Combining Static and Dynamic Contract Checking for Curry

Michael Hanus

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

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, dynamic checking of it can be omitted. 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 [23] as "well-typed expressions 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 [27], or add contracts with pre- and postconditions to operations. In this paper, we follow the latter approach since it provides a smooth integration into existing software development processes. For instance, 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 to restrict the applicability of an operation. Following the notation proposed in [6], 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 a precondition for checking `fac` invocations at run time, a preprocessor could transform each call to `fac` by attaching an additional test whether the precondition is satisfied (see [6]). 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 a full static verification, e.g., with proof assistants like Agda, Coq, or Isabelle, which is time-consuming and difficult, and a full dynamic checking, which might be inefficient.

For instance, one can verify (e.g., with an SMT solver [12]) 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 [21], 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 before we discuss the current implementation and first benchmark results, which are quite encouraging.

2 Functional Logic Programming and Curry

Functional logic languages combine the most important features of functional and logic programming in a single language (see [17] for a recent survey). In particular, the functional logic language Curry [21] 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 and Curry that are necessary to understand the contents of this paper. More details can be found in surveys on functional logic programming [17] and in the language report [21].

The syntax of Curry is close to Haskell [24]. In addition to Haskell, Curry applies rules with overlapping left-hand sides in a (don't know) non-deterministic manner

[1] The type inference depends on the underlying static type system. For instance, Haskell infers a more general overloaded type.

(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 unless they are anonymous. Function calls can contain free variables, in particular, variables without a value at call time. These calls are evaluated lazily where free variables as demanded arguments are non-deterministically instantiated [2].

Example 1. The following simple program shows the functional and logic features of Curry. It defines an operation "`++`" to concatenate two lists, which is identical to the Haskell encoding. The operation `ins` inserts an element at some (unspecified) position in a list:

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

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

Note that `ins` is a *non-deterministic operation* since it might deliver more than one result for a given argument, e.g., the evaluation of `ins 0 [1,2]` yields the values `[0,1,2]`, `[1,0,2]`, and `[1,2,0]`. Non-deterministic operations, which are interpreted as mappings from values into sets of values [15], 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 []     = []
perm (x:xs) = ins x (perm xs)
```

Non-deterministic operations are quite expressive since they can be used to completely eliminate logic variables in functional logic programs. Actually, it has been shown that non-deterministic operations and logic variables have the same expressive power [4,11]. For instance, a Boolean logic variable can be replaced by the non-deterministic *generator* operation for Booleans defined by

```
aBool = False ? True
```

This equivalence can be exploited when Curry is implemented by translation into a target language without support for non-determinism and logic variables. For instance, KiCS2 [9] 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 [30] for declarative input/output, set functions [5] to encapsulate non-deterministic search, functional patterns [3] and default rules [7] to specify complex transformations in a high-

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

Fig. 1. Syntax of the intermediate language FlatCurry

level manner, and a hierarchical module system together with a package manager[2] that provides access to dozens of packages with hundreds of 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 been used to specify the operational semantics of Curry programs [1]. 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., [1,17]), 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 [9]). 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 [1] as an extension of Launchbury's natural semantics for lazy evaluation [22]. 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 [1]. 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 [15], 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 *empty heap* is denoted by $[]$. $\Gamma[x \mapsto e]$ denotes a heap Γ' with $\Gamma'(x) = e$ and $\Gamma'(y) = \Gamma(y)$ for all $x \neq y$.

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

$$\textsf{Val} \qquad \Gamma : v \Downarrow \Gamma : v \quad \text{where } v \text{ is constructor-rooted}$$

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

$$\textsf{Fun} \qquad \frac{\Gamma : \rho(e) \Downarrow \Delta : v}{\Gamma : f(\overline{x_n}) \Downarrow \Delta : v} \quad \text{where } f(\overline{y_n}) = e \in P \text{ and } \rho = \{\overline{y_n \mapsto x_n}\}$$

$$\textsf{Let} \qquad \frac{\Gamma[\overline{y_k \mapsto \rho(e_k)}] : \rho(e) \Downarrow \Delta : v}{\Gamma : \mathit{let}\ \{\overline{x_k = e_k}\}\ \mathit{in}\ e \Downarrow \Delta : v} \quad \begin{array}{l}\text{where } \rho = \{\overline{x_k \mapsto y_k}\} \\ \qquad\quad \text{and } \overline{y_k} \text{ are fresh variables}\end{array}$$

$$\textsf{Or} \qquad \frac{\Gamma : e_i \Downarrow \Delta : v}{\Gamma : e_1\ \mathit{or}\ e_2 \Downarrow \Delta : v} \quad \text{where } i \in \{1,2\}$$

$$\textsf{Select} \qquad \frac{\Gamma : x \Downarrow \Delta : c(\overline{y_n}) \quad \Delta : \rho(e_i) \Downarrow \Theta : v}{\Gamma : \mathit{case}\ x\ \mathit{of}\ \{\overline{p_k \to e_k}\} \Downarrow \Theta : v} \quad \begin{array}{l}\text{where } p_i = c(\overline{x_n}) \\ \qquad\quad \text{and } \rho = \{\overline{x_n \mapsto y_n}\}\end{array}$$

Fig. 2. Natural semantics of normalized FlatCurry programs

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

Constructor-rooted expressions (i.e., head normal forms) are just returned by rule Val. Rule VarExp retrieves a binding for a variable from the heap and evaluates it. In order to avoid the re-evaluation of the same expression, VarExp updates the heap with the computed value, which models sharing. In contrast to the original rules [1], VarExp removes the binding from the heap. On the one hand, this allows the detection of simple loops ("black holes") as in functional programming. On the other hand, it is crucial in combination with non-determinism to avoid the binding of a variable to different values in the same derivation (see [8] for a detailed discussion on this issue). Rule Fun unfolds function calls by evaluating the right-hand side after binding the formal parameters to the actual ones. Let introduces new bindings in the heap and renames the variables in the expressions with the fresh names introduced in the heap. Or non-deterministically evaluates one of its arguments. Finally, rule Select deals with *case* expressions. When the discriminating 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 [17] 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 even 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 [6]. 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 [3] 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 the precondition of nested `fac` applications, like in the expression `fac (fac 3)`. If there is 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'spec_S 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 [5]).

Antoy and Hanus [6] describe a tool which transforms programs containing contracts and specifications into programs where these contracts and specifications are dynamically checked. This tool is available in recent distributions of the Curry implementations PAKCS [19] and KiCS2 [9] as a preprocessor so that the transformation can be automatically performed when Curry programs are compiled. Furthermore, the property-based testing tool CurryCheck [18] automatically tests contracts and specifications with generated input data.

Although these dynamic and static testing tools provide some confidence in the software under development, a 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 some of them is not present, we assume that

they are defined as predicates which 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:

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

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

Note that we specified *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. [10] 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, 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

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

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.

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

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

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

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

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

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

Fig. 3. Abstract assertion-collecting semantics

This example shows that we have to collect in expressions (the rules' right-hand sides) properties that are ensured to be valid when we reach particular points. For this purpose, we define an *abstract 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.

The abstract 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 this abstract semantics. 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 abstract evaluation of the expression. Note that the abstract 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 abstract semantics is deterministic, i.e., for each heap Γ, assertion C, variable z, and expression e, 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.

The abstract semantics allows to extract proof obligations to verify contracts. For instance, to verify that a postcondition $f'\,\texttt{post}$ 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 derive for the right-hand side of `coin` the following proof tree:

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

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:

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

Since there is no condition on the result variable z in the second argument 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          g x = const (f x) 42
                     f'post x z = x>0
```

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

$$[] : true \mid z \leftarrow e \Downarrow true$$

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

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

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

Nevertheless, we can improve our abstract semantics in cases where it is ensured that arguments are evaluated. For instance, primitive operations, like `+`, `*`, or `==`, evaluate their arguments. Thus, we can add the following rule (and restrict rule *Fun* to exclude these operations):

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

Since primitive operations are often known to the underlying verifier, we also collect the information about the call of the primitive operation. 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 [16].

If we construct a proof tree for the factorial function with these refined inference rules, 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:

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

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 relation between the concrete and the abstract semantics:

Theorem 1. *If $\Gamma : e \Downarrow \Gamma' : v$ is 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 $(\widehat{\Gamma'} \wedge z = v) \Rightarrow D$.*

Here, $\widehat{\Gamma}$ denotes the representation of heap information as a logic formula, i.e.,

$$\widehat{\Gamma} = \bigwedge \{x = e \mid x \mapsto e \in \Gamma,\ e \text{ not operation-rooted}\}$$

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

5 More Examples

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 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. The precondition on `take` defined by

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

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

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

```
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 the right-hand side to *true*. Hence, a reasonable verification strategy includes the simplication of proof obligations by symbolic evaluation before passing them to the external verifier.[3]

A more involved operation is the list index operator which selects the *n*th element of a list:

```
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

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

[3] Since Curry programs might contain non-terminating operations, one has to be careful when simplifying expressions. In 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 [20] contains such an analysis.

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.

6 Implementation and Benchmarks

We have implemented static contract verification as a fully automatic tool which tries to verify contracts at compile time and, in case of a successful verification, removes their run-time checking from the generated code. The complete compilation chain with this tool is as follows:

1. The Curry preprocessor performs a source-level transformation to add contracts as run-time checks, as sketched in Sect. 3 and described in [6].
2. The preprocessed program is compiled with the standard Curry front end into an intermediate FlatCurry program.
3. For each contract, the contract verifier extracts the proof obligation as described in Sect. 4.
4. Each proof obligation is translated into SMT-LIB format and sent to an SMT solver (here: Z3 [12]).
5. If the proof shows the validity of the contract, its check is removed from the FlatCurry program.

This general approach can be refined. For instance, if a pre- or postcondition is a conjunction of formulas, each conjunct can separately be verified and possibly removed. This allows to make dynamic contract checking more efficient even if the complete contract cannot be verified.

Although our tool is a prototype, we applied it to some initial benchmarks in order to get an idea about the efficiency improvement by static contract verification. 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) [9] 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. `sum` is similar to `fac` but adds all numbers instead of multiplying them. `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

Expression	dynamic	static+dynamic	speedup
`fac 20`	0.00	0.00	n.a.
`sum 1000000`	0.84	0.22	3.88
`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

Table 1. Benchmarks comparing dynamic and static contract checking

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.

7 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. [26] 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., [25], which might be used to verify assertions related to the size of data. On the other hand, many of our examples require symbolic reasoning on integer arithmetic with user-defined functions. For this purpose, SMT solvers are well suited and we have shown that they can be successfully applied to verify complex assertions (see example `nth` above).

Static contract checking has also been explored in purely functional languages. For instance, [31] presents 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. 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 [27], programming in such a language could be challenging if the proofs are difficult to construct. Therefore, we prefer a more practical method by checking properties which cannot be statically proved at run time. Another approach to express contracts as types is LiquidHaskell [28,29]. 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 [28]. Nevertheless, there might be assertions that cannot be verified in this way 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 [13] 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.

8 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 have shown 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 initial 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.

For future work, we will improve our tool in order to test the effectiveness of our approach on larger examples. This might provide also insights how to improve this approach in practice, e.g., how to use demand information 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 vi-

olation. Furthermore, it might also be interesting to improve the power of static contract checking by integrating abstract interpretation techniques, like [14,26].

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

References

1. E. Albert, M. Hanus, F. Huch, J. Oliver, and G. Vidal. Operational semantics for declarative multi-paradigm languages. *Journal of Symbolic Computation*, 40(1):795–829, 2005.
2. S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. *Journal of the ACM*, 47(4):776–822, 2000.
3. S. Antoy and M. Hanus. Declarative programming with function patterns. In *Proceedings of the International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'05)*, pages 6–22. Springer LNCS 3901, 2005.
4. S. Antoy and M. Hanus. Overlapping rules and logic variables in functional logic programs. In *Proceedings of the 22nd International Conference on Logic Programming (ICLP 2006)*, pages 87–101. Springer LNCS 4079, 2006.
5. S. Antoy and M. Hanus. Set functions for functional logic programming. In *Proceedings of the 11th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'09)*, pages 73–82. ACM Press, 2009.
6. S. Antoy and M. Hanus. Contracts and specifications for functional logic programming. In *Proc. of the 14th International Symposium on Practical Aspects of Declarative Languages (PADL 2012)*, pages 33–47. Springer LNCS 7149, 2012.
7. S. Antoy and M. Hanus. Default rules for Curry. *Theory and Practice of Logic Programming*, 17(2):121–147, 2017.
8. B. Braßel. *Implementing Functional Logic Programs by Translation into Purely Functional Programs*. PhD thesis, Christian-Albrechts-Universität zu Kiel, 2011.
9. B. Braßel, M. Hanus, B. Peemöller, and F. Reck. KiCS2: A new compiler from Curry to Haskell. In *Proc. of the 20th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2011)*, pages 1–18. Springer LNCS 6816, 2011.
10. O. Chitil, D. McNeill, and C. Runciman. Lazy assertions. In *Proceedings of the 15th International Workshop on Implementation of Functional Languages (IFL 2003)*, pages 1–19. Springer LNCS 3145, 2004.
11. J. de Dios Castro and F.J. López-Fraguas. Extra variables can be eliminated from functional logic programs. *Electronic Notes in Theoretical Computer Science*, 188:3–19, 2007.
12. 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.
13. C. Dimoulas, R. Pucella, and M. Felleisen. Future contracts. In *Proceedings of the 11th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'09)*, pages 195–206. ACM Press, 2009.
14. M. Fähndrich and F. Logozzo. Static contract checking with abstract interpretation. In *Proc. of the Conference on Formal Verification of Object-oriented Software (FoVeOOS 2010)*, pages 10–30. Springer LNCS 6528, 2011.
15. J.C. González-Moreno, M.T. Hortalá-González, F.J. López-Fraguas, and M. Rodríguez-Artalejo. An approach to declarative programming based on a rewriting logic. *Journal of Logic Programming*, 40:47–87, 1999.

16. M. Hanus. Improving lazy non-deterministic computations by demand analysis. In *Technical Communications of the 28th International Conference on Logic Programming*, volume 17, pages 130–143. Leibniz International Proceedings in Informatics (LIPIcs), 2012.
17. M. Hanus. Functional logic programming: From theory to Curry. In *Programming Logics - Essays in Memory of Harald Ganzinger*, pages 123–168. Springer LNCS 7797, 2013.
18. M. Hanus. CurryCheck: Checking properties of Curry programs. In *Proceedings of the 26th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2016)*, pages 222–239. Springer LNCS 10184, 2017.
19. M. Hanus, S. Antoy, B. Braßel, M. Engelke, K. Höppner, J. Koj, P. Niederau, R. Sadre, and F. Steiner. PAKCS: The Portland Aachen Kiel Curry System. Available at `http://www.informatik.uni-kiel.de/~pakcs/`, 2016.
20. M. Hanus and F. Skrlac. A modular and generic analysis server system for functional logic programs. In *Proc. of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation (PEPM'14)*, pages 181–188. ACM Press, 2014.
21. M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.9.0). Available at `http://www.curry-language.org`, 2016.
22. J. Launchbury. A natural semantics for lazy evaluation. In *Proc. 20th ACM Symposium on Principles of Programming Languages (POPL'93)*, pages 144–154. ACM Press, 1993.
23. R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.
24. S. Peyton Jones, editor. *Haskell 98 Language and Libraries—The Revised Report*. Cambridge University Press, 2003.
25. A. Serrano, P. López-García, F. Bueno, and M.V. Hermenegildo. Sized type analysis for logic programs. *Theory and Practice of Logic Programming*, 13(4-5-Online-Supplement), 2013.
26. N. Stulova, J.F. Morales, and M. Hermenegildo. 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)*, pages 90–103. ACM Press, 2016.
27. A. Stump. *Verified Functional Programming in Agda*. ACM and Morgan & Claypool, 2016.
28. 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.
29. N. Vazou, E.L. Seidel, R. Jhala, 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.
30. P. Wadler. How to declare an imperative. *ACM Computing Surveys*, 29(3):240–263, 1997.
31. D.N. Xu, S.L. Peyton Jones, and K. Claessen. Static contract checking for Haskell. In *Proc. of the 36th ACM Symposium on Principles of Programming Languages (POPL 2009)*, pages 41–52, 2009.