamma : C \mid x \leftarrow x \Downarrow D \quad \Gamma : D \mid y \leftarrow y \Downarrow E}{\Gamma : C \mid z \leftarrow x \,\oplus\, y \Downarrow E \wedge z = x \,\oplus\, y} \quad \text{where } \oplus \in \{\texttt{==}, \texttt{+}, \texttt{-}, \texttt{*}, \ldots\}$$

Since primitive operations are often known to the underlying verifier, we collect the information about the call of the primitive operations. In a similar way, one can also improve user-defined functions if some argument is known to be demanded, a property which can be approximated at compile time by a demand analysis [24].

If we construct a proof tree for the factorial function with these refined inference rules (see Fig. 4), we obtain the following (simplified) assertion:

$$(n \geq 0 \wedge n = 0 \wedge z = 1) \vee (n \geq 0 \wedge n \neq 0 \wedge n1 \geq 0 \wedge f > 0 \wedge z = n * f)$$

Since this assertion implies $z > 0$, the postcondition `fac'post` holds so that its checking can be omitted at run time.

Proof obligations for preconditions can also be extracted from the proof tree. For this purpose, one has to consider occurrences of operations with non-trivial preconditions. If such an operation occurs as a top-level expression or in a let binding associated to a top-level expression and the assertion before this expression implies the precondition, then one can omit the precondition checking for this call. For instance, consider again the proof tree for the right-hand side of the factorial function which contains the following (simplified) judgement:

$$[] : n \geq 0 \wedge n \neq 0 \mid z \leftarrow \textit{let}\ \{n1 = n - 1; f = \textit{fac}\, n1\}\ \textit{in}\ n * f \Downarrow \ldots$$

Since $n \geq 0 \wedge n \neq 0 \wedge n1 = n - 1$ implies $n1 \geq 0$, the precondition holds so that its check can be omitted for this recursive call.

The correctness of our approach relies on the following theorem stating a relation between the concrete and the assertion-collecting semantics. In this claim, $\widehat{\Gamma}$ denotes the representation of heap information as a logic formula, i.e.,

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

[Subtree $Case_1$]

$$\textit{Val}\;\dfrac{}{[\,] : n \geq 0 \wedge y = (n = 0) \wedge y = true \mid z \leftarrow 1 \Downarrow n \geq 0 \wedge n = 0 \wedge z = 1}$$

[Subtree $Case_2$]

$$\textit{Let}\;\dfrac{\textit{PrimOp}\;\dfrac{\textit{Val}\;\dfrac{}{\Gamma_1 : n \geq 0 \wedge n \neq 0 \mid n \leftarrow n \Downarrow n \geq 0 \wedge n \neq 0} \qquad \dfrac{\dfrac{}{\Gamma_2 : n \geq 0 \wedge n \neq 0 \mid f \leftarrow fac\, n1 \Downarrow n \geq 0 \wedge n \neq 0 \wedge n1 \geq 0 \wedge f > 0}\;\textit{Fun}}{\Gamma_1 : n \geq 0 \wedge n \neq 0 \mid f \leftarrow f \Downarrow n \geq 0 \wedge n \neq 0 \wedge n1 \geq 0 \wedge f > 0}\;\textit{VarExp}}{\Gamma_1 : n \geq 0 \wedge n \neq 0 \mid z \leftarrow n * f \Downarrow n \geq 0 \wedge n \neq 0 \wedge n1 \geq 0 \wedge f > 0 \wedge z = n * f}}{[\,] : n \geq 0 \wedge y = (n = 0) \wedge y = false \mid z \leftarrow let\ \{n1 = n - 1; f = fac\, n1\}\ in\ n * f \Downarrow n \geq 0 \wedge n \neq 0 \wedge n1 \geq 0 \wedge f > 0 \wedge z = n * f}$$

where $\Gamma_1 = [n1 \mapsto n - 1, f \mapsto fac\, n1]$ and $\Gamma_2 = [n1 \mapsto n - 1]$

$$\textit{Select}\;\dfrac{\textit{PrimOp}\;\dfrac{\ldots}{[\,] : n \geq 0 \mid y \leftarrow n\texttt{==}0 \Downarrow n \geq 0 \wedge y = (n = 0)} \qquad \text{[Subtree } Case_1\text{]} \qquad \text{[Subtree } Case_2\text{]}}{[\,] : n \geq 0 \mid z \leftarrow case\ n\texttt{==}0\ of \ldots \Downarrow (n \geq 0 \wedge n = 0 \wedge z = 1) \vee (n \geq 0 \wedge n \neq 0 \wedge n1 \geq 0 \wedge f > 0 \wedge z = n * f)}$$

Figure 4. Proof tree for the definition of `fac` (where assertions are slightly simplified)

Theorem 4.2. (Correctness)
Let $\Gamma : e \Downarrow \Delta : v$ be a valid judgement, z a variable, and C an assertion such that $\widehat{\Gamma} \Rightarrow C$ is valid. Then there is a valid judgement $\Gamma : C \mid z \leftarrow e \Downarrow D$ with $(\widehat{\Delta} \wedge z = v) \Rightarrow D$.

In this theorem and the subsequent propositions, we assume a *semantics with contract checking* which consists of the rules shown in Fig. 2 where rule Fun is replaced by rule FunCheck and the additional rule PrimOp is used to evaluate primitive operations.

Theorem 4.2 can be applied as follows. If some function f is defined by rule $f(\overline{x_n}) = e$, the judgement

$$[] : f\texttt{'pre}(\overline{x_n}) \mid z \leftarrow e \Downarrow D$$

is valid, and D implies $f\texttt{'post}(\overline{x_n}, z)$, we know that the postcondition holds for any call to f so that we can remove the postcondition check for f from the program code. Note that this relates to the partial correctness of postconditions. If the actual evaluation does not terminate, as for

```
loop = loop
```

any postcondition for `loop` can be verified (which is fine since removing the postcondition checking code from `loop` does not change the result).

In order to prove this theorem, we need a few lemmas. The first lemma shows that assertions that are valid w.r.t. an initial heap are also valid w.r.t. the result heap of a computation.

Lemma 4.3. Let Γ be a heap and C an assertion such that $\widehat{\Gamma} \Rightarrow C$. If $\Gamma : e \Downarrow \Delta : v$ is a valid judgement, then $\widehat{\Delta} \Rightarrow C$.

Proof:
We prove by induction on the height of the proof tree for the judgement $\Gamma : e \Downarrow \Delta : v$ (w.r.t. the natural semantics with contract checking) that $\widehat{\Gamma} \Rightarrow C$ implies $\widehat{\Delta} \Rightarrow C$ for any assertion C.

Base case: Rule Val is applied so that $\Delta = \Gamma$ and the claim vacuously holds.

For the induction step, we consider the different kinds of inference rules used to derive the judgement $\Gamma : e \Downarrow \Delta : v$.

- Rule VarExp is applied so that

$$\frac{\Gamma' : e \Downarrow \Delta' : v}{\Gamma'[x \mapsto e] : x \Downarrow \Delta'[x \mapsto v] : v}$$

  where $\Gamma = \Gamma'[x \mapsto e]$ and $\Delta = \Delta'[x \mapsto v]$.

  We distinguish the kind of expression e bound to x:

  1. e is constructor-rooted: Then $\widehat{\Gamma} = \widehat{\Gamma'} \wedge x = e$. Since e is constructor-rooted, rule Val is applied to the premise $\Gamma' : e \Downarrow \Delta' : v$ so that $\Delta' = \Gamma'$ and $v = e$. Hence

$$\widehat{\Delta} \;=\; \widehat{\Delta'[x \mapsto v]} \;=\; \widehat{\Delta'} \wedge x = v \;=\; \widehat{\Gamma'} \wedge x = e \;=\; \widehat{\Gamma'[x \mapsto e]} \;=\; \widehat{\Gamma}$$

     and $\widehat{\Delta} \Rightarrow C$ follows from our assumption.

2. e is some variable y: Then $\widehat{\Gamma} = \widehat{\Gamma'} \wedge x = y$ and $\widehat{\Gamma'} \wedge x = y \Rightarrow C$. We define C' by replacing all occurrences of x by y in C. Then $\widehat{\Gamma'} \Rightarrow C'$. By the induction hypothesis, $\widehat{\Delta'} \Rightarrow C'$. Since e is the variable y, rule VarExp has been applied to the premise $\Gamma' : e \Downarrow \Delta' : v$ so that $\Delta'(y) = v$. Hence, $\widehat{\Delta'} \Rightarrow C'$ is equivalent to $\widehat{\Delta'} \wedge y = v \Rightarrow C'$. This implies $\widehat{\Delta'} \wedge x = v \wedge y = v \Rightarrow C'$. Due to the definition of C', $\widehat{\Delta'} \wedge x = v \wedge y = v \Rightarrow C$. Since x and y are both bound to v in Δ, $\widehat{\Delta} \Rightarrow C$.
3. e is operation-rooted: Then $\widehat{\Gamma} = \widehat{\Gamma'}$. By the induction hypothesis, $\widehat{\Delta'} \Rightarrow C$ which implies $\widehat{\Delta'} \wedge x = v \Rightarrow C$. This proves the claim since $\widehat{\Delta} = \widehat{\Delta'} \wedge x = v$.

- Rule FunCheck is applied:

$$\frac{\Gamma : f\texttt{'pre}(\overline{x_n}) \Downarrow \Gamma' : \texttt{True} \quad \Gamma' : \rho(e) \Downarrow \Delta' : v \quad \Delta' : f\texttt{'post}(\overline{x_n}, v) \Downarrow \Delta : \texttt{True}}{\Gamma : f(\overline{x_n}) \Downarrow \Delta : v}$$

where $f(\overline{y_n}) = e \in P$ and $\rho = \{\overline{y_n \mapsto x_n}\}$. Assume that $\widehat{\Gamma} \Rightarrow C$. By induction hypothesis applied to the first premise, $\widehat{\Gamma'} \Rightarrow C$ holds. This implies $\widehat{\Delta'} \Rightarrow C$ (by induction hypothesis on the second premise) and $\widehat{\Delta} \Rightarrow C$ (by induction hypothesis on the third premise).

- Rule Let is applied:

$$\frac{\Gamma[\overline{y_k \mapsto \rho(e_k)}] : \rho(e) \Downarrow \Delta : v}{\Gamma : let\ \{\overline{x_k = e_k}\}\ in\ e \Downarrow \Delta : v}$$

Assume that $\widehat{\Gamma} \Rightarrow C$ holds. Since $\overline{y_k}$ are fresh variables, also $\widehat{\Gamma[\overline{y_k \mapsto \rho(e_k)}]} \Rightarrow C$ holds. By induction hypothesis, $\widehat{\Delta} \Rightarrow C$ holds.

- Rule Or is applied:

$$\frac{\Gamma : e_i \Downarrow \Delta : v}{\Gamma : e_1\ or\ e_2 \Downarrow \Delta : v}$$

for $i \in \{1, 2\}$. Then the claim follows from the induction hypothesis.

- Rule Select is applied:

$$\frac{\Gamma : x \Downarrow \Theta : c(\overline{y_n}) \quad \Theta : \rho(e_i) \Downarrow \Delta : v}{\Gamma : case\ x\ of\ \{\overline{p_k \to e_k}\} \Downarrow \Delta : v}$$

where, for some i, $p_i = c(\overline{x_n})$ and $\rho = \{\overline{x_n \mapsto y_n}\}$. Assume that $\widehat{\Gamma} \Rightarrow C$ holds. The induction hypothesis implies $\widehat{\Theta} \Rightarrow C$. Again, we can apply the induction hypothesis to show that $\widehat{\Delta} \Rightarrow C$ holds.

- Rule PrimOp is applied:

$$\frac{\Gamma : x \Downarrow \Theta : x' \quad \Theta : y \Downarrow \Delta : y'}{\Gamma :\ x \oplus y \Downarrow \Delta : x' \oplus y'}$$

where $\oplus \in \{\texttt{==}, \texttt{+}, \texttt{-}, \texttt{*}, \ldots\}$ is some primitive operation. Assume that $\widehat{\Gamma} \Rightarrow C$ holds. The induction hypothesis implies $\widehat{\Theta} \Rightarrow C$. Again, we can apply the induction hypothesis to the right premise to show that $\widehat{\Delta} \Rightarrow C$ holds.

□

The next lemma shows that assertions collected with a result heap can also be collected with the initial heap of a computation.

Lemma 4.4. Let $\Gamma : e \Downarrow \Delta : v$ and $\Delta : C \mid z \leftarrow e' \Downarrow D'$ be valid judgements where e' does not contain fresh variables introduced by evaluating e, i.e., variables from $\mathcal{D}om(\Delta)\backslash\mathcal{D}om(\Gamma)$. Then there exists an assertion D and a valid judgement $\Gamma : C \mid z \leftarrow e' \Downarrow D$ with $D' \Rightarrow D$.

Proof:
By Prop. 4.1, there is a unique proof tree for the judgement $\Gamma : C \mid z \leftarrow e' \Downarrow D$ for an assertion D. The difference between Γ and Δ are (1) additional bindings (by rule *Let*) and (2) updated bindings for non-constructor-rooted expressions (by rule *VarExp*). In the assertion collecting semantics, the heap is used only in rule *VarExp* to look up variables which are further inspected and might lead to additional assertions. Since e' does not contain variables from $\mathcal{D}om(\Delta)\backslash\mathcal{D}om(\Gamma)$, the only difference between D' and D is that D' might contain additional equations for variables which are bound to operation-rooted expressions in Γ. Hence, $D' \Rightarrow D$. □

Now we return to the proof of the main theorem.

Proof:
[of Theorem 4.2] The proof is by induction on the height h of the proof tree w.r.t. the natural semantics with contract checking.

Base case ($h = 0$): Rule Val is applied, i.e., $e = v$ and v is constructor-rooted. By rule *Val*, $\Gamma : C \mid z \leftarrow v \Downarrow C \wedge z = v$ is a valid judgement. If $\widehat{\Gamma} \Rightarrow C$, then $\widehat{\Gamma} \wedge z = v \Rightarrow C \wedge z = v$ also holds which shows the claim.

For the induction step ($h > 0$), we consider the different kinds of inference rules used to derive the judgement $\Gamma : e \Downarrow \Gamma' : v$.

- Rule VarExp is applied:

$$\frac{\Gamma : e \Downarrow \Delta : v}{\Gamma[x \mapsto e] : x \Downarrow \Delta[x \mapsto v] : v}$$

  Assume that $\widehat{\Gamma[x \mapsto e]} \Rightarrow C$.

  1. e is constructor-rooted: Then, by rule Val, $\Delta = \Gamma$, $v = e$, and

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

     is a valid derivation. Since $\widehat{\Gamma[x \mapsto e]} \wedge z = e \Rightarrow C \wedge z = e$, the claim holds in this case.

  2. e is operation-rooted: Then $\widehat{\Gamma[x \mapsto e]} = \widehat{\Gamma}$ so that $\widehat{\Gamma} \Rightarrow C$. Since the height of the proof tree for $\Gamma : e \Downarrow \Delta : v$ is smaller than h, by induction hypothesis, $\Gamma : C \mid z \leftarrow e \Downarrow D$ is valid and $\widehat{\Delta} \wedge z = v \Rightarrow D$. Hence,

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

is a valid derivation. Since $\widehat{\Delta[x \mapsto v]} = \widehat{\Delta} \wedge x = v$ and $\widehat{\Delta} \wedge z = v \Rightarrow D$, we have $\widehat{\Delta[x \mapsto v]} \wedge z = v \Rightarrow D$ so that the claim also holds in this case.

- Rule FunCheck is applied:

$$\frac{\Gamma : f\texttt{'pre}(\overline{x_n}) \Downarrow \Gamma' : \texttt{True} \quad \Gamma' : \rho(e) \Downarrow \Delta' : v \quad \Delta' : f\texttt{'post}(\overline{x_n}, v) \Downarrow \Delta : \texttt{True}}{\Gamma : f(\overline{x_n}) \Downarrow \Delta : v}$$

where $f(\overline{y_n}) = e \in P$ and $\rho = \{\overline{y_n \mapsto x_n}\}$. Assume that $\widehat{\Gamma} \Rightarrow C$. Since $\Gamma : f\texttt{'pre}(\overline{x_n}) \Downarrow \Gamma' : \texttt{True}$, the precondition holds w.r.t. Γ', i.e., $\widehat{\Gamma'} \Rightarrow f\texttt{'pre}(\overline{x_n})$. By Lemma 4.3, $\widehat{\Gamma'} \Rightarrow C$ so that we have $\widehat{\Gamma'} \Rightarrow C \wedge f\texttt{'pre}(\overline{x_n})$. Similarly, we have

$$\widehat{\Delta} \Rightarrow C \wedge f\texttt{'pre}(\overline{x_n}) \wedge f\texttt{'post}(\overline{x_n}, v) \tag{1}$$

The application of rule *Fun* shows

$$\Gamma : C \mid z \leftarrow f(\overline{x_n}) \Downarrow C \wedge f\texttt{'pre}(\overline{x_n}) \wedge f\texttt{'post}(\overline{x_n}, z)$$

Thus, the claim holds since $\widehat{\Delta} \wedge z = v \Rightarrow C \wedge f\texttt{'pre}(\overline{x_n}) \wedge f\texttt{'post}(\overline{x_n}, v)$ is a consequence of (1).

- Rule Let is applied:

$$\frac{\Gamma[\overline{y_k \mapsto \rho(e_k)}] : \rho(e) \Downarrow \Delta : v}{\Gamma : \mathit{let}\ \{\overline{x_k = e_k}\}\ \mathit{in}\ e \Downarrow \Delta : v}$$

Assume that $\widehat{\Gamma} \Rightarrow C$ holds. Let $\Gamma' = \Gamma[\overline{y_k \mapsto \rho(e_k)}]$. Then $\widehat{\Gamma'} \Rightarrow C$ also holds. By induction hypothesis,

$$\Gamma[\overline{y_k \mapsto \rho(e_k)}] : C \mid z \leftarrow \rho(e) \Downarrow D$$

is valid and $\Delta \wedge z = v \Rightarrow D$. Then the application of rule *Let*

$$\frac{\Gamma[\overline{y_k \mapsto \rho(e_k)}] : C \mid z \leftarrow \rho(e) \Downarrow D}{\Gamma : C \mid z \leftarrow \mathit{let}\ \{\overline{x_k = e_k}\}\ \mathit{in}\ e \Downarrow D}$$

is a valid derivation step so that the claim holds.

- Rule Or is applied:

$$\frac{\Gamma : e_i \Downarrow \Delta : v}{\Gamma : e_1\ \mathit{or}\ e_2 \Downarrow \Delta : v}$$

for $i \in \{1, 2\}$. Assume that $\widehat{\Gamma} \Rightarrow C$ holds and $i = 1$ (the other case is symmetric). By induction hypothesis, $\Gamma : C \mid z \leftarrow e_1 \Downarrow D_1$ is valid and $\widehat{\Gamma} \wedge z = v \Rightarrow D_1$. By Prop.4.1, there exists some assertion D_2 and a derivation tree showing that $\Gamma : C \mid z \leftarrow e_2 \Downarrow D_2$ is valid. Thus,

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

is a valid derivation step. Since $\widehat{\Gamma} \wedge z = v \Rightarrow D_1$, $\widehat{\Gamma} \wedge z = v \Rightarrow D_1 \vee D_2$ which shows the claim.

- Rule Select is applied:

$$\frac{\Gamma : x \Downarrow \Delta : c(\overline{y_n}) \quad \Delta : \rho(e_i) \Downarrow \Theta : v}{\Gamma : \mathit{case}\ x\ \mathit{of}\ \{\overline{p_k \to e_k}\} \Downarrow \Theta : v}$$

  where, for some i, $p_i = c(\overline{x_n})$ and $\rho = \{\overline{x_n \mapsto y_n}\}$. Assume that $\widehat{\Gamma} \Rightarrow C$ holds and $i = 1$ (the other cases are symmetric). By induction hypothesis applied to $\Gamma : x \Downarrow \Delta : c(\overline{y_n})$, $\Gamma : C \mid x \leftarrow x \Downarrow D$ is valid and $\widehat{\Delta} \wedge x = c(\overline{y_n}) \Rightarrow D$ holds. Since rule VarExp must be applied to derive $\Gamma : x \Downarrow \Delta : c(\overline{y_n})$, $\Delta = \Delta[x \mapsto c(\overline{y_n})]$ so that $\widehat{\Delta} \Rightarrow D \wedge x = c(\overline{y_n})$. Let $D_1 = D \wedge x = c(\overline{y_n})$. Hence, by induction hypothesis applied to $\Delta : \rho(e_1) \Downarrow \Theta : v$, $\Delta : D_1 \mid z \leftarrow \rho(e_1) \Downarrow E_1'$ is valid and $\widehat{\Theta} \wedge z = v \Rightarrow E_1'$. Since the difference between $\rho(e_1)$ and e_1 are the missing bindings of fresh pattern variables and rule *Val* adds simple equations for unbound variables, there is also a derivation tree for $\Delta : D_1 \mid z \leftarrow e_1 \Downarrow E_1''$ where E_1'' is weaker than E_1', i.e., $\widehat{\Theta} \wedge z = v \Rightarrow E_1''$. By Lemma 4.4, there is an assertion E_1 and a valid judgement $\Gamma : D_1 \mid z \leftarrow e_1 \Downarrow E_1$ with $E_1'' \Rightarrow E_1$. By Prop. 4.1, there are derivations for the remaining branches, i.e., $\Gamma : D_i \mid z \leftarrow e_i \Downarrow E_i$ $(i = 2, \ldots, k)$. Altogether, the inference step

$$\frac{\Gamma : C \mid x \leftarrow x \Downarrow D \quad \Gamma : D_1 \mid z \leftarrow e_1 \Downarrow E_1 \ \ldots \ \Gamma : D_k \mid z \leftarrow e_k \Downarrow E_k}{\Gamma : C \mid z \leftarrow \mathit{case}\ x\ \mathit{of}\ \{\overline{p_k \to e_k}\} \Downarrow E_1 \vee \ldots \vee E_k}$$

  is valid and $\widehat{\Theta} \wedge z = v \Rightarrow E_1'' \Rightarrow E_1 \Rightarrow E_1 \vee \ldots \vee E_k$ which shows the claim.

- Rule PrimOp is applied:

$$\frac{\Gamma : x \Downarrow \Delta : x' \quad \Delta : y \Downarrow \Theta : y'}{\Gamma : x \oplus y \Downarrow \Theta : x' \oplus y'}$$

  where $\oplus \in \{\texttt{==}, \texttt{+}, \texttt{-}, \texttt{*}, \ldots\}$ is some primitive operation. Assume that $\widehat{\Gamma} \Rightarrow C$ holds. By induction hypothesis applied to the left premise, $\Gamma : C \mid x \leftarrow x \Downarrow D$ is valid and $\widehat{\Delta} \wedge x = x' \Rightarrow D$ holds. Since rule VarExp must be applied to derive $\Gamma : x \Downarrow \Delta : x'$, $\Delta = \Delta[x \mapsto x']$ so that $\widehat{\Delta} \Rightarrow D$. Similarly, the induction hypothesis applied to the right premise yields a valid judgement $\Delta : D \mid y \leftarrow y \Downarrow E'$ with $\widehat{\Theta} \Rightarrow E'$. By Lemma 4.4, there is an assertion E and a valid judgement $\Gamma : D \mid y \leftarrow y \Downarrow E$ with $E' \Rightarrow E$. Altogether, the inference step

$$\frac{\Gamma : C \mid x \leftarrow x \Downarrow D \quad \Gamma : D \mid y \leftarrow y \Downarrow E}{\Gamma : C \mid z \leftarrow x \oplus y \Downarrow E \wedge z = x \oplus y}$$

  is valid and $\widehat{\Theta} \wedge z = x \oplus y \Rightarrow E \wedge z = x \oplus y$ which shows the claim.

□

5. Examples for Contract Verification

There are various recursively defined operations with pre- and postconditions that can be verified similarly to `fac` as shown above. For instance, the postcondition and the preconditions for both recursive calls to `fib` in

```
fib :: Int  →  Int
fib x | x == 0    = 0
      | x == 1    = 1
      | otherwise = fib (x-1) + fib (x-2)

fib'pre  n   = n >= 0
fib'post n f = f >= 0
```

can be verified with a similar reasoning.

SMT solvers like Z3 [4] provide good reasoning on integer theories. This can be successfully applied to verify more complex postconditions. For instance, consider the function that sums up all natural numbers:

```
sum :: Int  →  Int
sum n = if n==0 then 0
                else n + sum (n-1)
```

The precondition requires that the argument must be non-negative, and the postcondition specifies the correctness of this function by Gauss' formula:

```
sum'pre  n   = n>=0
sum'post n f = f == n * (n+1) `div` 2
```

Our method allows a fully automatic verification of this postcondition.

The precondition on the operation `take` defined by

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

take'pre n xs = n >= 0
```

can be verified similarly to `fac` or `fib` since the list structures are not relevant here. On the other hand, the verification of the precondition of the recursive call of function `last` defined by

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

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

requires the verification of the implication

$$not\ (null\ xs) \wedge xs = (y{:}ys) \wedge ys = (z{:}zs) \Rightarrow not\ (null\ (z{:}zs))$$

This can be proved by evaluating $not\ (null\ (z{:}zs))$ to $true$. Hence, a reasonable verification strategy includes the simplication of proof obligations by symbolic evaluation before passing them to the external verifier.[4]

A more involved operation is the list index operator which selects the nth element of a list:

[4]Since Curry programs might contain non-terminating operations, one has to be careful when simplifying expressions. In

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

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

The precondition ensures that the element to be selected always exists since the selected position is not negative and not larger than the length of the list. The use of the operation `take` (instead of the simpler condition `length xs > n`) is important to allow the application of `nth` also to infinite lists. To verify that the precondition holds for the recursive call, one has to verify that

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

implies

$$(n-1) \geq 0\ \wedge\ length\ (take\ ((n-1)+1)\ ys) = (n-1)+1$$

The proof of the first conjunct uses reasoning on integer arithmetic as in the previous examples. The second conjunct can also be proved by SMT solvers when the rules of the operations `length` and `take` are axiomatized as logic formulas (see below).

The final example is strongly related to functional logic programming since it states a property about a non-deterministic operation. We already presented in Sect. 2 the definition of the operation `perm` which non-deterministically returns some permutation of a list. Any correct implementation of a permutation algorithm should satisfy the property that the length of a permutation of a list l should be identical to the length of l. This can be expressed by the following postcondition:

```
perm'post :: [a] → [a] → Bool
perm'post xs ys = length xs == length ys
```

Antoy et al. [26] propose methods and a tool to translate Curry programs into Agda programs in order to verify properties of a given Curry program. Two different transformation methods to deal with non-deterministic operations are presented. Depending on the transformation method, the proof of the property `perm'post` requires between a few lines and one page of Agda code. Using our contract checker, the property can be verified in a fully automatic manner. Similarly to [26], we also state a property about the non-deterministic list insertion operation `ins`:

```
ins'post :: a → [a] → [a] → Bool
ins'post _ xs ys = length xs + 1 == length ys
```

Both postconditions `perm'sort` and `ins'sort` will automatically be verified with our implemented tool (see next section), where the postcondition `ins'sort` is necessary to verify the postcondition `perm'post`.

order to ensure the termination of the simplification process, one can either limit the number of simplification steps or use only operations for simplification that are known to be terminating. Since the latter property can be approximated by various program analysis techniques, the Curry program analyzer CASS [25] contains such an analysis.

6. Implementation

We implement static contract verification as a fully automatic tool which tries to verify contracts at compile time and, in case of a successful verification, does not generate code for dynamic (run-time) contract checking. The complete compilation chain with this tool is as follows:

1. The program under consideration is compiled with the standard Curry front end into an intermediate FlatCurry program.
2. For each postcondition f`'post`, the contract verifier extracts the proof obligation as described in Sect. 4.
3. Each proof obligation is translated into SMT-LIB format (the standard input language for SMT solvers [27]) and sent to an SMT solver (here: Z3 [4]).
4. If the validity of the postcondition cannot be verified by the SMT solver, the definition of the operation f is decorated with code to check the postcondition at run time.
5. Similarly, for each precondition f`'pre` and each call to f, the contract verifier extracts the proof obligation for this call together with the precondition and sends it to the SMT solver. If the validity of the precondition cannot be verified, f will be called with a run-time check for this precondition, otherwise the run-time check will be omitted.

Thus, if no static proof is successful, all contracts are added as run-time checks, as sketched in Sect. 3 and described in [3]. If all contracts can be verified, the program code is not modified and we have a high confidence in our code.

Although the general extraction of proof obligations from a given program with contracts is clear from the description in Sect. 4, the translation of these proof obligations into SMT-LIB format requires some design decisions caused by the specific nature of a functional logic language like Curry. We discuss some of these issues in the following.

When pre- and postconditions are constructed from a fixed set of operations that are known to the underlying SMT solver (e.g., as in LiquidHaskell [28]), one can directly translate the proof obligations for contract checking into SMT-LIB formulas. However, we also allow user-defined operations (like `length` or `take` in the precondition of `nth`) in contracts so that their meaning must be axiomatized in the SMT language. Due to the features of Curry, it might be necessary to transform

- polymorphic algebraic data types and
- polymorphic, possibly non-deterministic operations

into SMT-LIB. The transformation of data types can be directly implemented. Since version 2.6 of SMT-LIB [27], there are commands `declare-datatype` and `declare-datatypes` to introduce single and mutually recursive data types, respectively, which can also be parameterized by sorts. Furthermore, there is a `match` construct for pattern matching on values of algebraic data types.

The transformation of arbitrary Curry operations is more involved. Although mutually recursive functions can be defined in SMT-LIB via the `define-funs-rec` command, functions defined in Curry can be polymorphic and also non-deterministic. Both features are not supported by SMT-LIB (although some SMT solvers have extensions for polymorphic functions). Inspired by [29], we introduce a new SMT construct `define-pfuns-rec` to define a set of mutually recursive function signatures and corresponding terms defining the semantics of these functions. The general syntax is

(define-pfuns-rec (st_1 ... st_m))

Each st_i has the form

(par (s_1 ... s_k) (f ((x_1 σ_1) ... (x_n σ_n)) σ) t)

where $s_1, \ldots, s_k$ are sort parameters, $\sigma_1, \ldots, \sigma_n, \sigma$ are sorts, possibly containing the sort parameters, describing the signature of the function f, and t is a term over the parameters $x_1, \ldots, x_n$, specifying the meaning of the function f. For instance, the polymorphic predicate `null` (used in the example `last`) is defined in the prelude of Curry as follows:

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

Since pattern matching is represented in FlatCurry by case expressions, the translation into SMT is straightforward:

```
(define-pfuns-rec
   ((par (a) (null ((x1 (List a))) Bool)
             (match x1 ((nil true)
                        ((insert h t) false))))))
```

Since SMT-LIB does not support polymorphic operations, this definition cannot be directly translated into SMT. In order to reason about properties of polymorphic operations (as in the examples `take`, `last`, or `nth`), one could introduce a new "type variable" sort

```
(declare-sort TVar 0)
```

and instantiate all sort parameters to this specific sort so that the definition above is translated into the SMT statements

```
(declare-fun null ((List TVar)) Bool)
(assert
 (forall ((x1 (List TVar)))
  (= (null x1)
     (match x1 ((nil true)
                ((insert h t) false))))))
```

However, this does not work for operations that are applied to values of specific types. For instance, if we apply `null` to a list of Booleans and a list of integers, we need different type instantiations. Therefore, our contract prover collects all type instantiations of polymorphic operations used in an SMT script and specializes the generic operations on these types, e.g., type-specific operations like `null_Bool` or `null_Int` are generated. Since these operations might call other generic operations which must also be type-specialized, the entire process is implemented as a fixpoint computation on

Table 1. Benchmarks comparing dynamic and static contract checking

Expression	dynamic	static+dynamic	speedup
`fac 20`	0.00	0.00	n.a.
`sum 1000000`	0.99	0.19	5.10
`fib 35`	1.95	0.60	3.23
`last [1..20000000]`	0.63	0.35	1.78
`take 200000 [1..]`	0.31	0.19	1.68
`nth [1..] 50000`	26.33	0.01	2633
`allNats 200000`	0.27	0.19	1.40
`init [1..10000]`	2.78	0.00	>277
`[1..20000] ++ [1..1000]`	4.21	0.00	>420
`nrev [1..1000]`	3.50	0.00	>349
`rev [1..10000]`	1.88	0.00	>188

each `define-pfuns-rec` definition.

To translate non-deterministic operations into SMT functions, one can use existing approaches to transform non-deterministic operations into pure functions. For instance, methods to translate Curry programs into Agda programs are proposed in [26]. One of these techniques, called "planned choices," is quite appropriate here since it assumes an oracle for making the right non-deterministic choices. This oracle is added as an argument to each non-deterministic operation and its initial value can be modeled as a constant in an SMT script.

Note that the translation of user-defined operations is necessary only if such operations are used in pre- and postconditions. Thus, it is an acceptable limitation that not all features of Curry are fully modeled by our translator. For instance, higher-order features are replaced by free variables (i.e., properties introduced by their application are ignored), and the SMT solver is invoked with a timeout to deal with possibly non-terminating operations. This might imply that some valid contracts cannot be verified, but this does not cause a problem in our framework since such unverified contracts are checked at run time. However, the examples presented so far and the benchmarks shown in the next section demonstrate that our contract prover yields useful results.

7. Benchmarks

In order to get an idea about the efficiency improvement by static contract verification, we apply our tool described in the previous section to some benchmark programs. For this purpose, we compared the execution time of the program with and without static contract checking. Note that in case of preconditions, only verified preconditions for recursive calls can be omitted so that the operations can safely be invoked as before.

For the benchmarks, we used the Curry implementation KiCS2 (Version 0.6.0) [13] with the Glasgow Haskell Compiler (GHC 7.10.3, option `-O2`) as its back end on a Linux machine (Debian 8.9) with an Intel Core i7-4790 (3.60Ghz) processor and 8GiB of memory. Table 1 shows the execution times (in seconds, where "0.00" means less than 10 ms) of executing a program with the given main expression. Column "dynamic" denotes purely dynamic contract checking and column "static+dynamic" denotes the combination of static and dynamic contract checking as described in this paper. The column "speedup" is the ratio of the previous columns (where a lower bound is given if the execution time of the optimized program is below 10 ms).

Many of the programs that we tested are already discussed in this paper. `allNats` produces (non-deterministically) some natural number between `0` and the given argument, where the precondition requires that the argument must be non-negative. `init` removes the last element of a list, where the precondition requires that the list is non-empty and the postcondition states that the length of the output list is decremented by one. The list concatenation (++) has a postcondition which states that the length of the output list is the sum of the lengths of the input lists. `nrev` and `rev` are naive and linear list reverse operations, respectively, where their postconditions require that the input and output lists are of identical length.

As expected, the benchmarks show that static contract checking has a positive impact on the execution time. If contracts are complex, e.g., require recursive computations on arguments, as in `nth`, `init`, "++", or `rev`, static contract checking can improve the execution times by orders of magnitudes. Even if the improvement is small or not measurable (e.g., `fac`), static contract verification is useful since any verified contract increases the confidence in the correctness of the software and contributes to a more reliable software product.

Our contract verification tool is available as package `contract-prover` which can easily be installed with the Curry package manager.[5] The package also contains many further examples of successful static contract checking, ranging from arithmetic functions, like the McCarthy 91 or the Ackermann function, to list functions and non-deterministic operations.

8. Applications

Increasing reliability and efficiency of programs is the main motivation for static contract verification. However, there are also other areas where the results of static contract verification can be applied. In the following, we discuss two of them.

Precondition verification is used to optimize the call site of operations, i.e., it is related to a specific use of an operation. Postcondition verification is more general since a verified postcondition shows a property that holds for all valid calls of an operation. Therefore, our contract checker stores each successfully proved post-condition so that this information is available to other tools. For instance, if the postcondition of operation f defined in module M has been verified, the proof, i.e., the SMT-LIB script, is stored in file `PROOF_`M`_`f`_SatisfiesPostCondition.smt` so that it is available to other programming tools. Currently, such proofs are used by two tools: a property-based test tool and a verifier for non-failing programs.

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

CurryCheck [21] is a property-based test tool, i.e., it automatically tests properties parameterized over input data by generating test inputs and evaluating the properties on these inputs. CurryCheck uses EasyCheck [30] to generate test inputs in a systematic way by functional logic programming features. Usually, properties to be tested are defined by the programmer. However, there are also properties which are automatically generated and tested by CurryCheck. These properties are related to contracts and specifications. For instance, if an operation f of type $\tau \rightarrow \tau'$ has a precondition f'pre and a postcondition f'post, then CurryCheck generates the property

```
f_SatisfiesPostCondition :: τ → Prop
f_SatisfiesPostCondition x =
  f'pre x ==> always (f'post x (f x))
```

The property "always x" is satisfied if all values of x are True and the property "b ==> p" is satisfied if p is satisfied for all values where b evaluates to True. Thus, this generated property expresses the fact that the postcondition must be satisfied for all results computed from inputs satisfying the precondition. Usually, CurryCheck tests this property with hundreds of input values (see [30] for a description about the generation of these inputs). However, these tests are superfluous if the postcondition is already verified. Therefore, CurryCheck takes the results of the static contract checker described in this paper into account: if a proof file for a post condition exists, the property to check is not generated so that CurryCheck does not waste time to test it.

A tool to verify the absence of failures due to calling partially defined operations with unintended arguments is presented in [31]. It is based on the idea to express sufficient conditions about executing operations without failing as "non-fail conditions." For instance, the operation to compute the first element of a list

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

has the non-fail condition

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

Although non-fail conditions look similar to preconditions, non-fail conditions are weaker. If a precondition of f is not satisfied for a given argument, it is not allowed to invoke f with this argument. On the other hand, a non-fail condition provides a sufficient criterion to avoid failing computations, but it might be reasonable to invoke partially defined operations in logic-oriented computations. Thus, non-fail conditions have a different semantics so that they require different verification methods. Nevertheless, it has been shown in [31] that SMT solvers are a reasonable tool to verify non-fail conditions.

There are cases where the verification of non-fail conditions can be considerably improved by exploiting verified postconditions. For instance, consider the operation split that splits a list into components delimited by separators, where the separator elements are characterized by a given predicate:

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

```
                                        in (x : head sp) : tail sp

split'nonfail p xs = True
```

In order to verify that the trivial non-fail condition for `split` is correct, one has to show that the calls "`head sp`" and "`tail sp`" are non-failing. This demands to show that the result of `split p xs` is a non-empty list. This property can be stated as a postcondition:

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

Using the techniques presented in this paper, this postcondition can be verified so that our contract verifier stores a proof for this postcondition. The proof and, thus, the knowledge about the correctness of the postcondition are exploited to deduce that `not (null sp)` holds so that the non-fail conditions of the calls to `head` and `tail` are satisfied. Thus, the trivial non-fail condition of `split` is verified.

9. Related Work

As contract checking is an important contribution to obtain more reliable software, techniques for it have been extensively explored. Mostly related to our approach is the work of Stulova et al. [32] on reducing run-time checks of assertions by static analysis in logic programs. Although the objectives of this and our work are similar, the techniques and underlying programming languages are different. For instance, Curry with its demand-driven evaluation strategy prevents the construction of static call graphs that are often used to analyze the data flow as in logic programming. The latter is used by Stulova et al. where assertions are verified by static analysis methods. Hence, the extensive set of benchmarks presented in their work is related to typical abstract domains used in logic programming, like modes or regular types. There are also approaches to approximate argument/result size relations in logic programs, e.g., [33], which might be used to verify assertions related to the size of data. In contrast to these fixpoint-based approaches, we simply collect assertions from program expressions and use symbolic reasoning, e.g., integer arithmetic with user-defined functions, to solve them. SMT solvers are well suited for this purpose and we showed that they can be successfully applied to verify complex assertions (as in the example `nth` shown in Sect. 5).

Static contract checking has also been explored in purely functional languages. For instance, Xu et al. [34] present a method for static contract checking in Haskell by a program transformation and symbolic execution. Since an external verifier is not used, the approach is more limited. SMT solvers for static contract checking are also used in [35]. Similarly to our work, abstract assertions are collected and solved by an SMT solver in order to verify contracts. However, we consider a non-strict non-deterministic language which requires a different reasoning compared to the strict functional language used there. Another approach is the extension of the type system to express contracts as specific types. Dependent types are quite powerful since they allow to express size or shape constraints on data in the language of types. Although this supports the development of programs together with their correctness proofs [2], programming in such a language could be challenging if the proofs are difficult to construct. Therefore, we prefer a more practical method: properties which cannot be statically verified are checked at run time. One can also express contracts as refinement types as in LiquidHaskell [36, 28]. Similarly to our approach, LiquidHaskell uses an external SMT solver to verify contracts. Hence, LiquidHaskell can verify quite complex assertions, as shown by various case studies in [36]. Nevertheless, there might be assertions that cannot be verified so that a combination of static and dynamic checking is preferable in practice.

An alternative approach to make dynamic contract checking more efficient has been proposed in [37] where assertions are checked in parallel to the application program. Thus, one can exploit the power of multi-core computers for assertion checking by running the main program and the contract checker on different cores.

10. Conclusions

In this paper we proposed a framework to combine static and dynamic contract checking. Contracts are useful to make software more reliable, e.g., avoid invoking operations with unintended arguments. Since checking all contracts at run time increases the overall execution time, we presented a method to verify contracts in Curry at compile time by using an external SMT solver. Of course, this might not be successful in all cases so that unverified contracts are still required to be checked at run time. Nevertheless, our experiments show the advantages of this technique, in particular, to reduce dynamic contract checking for recursive calls. Since we developed this framework for Curry, a language combining functional and logic programming features, the same techniques can be applied to purely functional or purely logic languages.

We do not expect that all contracts can be statically verified. Apart from the complexity of some contracts, preconditions of operations of the API of some libraries or packages cannot be checked since their use is unknown at compile time. However, one could provide two versions of such operations, one with a dynamic precondition check and one ("unsafe") without this check. Whenever one can verify that the precondition is satisfied at the call site, one can invoke the version without the precondition check. If all versions with precondition checks become dead code in a complete application, one has a high confidence in the quality of the entire application.

Another refinement of our approach is the consideration of the individual conjuncts of contracts. For instance, if a pre- or postcondition is a conjunction of formulas, each conjunct can separately be verified so that only the remaining unverified conjuncts have to be added for run-time checking. This allows to make dynamic contract checking more efficient even if the complete contract cannot be verified.

Our tool could be improved by exploiting strictness information. Currently, information about functions as arguments to other functions is ignored except for primitive operations where the demand on arguments is known. This can be extended to any user-defined function if its demand is known at compile time. The latter can be approximated by a demand analysis. A simple approach to it is shown in [24] but needs to be improved for our purposes.

It could also be interesting to combine our tool with other tools related to program testing and verification. Potential benefits of such a combination were discussed in Sect. 8 but should be further explored. On the other hand, program analysis tools might be useful to improve static verification, e.g., demand information can be used to generate more precise proof obligations. If the contract verifier finds counter-examples to some proof obligation, one could also analyze these in order to check whether they show an actual contract violation. Furthermore, it might also be interesting to improve the power of static contract checking by integrating abstract interpretation techniques, like [38, 32].

Acknowledgments. The author is grateful to John Gallagher, Grigore Rosu, and the anonymous reviewers for their helpful comments to improve the paper.

References

[1] Milner R. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 1978. **17**:348–375.

[2] Stump A. Verified Functional Programming in Agda. ACM and Morgan & Claypool, 2016. doi:10.1145/2841316.

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

[4] de Moura L, Bjørner N. Z3: An Efficient SMT Solver. In: Proc. of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008). Springer LNCS 4963, 2008 pp. 337–340. doi:10.1007/978-3-540-78800-3.

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

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

[7] Peyton Jones S (ed.). Haskell 98 Language and Libraries—The Revised Report. Cambridge University Press, 2003.

[8] Antoy S, Echahed R, Hanus M. A Needed Narrowing Strategy. *Journal of the ACM*, 2000. **47**(4):776–822. doi:10.1145/347476.347484.

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

[10] Antoy S. Constructor-based Conditional Narrowing. In: Proc. of the 3rd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP 2001). ACM Press, 2001 pp. 199–206.

[11] Antoy S, Hanus M. Overlapping Rules and Logic Variables in Functional Logic Programs. In: Proceedings of the 22nd International Conference on Logic Programming (ICLP 2006). Springer LNCS 4079, 2006 pp. 87–101.

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

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

[14] Wadler P. How to Declare an Imperative. *ACM Computing Surveys*, 1997. **29**(3):240–263.

[15] Antoy S, Hanus M. Set Functions for Functional Logic Programming. In: Proceedings of the 11th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'09). ACM Press, 2009 pp. 73–82. doi:10.1145/1599410.1599420.

[16] Antoy S, Hanus M. Declarative Programming with Function Patterns. In: Proceedings of the International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'05). Springer LNCS 3901, 2005 pp. 6–22.

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

[18] Albert E, Hanus M, Huch F, Oliver J, Vidal G. Operational Semantics for Declarative Multi-Paradigm Languages. *Journal of Symbolic Computation*, 2005. **40**(1):795–829.

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

[20] Braßel B. Implementing Functional Logic Programs by Translation into Purely Functional Programs. Ph.D. thesis, Christian-Albrechts-Universität zu Kiel, 2011.

[21] Hanus M. CurryCheck: Checking Properties of Curry Programs. In: Proceedings of the 26th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2016). Springer LNCS 10184, 2017 pp. 222–239. doi:10.1007/978-3-319-63139-4_13.

[22] Chitil O, McNeill D, Runciman C. Lazy Assertions. In: Proceedings of the 15th International Workshop on Implementation of Functional Languages (IFL 2003). Springer LNCS 3145, 2004 pp. 1–19.

[23] Degen M, Thiemann P, Wehr S. True Lies: Lazy Contracts for Lazy Languages (Faithfulness is Better than Laziness). In: 4. Arbeitstagung Programmiersprachen (ATPS'09). Springer LNI 154, 2009 pp. 370; 2946–2259.

[24] Hanus M. Improving Lazy Non-Deterministic Computations by Demand Analysis. In: Technical Communications of the 28th International Conference on Logic Programming, volume 17. Leibniz International Proceedings in Informatics (LIPIcs), 2012 pp. 130–143. doi:10.4230/LIPIcs.ICLP.2012.130.

[25] Hanus M, Skrlac F. 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). ACM Press, 2014 pp. 181–188. doi:10.1145/2543728.2543744.

[26] Antoy S, Hanus M, Libby S. Proving Non-Deterministic Computations in Agda. In: Proc. of the 24th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2016), volume 234 of *Electronic Proceedings in Theoretical Computer Science*. Open Publishing Association, 2017 pp. 180–195. doi:10.4204/EPTCS.234.13.

[27] Barrett C, Fontaine P, Tinelli C. The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa, 2017. Available at `www.SMT-LIB.org`.

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

[29] Claessen K, Johansson M, Rosén D, Smallbone N. TIP: Tons of Inductive Problems. In: Int. Conf. on Intelligent Computer Mathematics (CICM 2015). Springer LNCS 9150, 2015 pp. 333–337. doi:10.1007/978-3-319-20615-8_23.

[30] Christiansen J, Fischer S. EasyCheck - Test Data for Free. In: Proc. of the 9th International Symposium on Functional and Logic Programming (FLOPS 2008). Springer LNCS 4989, 2008 pp. 322–336.

[31] Hanus M. Verifying Fail-Free Declarative Programs. In: Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming(PPDP 2018). ACM Press, 2018 pp. 12:1–12:13. doi:10.1145/3236950.3236957.

[32] Stulova N, Morales J, Hermenegildo M. Reducing the Overhead of Assertion Run-time Checks via Static Analysis. In: Proc. 18th International Symposium on Principles and Practice of Declarative Programming (PPDP 2016). ACM Press, 2016 pp. 90–103.

[33] Serrano A, López-García P, Bueno F, Hermenegildo M. Sized Type Analysis for Logic Programs. *Theory and Practice of Logic Programming*, 2013. **13**(4-5-Online-Supplement). doi: http://static.cambridge.org/resource/id/urn:cambridge.org:id:binary:20161018085635834-0697:S1471068413000112:tlp2013011.pdf.

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

[35] Nguyen P, Tobin-Hochstadt S, Van Horn D. Soft Contract Verification. In: Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP 2014). ACM Press, 2014 pp. 139–152. doi:10.1145/2628136.2628156.

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

[37] Dimoulas C, Pucella R, Felleisen M. Future Contracts. In: Proceedings of the 11th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'09). ACM Press, 2009 pp. 195–206.

[38] Fähndrich M, Logozzo F. Static contract checking with Abstract Interpretation. In: Proc. of the Conference on Formal Verification of Object-oriented Software (FoVeOOS 2010). Springer LNCS 6528, 2011 pp. 10–30. doi:10.1007/978-3-642-18070-5_2.