1 `curry check`: A Tool for Testing Properties of Curry Programs

CurryCheck is a tool that supports the automation of testing Curry programs. The tests to be executed can be unit tests as well as property tests parameterized over some arguments. The tests can be part of any Curry source program and, thus, they are also useful to document the code. CurryCheck is based on EasyCheck [5]. Actually, the properties to be tested are written by combinators proposed for EasyCheck, which are actually influenced by QuickCheck [6] but extended to the demands of functional logic programming.

1.1 Testing Properties

To start with a concrete example, consider the following naive definition of reversing a list:

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

To get some confidence in the code, we add some unit tests, i.e., test with concrete test data:

```
revNull = rev []      -=- []
rev123  = rev [1,2,3] -=- [3,2,1]
```

The operator "`-=-`" specifies a test where both sides must have a single identical value. Since this operator (as many more, see below) are defined in the library `Test.Prop`,[1] we also have to import this library. Apart from unit tests, which are often tedious to write, we can also write a property, i.e., a test parameterized over some arguments. For instance, an interesting property of reversing a list is the fact that reversing a list two times provides the input list:

```
revRevIsId xs = rev (rev xs) -=- xs
```

Note that each property is defined as a Curry operation where the arguments are the parameters of the property. Altogether, our program is as follows:

```
module Rev(rev) where

import Test.Prop

rev :: [a] → [a]
rev []     = []
rev (x:xs) = rev xs ++ [x]

revNull = rev []      -=- []
rev123  = rev [1,2,3] -=- [3,2,1]

revRevIsId xs = rev (rev xs) -=- xs
```

Now we can run all tests by invoking the CurryCheck tool. If our program is stored in the file

[1]The library `Test.Prop` is a clone of the library `Test.EasyCheck` which defines only the interface but not the actual test implementations. Thus, the library `Test.Prop` has less import dependencies. When CurryCheck generates programs to execute the tests, it automatically replaces references to `Test.Prop` by references to `Test.EasyCheck` in the generated programs.

`Rev.curry`, we can execute the tests as follows:

```
> curry check Rev
...
Executing all tests...
revNull (module Rev, line 7):
 Passed 1 test.
rev123 (module Rev, line 8):
 Passed 1 test.
revRevIsId_ON_BASETYPE (module Rev, line 10):
 OK, passed 100 tests.
```

Since the operation `rev` is polymorphic, the property `revRevIsId` is also polymorphic in its argument. In order to select concrete values to test this property, CurryCheck replaces such polymorphic tests by defaulting the type variable to prelude type `Ordering` (the actual default type can also be set by a command-line flag). If we want to test this property on integers numbers, we can explicitly provide a type signature, where `Prop` denotes the type of a test:

```
revRevIsId :: [Int]  → Prop
revRevIsId xs = rev (rev xs) -=- xs
```

The command `curry check` has some options to influence the output, like "`-q`" for a quiet execution (only errors and failed tests are reported) or "`-v`" for a verbose execution where all generated test cases are shown. Moreover, the return code of `curry check` is `0` in case of successful tests, otherwise, it is `1`. Hence, CurryCheck can be easily integrated in tool chains for automatic testing.

In order to support the inclusion of properties in the source code, the operations defined the properties do not have to be exported, as show in the module `Rev` above. Hence, one can add properties to any library and export only library-relevant operations. To test these properties, CurryCheck creates a copy of the library where all operations are public, i.e., CurryCheck requires write permission on the directory where the source code is stored.

The library `Test.Prop` defines many combinators to construct properties. In particular, there are a couple of combinators for dealing with non-deterministic operations (note that this list is incomplete):

- The combinator "`<~>`" is satisfied if the set of values of both sides are equal.
- The property x `~>` y is satisfied if x evaluates to every value of y. Thus, the set of values of y must be a subset of the set of values of x.
- The property x `<~`y is satisfied if y evaluates to every value of x, i.e., the set of values of x must be a subset of the set of values of y.
- The combinator "`<~~>`" is satisfied if the multi-set of values of both sides are equal. Hence, this operator can be used to compare the number of computed solutions of two expressions.
- The property `always` x is satisfied if all values of x are true.
- The property `eventually` x is satisfied if some value of x is true.
- The property `failing` x is satisfied if x has no value, i.e., its evaluation fails.

- The property *x* `#` *n* is satisfied if *x* has *n* different values.

For instance, consider the insertion of an element at an arbitrary position in a list:

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

The following property states that the element is inserted (at least) at the beginning or the end of the list:

```
insertAsFirstOrLast :: Int → [Int] → Prop
insertAsFirstOrLast x xs = insert x xs ~> (x:xs ? xs++[x])
```

A well-known application of `insert` is to use it to define a permutation of a list:

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

We can check whether the length of a permuted lists is unchanged:

```
permLength :: [Int] → Prop
permLength xs = length (perm xs) <~> length xs
```

Note that the use of "`<~>`" is relevant since we compare non-deterministic values. Actually, the left argument evaluates to many (identical) values.

One might also want to check whether `perm` computes the correct number of solutions. Since we know that a list of length n has $n!$ permutations, we write the following property:

```
permCount :: [Int] → Prop
permCount xs = perm xs # fac (length xs)
```

where `fac` is the factorial function. However, this test will be falsified with the argument `[1,1]`. Actually, this list has only one permuted value since the two possible permutations are identical and the combinator "`#`" counts the number of *different* values. The property would be correct if all elements in the input list `xs` are different. This can be expressed by a conditional property: the property *b* `==>` *p* is satisfied if *p* is satisfied for all values where *b* evaluates to `True`. Therefore, if we define a predicate `allDifferent` by

```
allDifferent []     = True
allDifferent (x:xs) = x ‘notElem‘ xs && allDifferent xs
```

then we can reformulate our property as follows:

```
permCount xs = allDifferent xs ==> perm xs # fac (length xs)
```

Now consider a predicate to check whether a list is sorted:

```
sorted :: [Int] → Bool
sorted []       = True
sorted [_]      = True
sorted (x:y:zs) = x<=y && sorted (y:zs)
```

This predicate is useful to test whether there are also sorted permutations:

```
permIsEventuallySorted :: [Int]  → Prop
permIsEventuallySorted xs = eventually $ sorted (perm xs)
```

The previous operations can be exploited to provide a high-level specification of sorting a list:

```
psort :: [Int]  → [Int}
psort xs | sorted ys = ys
 where ys = perm xs
```

Again, we can write some properties:

```
psortIsAlwaysSorted xs = always $ sorted (psort xs)

psortKeepsLength xs = length (psort xs) <~> length xs
```

Of course, the sort specification via permutations is not useful in practice. However, it can be used as an oracle to test more efficient sorting algorithms like quicksort:

```
qsort :: [Int]  → [Int]
qsort []     = []
qsort (x:l)  = qsort (filter (<x) l) ++ x : qsort (filter (>x) l)
```

The following property specifies the correctness of quicksort:

```
qsortIsSorting xs = qsort xs <~> psort xs
```

Actually, if we test this property, we obtain a failure:

```
> curry check ExampleTests
...
qsortIsSorting (module ExampleTests, line 53) failed
Falsified by third test.
Arguments:
[1,1]
Results:
[1]
```

The result shows that, for the given argument `[1,1]`, an element has been dropped in the result. Hence, we correct our implementation, e.g., by replacing `(>x)` with `(>=x)`, and obtain a successful test execution.

For I/O operations, it is difficult to execute them with random data. Hence, CurryCheck only supports specific I/O unit tests:

- *a* `‘returns‘` *x* is satisfied if the I/O action *a* returns the value *x*.
- *a* `‘sameReturns‘` *b* is satisfied if the I/O actions *a* and *b* return identical values.

Since CurryCheck executes the tests written in a source program in their textual order, one can write several I/O tests that are executed in a well-defined order.

1.2 Generating Test Data

CurryCheck test properties by enumerating test data and checking a given property with these values. Since these values are generated in a systematic way, one can even prove a property if the number of test cases is finite. For instance, consider the following property from Boolean logic:

```
neg_or b1 b2 = not (b1 || b2) -=- not b1 && not b2
```

This property is validated by checking it with all possible values:

```
> curry check -v ExampleTests
...
0:
False
False
1:
False
True
2:
True
False
3:
True
True
neg_or (module ExampleTests, line 67):
 Passed 4 tests.
```

However, if the test data is infinite, like lists of integers, CurryCheck stops checking after a given limit for all tests. As a default, the limit is 100 tests but it can be changed by the command-line flag "`-m`". For instance, to test each property with 200 tests, CurryCheck can be invoked by

```
> curry check -m 200 ExampleTests
```

For a given type, CurryCheck automatically enumerates all values of this type (except for function types). In KiCS2, this is done by exploiting the functional logic features of Curry, i.e., by simply collecting all values of a free variable. For instance, the library `Test.EasyCheck` defines an operation

```
valuesOf :: a → [a]
```

which computes the list of all values of the given argument according to a fixed strategy (in the current implementation: randomized level diagonalization [5]). For instance, we can get 20 values for a list of integers by

```
Test.EasyCheck> take 20 (valuesOf (_::[Int]))
[[],[-1],[-3],[0],[1],[-1,0],[-2],[0,0],[3],[-1,1],[-3,0],[0,1],[2],
[-1,-1],[-5],[0,-1],[5],[-1,2],[-9],[0,2]]
```

Since the features of PAKCS for search space exploration are more limited, PAKCS uses in CurryCheck explicit generators for search tree structures which are defined in the module `SearchTreeGenerators`. For instance, the operations

```
genInt :: SearchTree Int

genList :: SearchTree a → SearchTree [a]
```

generates (infinite) trees of integer and lists values. To extract all values in a search tree, the library `Test.EasyCheck` also defines an operation

```
valuesOfSearchTree :: SearchTree a → [a]
```

so that we obtain 20 values for a list of integers in PAKCS by

```
...> take 20 (valuesOfSearchTree (genList genInt))
[[],[1],[1,1],[1,-1],[2],[6],[3],[5],[0],[0,1],[0,0],[-1],[-1,0],[-2],
[-3],[1,5],[1,0],[2,-1],[4],[3,-1]]
```

Apart from the different implementations, CurryCheck can test properties on predefined types, as already shown, as well as on user-defined types. For instance, we can define our own Peano representation of natural numbers with an addition operation and two properties as follows:

```
data Nat = Z | S Nat

add :: Nat → Nat → Nat
add Z     n = n
add (S m) n = S(add m n)

addIsCommutative x y = add x y -=- add y x

addIsAssociative x y z = add (add x y) z -=- add x (add y z)
```

Properties can also be defined for polymorphic types. For instance, we can define general polymorphic trees, operations to compute the leaves of a tree and mirroring a tree as follows:

```
data Tree a = Leaf a | Node [Tree a]

leaves (Leaf x) = [x]
leaves (Node ts) = concatMap leaves ts

mirror (Leaf x) = Leaf x
mirror (Node ts) = Node (reverse (map mirror ts))
```

Then we can state and check two properties on mirroring:

```
doubleMirror t = mirror (mirror t) -=- t

leavesOfMirrorAreReversed t = leaves t -=- reverse (leaves (mirror t))
```

In some cases, it might be desirable to define own test data since the generated structures are not appropriate for testing (e.g., balanced trees to check algorithms that require work on balanced trees). Of course, one could drop undesired values by an explicit condition. For instance, consider the following operation that adds all numbers from 0 to a given limit:

```
sumUp n = if n==0 then 0 else n + sumUp (n-1)
```

Since there is also a simple formula to compute this sum, we can check it:

```
sumUpIsCorrect n = n>=0 ==> sumUp n -=- n * (n+1) `div` 2
```

Note that the condition is important since `sumUp` diverges on negative numbers. CurryCheck tests this property by enumerating integers, i.e., also many negative numbers which are dropped for the tests. In order to generate only valid test data, we define our own generator for a search tree containing only valid data:

```
genInt = genCons0 0 ||| genCons1 (+1) genInt
```

The combinator `genCons0` constructs a search tree containing only this value, whereas `genCons1` constructs from a given search tree a new tree where the function given in the first argument is applied to all values. Similarly, there are also combinators `genCons2`, `genCons3` etc. for more than one argument. The combinator "`|||`" combines two search trees.

If the Curry program containing properties defines a generator operation with the name genτ, then CurryCheck uses this generator to test properties with argument type τ. Hence, if we put the definition of `genInt` in the Curry program where `sumUpIsCorrect` is defined, the values to check this property are only non-negative integers. Since these integers are slowly increasing, i.e., the search tree is actually degenerated to a list, we can also use the following definition to obtain a more balanced search tree:

```
genInt = genCons0 0 ||| genCons1 (\n  →  2*(n+1)) genInt
                    ||| genCons1 (\n  →  2*n+1)   genInt
```

The library `SearchTree` defines the structure of search trees as well as operations on search trees, like limiting the depth of a search tree (`limitSearchTree`) or showing a search tree (`showSearchTree`). For instance, to structure of the generated search tree up to some depth can be visualized as follows:

```
...SearchTree> putStr (showSearchTree (limitSearchTree 6 genInt))
```

If we want to use our own generator only for specific properties, we can do so by introducing a new data type and defining a generator for this data type. For instance, to test only the operation `sumUpIsCorrect` with non-negative integers, we do not define a generator `genInt` as above, but define a wrapper type for non-negative integers and a generator for this type:

```
data NonNeg = NonNeg { nonNeg :: Int }

genNonNeg = genCons1 NonNeg genNN
 where
   genNN = genCons0 0 ||| genCons1 (\n  →  2*(n+1)) genNN
                      ||| genCons1 (\n  →  2*n+1)   genNN
```

Now we can either redefine `sumUpIsCorrect` on this type

```
sumUpIsCorrectOnNonNeg (NonNeg n) = sumUp n -=- n * (n+1) `div` 2
```

or we simply reuse the old definition by

```
sumUpIsCorrectOnNonNeg = sumUpIsCorrect . nonNeg
```

1.3 Checking Contracts and Specifications

The expressive power of Curry supports writing high-level specifications as well as efficient implementations for a given problem in the same programming language, as discussed in [3]. If a specification or contract is provided for some function, then CurryCheck automatically generates properties to test this specification or contract.

Following the notation proposed in [3], a *specification* for an operation f is an operation f`'spec` of the same type as f. A *contract* consists of a pre- and a postcondition, where the precondition could be omitted. A *precondition* for an operation f of type $\tau \rightarrow \tau'$ is an operation

```
f'pre :: τ  →  Bool
```

whereas a *postcondition* for f is an operation

```
f'post :: τ  →  τ'  →  Bool
```

which relates input and output values (the generalization to operations with more than one argument is straightforward).

As a concrete example, consider again the problem of sorting a list. We can write a postcondition and a specification for a sort operation `sort` and an implementation via quicksort as follows (where `sorted` and `perm` are defined as above):

```
-- Postcondition: input and output lists should have the same length
sort'post xs ys = length xs == length ys

-- Specification:
-- A correct result is a permutation of the input which is sorted.
sort'spec :: [Int] → [Int]
sort'spec xs | ys == perm xs && sorted ys = ys  where ys free

-- An implementation of sort with quicksort:
sort :: [Int] → [Int]
sort []     = []
sort (x:xs) = sort (filter (<x) xs) ++ [x] ++ sort (filter (>=x) xs)
```

If we process this program with CurryCheck, properties to check the specification and postcondition are automatically generated. For instance, a specification is satisfied if it yields the same values as the implementation, and a postcondition is satisfied if each value computed for some input satisfies the postcondition relation between input and output. For our example, CurryCheck generates the following properties (if there are also preconditions for some operation, these preconditions are used to restrict the test cases via the condition operater "`==>`"):

```
sortSatisfiesPostCondition :: [Int] → Prop
sortSatisfiesPostCondition x =
  let r = sort x
  in (r == r) ==> always (sort'post x r)

sortSatisfiesSpecification :: [Int] → Prop
sortSatisfiesSpecification x = sort x <~> sort'spec x
```

1.4 Checking Equivalence of Operations

CurryCheck supports also equivalence tests for operations. Two operations are considered as *equivalent* if they can be replaced by each other in any possible context without changing the computed values (this is also called *contextual equivalence* and precisely defined in [3] for functional logic programs). For instance, the Boolean operations

```
f1 :: Bool → Bool                 f2 :: Bool → Bool
f1 x = not (not x)                f2 x = x
```

are equivalent, whereas

```
g1 :: Bool → Bool                 g2 :: Bool → Bool
g1 False = True                   g2 x = True
g1 True  = True
```

are not equivalent: `g1 failed` has no value but `g2 failed` evaluates to `True`.

To check the equivalence of operations, one can use the property combinator `<=>`:

```
f1_equiv_f2 = f1 <=> f2
g1_equiv_g2 = g1 <=> g2
```

The left and right argument of this combinator must be a defined operation or a defined operation with a type annotation in order to specify the argument types used for checking this property.

CurryCheck transforms such properties into properties where both operations are compared w.r.t. all partial values and partial results. The details are described in [4].

It should be noted that CurryCheck can test the equivalence of non-terminating operations provided that they are *productive*, i.e., always generate (outermost) constructors after a finite number of steps (otherwise, the test of CurryCheck might not terminate). For instance, CurryCheck reports a counter-example to the equivalence of the following non-terminating operations:

```
ints1 n = n : ints1 (n+1)

ints2 n = n : ints2 (n+2)

-- This property will be falsified by CurryCheck:
ints1_equiv_ints2 = ints1 <=> ints2
```

This is done by iteratively guessing depth-bounds, computing both operations up to these depth-bounds, and comparing the computed results. Since this might be a long process, CurryCheck supports a faster comparison of operations when it is known that they are terminating. If the name of a test contains the suffix `'TERMINATE`, CurryCheck assumes that the operations to be tested are terminating, i.e., they always yields a result when applied to ground terms. In this case, CurryCheck does not iterate over depth-bounds but evaluates operations completely. For instance, consider the following definition of permutation sort (the operations `perm` and `sorted` are defined above):

```
psort :: Ord a => [a] → [a]
psort xs | sorted ys = ys
  where ys = perm xs
```

A different definition can be obtained by defining a partial identity on sorted lists:

```
isort :: Ord a => [a] → [a]
isort xs = idSorted (perm xs)
 where idSorted []              = []
       idSorted [x]             = [x]
       idSorted (x:y:ys) | x<=y = x : idSorted (y:ys)
```

We can test the equivalence of both operations by specializing both operations on some ground type (otherwise, the type checker reports an error due to an unspecified type `Ord` context):

```
psort_equiv_isort = psort <=> (isort :: [Int] → [Int])
```

CurryCheck reports a counter example by the 274th test. Since both operations are terminating, we can also check the following property:

```
psort_equiv_isort'TERMINATE = psort <=> (isort :: [Int] → [Int])
```

Now a counter example is found by the 21th test.

Instead of annotating the property name to use more efficient equivalence tests for terminating operations, one can also ask CurryCheck to analyze the operations in order to safely approximate termination or productivity properties. For this purpose, one can call CurryCheck with the option "`--equivalence=`*equiv*" or "`-e`*equiv*". The parameter *equiv* determines the mode for equivalence checking which must have one of the following values (or a prefix of them):

`manual`: This is the default mode. In this mode, all equivalence tests are executed with first technique described above, unless the name of the test has the suffix `'TERMINATE`.

`autoselect`: This mode automatically selects the improved transformation for terminating operations by a program analysis, i.e., if it can be proved that both operations are terminating, then the equivalence test for terminating operations is used. It is also used when the name of the test has the suffix `'TERMINATE`.

`safe`: This mode analyzes the productivity behavior of operations. If it can be proved that both operations are terminating or the test name has the suffix `'TERMINATE`, then the more efficient equivalence test for terminating operations is used. If it can be proved that both operations are productive or the test name has the suffix `'PRODUCTIVE`, then the first general test technique is used. Otherwise, the equivalence property is *not* tested. Thus, this mode is useful if one wants to ensure that all equivalence tests always terminate (provided that the additional user annotations are correct).

1.5 Checking Usage of Specific Operations

In addition to testing dynamic properties of programs, CurryCheck also examines the source code of the given program for unintended uses of specific operations (these checks can be omitted via the option "`--nosource`"). Currently, the following source code checks are performed:

- The prelude operation "`=:<=`" is used to implement functional patterns [1]. It should not be used in source programs to avoid unintended uses. Hence, CurryCheck reports such unintended uses.
- Set functions [2] are used to encapsulate all non-deterministic results of some function in a set structure. Hence, for each top-level function f of arity n, the corresponding set function can be expressed in Curry (via operations defined in the library `SetFunctions`) by the application "`set`n f" (this application is used in order to extend the syntax of Curry with a specific notation for set functions). However, it is not intended to apply the operator "`set`n" to lambda abstractions, locally defined operations or operations with an arity different from n. Hence, CurryCheck reports such unintended uses of set functions.

References

[1] 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.

[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. 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.

[4] S. Antoy and M. Hanus. Equivalence checking of non-deterministic operations. In *Proc. of the 14th International Symposium on Functional and Logic Programming (FLOPS 2018)*, to appear in Springer LNCS.

[5] 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.

[6] K. Claessen and J. Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. In *International Conference on Functional Programming (ICFP'00)*, pages 268–279. ACM Press, 2000.