Contracts and Specifications for Functional Logic Programming

Sergio Antoy[1] Michael Hanus[2]

[1] Computer Science Dept., Portland State University, Oregon, U.S.A.
antoy@cs.pdx.edu

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

Abstract. The expressive power of functional logic languages supports high-level specifications as well as efficient implementations of problems in the same language. If specifications are executable, they can be used both as initial prototypical implementations and as contracts for checking the reliable execution of implementations intended to satisfy the specification. In this paper, we propose a practical framework to support this general approach to coding. We discuss the notions of specifications and contracts for functional logic programming and present a tool that supports the development of declarative programs based on these notions.

1 Introduction

Functional logic programming languages [3,15] support a wide spectrum of programming styles. One can apply logic programming features like nondeterminism and logic variables to specify the basic knowledge about a problem and let the run-time system search for appropriate solutions. Or one can use a deterministic (functional) programming style to implement sophisticated and efficient algorithms [22].

The combination of both styles can be leveraged for increased reliability: high-level ("obviously correct") specifications can be formulated as functional logic programs. Since these specifications are executable, they can serve as initial prototypical implementations. Executable specifications are useful to run experiments which may expose defects and ultimately raise the confidence that a specification captures the intent. If the direct execution of the specification is too inefficient, one can choose more efficient data structures (e.g., balanced search trees instead of lists) and/or better algorithms for production software. In this case, the initial specification remains valuable since one can use it as an oracle to test the implementation on a large set of test data [8,13] or to check, via run-time assertions, that the implementation behaves as intended on particular executions.

In this paper we show the feasibility of this idea by formalizing specifications, contracts, and assertions, by showing some important relations between them, and by providing tools to support this approach to program design and development. The concrete language for our presentation is the multi-paradigm declarative language Curry [17]. We demonstrate that Curry can be used as a wide-spectrum language [5] for software

development. In particular, we have implemented a tool that either transforms a specification into an executable program or, if the implementation of the specification is also provided, into a contract attached to this implementation.

Although we assume familiarity with the general concepts of functional logic programming [3,15], we review in the next section the concepts crucial for this paper. Section 3 presents the fundamental notions of our framework. The corresponding tool support is sketched in Section 4 together with some examples.

2 Functional Logic Programming and Curry

The declarative multi-paradigm language Curry [17] extends non-strict functional programming languages such as Haskell [23] with logic programming features, e.g., nondeterminism and equational constraints. Consequently, Curry has a Haskell-like syntax[3] extended by the possible inclusion of free (logic) variables in conditions and right-hand sides of defining rules. The operational semantics is based on an optimal evaluation strategy [1] which is a conservative extension of lazy functional programming and (concurrent) logic programming.

Expressions in Curry programs contain *operations* (defined functions), *constructors* (introduced in data type declarations), and *variables* (arguments of operations or free variables). The goal of a computation is to obtain a value of some expression, where a *value* is an expression that does not contain any operation. Note that in a functional logic language expressions might have more than one value due to nondeterministically defined operations. For instance, Curry contains a *choice* operation defined by:

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

Thus, the expression "`0 ? 1`" has two values: `0` and `1`. If expressions have more than one value, these values are typically constrained by conditions in the rules defining operations according to the program intent. A *rule* has the form "$f\ t_1 \ldots t_n$ `|` c `=` e" where c is a *constraint*, i.e., an expression of the built-in type `Success`. For instance, the trivial constraint `success` is a value of type `Success` that denotes the always satisfiable constraint. An *equational constraint* e_1 `=:=` e_2 is satisfiable if both sides e_1 and e_2 are reducible to unifiable values. Furthermore, if c_1 and c_2 are constraints, c_1 `&` c_2 denotes their concurrent conjunction (i.e., both constraints are concurrently evaluated) and c_1 `&>` c_2 denotes their sequential conjunction (i.e., c_2 is evaluated after the successful evaluation of c_1).

Nondeterministic expressions could cause a semantical ambiguity when bound to variables. Consider the operations

```
coin = 0 ? 1
double x = x + x
```

Standard term rewriting produces, among others, the derivation

```
double coin  →  coin + coin →  0 + coin →  0 + 1 →  1
```

[3] Variables and function names usually start with lowercase letters and the names of type and data constructors start with an uppercase letter. The application of f to e is denoted by juxtaposition ("$f\ e$").

whose result is unintended. Therefore, González-Moreno et al. [14] proposed the rewriting logic CRWL as a logical foundation for declarative programming with non-strict and nondeterministic operations. This logic specifies the *call-time choice* semantics [18] where values of the arguments of an operation are determined before the operation is evaluated. In a lazy strategy, this is naturally obtained by sharing. For instance, the two occurrences of `coin` in the derivation above are shared so that "`double coin`" has only the results: `0` or `2`. Since standard term rewriting does not conform to the intended call-time choice semantics, other notions of rewriting have been proposed to formalize this idea, like graph rewriting [11,12] or let rewriting [19]. For our purposes, it is sufficient to use a simple reduction relation that we sketch without giving all details (which can be found in [19]).

To cover non-strict computations, expressions can also contain the special symbol $\bot$ to represent *undefined or unevaluated values*. A *partial value* is a value containing occurrences of $\bot$. A *partial constructor substitution* is a substitution that replaces variables by partial values. A *context* $\mathcal{C}[\cdot]$ is an expression with some "hole". Then the reduction relation we use throughout this paper is defined as follows (conditional rules are not considered for the sake of simplicity):

$$\begin{array}{rcll} \mathcal{C}[f\ \sigma(t_1) \ldots \sigma(t_n)] & \to & \mathcal{C}[\sigma(r)] & f\ t_1 \ldots t_n \to r \text{ program rule,} \\ & & & \sigma \text{ partial constructor substitution} \\ \mathcal{C}[e] & \to & \mathcal{C}[\bot] & \end{array}$$

The first rule models the call-time choice: if a rule is applied, the actual arguments of the operation must have been evaluated to partial values. The second rule models non-strictness by allowing the evaluation of any subexpression to an undefined value (which is intended if the value of this subexpression is not demanded). As usual, $\stackrel{*}{\to}$ denotes the reflexive and transitive closure of this reduction relation. The equivalence of this rewrite relation and CRWL is shown in [19].

Sometimes we use `let`-expressions to enforce the call-time choice semantics. In order to avoid the explicit handling of `let`-expressions in the reduction relation (as proposed in [19]), we consider `let`-expressions as syntactic sugar for auxiliary functions. For instance, the definition

```
f x = let z = coin*x in z+coin
```

is syntactic sugar for

```
f x = g (coin*x)
g z = z+coin
```

where `g` is a fresh name.

In nondeterministic programming, it is sometimes useful to examine the set of all the values of some expression. A "set-of-values" operation applied to an arbitrary argument might produce results that depend on the degree of evaluation of the argument (see [6] for a detailed discussion). *Set functions* overcome this problem [2]. For each defined function f, $f_{\mathcal{S}}$ denotes the corresponding set function. $f_{\mathcal{S}}$ encapsulates the nondeterminism of f, but excludes the potential nondeterminism of the arguments to which f is applied. For instance, consider the operation `negOrPos` defined by:

```
negOrPos x = -x ? x
```

Then "`negOrPos`$_{\mathcal{S}}$ `2`" evaluates to the set $\{$`-2,2`$\}$, i.e., the nondeterminism originating from `negOrPos` is encapsulated into a set. However, "`negOrPos`$_{\mathcal{S}}$ `(1?2)`" evaluates to two different sets $\{$`-1,1`$\}$ and $\{$`-2,2`$\}$ due to its nondeterministic argument, i.e., the nondeterminism originating from the argument produces different sets. The type *set* is abstract, i.e., the implementation is hidden, but there are operations, e.g., to determine whether a set is empty, `isEmpty`, or an element belongs to a set.

3 Specifications and Contracts

Our framework to support the development of reliable declarative programs is based on the idea of using a single language for specifications, contracts, and implementations. Specifications differ from programs because they may be nondeterministic and/or refer to existentially quantified quantities. A functional logic language such as Curry is appropriate to express specifications because it is nondeterministic and it has equation-solving capabilities.

Using the same language makes specifications and implementations similar. In fact, a specification is like any other operation but with a specific tag so that the specification is more versatile:

- If there is only a specification but no implementation of an operation, the specification can be used as an initial implementation for this operation.
- If there are both a specification and an implementation of an operation, the specification can be used to check the implementation in two different ways:
  Dynamic checking: If the implementation computes some result when the operation is executed, test whether this result conforms to the specification.
  Static checking: If one formally proves that the implementation is correct w.r.t. the specification, run-time checking is not necessary.

We distinguish between a specification and a contract for an operation. A *specification* describes precisely the intended meaning of an operation. However, a *contract* describes conditions that must be satisfied by the implementation. These conditions can be weaker than a specification. Contracts have been introduced in the context of imperative and object-oriented programming languages [21] to improve the quality of software. Typically, a contract consists of both a pre- and a postcondition. The *precondition* is an obligation for the arguments of an operation application. The *postcondition* is an obligation for both the arguments of an operation application and the result of the operation application to those arguments. Intuitively, the application of or call to each operation must satisfy its precondition, and, if both the precondition is satisfied and the operation returns a result, this result must satisfy the postcondition. When a contract is checked at run-time, the pre- and postcondition are called *assertions*.

Specifications, preconditions, and postconditions are independent notions separately useful for software development. A precondition for an operation states general restrictions on arguments that must be satisfied in order to apply this operation. Hence, a specification is intended only for inputs satisfying the precondition. Likewise, a postcondition must only be satisfied for these inputs. In a strongly typed language, a type restriction on arguments can be considered a precondition. In general, one is interested

in preconditions that are more expressive than a traditional type system. For instance, a precondition for a factorial function could require the argument to be non-negative. A postcondition is some requirement on all the results of an operation. It could be a type restriction, but it could also be much stronger. For instance, a postcondition for an operation to sort a list of values could state that the length of the output list is identical to the length of the input list. If a postcondition specifies all and only the intended results of an operation, it can be considered a specification. As we will see later, we can exploit the logic programming features of our language to execute a postcondition as a prototypical implementation by generating result values satisfying the postcondition.

The following definition fixes the notions discussed so far. For the sake of simplicity, we formally define our notions only for unary operations, but the extension to operations with several arguments is straightforward and, thus, it will be used in the subsequent examples.

Definition 1 (Specification, Contract). *Let f be an operation of type $\tau \to \tau'$. 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`. A* postcondition *for f is an operation f^{post} of type $\tau \to \tau' \to$ `Bool`. A precondition and postcondition pair is also called a* contract *for the operation. If a precondition is not explicitly defined, the most general precondition "f^{pre} `_ = True`" is assumed.*

Similarly to other proposals for assertions or contracts for functional (logic) programs (e.g., [7,9,16]), we define pre- and postconditions as Boolean-valued functions. An exception is [4] where constraints are used as conditions which was motivated by the use of postconditions as specifications instead of an unequivocal specification as in this work.

As an example, consider an operation, `sort`, to sort a list of integers. The type of `sort` is:

```
sort :: [Int] → [Int]
```

Since we have no further requirements on arguments (apart from its type), our precondition for `sort` is the constant operation[4]

```
sort'pre :: [Int] → Bool
sort'pre _ = True
```

As an example for a postcondition, we require that the length of the input and output lists must be equal:

```
sort'post :: [Int] → [Int] → Bool
sort'post xs ys = length xs == length ys
```

However, an unequivocal specification states that the result of `sort` is a permutation in ascending order of its input:

```
sort'spec :: [Int] → [Int]
sort'spec xs | sorted ys = ys  where ys = perm xs
```

This specification requires the definition of permutations and sorted lists which are easily formalized in Curry ("`<=:`" denotes the less-than-or-equal-to constraint):

[4] Note that in the concrete syntax we use in our tool (see below) we write f`'pre` instead of f^{pre} (and similarly for postconditions and specifications).

```
perm []     = []
perm (x:xs) = ndinsert x (perm xs)
  where ndinsert x ys     = x : ys
        ndinsert x (y:ys) = y : ndinsert x ys

sorted []       = success
sorted [_]      = success
sorted (x:y:ys) = x<=:y & sorted (y:ys)
```

We can use the specification `sort'spec` to sort lists since it is a Curry program and, as such, executable. Obviously, it is inefficient for large lists, so we implement it more efficiently using the well-known quicksort algorithm:

```
sort :: [Int] → [Int]
sort []     = []
sort (x:xs) = sort (filter (<x) xs) ++ [x] ++ sort (filter (>x) xs)
```

If we apply our tool, DSDCurry, to this program, the specification is transformed into an additional postcondition and all existing pre- and postconditions are attached to the `sort` operation for dynamic assertion checking. The assertions checked during the execution of this transformed program reveal an error in our implementation:

```
SortC> sort [5,1,2,6,5,3]
ERROR: Postcondition of operation 'sort' violated for:
[5,1,2,6,5,3] → [1,2,3,5,6]
```

If we correct the error, by replacing the condition `(>x)` with `(>=x)`, the transformed program executes as intended and without error messages.

Before discussing some details of our tool, we have to define the precise meaning of correct implementations and violated assertions. In imperative or strict functional languages, this seems obvious. However, in a functional logic language like Curry, operations might have multiple results or reduce to infinite structures (i.e., their evaluation does not terminate). In order to support contract checking also in these situations, we have to prepare an appropriate setup.

First, we consider the possible violation of contracts. Obviously, a precondition f^{pre} is violated for some expression e if $f^{pre}\ e$ is reducible to `False`, since we want to avoid any calls on operations where the argument does not satisfy the precondition. For postconditions, the situation is less clear for nondeterministic functions. Consider a value v such that $f^{pre}\ v$ is reducible to `True`, $f\ v \xrightarrow{*} v_1$, $f\ v \xrightarrow{*} v_2$, and $f^{post}\ v\ v_1 \xrightarrow{*}$ `True`, but $f^{post}\ v\ v_2 \xrightarrow{*}$ `False`, i.e., one result, v_1, satisfies the postcondition but another result for the same input, v_2, does not satisfy the postcondition. In a complete implementation, all results of an operation could be produced. Therefore, we propose the strong view that *any* result that a function produces must satisfy the function's postcondition.

Definition 2 (Violation). *Let f be an operation of type $\tau \to \tau'$, f^{pre} and f^{post} be pre- and postconditions for f, and e an expression of type τ. A* violation of the precondition f^{pre} of f at e *is a derivation of $f^{pre}\ e$ to* `False`*. A* violation of the postcondition f^{post} of f at e *is a derivation of*

```
let x = e in not (f^pre x) || f^post x (f x)
```

to `False`*, where* `x` *is a fresh variable.*

The definition of a postcondition violation considers the fact that a violation should be reported only if the precondition holds for the given argument. Note that the

`let`-expression is reasonable for nondeterministic arguments since the condition "`not (`f^{pre} e`) || `f^{post} e `(`f e`)`" is different from the one given in the above definition if e is nondeterministic. For instance, consider

```
id'post x y = x==y
id x = x
e = 0 ? 1
```

Then "`id'post e (id e)`" reduces to both `True` and `False` whereas

```
let x = e in id'post x (id x)
```

cannot reduce to `False` due to the call-time choice semantics. The intent is that the postcondition should be satisfied for the same values used in the precondition; thus, our definition captures this demand.

Next we have to define the correctness of an implementation w.r.t. a given specification. A simple approach could require that the *values* of the specification are all and only the *values* of the implementation. However, this is not reasonable for non-strict languages. For instance, consider

```
nums'spec n = n : nums'spec (n+1)
```

Since `nums'spec` does not reduce to a value (its evaluation does not terminate), any other operation (of the same type) that does not reduce to a value would be correct w.r.t. this specification, e.g.:

```
nums n = n : nums n
```

Obviously, this is not intended. If we put the specification and the implementation in an identical context (e.g., by applying "`take 2`" to `nums'spec` and `nums`), then we might obtain different results. This motivates the following definition.

Definition 3 (Equivalence, Correctness). *Let f_1, f_2 be operations of type $\tau \to \tau'$. f_1 is* equivalent *to f_2 iff, for any expression E_1, $E_1 \xrightarrow{*} v$ iff $E_2 \xrightarrow{*} v$, where v is a value and E_2 is obtained from E_1 by replacing any occurrence of f_1 with f_2. An implementation f is* correct *w.r.t. a specification f^{spec} iff f and f^{spec} are equivalent when applied to expressions satisfying f^{pre}.*

The correctness of an implementation w.r.t. a specification imposes an equality of two sets of result values. The implementation could produce a value more or less times than the specification in the sense that the same expression has "substantially" distinct derivations to the same value. Furthermore, equivalent operations could differ in contexts that do not yield any result. For instance, the evaluation of one operation could diverge where an equivalent operation might terminate with a failure or some exception.

Intuitively, two operations are equivalent if it is impossible to detect any difference between them in any application context. If operations do not produce values or produce some values as well as failures, the consideration of an application context is important. For instance, consider the following alternative implementation of sorting a list based on an operation `idSorted` that is the identity on sorted lists:

```
sort' xs = idSorted (perm xs)
  where idSorted []  = []
        idSorted [x] = [x]
        idSorted (x:y:ys) | x<=y = x : idSorted (y:ys)
```

Although this implementation only returns values that are sorted lists, it is not correct w.r.t. the specification `sort'spec`. For instance, consider the operation `head` that returns the first element of the list. Then there is a derivation

$$\begin{aligned}\texttt{head (sort' [2,3,1])} &\xrightarrow{*} \texttt{head (idSorted [2,3,1])}\\ &\xrightarrow{*} \texttt{head (2 : idSorted [3,1])}\\ &\xrightarrow{*} \texttt{2}\end{aligned}$$

whereas "`head (sort'spec [2,3,1])`" cannot be reduced to `2`. The implementation `sort'` is incorrect with respect to the specification of sort: if we want to compute the minimum of a list by sorting the list and taking the first element, the previous derivation shows that we obtain an unintended result.

Specifications can be used to verify programs. This is a complex task that could be supported by proof systems. In this paper we exploit the property that specifications are executable so that we can use them to detect an incorrect execution of the implementation. For this purpose, we use a specification as a contract for an implementation. Thus, if we detect a violation at run-time, we can deduce that the implementation is not correct. This demands for a postcondition that is generated from a specification. In a naive approach, we could try to define such a postcondition as

$$f^{post}\ \texttt{x y = y} \in f_{\mathcal{S}}^{spec}\ \texttt{x}$$

i.e., the postcondition checks whether the actual result is in the set of all the results according to the specification. Unfortunately, this simple definition does not work as intended due to the following problems:

1. For partially defined operations, this postcondition could be violated even though the implementation is correct. For instance, consider the simple example

   ```
   head'spec (x:_) = x
   head (x:_) = x
   ```

   Obviously, `head` is correct w.r.t. `head'spec`. However, the set $\texttt{head'spec}_{\mathcal{S}}\ \texttt{[]}$ is empty so that the condition "$\texttt{head []} \in \texttt{head'spec}_{\mathcal{S}}\ \texttt{[]}$" could reduce to `False`. Therefore, this condition should be checked only if the actual result is a value and not a failure. However, the implementation of "$\in$" may not require the evaluation of its left argument when its right argument is empty.[5]
2. The membership test requires the decision that two entities are equal. Since in functional logic languages, this test is evaluated by strict equality on (finite) values, the test will never be successful for operations delivering infinite structures.

The first problem can be handled by the addition of an equality test "`y==y`". Since the equality "`==`" compares *values*, the test is successful only if `y` is a value. This has the consequence that postconditions are not checked for failure cases. From a conceptual point of view, it would be better to exclude such cases by appropriate preconditions. Since the test for such an exclusion is undecidable in general, we add this sufficient condition to the postcondition.

The second problem can be handled in part by avoiding the comparison of complete results, and comparing only some computed parts, instead. For this purpose, we define a postcondition that is parametric w.r.t. some observation operation g.

[5] Although this problem can be avoided by excluding the application `head []` using an appropriate precondition, in general it is difficult to avoid failing computations by preconditions.

Definition 4. *Let f^{spec} be a specification of type $\tau \to \tau'$ and g an operation of type $\tau' \to \tau''$. The* postcondition f^{post}_g generated from f^{spec} *w.r.t. g is defined by*

```
f^post_g x y = let z = g y
                   g' a = g (f^spec a)
               in  z==z && z ∈ g'_S x
```

If we use $g = \texttt{id}$ (the identity function), the generated postcondition checks whether a result `y` is a value and it is contained in the set of all the results according to the specification. For instance, consider

```
f'spec = 0 ? 1
f = 1 ? 0
```

The generated postcondition $f^{post}_{\texttt{id}}$ requires that each value of the implementation `f` is contained in the set $\{0, 1\}$.

If we know that a specification is deterministic, i.e., it yields at most one result for a given input, then we can provide a simpler postcondition without using an observation operation and set functions:

```
f^post x y = y == f^spec x
```

Although this definition does not support the detection of violations for failed computations (if the evaluation of `y` fails, the evaluation of `f^post x y` also fails so that it will never reduce to `False`), it might report violations when computing infinite structures, if the equality is checked in a demand-driven manner (e.g., the expression `[1..]==[2..]` evaluates to `False`). Hence, this optimized formulation is supported by our tool.

The use of a postcondition generated from a specification to check an implementation is justified by the following propositions. The first proposition shows that equivalent operations have the same violations.

Proposition 1. *Let f^{post} be a postcondition for f. If f is equivalent to f' and there is a violation of the postcondition f^{post} for f at e, then there is also a violation of the postcondition f^{post} for f' at e.*

The next proposition shows that any postcondition f^{post} derived from a specification f^{spec} cannot cause any violation when f^{post} is used to check an execution of f^{spec}.

Proposition 2. *If f^{post}_g is the postcondition generated from f^{spec} w.r.t. some operation g, then there is no e such that there is a violation of the postcondition f^{post}_g for f^{spec} at e.*

As a consequence, we can use the postcondition generated from f^{spec} to detect an incorrect implementation:

Corollary 1. *Let f^{post}_g be the postcondition generated from f^{spec} w.r.t. some operation g. If there is a violation of f^{post}_g for f at e, then f is not correct w.r.t. f^{spec}.*

Similarly to testing, the correctness of an implementation cannot be determined by individual executions of a program. Nevertheless, we can infer from a satisfied postcondition which is generated from f^{spec} and an observation operation g that the observed part of the computation is correct w.r.t. the specification:

Proposition 3. *Let f_g^{post} be the postcondition generated from f^{spec} w.r.t. some operation g and e an expression such that f_g^{post} `e (f e)` $\stackrel{*}{\rightarrow$ `True`. Then there is a value s with g `(f e)` $\stackrel{*}{\rightarrow}$ s and g `(`f^{spec} `e)` $\stackrel{*}{\rightarrow}$ s.*

Now we are ready to put this theoretical framework into a tool to support the development of reliable declarative programs.

4 Tool Support

In this section we discuss a tool, DSDCurry[6], based on the ideas described in the previous sections. Basically, the tool transforms a Curry module M containing specifications, pre- and/or postconditions for some operations into a new Curry module MC providing the same interface, but where some operations are checked against the provided specifications and/or contracts. Providing specifications and/or contracts is not mandatory. However, when they are provided, they are used as follows in the transformed module:

- If there is a specification f^{spec}, then a corresponding postcondition is generated according to Definition 4 (if an observation operation is not provided by the programmer, the identity function `id` is used for g). If there is also a user-defined postcondition, it is combined with the generated postcondition by conjunction.
- If there is only a specification f^{spec} but no implementation[7] of operation f is provided, then an implementation for f is generated by the rule f `=` f^{spec}.
- If there is neither a specification nor an implementation but a postcondition f^{post} for some operation f, the postcondition is used as a (weak) specification for f, i.e., an initial implementation is generated for f by the following definition:

  f `x |` f^{post} `x y = y where y free`
- If there is a contract f^{pre}/f^{post} for some operation f, the implementation of f is replaced by

  f `x | checkPre "`f`" (`f^{pre} `x) &> checkPost "`f`" (`f^{post} `x y)`
  `  = y`
  `where y =` f`' x`
  `  `f`' ...`

  where "f' ..." contains the original definition of f with every occurrence of f replaced by f'. Thus, the original interface of any function is preserved by DSDCurry. The auxiliary operations `checkPre` and `checkPost` produce an error message if their second argument evaluates to `False`. For instance, `checkPre` is defined by:[8]

```
checkPre fname checkresult =
  if checkresult then success else
   error ("Precondition of operation '"++fname++"' violated!")
```

[6] The tool together with more examples is available at:
`http://www.informatik.uni-kiel.de/~pakcs/dsdcurry/`.

[7] An operation defined by the rule "f `= unknown`" is considered as undefined. Such a vacuous definition might be necessary if f is referenced in the definition of other operations in M.

[8] The actual implementation provides more information, e.g., about the concrete arguments of the pre- and postcondition.

The postcondition checker, `checkPost`, is similarly defined. Note that the pre- and postcondition checkers are constraints rather than Boolean operations. This is useful for lazy assertion checking [16] since constraints can be concurrently evaluated.

We demonstrate the development of a simple program using DSDCurry. Consider the specification `sort'spec` and the contract `sort'pre`/`sort'post` for sorting a list as shown in Section 3. According to Definition 4, the specification and postcondition are combined into a new postcondition of the form

```
sort'post x y = sort'post'org x y && y == y && y ∈ sort'spec_S x
  where sort'post'org xs ys = length xs == length ys
```

where `sort'post'org` is the original, user-supplied postcondition. If we do not provide any implementation of the operation `sort`, an implementation is generated from its specification where contract checking is added:

```
sort x | checkPre "sort" (sort'pre x)
         &> checkPost "sort" (sort'post x y)
       = y
  where y = sort'spec x
```

In principle, postcondition checking should be superfluous for specifications since any user-defined postcondition should be a logical consequence of the specification. Nevertheless, it is included since this entailment is not checked at compile time by our tool.

This prototypical implementation is not efficient because it does not exploit any knowledge about sorting algorithms developed over decades of research in computer science. We improve the efficiency of this implementation by adopting one of these algorithms known as *straight selection sort*. Informally, a list is sorted by selecting its smallest element, sorting the remaining elements, and placing the smallest element in front of the sorted remaining elements. If we know how to select the smallest element of a list, the implementation of this sort method is straightforward by a case distinction on the form of the input list:

```
sort [] = []
sort (x:xs) = min : sort rest    where (min,rest) = minRest (x:xs)
```

Here, we assume that the essential operation of selecting the smallest element is encoded by the operation `minRest` that, for a non-empty input list, returns both the smallest element and the remaining elements. Since finding the smallest element is a non-trivial task, we define a contract for `minRest`:

```
minRest'pre = not . null

minRest'post xs (min,rest) = (min:rest) ∈ perm_S xs && all (>= min) xs
```

The precondition requires that `minRest` is only applied to non-empty lists. Since there might be different methods to select a minimal element and return the remaining ones, we do not put any requirements on the order of the remaining elements in the postcondition, hence `(min:rest)` is some permutation of the input list. This is also the reason why it would be too restrictive to provide a specification of `minRest`. However, we can use the postcondition as an initial implementation.[9] This implementation of `minRest` has the undesirable consequence of producing many values, i.e., the minimal element

[9] In this case, we slightly change the postcondition and replace the Boolean operation "$\in$" by a constraint since the equality test implicitly performed by "$\in$" suspends on free variables [17].

together with all permutations of the remaining elements. We can either restrict this implementation to return only one value and ignore the others (for this reason, DSDCurry has an option to enforce this behavior), or provide a more informed implementation of the operation `minRest` as follows.

A direct implementation of `minRest` could be obtained via two auxiliary operations, `min` and `del`, that return the minimal element of a list and delete an occurrence of an element in a list, respectively:

```
minRest (x:xs) = let m = min x xs
                 in (m, del m (x:xs))
 where min x [] = x
       min x (y:ys) = if x<=y then min x ys else min y ys

       del x (y:ys) = if x==y then ys else y : del x ys
```

If we transform this augmented program with DSDCurry, it works as intended without any contract violation. We observe that our implementation of `minRest`, in the worst case, performs two traversals of the input list, whereas it is possible to compute the minimal element and the remaining elements with a single traversal. To improve the performance, we re-code `minRest` as

```
minRest (x:xs) = mr x [] xs
  where mr m r [] = (m,r)
        mr m r (y:ys) = if m<=y then mr m (y:r) ys else mr y (m:r) ys
```

This implementation is more efficient, but also more complicated and its correctness is not as apparent as before. Thus, we apply again our transformation tool to integrate the contract into this implementation and execute the program to increase our confidence in its correctness. Now that we are satisfied with the implementation, we could attempt a formal correctness proof of this implementation. However, this is outside the scope of this paper.

As a further example, consider a program to compute the infinite list, `fibs`, of all the Fibonacci numbers. The specification maps the operation, `fib`, to compute the n-th Fibonacci number defined by the immediate recursive definition, onto the list of all naturals:

```
fibs'spec = map fib [0..]
  where fib n | n == 0    = 0
              | n == 1    = 1
              | otherwise = fib (n-1) + fib (n-2)
```

The application of DSDCurry immediately gives us a correct implementation of `fibs` from this specification, e.g., the expression "`take 10 fibs`" reduces to `[0,1,1,2,3,5,8,13,21,34]`. Since each number in the list is computed by applying operation `fib`, the implementation is quite inefficient due to the exponential complexity of `fib`. Hence, we improve the implementation and construct the list (in linear time) by creating the next element by adding the two previous ones:

```
fibs = fiblist 0 1  where fiblist x y = x : fiblist (x+y) y
```

When we execute "`take 10 fibs`" again after transforming our program with DSDCurry, a violation is reported for the third element, `2`, of the result list. We made a typical error in iterative definitions by swapping some arguments. If we correct the program to

```
fibs = fiblist 0 1  where fiblist x y = x : fiblist y (x+y)
```

and transform and run it again, no more violations are reported.

Contract checking in the presence of infinite structures requires the lazy evaluation of assertions. Thus, our simple implementation where the contract is completely checked in the condition of an operation would lead to an infinite loop in the transformed `fibs` operation. In general, the eager or strict checking of assertions might influence the execution behavior of a program. To avoid this problem, Chitil et al. [7] proposed *lazy assertions*. Lazy assertions do not evaluate their arguments, but check them when they become evaluated by the application program. Thus, as long as every assertion is satisfied, program executions with or without lazy assertion checking deliver the same results.

On the other hand, lazy assertion checking might not detect contract violations if the assertion arguments are not sufficiently evaluated by the main program. Thus, it is debatable whether full assertion checking should be avoided in order to preserve the behavior of programs [9,16]. Lazy assertions do not modify the behavior, but a lazily computed result cannot be trusted as long as some assertion has not been checked. As a compromise between these conflicting goals, *enforceable assertions* are proposed in [16]. These assertions behave like lazy assertions, but they can also be checked upon an explicit request of the programmer, e.g., at the end of a program execution or at key intermediate execution points.

Making the appropriate choice might be dependent on the application or require some sophisticated program analysis. Therefore, DSDCurry supports strict, lazy, and enforceable assertions by transformation options so that it can be easily adapted to future insights.

5 Conclusions and Related Work

We have discussed some notions that are essential for a methodology intended to develop reliable declarative programs. Specifications are executable so that they can be used as initial prototypes as well as contracts for implementations that might later be developed. We have shown some relationships between these notions that are the basis of a transformation tool to support this development. Our tool, DSDCurry, transforms a specification into an initial implementation, if an implementation is not provided, otherwise it transforms the specification into a contract that checks the results computed by the implementation. Furthermore, our tool supports various forms of contract checking, such as eager, lazy, or enforceable assertions.

In principle, our method and tool support can be seen as a proposal to use Curry as a wide-spectrum language. In contrast to a wide-spectrum language like CIP-L [5] that supports the development of correct programs by applying a stepwise transformation process to specifications, our approach is more flexible. It does not guarantee correct implementations, but it allows very efficient implementations. The correctness is only checked at each concrete program execution w.r.t. some observation operation.

The use of contracts or assertions to obtain more reliable programs has been proposed for many programming languages and paradigms. Concepts for assertions in strict languages, like imperative, logic, or strict functional languages, are easier to handle

than in non-strict languages. For instance, [24] proposes an assertion language for (constraint) logic programming that is combined in [20] with a static verification framework. [10] considered a strict language with side effects and proposed the evaluation of assertions in parallel to the application program to exploit the power of multi-core computers. In non-strict languages, one has the option between lazy assertions [7], which do not change the meaning of a program (apart from reporting violated assertions) but might not report some violations, and strict assertions which could influence the evaluation order. Degen et al. [9] discussed the different approaches and came to the conclusion that there seems to be no way to satisfy both objectives, meaning preservation and violation reporting, in a non-strict language.

ESC/Haskell [25] is an approach to add pre- and postconditions to Haskell programs which are checked at compile time by sophisticated program transformations. Similarly to our approach, pre- and postconditions are arbitrary Boolean operations implemented in the source language. These conditions are considered as violated if the evaluation of an operation might fail due to incompletely defined operations (e.g., applying the operation `head` to the empty list). Such an interpretation of pre- and postconditions is too restrictive for functional logic languages where failures are used as a programming technique. Moreover, we distinguish between precise specifications and (weak) postconditions. For instance, [25] considers a sorting algorithm as verified if the output is a sorted list. We consider such a property as a weak postcondition whereas a precise specification should additionally require that the output is a permutation of the input list in order to exclude non-intended implementations.

An obvious challenge for future work is to provide proof support for contracts and specifications. If it can be shown at compile time that a contract is always satisfied by the corresponding implementation, its run-time checking can be omitted. This improves the efficiency of reliable software and reduces the need to test the developed software with large sets of test data [8,13]. Furthermore, a static proof guarantees the correctness of the implementation for all inputs rather than for particular executions.

References

1. S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. *Journal of the ACM*, 47(4):776–822, 2000.
2. 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.
3. S. Antoy and M. Hanus. Functional logic programming. *Communications of the ACM*, 53(4):74–85, 2010.
4. S. Antoy and M. Hanus. A transformation tool for functional logic program development. In *Proc. of the 24th Workshop on (Constraint) Logic Programming (WLP 2010)*, pages 23–33. German University of Cairo, 2010.
5. F.L. Bauer, M. Broy, R. Gnatz, W. Hesse, B. Krieg-Brückner, H. Partsch, P. Pepper, and H. Wössner. Towards a wide spectrum language to support program specification and program development. *ACM SIGPLAN Notices*, 13(12):15–24, 1978.
6. B. Braßel, M. Hanus, and F. Huch. Encapsulating non-determinism in functional logic computations. *Journal of Functional and Logic Programming*, 2004(6), 2004.

7. 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.
8. J. Christiansen and S. Fischer. EasyCheck - test data for free. In *Proc. of the 9th International Symposium on Functional and Logic Programming (FLOPS 2008)*, pages 322–336. Springer LNCS 4989, 2008.
9. M. Degen, P. Thiemann, and S. Wehr. True lies: Lazy contracts for lazy languages (faithfulness is better than laziness). In *4. Arbeitstagung Programmiersprachen (ATPS'09)*, pages 370; 2946–2259. Springer LNI 154, 2009.
10. 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.
11. R. Echahed and J.-C. Janodet. On constructor-based graph rewriting systems. Research report imag 985-i, IMAG-LSR, CNRS, Grenoble, 1997.
12. R. Echahed and J.-C. Janodet. Admissible graph rewriting and narrowing. In *Proc. Joint International Conference and Symposium on Logic Programming (JICSLP'98)*, pages 325–340, 1998.
13. S. Fischer and H. Kuchen. Systematic generation of glass-box test cases for functional logic programs. In *Proceedings of the 9th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'07)*, pages 75–89. ACM Press, 2007.
14. 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.
15. M. Hanus. Multi-paradigm declarative languages. In *Proceedings of the International Conference on Logic Programming (ICLP 2007)*, pages 45–75. Springer LNCS 4670, 2007.
16. M. Hanus. Lazy and enforceable assertions for functional logic programs. In *Proc. of the 19th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2010)*, pages 84–100. Springer LNCS 6559, 2011.
17. M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.8.2). Available at `http://www.curry-language.org`, 2006.
18. H. Hussmann. Nondeterministic algebraic specifications and nonconfluent term rewriting. *Journal of Logic Programming*, 12:237–255, 1992.
19. F.J. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández. A simple rewrite notion for call-time choice semantics. In *Proceedings of the 9th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'07)*, pages 197–208. ACM Press, 2007.
20. E. Mera, P. López-García, and M. Hermenegildo. Integrating software testing and run-time checking in an assertion verification framework. In *25th International Conference on Logic Programming (ICLP 2009)*, pages 281–295. Springer LNCS 5649, 2009.
21. B. Meyer. *Object-oriented Software Construction*. Prentice Hall, second edition, 1997.
22. C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.
23. S. Peyton Jones, editor. *Haskell 98 Language and Libraries—The Revised Report*. Cambridge University Press, 2003.
24. G. Puebla, F. Bueno, and M. Hermenegildo. An assertion language for constraint logic programs. In *Analysis and Visualization Tools for Constraint Programming*, pages 23–62. Springer LNCS 1870, 2000.
25. D.N. Xu. Extended static checking for Haskell. In *Proc. of the 36th ACM SIGPLAN Workshop on Haskell (Haskell 2006)*, pages 48–59, 2006.