Programming Languages and Compiler Construction
Department of Computer Science
Christian-Albrechts-University of Kiel

Master's Thesis

CurryCheck - Unifying Curry's Testing Frameworks

Jan-Patrick Baye

2015

Advisors:
Prof. Michael Hanus
M.Sc. Sandra Dylus

Statutory Declaration

date	Jan-Patrick Baye

Abstract

With the growing complexity of software and its importance in vital parts of our infrastructure testing software becomes more and more vital. For this reason testing should be as simple as possible. In this thesis we try to design and implement a testing framework for Curry that is easy to use, provides all the functionalities of existing frameworks, and is readily extendable if new frameworks with additional functionality arise. To create this new framework, we will look at two existing ones for Curry, *CurryTest* and *EasyCheck*. We will analyse these two frameworks to find their assets and shortcomings and use both *EasyCheck* and *CurryTest* to implement our new framework.

Contents

Listings

1. Introduction

In this first chapter we will look at the motivations and goals of this thesis and finally give a short overview over the structure and content of the remaining chapters.

1.1. Motivation

In today's world software is all around us and more importantly controls many aspects of our lives. Be it as operating system on our smartphones or the controlling software of a nuclear reactor, we interact and depend on software as part of our lifestyle. Because software plays such a vital role in our society it is important that many programs work as reliable as possible. There are many reasons why software has to work reliable. Let us take a look at the website of an online shop. It might not be a big problem for a customer if the website does not work properly, he just might get annoyed and buys at another store. For the owner of the website, on the other hand, it might be a big problem. Because, if the site does not work properly, he will not sell anything. Thus, financial reasons are one argument for reliable software. A prominent example for a big financial loss due to software errors is the explosion of the Ariane 5 in 1996, which cost was valued at \$500 million[12].

Another reason for tests is safety. Since a malfunction in many software system would probably cost lives. Examples for these kind of safety critical system are the autopilot of an air plane or the coming autonomous driving cars.

For programmers the creation of tests has to be as simple as possible, as they are then more inclined to write tests. If writing or executing tests is hard, chances are that nobody writes any tests or the tests get not executed as often as necessary.

The idea behind this thesis is to create a new testing framework for Curry, with a uniform syntax and a way to automatically execute all written tests in a Curry module, that combines the features of existing frameworks.

1.2. Goal

As mentioned before, one of the goals is, to create a simple syntax for writing tests. Since tests basically boil down to comparing two values, a trivial way to write them would be similar to `func1 == func2`. Accordingly, the objective is to provide a small set of operators that allow us to write the tests in the same way. Listing 1.1 shows a simple test example using this kind of syntax.

Listing 1.1: simple test

```
lengthOfEmptyList = length [] -==- 0
```

EasyCheck provides the ability to define properties and test these. Since we want our syntax to be as simple as possible, we want to use the same operators for defining properties as well. Listing 1.2 shows an example of a property being defined using `(-==-)`.

Listing 1.2: simple property test

```
reversePreservesLength = length . reverse -==- length
```

Besides providing the possibility to define properties by using the form shown in Listing 1.2, we also want to provide the alternative way shown in Listing 1.3.

Listing 1.3: simple property test - alternative syntax

```
reversePreservesLength xs = length (reverse xs) -==- length xs
```

In chapter 4 we will take a more detailed look at the syntax used in this framework and also at the semantics, especially when dealing with non-determinism.

Besides the syntax, we want to take a look at where we write our tests. Oftentimes tests are written in separated modules like unit tests[2]. Splitting tests and functions has some disadvantages. Firstly, if a test fails, there is no direct way of linking the failed test to the tested function, more specifically the exact location (file, line), except if we hard code this information into the test. Secondly, a test does not only provide a good way of ensuring the quality of the tested function, it can also provide a good way of documenting a function. A simple example of this would be the addition of numbers. One property such an implementation should probably fulfil is commutativity, so it would be sensible to write a corresponding test. If we write our tests in a different module than our function, we would have to write a comment for our function detailing this property to properly document it. But then we have a new problem, since we need to keep track of a comment in one file (our program) that actually describes something in a different module (our test). Accordingly, one of our goals is to support the declaration of tests in the same module as the functions we want to test. This way, if we write our tests next to the function itself, we not only add additional documentation of the tested function, but also provide a better way of linking the test with the function. When a function fails, we can provide the file and line number of the failed test and the spacial locality allows us to review the test and the function directly.

Since we want to be able to write our tests in the same file as the functions we test, we want our tests to be valid Curry code. We then do not have to remove the tests before compiling the program, and we also have the benefit of the same syntax highlighting and type checking as everything else.

The last requirement we want to achieve is automatic test detection and execution. Because the tests are mixed with the main program, one goal is providing a program that finds and executes all tests in a module. Furthermore, it would be desirable that we do not need to export our tests. If we would need to export the tests, they would needlessly clutter the modules interface and hide the module's actually important interface functions. Moreover, when compiling the program, a good compiler could do a dead code analysis and remove all tests as they are neither locally used nor exported.

To summarize, the created framework should realize the following goals:

- simple syntax
- tests are valid Curry code
- provide all features of existing frameworks (*EasyCheck* and *CurryTest*)
- tests defined inside the same module
- automatic test detection and execution

1.3. Structure

The rest of this thesis is structured as follows. In chapter 2 and chapter 3 we will look at some preliminaries of this thesis. More precisely, we take a quick look at functional and logic programming (section 2.1 and section 2.2, respectively). We then provide a short introduction to Curry (section 2.3), as Curry is the language for which we provide a testing framework. In section 2.4 we glance at*AbstractCurry*, which provides meta-programming capabilities for Curry and is used extensively in the implementation. In chapter 3 we go through two existing testing frameworks for Curry, namely *CurryTest* (section 3.1) and *EasyCheck* (section 3.2). As our framework tries to combine and expand these two frameworks, we take a more detailed look at each framework's syntax and capabilities.

After handling the preliminaries in chapter 2 and chapter 3, we will then discuss the new framework CurryCheck in chapter 4 and chapter 5. Firstly we examine the provided syntax and semantics of *CurryCheck* in chapter 4 and then review the implementations as well as some modifications to *EasyCheck* in chapter 5.

Lastly we discuss some problems that remain in *CurryCheck* and how to resolve them in future versions in chapter 6 and finally finish with a short summary of which goals have been achieved in chapter 7.

2. Preliminaries

In this chapter we will look at the programming language Curry, its underlying concepts and introduce *AbstractCurry*, which provides a means to do meta-programming with Curry programs.

As Curry is a multi-paradigm language, first off we will quickly summarize these concepts, i.e. functional (section 2.1) and logic (section 2.2) programming. After that we will introduce the reader to Curry's non-determinism (section 3.1), as we have to deal with it semantically, when providing a test framework and at last say a few words about *AbstractCurry*. Readers already proficient with these concepts can safely skip ahead to chapter 3.

2.1. Functional Programming

Functional programming is one of the two main concepts underlying Curry. Functional programs are declarative programs, which means that a program is a set of function definitions. The programmer concentrates more on how the program should do its task on a logical layer and not so much in which order basic instruction are executed. Thus, a functional language abstracts from the concept of memory. In the rest of the document we expect the reader to be familiar with functional programming. Since Curry is heavily influenced by Haskell and many Haskell programs are indeed also valid Curry programs, we especially expect the reader to be seasoned in Haskell and its syntax.

2.2. Logic Programming

The second important paradigm behind Curry is logic programming. Logic programming is a declarative paradigm as well, but in contrast to functional programming logic programming does not only provide for functions but relations. A logic program is a set of facts and relations. The compiler has to provide a built-in search as a way to test these relations for possible solutions. This built-in search enables us to use many relations in multiple directions. The reader is expected to become adept with logic programming and its principal on his or her own, if not already familiar with it.

2.3. Curry

As mentioned in section 2.1, we will assume knowledge and familiarity with functional and logic programming and especially Haskell and its syntax. Concepts like algebraic data types, higher-order functions and pattern matching should be well understood, as we will only look at one feature Curry integrates from logic programming: non-determinism. The reason we will cover non-determinism is that our test framework has to deal with non-deterministic functions. So in order to test non-deterministic functions, we should understand what they are and how they work. All programs and examples in this document are compiled or executed with KiCS2 version 0.4.0.

2.3.1. Non-determinism in Curry

Just as in Haskell, a Curry function is a set of rules, but the evaluation of these rules differs a little bit, as we will see. Let us write a simple function that tests if a list is empty.

```
isEmpty :: [a] -> Bool
isEmpty []     = True
isEmpty (x:xs) = False
```

The first rule matches the empty list and thus returns `True`. The second rule, on the other hand, matches a non-empty list and consequently returns `False`. In Haskell we would probably rewrite that function definition to use the wild-card `_`, as we do not need the values of `x` or `xs`. In fact, if the second rule is tried, we know that the first one did not match and as such, the list has to be non-empty.

```
isEmpty' :: [a] -> Bool
isEmpty' [] = True
```

```
isEmpty' _  = False
```

Although this is perfectly valid Curry code, if we execute this version in KiCS2, we do not get the expected result.

```
> isEmpty' []
True
False
```

Instead of just `True`, we get `True` as well as False as a result of `isEmpty' []`. This is a result of Curry testing all rules of a function definition, regardless of the other ones matching or not. In the case of `isEmpty' []` both rules match, so we get both results.

Even though we have to be careful with overlapping rules in Curry, we can use them to purposely introduce non-determinism. A very simple example of a non-deterministic function is the definition of a coin, which can be either heads or tails (or in this case 0 or 1). Using overlapping rules we can define a coin the following way.

```
coin1 :: Int
coin1 = 1
coin1 = 0
```

Since non-determinism is a key feature of Curry, Curry also provides the `(?)` - operator to introduce non-determinism into programs. The definition is shown in Listing 2.1. The operator is defined using two rules. The first one ignores its second argument and returns the first. The second rule ignores the first argument and returns the second. As both rules will always match, the result is the non-deterministic return of one of the two arguments.

Listing 2.1: definition of (?)

```
(?) :: a -> a -> a
x ? _ = x
_ ? y = y
```

Using `(?)` we can rewrite the definition of coin using a single rule. In this way we do not have to use overlapping rules to introduce non-determinism into our programs anymore.

```
coin2 :: Int
coin2 = 1 ? 0
```

With the help of `(?)`, we can transform a list into a non-deterministic value selecting one of the lists elements. The return value is either the first element or any value of the remaining elements. The following listing shows the implementation of `anyOf` using the `(?)` - operator.

```
anyOf :: [a] -> a
anyOf (x:xs) = x ? anyOf xs
```

We can simplify `anyOf`'s definition using the higher-order-function `foldr1`.

```
anyOf :: [a] -> a
anyOf = foldr1 (?)
```

2.4. AbstractCurry

To realize most of the goals (section 1.2), we mainly use AbstractCurry. AbstractCurry is part of the standard Curry library and allows meta programming in Curry programs. We will not cover AbstractCurry in depth in this section, but rather look at some small examples to convey a basic understanding of AbstractCurry. We will write three small functions that are used in the final program and the data types needed by these functions.

The first interesting question is how to obtain the AbstractCurry representation of a program. The Curry library provides a function called `readCurry` (Listing 2.2). The functions parameter is the name of the module it should parse.

Listing 2.2: readCurry signature

```
readCurry :: String -> IO CurryProg
```

`readCurry` returns an abstract representation of the parsed module. The definition of the data type `CurryProg` is shown in Listing 2.3. The type contains the module's name, a list of imported modules, the data type and function declarations and at last the operator precedence declarations.

Listing 2.3: definition of CurryProg

```
type MName = String
data CurryProg
  = CurryProg MName        -- module name
              [MName]      -- imports
```

```
                [CTypeDecl] -- type declarations
                [CFuncDecl] -- function declarations
                [COpDecl]   -- operator precedence declarations
```

This abstract representation enables us to transform programs. But if we do modify the code, we want to be able to use the new version. We have to generate a normal Curry file, as AbstractCurry cannot be directly used by the compiler. To generate the Curry code, we use `showCProg :: CurryProg -> String` from module *PrettyAbstract*. As a module has to be saved as *ModuleName.curry*, if we want to be able to import it later, it is a good idea to directly save it in this way. Since there is no library function that does this way of saving for us, we have to get our own hands dirty. First we need to get the name of the module from its abstract representation. Listing 2.4 shows a simple function to obtain the name. We simply match for the first argument of `CurryProg` and return the value.

Listing 2.4: getModuleName

```
-- get a module's name
getModuleName :: CurryProg -> String
getModuleName (CurryProg modname _ _ _ _) = modname
```

For saving we need to use three `IO` functions, namely `openFile, hPutStr` and `hClose`. The signatures of these three functions are listed in Listing 2.5. As typical for file I/O we have to obtain a *Handle* by opening the file (`openFile`). For this we can use three modes: `ReadMode`, `WriteMode` and `AppendMode`. We can than write to the open `Handle` (`hPutStr`) and when we are done, we have to close the `Handle` to conclude the operation (`hClose`).

Listing 2.5: openFile hPutStr hClose signatures

```
openFile :: String -> IOMode -> IO Handle
hPutStr  :: Handle -> String -> IO ()
hClose   :: Handle -> IO ()
```

Putting all this together, we can now write `saveCurryCode` (Listing 2.6), a function that takes an abstract representation of a Curry module and saves it as a normal Curry program under *'ModuleName'.curry.*

Listing 2.6: saveCurryCode definition

```
saveCurryCode :: CurryProg -> IO ()
saveCurryCode p = do
  file <- openFile (getModuleName p ++ ".curry") WriteMode
  hPutStr file $ showCProg p
  hClose file
```

Now that we can obtain a program's AbstractCurry representation and save it as a normal Curry program again, we can think about modifying the code. One thing we will have to do later on is changing a functions visibility to public, i.e. export it. So let us look at how to do this. First we need to examine the data type representing a function. This data type is called `CFuncDecl`.

Listing 2.7: definition of CFuncDecl

```
data CFuncDecl
  = CFunc          QName Arity CVisibility CTypeExpr [CRule]
  | CmtFunc String QName Arity CVisibility CTypeExpr [CRule]
```

The difference between the two constructors `CFunc` and `CmtFunc` of `CFuncDecl` is that `CmtFunc` has an additional parameter (the first) to include a comment. The parameter of type `CVisibility` is the functions visibility level. There are only two different levels: `Public` (exported) and `Private`. Since we want to make the function public, we simply have to pattern match for the different arguments, whereat we can ignore the original visibility (`_`), as we do not care about the original visibility. We then yield a new function using the same constructor and arguments, except for the third (comment not counting) one set to `Public` (Listing 2.8). Thusly, we created a new version of the function that is exported.

Listing 2.8: makePublic

```
makePublic :: CFuncDecl -> CFuncDecl
makePublic (CmtFunc c name arity _ typeExpr rules)
  = CmtFunc c name arity Public typeExpr rules
makePublic (CFunc     name arity _ typeExpr rules)
  = CFunc     name arity Public typeExpr rules
```

3. CurryTest and EasyCheck

This chapter will introduce two existing frameworks to test Curry programs. These frameworks are *CurryTest*[2] and *EasyCheck*[11][3][8]. We will examine what functionalities each of these test suites provide and how to use them. The first framework we look at is *CurryTest*.

3.1. CurryTest

CurryTest is part of both KiCS2 as well as PACKS[1]. Both compiler's test suites are currently exclusively using *CurryTest*. *CurryTest* consists of the Curry module *Assertion* and the tool *currytest*. The *currytest* program is used to automatically execute all tests in a given module. The *Assertion* module provides the following six functions to define tests.

```
assertTrue      :: String -> Bool -> Assertion ()
assertEqual     :: String -> a -> a -> Assertion a
assertValues    :: String -> a -> [a] -> Assertion a
assertSolutions :: String -> (a->Success) -> [a] -> Assertion a
assertIO        :: String -> IO a -> a -> Assertion a
assertEqualIO   :: String -> IO a -> IO a -> Assertion a
```

To define a test one has to import the module *Assertion* and then fully apply one of the above functions. But to make the tests executable by the *currytest* program, each test has to be exported as well. For this reason one would normally define tests with *CurryTest* in a separate module from the functions that are tested. The module containing the tests than simply exports all its top level functions. Through this separation the normal modules can still export only the relevant functions, without the tests cluttering the modules' interfaces.

[1]Portland Aachen Kiel Curry System

Next let us look at the different functions provided by *Assertion*. One thing all six of them have in common is the first argument of type `String`. This first argument is a message that is displayed when the test is executed. In the rest of this section we will ignore this argument when talking about the test functions. Accordingly, we will refer to the first argument after the `String` -argument as the first of the function and so on.

The simplest function to define a test with is `assertTrue`. `assertTrue` takes only one argument of type `Bool`. The test fails if the argument evaluates to `False`. So let us test if our function `isEmpty` is correct for the empty list.

```
testEmptyList = assertTrue "[]␣is␣empty" (isEmpty [])
```

The next function is `assertEqual` and it expects two arguments. The test is passed if both arguments evaluate to the same value. A common use case is to test a set of inputs against known results of a function. The example code for `assertEqual` shows a small test set for *reverse*. It is customary to use multiple input values.

```
testReverse1 = assertEqual "reverse1" (reverse      []) []
testReverse2 = assertEqual "reverse2" (reverse     [1]) [1]
testReverse3 = assertEqual "reverse3" (reverse [1,2,3]) [3,2,1]
```

`assertEqual` and `assertTrue` can be expressed via one another. All tests using `assertTrue "" boolean` can be rewritten as `assertEqual "" boolean True`. `assertEqual "" val1 val2`, on the other hand, can be rewritten as `assertTrue "" (val1 == val2`.

All test functions used so far expect a deterministic result and generate an error if the tested functions are non-deterministic. The next two test functions work with non-determinism. The first one up is `assertValues`. It takes a
(non-)deterministic value as first and a list of values as second argument. Thereby, the list is interpreted as the multi-set of values the first argument should produce. This means `[True, False, True]` and `[True, False]` are two different values. `[True, False]` and `[False, True]`, on the other hand, are interchangeable. So, to test our coin function we can use the following two variations.

```
testCoin1 = assertValues "coin1" coin [0, 1]
testCoin2 = assertValues "coin2" coin [1, 0]
```

As deterministic functions are just a special case of non-deterministic functions, all tests using `assertTrue` or `assertEqual` can be rewritten using `assertValues`. Since we have already shown that `assertTrue` and `assertEqual` are interchangeable, we only show this for one of them. `assertTrue "" boolean` can also be expressed as `assertValues "" boolean [True]`.

The second test function to deal with non-determinism is `assertSolutions`. This function deals with constraint abstractions and passes if the constraint abstraction (first argument) yields the multi-set (second argument) as solution. The online guide to *CurryTest* lists the following example.

```
testPrefix = assertSolutions "prefix" (\x -> x++_ =:= [1,2])
                                      [[],[1],[1,2]]
```

The remaining two functions are designed to test `IO`-functions. Although, it is good practise to avoid the `IO`-monad as long as possible in the code, at some point we probably have to use it. `assertIO` expects one `IO`-action and the return value of this operation. The test passes if the `IO`-operation's return value matches the second argument of `assertIO`. Possible changes to the *World*, which the `IO`-action might introduce, are not considered, as it would be impossible.

As usage example of `assertIO` we will use the function `nth`.

```
nth :: String -> Int -> IO String
nth filename n = do
  content <- readFile filename
  let lines = splitOn "\n" content
  return $ lines !! n
```

`nth` expects a filename and a positive integer and returns the n-th line in the file. To test it we have two choices: we can either provide a test file as part of our test suite or we can create a temporary file on the fly. `wrapNth` demonstrates a very rudimentary version of the second option, creating the file on the fly. The `wrapNth` function creates a new file "tmp.txt" with six lines containing the strings "1" through "6". It then calls `nth`, deletes the file and yields the result of the call to `nth`.

```
wrapNth :: IO String
wrapNth = do
  writeFile "tmp.txt" "1\n2\n3\n4\n5\n6\n"
  ret <- nth "tmp.txt" 5
  system "rm␣-f␣tmp.txt"
  return ret
```

Our test can now be written as follows.

```
testNth = assertIO "nth" wrapNth "5"
```

The final test function is `assertEqualIO`. In contrast to `assertIO` it expects two `IO`-actions, but also just compares the return values. Every test with `assertIO` can be written as `assertEqualIO` test by simply wrapping the second argument in a `return` statement.

The final test module would look like the following:

```
module TestModule where

import Assertion
import ModuleToTest
import HelperModules
-- ...

-- tests and helpers
-- ...
```

The test module simply exports all contained functions because the tests have to be exported anyway. We then import the *Assertion* module and all other modules containing functions we want to test or otherwise need to write our test. For example, to write our wrapper for `nth` (`wrapNth`) we would need to import the `system` function from the module *System*. We then write our tests and helper functions as shown in the examples above.

To execute the tests, *currytest* has to be called with the module name that contains the tests. So in our example we would call `currytest TestModule`.

```
$ currytest TestModule
[...]
=============================================================
Testing module "TestList"...
OK: [] is empty
OK: reverse1
OK: reverse2
OK: reverse3
OK: coin1
OK: coin2
OK: prefix
OK: nth
```

```
FAILURE of nth: IO assertion not satisfied:
Computed answer: "6"
Expected answer: "5"

FAILURE occurred in some assertions!

FAILURE IN SOME TEST OCCURRED!!!
FAILED TEST MODULES: TestModule
All tests successfully passed.
==============================================================
```

As output the tool lists the tests (the `String` argument of the `asserts`) and the status of the tests. If a test fails, the tool also states details as to why the test failed. In our example the last test failed. The output indicates that `nth` yielded `6`, also we expected it to compute `5`. A look at our definition of `nth` reveals that we forgot to subtract `1` from `n` as the first element of `lines` has index `0`. When we run the corrected version the output changes to this:

```
$ currytest TestModule
[...]
OK: prefix
OK: nth
All tests successfully passed.
```

Furthermore, the *currytest* program uses its exit code to specify whether all tests passed or not. As typical for *nix programs a status of zero indicates success (all tests passed) and other values an error (in this case: some tests failed). The exit status can then be used to automate the testing and inform the user of failures so he can look at the generated log.

3.2. EasyCheck

The second existing test framework we look at is *EasyCheck*. In contrast to *CurryTest*, *EasyCheck* is only included in *KiCS* (Version one in this case). Additionally, a newer version named *curry-test* exists which is only available for the *MCC*[2] as it uses type classes and some libraries differ between the *MCC* and *KiCS(2)/PACKS*.

[2]Münster Curry Compiler

EasyCheck is inspired by Haskell's *QuickCheck*[11][9][8]. The idea behind it is to write tests as properties and generate random input values. The *EasyCheck* library provides a small set of operators to create properties and the `easyCheck` and `verboseCheck` functions to test properties.

The first operator *EasyCheck* provides is `(-=-)`. `(-=-)` is used to test deterministic functions and produces an error when used on non-deterministic ones. Like all the other operators, `(-=-)`'s arguments cannot be functions. If we want to generate test data, our test function has to have the corresponding arguments and then apply them to the functions we want to compare. Let us look at the following example.

```
reverseLeavesSingletonUntouched :: Bool -> Prop
reverseLeavesSingletonUntouched x = reverse [x] -=- [x]
```

We defined a property named `reverseLeavesSingletonUntouched`. As this holds for all singleton lists, we introduced a parameter `x` to generate a random value for our list. The right side of our test function is the actual property. Since `reverse [x]` and `[x]` are deterministic values (as long as `x` is deterministic), we use `(-=-)` to define the property. Although this property holds for all types, the signature contains a specific type for `x`, namely `Bool`. We could have chosen any other type like `Int` or `(Int, Maybe Bool)`, as long as we choose a concrete type, because *EasyCheck* has to generate values and needs a concrete type declaration for this.

The next three operators (`(~>)`, `(<~)`, `(<~>)`) we look at deal with non-deterministic tests. All three operators expect a non-deterministic function of type `a` as first and second argument. In contrast to `assertValues` and `assertSolutions` the result sets of the non-deterministic operations are not interpreted as multi-sets but as sets. The first of the three operators (`(~>)`) tests, if the first result set is a superset of the second. This way we can test for a function to return at least some known values. The following property tests for our coin to return at least zero.

```
coinCanBeHeads :: Prop
coinCanBeHeads = coin1 ~> 0
```

A more productive example would be the following function:

```
insert :: a -> [a] -> [a]
insert x ys     = x : ys
insert x (y:ys) = y : insert x ys
```

`insert` places a value at an arbitrary position in a list. One possible result of `insert x xs` should obviously be `x:xs`. Using `(~>)` we can write this as follows.

```
insertAtFirstPlace :: Int -> [Int] -> Prop
insertAtFirstPlace x xs = insert x xs ~> x:xs
```

Since we also know a second trivial solution, namely inserting `x` at the last position, we can write a second test.

```
insertAtLastPlace :: Int -> [Int] -> Prop
insertAtLastPlace x xs = insert x xs ~> xs ++ [x]
```

As stated earlier, *EasyCheck* interprets both sides of the operator as non-deterministic values. We can use this to rewrite our two tests into a single test containing all trivial solutions.

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

The `(<~)` operator simply checks, if the first result set is a subset of the second one. This can be useful when calculating a superset of the possible solutions is easy. A very simple example would be our `coin` again. We know that only `0` and `1` should be in the result set.

```
coinHasOnlyHeadsAndTails :: Prop
coinHasOnlyHeadsAndTails = coin1 <~ (0 ? 1)
```

It is important to note that `coinHasOnlyHeadsAndTails` does not fail if `coin` has a multi-set of solutions containing more than one `0` or `1`.

With `(~>)`'s and `(<~)`'s semantic consistent, `(<~>)` tests if the first and second argument's return sets are equal. A good way to use this function, is to test two different versions of the same function. In section 2.3 we introduced two different versions of `coin`: `coin1` and `coin2`. Using `(<~>)` we can test if both produce the same result set.

```
coin1AndCoin2 :: Prop
coin1AndCoin2 = coin1 <~> coin2
```

Since a key feature of *EasyCheck* is the ability to generate test data, it might be necessary to restrict the generated values. Let us consider the following example. The `tail` function returns all elements of a list, except for the first one. A simple property that could be deduced is that the result of tail is a list one element shorter than the original list.

```
tailReducesLength :: [Bool] -> Prop
tailReducesLength xs = length (tail xs) -=- length xs - 1
```

So far we only defined properties, to actually test them we have to call them using a set of functions *EasyCheck* provides. For properties without parameters we have to use `easyCheck property` and for properties with parameters we have to call `easyCheckX property`, where `X` is the number of parameters the property expects. *EasyCheck* provides functions for up to five parameters. As there is no possibility to automatically execute all *EasyCheck* tests in a module, we have to write a main function like this:

```
main = do
  easyCheck1 reverseLeavesSingletonUntouched
  easyCheck  coinCanBeHeads
  easyCheck2 insertTrivial
  easyCheck  coinHasOnlyHeadsAndTails
  easyCheck  coin1AndCoin2
  easyCheck1 tailReducesLength
```

When we execute `main`, *EasyCheck* produces the number of tests executed for each property.

```
Passed 2 tests.
Passed 1 test.
OK, passed 1000 tests.
Passed 1 test.
Passed 1 test.
Falsified by first test.
Arguments:
[]
no result
```

In this example our last test failed. The output states three different things:

1. The number of the test that failed, in this case the first.
2. The generated arguments (`[]`).
3. All the results calculated for the left side of the operator. Or in this case `no result` because `tail` simply fails when called with the empty list.

In this example the problem is our defined property: it does not hold for the empty list. Since it is a common occurrence that properties do not hold for some values, *EasyCheck* provides the `(==>)` operator. `(==>)` allows us to define an additional condition that must be true for the property to be tested. Using `(==>)` we can rewrite `tailReducesLength` to exclude the empty list as possible value.

```
tailReducesLength' :: [Bool] -> Prop
tailReducesLength' xs
  = not (isEmpty xs) ==> length (tail xs) -=- length xs - 1
```

The `(==>)` operator can be used in combination with every other operator discussed in this section. After rewriting the main function to use `tailReducesLength'` all tests pass.

```
[...]
OK, passed 1000 tests.
```

Besides `easyCheck` and its variations, the *EasyCheck* library also provides `verboseCheck` and the corresponding versions for properties expecting arguments. These functions provide additional information for each executed test. Specifically the generated arguments for each test.

As the outputs shown above demonstrate, *EasyCheck* in no way identifies the property being tested. Because of this lack of information, it is reasonable to include some additional information in the main function, for example printing the name of each property before testing it.

In contrast to *CurryTest* *EasyCheck* provides no other means than the textual output to identify if a test failed.

4. Syntax and Semantics

In this chapter we will introduce *CurryCheck*'s interface. For each function *CurryCheck* provides we will show some examples of usage and define the semantics of the operation. We will also show how previous tests using *CurryTest* or *EasyCheck* can be ported to *CurryCheck*. All in all, *CurryCheck* provides one data type, seven operators and two helper functions to define tests.

4.1. Basic structure of CurryCheck

CurryCheck consist of two parts: a library and a program. The library module (*CurryCheck*) has to be imported by any module in which tests will be defined. The library provides all data types and functions discussed in the remainder of this chapter. The second part is an executable (*currycheck*). The *currycheck* program provides the means to automatically execute all defined tests. To identify which test runs, *CurryCheck* uses the name of the test function as an identifier. It is therefore advisable to use meaningful names for the tests.

4.2. `CTest a`

The first data type *CurryCheck* provides is `CTest a`. All of `CTest`'s constructors are hidden by the library and it can only be constructed via the provided functions. The importance of `CTest` is that all top level functions evaluating to *CTest a* will be executed by the *currycheck* program. Thereby the visibility of the functions does not matter. So in contrast to *CurryTest*, private global functions will be executed as well.

4.3. (-===-)

(-===-) is the heart of the *CurryCheck* framework and most test can be defined with this operator. In contrast to *EasyCheck*'s (-==-) the (-===-)-operator does not restrict itself to deterministic arguments and partially applied functions are possible as well. We will define a series of example tests examining different argument types and discuss the semantics of (-===-) in each case, but simply put: both sides should evaluate to the same value.

```
(-===-) :: a -> a -> CTest a
```

(-===-) takes two arguments of type a and returns a CTest a. The simplest usage of (-===-) is to compare two deterministic values. To test our isEmpty function with the empty list we can write the following code:

```
testIsEmpty :: CTest Bool
testIsEmpty = isEmpty [] -===- True
```

As both sides are deterministic values, the semantics are simple: if the values are the same, the test passes, otherwise it fails. As stated earlier, partial functions are possible as well. The use of partial functions allows us to define properties similar to *EasyCheck*. Let us consider the associativity of (+). When working with Ints in Curry the order of the arguments should not affect the outcome of the (+) - operation. To test this with *CurryCheck* we can define the following test case.

```
plusIsAssociative :: CTest (Int -> Int -> Int)
plusIsAssociative = (+) -===- flip (+)
```

Both arguments are still deterministic, but now we deal with functions instead of simple values. In *CurryCheck* two deterministic functions are equal ((-===-)) if they both evaluate to the same value when called with the same arguments.

When using non-deterministic values with (-===-), we compare the multi-sets of the results. For the test to pass, both sides have to have the same multi-set of results. So (1 ? 0) and (0 ? 1) are considered equal, but (0 ? 1 ? 1) and (0 ? 1) are not, as the first expression has a multi-set of {0, 1, 1} and the second expression only of {0, 1}. The use of multi-sets allows us to not consider the order in which the values are produced.

To use our two coin implementations again, we can compare them using *CurryCheck* like this:

```
coin1AndCoin2 :: CTest Int
coin1AndCoin2 = coin1 -===- coin2
```

Lastly we can also use non-deterministic functions as arguments of `(-===-)`. As with deterministic functions, the results of each generated input are compared and, like non-deterministic values, this comparison considers the multi-set of results. Using this we can compare two versions of a function to test if they produce exactly the same results. As an example let us consider `insert` again. More exactly the Curry system should produce the same results when we reorder the rules.

```
insert' :: a -> [a] -> [a]
insert' x (y:ys) = y : insert x ys
insert' x ys     = x : ys
```

To test this, we can use `(-===-)`:

```
testInsert :: CTest (Bool -> [Bool] -> [Bool])
testInsert = insert -===- insert'
```

Although `insert` and `insert'` are both defined as polymorphic functions, we must use a concrete type in our test definition because the test system has to know a concrete type to generate test data.

So far we only used plain values and partially applied functions in our tests. As stated in section 4.2, all global functions evaluating to `CTest a` are executed. This means that, like with *EasyCheck*, we can write our tests as functions with parameters.

```
plusIsAssociative' :: Int -> Int -> CTest Int
plusIsAssociative' x y = x + y -===- y + x
```

`plusIsAssociative` and `plusIsAssociative'` are essentially two different syntaxes for the same test. Furthermore, *CurryCheck* allows a mixed style.

```
plusIsAssociative'' :: Int -> CTest (Int -> Int)
plusIsAssociative'' x = (x+) -===- (+x)
```

4.4. (<~~), (~~>), and (<~~>)

`(<~~)`, `(~~>)` and `(<~~>)` are *CurryCheck*'s set operators. Like their *EasyCheck* counterparts, `(<~)`, `(<~>)`, and `(<~>)`, they evaluate both their arguments and compare the result sets, so semantically they are identical.But unlike *EasyCheck*'s versions, *CurryCheck*'s set operators also support functions as arguments and the mixed syntax seen in section 4.3.

```
(<~~), (~~>), (<~~>) :: a -> a -> CTest a
```

4.5. (===>)

As *CurryCheck* provides automatic test data generation, like in *EasyCheck*, it is important to be able to restrict the generated data. Following *EasyCheck*'s example we provide the `(===>)` operator.

```
(===>) :: Bool -> CTest a -> CTest a
```

The operator's two arguments are a boolean value and a `CTest a`. It is important to note that `a` cannot be a functional type in this context. That means, `(===>)` only supports the syntax style of *EasyCheck*. As an example of using `(===>)` let us revisit `tailReducesLength` from section 3.2.

```
tailReducesLength :: [Bool] -> CTest Int
tailReducesLength xs
  = not (isEmpty xs) ===> length (tail xs) -==- length xs
```

Like *EasyCheck*, *CurryCheck* uses a generated value only if `(===>)`'s first argument evaluates to `True`.

4.6. IO tests

So far all operators work with pure functions. To test `IO`-functions *CurryCheck* provides two operators: `‘resultsIn‘` and `‘sameAs‘`. These two functions provide semantics equal to *CurryTest*'s `IO`-assertions.

```
resultsIn :: IO a -> a    -> CTest a
sameAs    :: IO a -> IO a -> CTest a
```

Thereby `resultsIn` directly compares the result of an `IO`-action with the second argument, like `assertIO`. Whereas `sameAs` compares the results of two `IO`-operations, like `assertEqualIO`. Although both `resultsIn` as well as `sameAs` yield a value of type `CTest a`, they cannot be used with `(===>)` or generated arguments.

4.7. `anyOf` and `anySolutionOf`

In section 3.2 we defined `insertTrivial` to test `insert` for two trivial results: the insertion at first and last position. To specify the two possible results, we had to use the `(?)`-operator, although on a semantic level we want to specify a set of values. *CurryCheck* provides the `anyOf` function to grant a more set like syntax.

```
insertTrivial :: Bool -> [Bool] -> CTest [Bool]
insertTrivial x xs = insert x xs ~~> anyOf [x:xs, xs++[x]]
```

`anyOf` also enables us to more easily port tests from *CurryTest*, as we will see in section 4.10. The `anyOf` function yields one element of its first argument non-deterministically.

As *CurryCheck* does not provide a direct way for testing constraint abstractions like *CurryTest*. `anySolutionOf` takes a constraint abstraction and returns a single solution. Like `anyOf`, it is a non-deterministic function and as such has a result set of all possible solutions of its argument.

```
anySolutionOf :: (a -> Success) -> a
```

4.8. Limits when using partially applied functions

It was mentioned in section 4.3 and section 4.4 that partially applied functions can be used as arguments of *CurryCheck*'s operators. It is important to note that this syntax has some limitations. It is only possible to use partially applied functions when the operator is used directly on the right hand side of the tests definition. To clarify here an example:

We have seen `(-===-)` being used with partially applied functions in section 4.3.

```
testInsert = insert -===- insert'
```

Here `(-===-)` is used directly on the right hand side of `testInsert`'s definition. However, it is not possible to rewrite this test like this:

```
testInsert = helper
 where helper = insert -===- insert'
```

This restriction comes from the fact that *CurryCheck* has to transform tests with partially applied functions into tests using only fully applied ones. This transformation is necessary for *CurryCheck* to be able to pass the tests to *EasyCheck*. As part of the transformation, *CurryCheck* has to modify the right hand side of the test. Specifically, it has to add additional parameters to the operator's arguments (subsection 5.4.1), `insert` and `insert'` in this example.

4.9. Porting from EasyCheck

In this section we will look at how to port the different types of tests from *EasyCheck* to *CurryCheck*. As the syntax of both frameworks is very similar this is a rather straight forward process and mainly involves renaming the operators. Table 4.1 shows how to rename the operators.

Table 4.1.: operator mapping between EasyCheck and CurryCheck

EasyCheck	**CurryCheck**
(-=-)	(-===-)
(<~)	(<~~)
(~>)	(~~>)
(<~>)	(<~~>)
(==>)	(===>)

After changing the operators the only thing left to do is to adjust the signatures. More exactly, the `Prop` part of the signatures has to be changed to the correct instantiation of `CTest a`.

The following listing shows three examples.

```
-- EasyCheck
reverseLeavesSingletonUntouched :: Bool -> Prop
reverseLeavesSingletonUntouched x = reverse [x] -=- [x]

-- CurryCheck
reverseLeavesSingletonUntouched :: Bool -> CTest [Bool]
reverseLeavesSingletonUntouched x = reverse [x] -==- [x]

-- EasyCheck
coinCanBeHeads :: Prop
coinCanBeHeads = coin1 ~> 0

-- CurryCheck
coinCanBeHeads :: CTest Int
coinCanBeHeads = coin1 ~~> 0

-- EasyCheck
isEmptyIsFalseForNonEmptyLists :: [Bool] -> Prop
isEmptyIsFalseForNonEmptyLists xs
  = length xs > 0 ==> isEmpty xs -=- False

-- CurryCheck
isEmptyIsFalseForNonEmptyLists :: [Bool] -> CTest Bool
isEmptyIsFalseForNonEmptyLists xs
  = length xs > 0 ===> isEmpty xs -==- False
```

4.10. Porting from CurryTest

In this section we will examine the porting of tests from *CurryTest* to *CurryCheck*. As the interface is designed after *EasyCheck*, a little more work has to be done than when porting *EasyCheck* to *CurryCheck*. In section 3.1 we already showed that `assertTrue` and `assertEqual` can easily be rewritten using `assertValues`. `assertValues` was based on the comparison of multi-sets. The only operator *CurryCheck* provides which compares multi-sets for equality is `(-==-)`. We therefore have to rewrite tests using one of these three functions using `(-==-)`.

In the case of `assertTrue` and `assertEqual` we can simply use `(-==-)` to compare the values. As *CurryCheck* uses the name of the test function to identify the test instead of a additional `String` argument, we simply use the `String` argument as function name (we have to be careful with special characters).

```
-- CurryTest
testEmptyList :: Assertion ()
testEmptyList = assertTrue "empty␣list␣is␣empty" (isEmpty [])
```

```
-- CurryCheck
emptyListIsEmpty :: CTest Bool
emptyListIsEmpty = isEmpty -==- True

-- CurryTest
testReverse1 :: Assertion [Int]
testReverse1 = assertEqual "reverse1" (reverse []) []

-- CurryCheck
reverse1 :: CTest [Int]
reverse1 = reverse [] -==- []
```

In contrast to `(-==-)`, `assertValues` expects the multi-set as argument, whereas `(-==-)` expects a non-deterministic function, which results are the elements of the multi-set. To simplify the transition from *CurryTest* we can use the helper function `anyOf`.

```
-- CurryTest
testCoin1 :: Assertion Int
testCoin1 = assertValues "coin1" coin1 [0, 1]

-- CurryCheck
testCoin1 :: CTest Int
testCoin1 = coin1 -==- anyOf [0, 1]
```

To port tests using `assertSolutions` to *CurryCheck* we can use a combination of `anySolutionOf` and `anyOf`. We use `anySolutionOf` to transform the constraint abstraction into a non-deterministic value and `anyOf` to transform the multi-set of expected results like before. Since `assertSolutions` also uses multi-set comparison, we use `(-==-)` again to define the test.

```
-- CurryTest
testPrefix :: Assertion [Int]
testPrefix = assertSolutions "prefix" (\x -> x++_ =:= [1,2])
                                      [[],[1],[1,2]]

-- CurryCheck
testPrefix :: CTest [Int]
testPrefix = anySolutionOf (\x -> x++_ =:= [1,2]) -==-
             anyOf [[], [1], [1,2]]
```

The last two functions are `assertIO` and `assertEqualIO`. `CurryTest`'s `IO`-tests can be ported to *CurryCheck* using `resultsIn` and `sameAs`. `resultsIn` in used to port `assertIO` tests and `sameAs` for `assertEqualIO`, respectively. As the parameter types are the same for *CurryCheck*'s and *CurryTest*'s `IO` functions, no arguments have to be altered, except for the `String`. The following code shows the conversion of an `IO`-test using `resultsIn`.

```
-- CurryTest
testDir :: Assertion (Bool, Bool, Bool, Bool)
testDir = assertIO "test create/rename/delete directory"
                   dirOps (True,False,True,False)
 where
  dirOps = do
    -- ...

-- CurryCheck
createRenameDeleteDir :: CTest (Bool, Bool, Bool, Bool)
createRenameDeleteDir
  = dirOps `resultsIn` (True,False,True,False)
 where
  dirOps = do
    -- ...
```

5. Implementation

In this chapter we will examine the implementation of *CurryCheck*. First of all we look at what *CurryCheck* has to generate to execute the tests. After that we discuss some necessary changes and extensions to *EasyCheck* to support the semantics defined in the previous chapter. Lastly we address how *CurryCheck* is implemented. More precisely, how *CurryCheck* analyses the source code and how it generates and executes the tests.

As stated in section 2.4, *AbstractCurry* is used extensively in the implementation of *CurryCheck*. Most functions and data types provided by *AbstractCurry* will not be discussed in detail. The underlying data structures defined by *AbstractCurry* are quite cumbersome and it is more important to understand what is being generated (the actual Curry code) and what information are used than it is to understand how the underlying data types work. Furthermore, we will not always explicitly distinguish between the *AbstractCurry* representation of a piece of code and the code itself. It should be clear that our code works on the abstract representation and the actual code gets only generated when calling `showCProg`/`saveCurryCode`. We may for example say that the function `cfunc` generates a function although it actually yields the abstract representation of a function (`CFuncDecl`). To make it easier to follow the code without diving into the *AbstractCurry*, we will often refer to the listings provided in section 5.1 to identify what actually gets analysed, modified or generated in a specific part of the program using an example.

5.1. Generated code

In this section we will look at the code generated by *CurryCheck*. On this account, this chapter contains multiple larger code listings. These listings will also get referenced by late sections in this chapter when discussing the implementation in more detail.

Listing 5.1 shows an example file (*Demo.curry*) containing some tests using *CurryCheck*. It basically consists of a selection of tests shown in chapter 4. In short, *Demo.curry* exports three of its top-level functions, namely `isEmpty`, `insert`, and `insert'`. All of these have been introduced in previous chapters.

After the module's imports, the function definitions mixed with the tests follow. In this Demo the tests follow directly after the corresponding function definition. As one of the ideas behind *CurryCheck* was to use this locality to generate a more useful output. In this example the tests follow after the function definition, but the other way around is just as reasonable. After the definitions of `isEmpty`, `insert`, and `insert'` and a total of four corresponding tests. Another five non related tests follow. The first two of these five tests are really minimalistic `IO`-tests to show how *CurryCheck* handles these kind of tests. The last three tests are the three different versions of `plusIsAssociative` from section 4.3, as they demonstrate the three possible syntaxes to directly compare functions.

Listing 5.1: Demo.curry - Curry file containing tests

```
module Demo
  ( isEmpty
  , insert
  , insert'
  ) where

import CurryCheck
import Assertion
import EasyCheck
import System    (system)
import List      (splitOn)

-- tests if a list is empty
isEmpty :: [a] -> Bool
isEmpty []    = True
isEmpty (_:_) = False

testIsEmpty :: CTest Bool
testIsEmpty = isEmpty [] -===- True

-- usage of (===>)
```

```
isEmptyIsFalseForNonEmptyLists :: [Bool] -> CTest Bool
isEmptyIsFalseForNonEmptyLists xs
  = length xs > 0 ===> isEmpty xs -==- False

-- two versions of insert and tests
insert :: a -> [a] -> [a]
insert x ys     = x : ys
insert x (y:ys) = y : insert x ys

insert' :: a -> [a] -> [a]
insert' x (y:ys) = x : insert y ys
insert' x ys     = x : ys

testInsert :: CTest (Bool -> [Bool] -> [Bool])
testInsert = insert' -==- insert

insertTrivial :: Bool -> [Bool] -> CTest [Bool]
insertTrivial x xs = insert x xs ~~> anyOf [x:xs, xs++[x]]

-- minimalistic IO examples
ioExample1 :: CTest Int
ioExample1 = return 2 `resultsIn` 2

ioExample2 :: CTest Int
ioExample2 = return 2 `sameAs` return 2

-- test associativity of (+)
-- 1. function as argument
plusIsAssociative :: CTest (Int -> Int -> Int)
plusIsAssociative = (+) -==- flip (+)

-- 2. fully applied function arguments
plusIsAssociative' :: Int -> Int -> CTest Int
plusIsAssociative' x y = x + y -==- y + x

-- 3. mixed syntax
plusIsAssociative'' :: Int -> CTest (Int -> Int)
plusIsAssociative'' x = (x+) -==- (+x)
```

To test *Demo.curry*, we have to execute the *currycheck* executable with the module name as argument: `currycheck Demo`. *CurryCheck* then generates two different kinds of files. For each module (in this case only *Demo*) it generates a copy called *ModuleName_test.curry*. This copy contains all functions of the original module and a possibly modified version of each test. More specifically, *CurryCheck* modifies each test, that uses direct function comparison without applied arguments (cf. `plusIsAssociative`) or the mixed syntax (cf. `plusIsAssociative''`). Besides these changes to the tests, all global functions in this new module are public. Listing 5.2 shows the modified version of *Demo.curry*.

Listing 5.2: Demo_test.curry - generated file; containing modified tests

```
module Demo_test where

import CurryCheck
import Assertion
import EasyCheck
import System
import List

testIsEmpty :: CurryCheck.CTest Bool
testIsEmpty
  = isEmpty [] -==- True

isEmptyIsFalseForNonEmptyLists :: [Bool] -> CurryCheck.CTest Bool
isEmptyIsFalseForNonEmptyLists xs
  = (length xs > 0) ===> (isEmpty xs -==- False)

testInsert :: Bool -> [Bool] -> CurryCheck.CTest [Bool]
testInsert x1 x2
  = insert' x1 x2 -==- insert x1 x2

insertTrivial :: Bool -> [Bool] -> CurryCheck.CTest [Bool]
insertTrivial x xs
  = insert x xs ~~> CurryCheck.anyOf [x : xs,xs ++ [x]]

ioExample1 :: CurryCheck.CTest Int
ioExample1
  = CurryCheck.resultsIn (return 2) 2

ioExample2 :: CurryCheck.CTest Int
ioExample2
  = CurryCheck.sameAs (return 2) (return 2)

```

```
plusIsAssociative :: Int -> Int -> CurryCheck.CTest Int
plusIsAssociative x1 x2
  = (x1 + x2) -===- flip (+) x1 x2

plusIsAssociative' :: Int -> Int -> CurryCheck.CTest Int
plusIsAssociative' x y
  = (x + y) -===- (y + x)

plusIsAssociative'' :: Int -> Int -> CurryCheck.CTest Int
plusIsAssociative'' x x1
  = (\ x0 -> x + x0) x1 -===- (\ x0 -> x0 + x) x1

isEmpty :: [a] -> Bool
isEmpty []
  = True
isEmpty (_ : _)
  = False

insert :: a -> [a] -> [a]
insert x ys
  = x : ys
insert x (y : ys)
  = y : insert x ys

insert' :: a -> [a] -> [a]
insert' x (y : ys)
  = x : insert y ys
insert' x ys
  = x : ys
```

In contrast to the first kind of generated files (the module copies), the second kind is only generated once for all modules. As of now all tests are still of type `CTest a`. Since we are using *EasyCheck* and *CurryTest* as the underlying frameworks to execute our tests, we have to convert our tests. This is done in *execTests.curry* (Listing 5.3). *execTests.curry* consists of two parts: a main function that executes the tests if called, and one function for each test it has to execute. Since this module comprises all converted tests, it has to import all generated copies of test modules and both *Assertion* and *EasyCheck*.

The *main*-functions first calls the *currytest* tool with itself as argument to automatically execute all *Assertion*s. We have seen in section 3.2 that

EasyCheck does not automatically find and execute tests. Because of this, it afterwards executes all the remaining tests using *EasyCheck*.

The converted tests have on of two types: `Assertion a` or `IO Bool`. Every test that uses *EasyCheck* is of type `IO Bool` and gets called from within the `main`-function.

Listing 5.3: execTests.curry - generated file; this is executed to run the tests

```
module execTests where

import Assertion
import CurryCheck
import EasyCheck
import Demo_test

main :: IO ()
main
  = do x <- CurryCheck.execAsserts "execTests"
       CurryCheck.execProps x
        [isEmptyIsFalseForNonEmptyLists_Demo,testInsert_Demo
        ,insertTrivial_Demo,plusIsAssociative_Demo
        ,plusIsAssociative'_Demo
        ,plusIsAssociative''_Demo,testIsEmpty_Demo]

isEmptyIsFalseForNonEmptyLists_Demo :: IO Bool
isEmptyIsFalseForNonEmptyLists_Demo
  = EasyCheck.easyCheck1
     "isEmptyIsFalseForNonEmptyLists␣in␣module␣Demo␣(line␣22)"
     (\ x1 -> CurryCheck.testProp
              (Demo_test.isEmptyIsFalseForNonEmptyLists x1))

testInsert_Demo :: IO Bool
testInsert_Demo
  = EasyCheck.easyCheck2 "testInsert␣in␣module␣Demo␣(line␣35)"
     (\ x1 x2 -> CurryCheck.testProp (Demo_test.testInsert x1 x2))

insertTrivial_Demo :: IO Bool
insertTrivial_Demo
  = EasyCheck.easyCheck2 "insertTrivial␣in␣module␣Demo␣(line␣38)"
     (\ x1 x2 -> CurryCheck.testProp
                (Demo_test.insertTrivial x1 x2))

plusIsAssociative_Demo :: IO Bool
```

```
plusIsAssociative_Demo
  = EasyCheck.easyCheck2
     "plusIsAssociative␣in␣module␣Demo␣(line␣50)"
     (\ x1 x2 -> CurryCheck.testProp
                  (Demo_test.plusIsAssociative x1 x2))

plusIsAssociative'_Demo :: IO Bool
plusIsAssociative'_Demo
  = EasyCheck.easyCheck2
     "plusIsAssociative'␣in␣module␣Demo␣(line␣54)"
     (\ x1 x2 -> CurryCheck.testProp
                  (Demo_test.plusIsAssociative' x1 x2))

plusIsAssociative''_Demo :: IO Bool
plusIsAssociative''_Demo
  = EasyCheck.easyCheck2
     "plusIsAssociative''␣in␣module␣Demo␣(line␣58)"
     (\ x1 x2 -> CurryCheck.testProp
                  (Demo_test.plusIsAssociative'' x1 x2))

testIsEmpty_Demo :: IO Bool
testIsEmpty_Demo
  = EasyCheck.easyCheck0 "testIsEmpty␣in␣module␣Demo␣(line␣18)"
     (CurryCheck.testProp Demo_test.testIsEmpty)

ioExample1_Demo :: Assertion.Assertion Int
ioExample1_Demo
  = CurryCheck.testFunc Demo_test.ioExample1
     "ioExample1␣in␣module␣Demo␣(line␣42)"

ioExample2_Demo :: Assertion.Assertion Int
ioExample2_Demo
  = CurryCheck.testFunc Demo_test.ioExample2
     "ioExample2␣in␣module␣Demo␣(line␣45)"
```

So far we only looked at what *CurryCheck* generates to execute the tests. Listing 5.4 shows the generated output of `currycheck Demo` in which the output of the Curry compiler is omitted.

Listing 5.4: 'currycheck Demo' output

```
$ currycheck Demo
[...]
execTests Assertion System> ===========================
Testing module "execTests"...
OK: ioExample1 in module Demo (line 42)
OK: ioExample2 in module Demo (line 45)
All tests successfully passed.
isEmptyIsFalseForNonEmptyLists in module Demo (line 22):
 OK, passed 1000 tests.
testInsert in module Demo (line 35) failed
Falsified by 5th test.
Arguments:
True
[False]
Results:
[True,False]
insertTrivial in module Demo (line 38):
 OK, passed 1000 tests.
plusIsAssociative in module Demo (line 50):
 OK, passed 1000 tests.
plusIsAssociative' in module Demo (line 54):
 OK, passed 1000 tests.
plusIsAssociative'' in module Demo (line 58):
 OK, passed 1000 tests.
testIsEmpty in module Demo (line 18):
 Passed 1 test.
Evaluation terminated with non-zero status 1
```

At the top of the output we see the results of our two `IO`-tests. Since *CurryCheck* first calls *CurryTest*, and *CurryTest* is only used for `IO`-tests, all `IO`-tests will be shown at the top of the output, independently of its module or position in relation to non-`IO`-tests. The message shown is generated by *CurryCheck*, and uses the test function's name, module, and line number to identify the test. In this example we can identify the failed test as `testInsert` and use the module name and line number to easily find the test's definition and the function it tests. It turns out that we mode a mistake in our implementation of `insert'` and switched `x` and `y` in the right hand side of the first rule (Listing 5.1, line 32).

After the `IO`-tests the output of the remaining tests is shown. In contrast to the original *EasyCheck* it displays the name of the test function, module name, and line number before the tests result.

5.2. Modifications to EasyCheck

In the previous section, we discussed the different files generated by *CurryCheck* and the generated test output. Before we dive into the details of how *CurryCheck* generates these files, we first look at some small modifications to the original *EasyCheck*. These modifications where necessary, considering that we want to provide a more concise output and use multi-sets to compare functions. We have seen in section 3.2 that the original *EasyCheck* only uses a set based approach to compare results of non-deterministic expressions.

Multi-set semantics

First of we look at the implementation of multi-set semantics. Since we do not want to replace the set semantics altogether, but only add multi-set semantics as an supplementary feature, the original operators where not modified. Instead we introduce two new functions to *EasyCheck*: `isSameMSet` and `(<=>)`.

The following listing shows the definition of `isSameMSet`.

```
isSameMSet :: [a] -> [a] -> Bool
[]     `isSameMSet` ys = ys == []
(x:xs) `isSameMSet` ys
  | x `elem` ys        = xs `isSameMSet` (delete x ys)
  | otherwise          = False
```

Just like `isSameSet` and `isSubsetOf`, `isSameMSet` provides set (or in this case multi-set) semantics on top of lists. The current implementation is straight forward. If the first list is empty, the second one has to be as well, otherwise the two lists represent different multi-sets. If, on the other hand, the first list is not empty, its first element (`x`) has to be part of the second list at least once. In that case the rest list of the first argument (`xs`) is compared to the second list minus the first occurrence of `x`. Should the second list not contain x the two lists represent different multi-sets.

`isSameMSet` is then used to implement `(<=>)` like `(<~>)`.

```
(<=>) :: a -> a -> Prop
x <=> y = test x (isSameMSet (valuesOf y))
```

Adding output

Since we want to provide *EasyCheck* with additional information to print (e.g. line number), we have to add an additional argument to all `easyCheck` and `verboseCheck` variations. This new `String` argument is the same as used with *CurryTest*. As a result `easyCheck2` for example looks like this.

```
easyCheck2 :: String -> (_ -> _ -> Prop) -> IO ()
```

The string argument is then relayed to *EasyCheck*'s `test` function, which also generates the output. A second modification to all `easyCheck` and `verboseCheck` versions is a change in the functions' result type. Instead of *IO ()* the new versions have a result type of `IO Bool`. The `boolean` value is used to indicate whether the test passed (`True`) or not (`False`). This information is used by *CurryCheck* to calculate its exit code.

In summary the new function signatures for `easyCheck`/`verboseCheck` look like this.

```
easyCheckN :: String -> (_ -> ... -> Prop) -> IO Bool
```

5.3. Mapping to EasyCheck and CurryTest

Now that we have seen the different kinds of files generated by *CurryCheck*, it is time to look at the mapping from *CurryCheck* to *EasyCheck* and *CurryTest*. More exactly what operator gets mapped to what framework.

`IO` / `resultsIn`, `sameAs`

These two operators are the only ones that get mapped to *CurryTest*. Since *CurryTest* is the only underlying framework that supports `IO`-tests, it is the only choice for `IO`-tests. As already seen in section 4.10, albeit in the other direction, there is a direct correlation between `resultsIn` and `assertIO`, respectively, `sameAs` and `assertEqualIO`.

`(<~~>)`, `(<~~)`, and `(~~>)`

Like `resultsIn` and `sameAs` with *CurryTest*, these three operators are directly correlated with *EasyCheck* as they have the same semantics as there

EasyCheck counterparts `(<~>)`, `(<~)`, and `(~>)`. Furthermore, there is no support for set semantics in *CurryTest*. For these reasons, the three operators get mapped to *EasyCheck*.

`(-===-)`

The only remaining operator is `(-===-)`. Although *CurryTest* supports multiset semantics with `assertValues` and `assertSolutions`, we can only use them with fully applied functions, as *CurryTest* does not support test data generation. Because of this the use of *CurryTest* for `(-===-)` is severely limited. It is, on the other hand, also possible to test fully applied functions with *EasyCheck*. As this possibility combined with the modifications to *EasyCheck* discussed in section 5.2 enables us to map all applications of `(-===-)` directly to *EasyCheck*, we will do precisely that.

5.4. Implementation details

Now it is time to take a more detailed look at the implementation of *CurryCheck*. The process of executing a test can be loosely split into three phases.

1. generate copies of all modules and analyse them
2. generate *execTests.curry*
3. execute `execTests.main`

This structure can be seen in the main function of *currycheck*.

```
main :: IO ()
main = do
  [...]
  testModules <- genTestEnvironment mode args
  genTestModule mode testModules
  ret <- execTests mode
  cleanup mode testModules
  exitWith ret
```

After processing the command line arguments (omitted in the code above), the main functions calls three different functions, namely `genTestEnvironment`, `getTestModule`, and `execTests`, which implement the three different phases. For the rest of this chapter we will discuss each phase and its implementation in more detail.

5.4.1. Phase 1 - `genTestEnvironment`

The first step is to create a modified copy of the input modules and analyse the code for tests. We refer to the created files in this step as the test environment, as they are necessary for *execTests.curry*, which actually executes the tests. The function `genTestEnvironment` implements this first step. It transforms a list of module names into a list of *TestModule*s. During this process it also creates the modified copies of the modules (cf. Listing 5.2).

```
genTestEnvironment :: VerbosityMode
                   -> [String]
                   -> IO [TestModule]
genTestEnvironment m = mapIO (genAndAnalyseModule m)
```

Internal data types

The first data type to look at is `CTest a` as all tests are of this type.

```
data CTest a
  = EqualTest a a
  | CondTest Bool (CTest a)
  | SubSetTest a a
  | SetTest a a
  | IOAssertion (IO a) a
  | IOEqualAssertion (IO a) (IO a)
```

`CTest a` basically consists of one constructor per operator *CurryCheck* provides to define tests. The exception being (<~~) and (~~>) both using `SubSetTest`. *CurryCheck* pattern matches over these constructors during the analysis to classify the tests (Listing 5.7).

A *TestModule* represents all the information *CurryCheck* needs to execute the next two phases. This allows us to parse the input modules only during the first phase. There are four types of information needed during the next two stages. The original name of the module, as this has to be used to identify the test in the output, the modified name, and a representation of the contained tests.

```
data TestModule = TestModule
  { moduleName :: String
  , newName    :: String
  , tests      :: Tests
  }
```

Before we look at how these information are generated, we will examine the representation of the tests.

```
data Test = PropTest   QName Int       Int
          | AssertTest QName CTypeExpr Int
```

Each test is either a property based test or an assertion based test, which get mapped to *EasyCheck* or *CurryTest*, respectively. In both cases we need the functions name (`QName`) and the line number in which the test is defined (the last argument). In the case of a property based test we additionally need the information about the test's arity (the second argument) to determine which version of `easyCheck`/`verboseCheck` has to be used. For assertion based tests, on the other hand, we need the original type signature (`CTypeExpr`).

genAndAnalyseModule

Listing 5.5: genAndAnalyseModule

```
genAndAnalyseModule :: VerbosityMode -> String -> IO TestModule
genAndAnalyseModule m moduleName = do
  prog <- readCurry moduleName
  lines <- getLines $ moduleName ++ ".curry"
  let words = firstWordsOfLines lines
  let (rawTests, newMod) = transformModule prog
  saveCurryCode newMod
  renameModule2 moduleName newModName
  tests <- classifyTests m newModName rawTests
  return $ TestModule moduleName newModName
                     (addLinesNumbers words tests)
 where
  transformModule :: CurryProg -> ([CFuncDecl], CurryProg)
  transformModule
```

```
15      = transformTests . renameModule1 newModName
16                       . makeAllPublic
17    [...] -- add line numbers
```

Listing 5.5 shows a shortened version of `genAndAnalyseModule`. After parsing the module to *AbstractCurry* (line 3), the original module gets also broken down into lines and these into their first words (lines 4 and 5). A 'first word' is thereby defined as everything before the first white space of a line. This list of words is later (line 11) used to add the line number information to the tests. To determine the line number of a test in the original file, the first occurrence of the functions name in the list of words is used. This should correspond to either the tests type signature or the test definition itself if no signature is present (cf. Listing 5.1). Since type signature and test definition should be directly adjacent (good coding style), this method of finding the line number should produce a usable result.

After parsing the source code, `transFormModule` creates the modified version of the input module (Listing 5.2). Besides returning the new module definition (`newMod`) it also returns the definitions of each test included in the module (`rawTests`). Transforming the module consists of several steps. We have to make all functions public because `execTests` has to be able to execute them. How all functions can be transformed into public functions was already discussed at the end of section 2.4.

The new module also has to be renamed(cf. Listing 5.1, line 1 and Listing 5.2 line 1). Renaming is done in two steps. `renameModule1` simply replaces the module name in the *AbstractCurry* representation of the module header. This also enables us to use `saveCurryCode` to create a file containing the modified code base. Because *AbstractCurry* uses qualified names throughout, so it has to be replaced there as well. But replacing the module name in the body of the module using *AbstractCurry* is rather cumbersome, as a rule for nearly every *AbstractCurry*-data type has to be defined. So instead *CurryCheck* replaces all other occurrences after the file has been saved using simple textual substitution (`renameModule2`, line 8).

The last step in transforming the module is done by `transformTests`. We have seen in section 5.1 that only tests using partially applied functions have to be modified to use the fully applied function syntax.

After transforming the module, *CurryCheck* has to analyse the tests to determine of what kind they are (`PropTest`/`AssertTest`). This classification is done by `classifyTests` (line 9).

transformTests

`transformTests`'s task is to transform tests that use partially applied functions. We will refer to tests using partially applied functions as functional tests from now on. To do this transformation we have to extract the tests first (Listing 5.6, line 7) and then identify the functional tests (cf. Listing 5.1 lines 50-51,58-59).

Except for the alterations to the test functions the rest of the module remains unchanged (cf. Listing 5.1 and Listing 5.2). Accordingly, the new functions of the module consist of the modified tests and the unchanged remaining functions (Listing 5.6, line 11; cf. Listing 5.2).

Listing 5.6: transformTests

```
 1 transformTests :: CurryProg -> ([CFuncDecl], CurryProg)
 2 transformTests (CurryProg modName imports
 3                           typeDecls functions opDecls)
 4   = (tests
 5   , CurryProg modName imports typeDecls newFunctions opDecls)
 6  where
 7   (rawTests, funcs) = partition isTest functions
 8
 9   tests = map transformTest rawTests
10
11   newFunctions = tests ++ funcs
12
13   transformTest :: CFuncDecl -> CFuncDecl
14   transformTest f | isFunctionalTest f = transformTest' f
15                   | otherwise          = f
16
17   isFunctionalTest :: CFuncDecl -> Bool
18   isFunctionalTest
19     = isFunctionalType . stripTCons . resultType . funcType
20
21   stripTCons :: CTypeExpr -> CTypeExpr
22   stripTCons (CTCons _ [t]) = t
23
```

```
  transformTest' :: CFuncDecl -> CFuncDecl
  transformTest' (CFunc name arity visibility fType [rule])
    = CFunc name newArity visibility newType [updateRule rule]
   where
    nestedType = stripTCons $ resultType fType
    nestedArity = calcArity nestedType
    newArity = arity + nestedArity

    newType = foldr (~>) (CTCons ("CurryCheck", "CTest")
                                 [resultType nestedType])
                       $ argTypes fType ++ argTypes nestedType

    updateRule :: CRule -> CRule
    updateRule (CRule ps rhs)
      = CRule (ps ++ (genPVars nestedArity))
              (updateRhs rhs)

    newVars = genEVars nestedArity

    updateRhs :: CRhs -> CRhs
    updateRhs (CSimpleRhs expr lDecls)
      = CSimpleRhs (updateExpr expr) lDecls

    updateExpr :: CExpr -> CExpr
    updateExpr (CApply (CApply s@(CSymbol ("CurryCheck", op))
                       lhs) rhs)
      | op `elem` ["-===-", "<~~>", "<~~", "~~>"]
        = CApply (CApply s (applyE lhs newVars))
                 (applyE rhs newVars)
```

As all of this is done using *AbstractCurry*, we will look at some smaller functions to further familiarize ourself with it. `isTest` is used to determine if a function is a test or not. We stated earlier that all top-level functions of result type `CTest _` are categorized as tests. Therefore, we only have to look at the result type of a function to ascertain this information.

```
isTest :: CFuncDecl -> Bool
isTest = checkType . funcType
 where
  checkType :: CTypeExpr -> Bool
  checkType ct = case resultType ct of
    (CTCons ("CurryCheck", "CTest") _) -> True
    _                                  -> False
```

We have already seen the definition of `CFuncDecl` during our short introduction to *AbstractCurry* in section 2.4. `funcType`'s result is the type signature of its argument, and `resultType` yields the result type of a functional type. We use these two functions to obtain a function's result type and then compare it to `CTest _`.

As only functional tests have to be transformed, `transformTest` uses `isFunctionalTest` to distinguish if a test has to be modified (Listing 5.6, lines 14-15). `isFunctionalTest` uses `stripTCons . resultType . funcType` to obtain the inner type (`a`) of `CTest a`. `isFunctionalType` then yields whether `a` is of a functional nature (lines 17-19).

If a test is functional test (cf. Listing 5.1, ll. 50-51,58-59), the actual transformation is done by `transformTest'`. `transformTest'` updates the arity of the new function (Listing 5.6, ll. 28-30), as the new test has a new argument for each of the inner type's parameters (cf. Listing 5.1, ll. 50-51 and Listing 5.2, ll. 33-35). Since we lift all the inner parameters to the outer function, we also have to update the test's signature (lines 32-34). `CTest`'s new inner type is the result type of its former inner type (lines 32-33) and all the inner type's parameter types get appended to the tests existing arguments (line 34).

Lastly we have to modify the tests rule (cf. Listing 5.1, l. 59 and Listing 5.2, ll. 42-43). We have to add some variable patterns to the left side and then modify the function calls on the right side to use these variables. `ps ++ (genPVars nestedArity)` (line 38) adds the new variable patterns to the left side of the rule. `updateRhs` and `updateExpr` generate the new right hand side of the test's rule, whereat he the only transformation is done by `updateExpr`, which applies the functions to the added arguments (lines 51-52).

classifyTests

The test have to classified in two categories: property based tests and assertion based tests. Since only property based tests can have arguments, we can use parsing to easily classify a test as property based if the test's type is of a functional nature (cf. Listing 5.1, l. 54). Listing 5.7 lines 21-22 show this partitioning. `arity` is used to get the tests arity. Because all functional tests are eliminated at this stage if the arity is greater than zero, the test is of a functional nature and expects generated arguments.

All remaining tests (`maybeProps`) cannot be classified through its type signature, as these test can either be `IO`-tests, which are assertion based, or direct comparison of values, for which we use the property based system (cf. section 5.3).

Listing 5.7: classifyTests

```
classifyTests :: VerbosityMode -> String
              -> [CFuncDecl] -> IO Tests
classifyTests m moduleName tests = do
  bs <- mapIO isProp names
  return $ makeProperties trivialProps
        ++ classify maybeProps bs
 where
  isProp :: String -> IO Bool
  isProp fname = do
    system $ cmdTemplate fname
    b <- readQTermFile "classify.txt" :: IO Bool
    return $## b

  cmdTemplate fname
    = makeCmdQuiet ("kics2␣:add␣CurryCheck␣:add␣" ++ moduleName
                 ++ "␣:eval␣CurryCheck.isProp␣" ++ fname ++ "␣:q")
                   m

  names = funcNames maybeProps

  (trivialProps, maybeProps)
    = partition (\f -> arity f > 0) tests

  classify :: [CFuncDecl] -> [Bool] -> Tests
  classify [] [] = []
  classify (x:xs) ( True:bs) = property  x : classify xs bs
  classify (x:xs) (False:bs) = assertion x : classify xs bs

  property :: CFuncDecl -> Test
  property f = PropTest (funcName f) (arity f) 0

  assertion :: CFuncDecl -> Test
  assertion f = AssertTest (funcName f) (funcType f) 0

  makeProperties = map property
```

Since we cannot evaluate *AbstractCurry* code directly, we use the modified module to evaluate all unclassified tests and then classify them based on their result. To do this evaluation, we use a function provided in the *CurryCheck* module: `isProp`. `isProp` evaluates a test to its head normal form and then determines whether it is assertion based or property based by analysing the constructor. If `IOAssertion` or `IOEqualAssertion` is used the test is assertion based and `isProp` writes `False` into the temporary file *classify.txt*, otherwise `isProp` writes `True`.

```
isProp :: CTest _ -> IO ()
isProp = writeQTermFile "classify.txt" . isProp'
 where
  isProp' t = case t of
    (IOAssertion       _ _) -> False
    (IOEqualAssertion _ _) -> False
    _                       -> True
```

To classify a single `test` in `maybeProps classifyTests` executes `CurryCheck.isProp test` using the curry system and parses the result from the temporary file (lines 8-17). This way we get a list of booleans indicating if the tests are property based or assertion based. It is then possible to generate the list of classified tests (lines 5-6, 24-35). Considering we do not know the line numbers for the tests, we use `0` as value for now (lines 30 and 33).

5.4.2. Phase 2 - `genTestModule`

The second phase of *CurryCheck* is the generation of *execTests.curry* (cf. Listing 5.3). This module contains the mapped tests of all modules and the main function that runs the tests. *execTests.curry* is generated by the `genTestModule` function.

genTestModule

`genTestModule` creates the *execTests* module. To do this, it maps the tests to the underlying frameworks (Listing 5.8, line 4), generates the main function (lines 5-6), and a list of imports (lines 10 and 11).

Listing 5.8: genTestModule

```
1  genTestModule :: VerbosityMode -> [TestModule] -> IO ()
```

```
genTestModule m modules = saveCurryCode testProg
 where
  funcs = concatMap (createTests m) modules
  mainFunction = genMainFunction moduleName
                                (concatMap tests modules)
  moduleName = "execTests"
  testProg = CurryProg moduleName imports []
                        (mainFunction : funcs) []
  imports = ["Assertion", "CurryCheck", "EasyCheck", "System"]
          ++ map newName modules
```

The list of imported modules is straight forward, as it contains all temporary modules of the first phase (`map newName modules`), the *CurryCheck* library, the frameworks *CurryCheck* is based upon (*EasyCheck* and *CurryTest*), and the *System* library for `exitWith`.

createTest

`createTests` is basically a `map` of `createTest` over the modules and as such is not discussed any further here. The mapping of the tests to the underlying frameworks is implemented by `createTest`. This function maps all assertion based tests to `Assertion a` and all property based tests to `easyCheck`/`verboseCheck` calls (cf. Listing 5.2 and Listing 5.3). Listing 5.9 shows a partial version of `createTest` which does not include most parts relevant for assertion based tests. Both modifications are of a similar nature. We will only discuss the mapping to *EasyCheck* in this section and therefore omitted the irrelevant code from the listing.

In both cases, mapping to *EasyCheck* as well as mapping to *CurryTest*, a new function is created. To make these names unique, the original test's name supplemented by ''*_ moduleName*'', whereat `moduleName` corresponds to the module name that contains the original test definition. So in our example in section 5.1 `moduleName` equals "Demo" (cf. Listing 5.2, l.17 and Listing 5.3, l. 24). Complementing the function name is necessary as tests from different source modules might have the same name.

Listing 5.9: createTest

```
createTest :: VerbosityMode -> String -> String
           -> Test -> CFuncDecl
createTest m origName moduleName test
```

```
4    = uncurry (cfunc ("execTests", (genTestName $ getName test))
5                     0 Public)
6               createTest'
7   where
8    createTest' = case test of
9      (PropTest   name arity _)
10       -> (ioType boolType, propBody name arity)
11     (AssertTest name     t _)
12       -> (assertionType t, assertBody name)
13
14   extractInnerType (CTCons _ [t]) = t
15
16   genTestName (modName, fName) = fName ++ "_" ++ modName
17
18   msg = string2ac $ genMsg (getLine test) origName (getName test)
19
20   easyCheckFuncName :: String
21   easyCheckFuncName = case m of
22     Verbose -> "verboseCheck"
23     _       -> "easyCheck"
24
25   easyCheck arity = ("EasyCheck", easyCheckFuncName ++ show arity)
26
27   propBody :: QName -> Int -> [CRule]
28   propBody (_, name) arity = [simpleRule []
29                              $ applyF (easyCheck arity)
30                                       [msg, makeProp arity]]
31    where
32     makeProp n
33       | n == 0    = applyF ("CurryCheck", "makeProp")
34                            [CSymbol (moduleName, name)]
35       | otherwise = CLambda (genPVars arity)
36                   $ applyF ("CurryCheck", "makeProp")
37                            [applyF (moduleName, name)
38                                    (genEVars arity)]
```

`createTest'` (Listing 5.9, lines 8-12) generates the function'ss type signature and the function's rules. As stated earlier, we will only focus on `PropTests`. All versions of `easyCheck` and `verboseCheck` evaluate to `IO Bool`, thus the type signature of all `PropTests` is exactly that: `ioType boolType` (cf. Listing 5.3, ll. 24,35).

`propBody` generates the function's rules. More precisely, `propBody` generates a single rule for the function as no arguments exist. In the case of mapping to *EasyCheck* the test has to be transformed into a property (`Prop`) and right the `easyCheck`/`verboseCheck` variation has to be called.

`propBody` determines the right function name using `easyCheckFuncName` and the test's arity (Listing 5.9, lines 20-25, 29). As all variations of `easyCheck` and `verboseCheck` take two arguments, a message string and a property, we can generate the corresponding code with `applyF` (lines 29-30). `applyF` is a function provided by the *AbstractCurry* library that takes a qualified function name and a list of arguments to create the representation of the corresponding function call (cf. Listing 5.3, ll. 19-22).

The transformation of the test into a property is done by calling `CurryCheck.makeProp`. This function transforms a value of type `CTest a` into a value of type `Prop`, whereat `a` cannot be a functional value. If the arity of the test is `0` we can call `CurryCheck.makeProp` directly with the test as argument (Listing 5.9, lines 33-34). If, on the other hand, the test has an arity greater than zero, we have to create an extra lambda expression of the same arity as the test with the fully applied test function as body (Listing 5.9, ll. 35-38; cf. Listing 5.3, ll. 27,32). `genPVars` and `genEVars` are two small helper functions that generate a list of pattern variable and a list of corresponding variable expressions, respectively.

genMainFunction

The core of the generated main function are the calls to two functions: `execAsserts` and `execProps`. These two functions are provided by the *CurryCheck* library and are listed below.

```
execAsserts :: String -> IO Int
execAsserts filename
  = system $ "currytest␣" ++ filename

execProps :: Int -> [IO Bool] -> IO Int
execProps failed ps = do
  x <- sequenceIO ps
  if (failed /= 0) || not (and x)
   then return 1
   else return 0
```

Since *CurryTest* already provides a way to automatically execute all associated tests in a module, `execAsserts` simply executes the *currytest* command with the given `filename` as parameter.

`execProps` takes an existing exit code and a list of `IO`-actions. It then executes all actions and calculates a new exit code based on the previous one and the results of the `IO`-actions.

Listing 5.10: genMainFunction

```
genMainFunction :: String -> Tests -> CFuncDecl
genMainFunction testModule tests
  = CFunc (testModule, "main") 0 Public typeExpr body
 where
  typeExpr = ioType unitType -- IO ()

  body = [simpleRule [] expr]

  expr = CDoExpr $
       [ CSPat (cpvar "x") execAsserts
       , CSPat (cpvar "x1")
               $ applyF ("CurryCheck", "execProps")
                        $ (cvar "x") : [testExprs]
       , CSExpr $ applyF ("System", "exitWith") [(cvar "x1")]
       ]

  testExprs = list2ac $ map makeExpr
                      $ filter isPropTest tests

  isPropTest (PropTest   _ _ _) = True
  isPropTest (AssertTest _ _ _) = False

  makeExpr :: Test -> CExpr
  makeExpr (PropTest (modName, name) _)
    = constF (testModule, name ++ "_" ++ modName)

  execAsserts :: CExpr
  execAsserts = applyF ("CurryCheck", "execAsserts")
                       [string2ac testModule]
```

`execTests`'s main function (cf. Listing 5.3, ll. 8-15) is generated in `genMainFunction`. The body of the generated function is a *do*-expression with three statements. First

`CurryCheck.execAsserts` gets executed and the result saved to `x` (Listing 5.10, lines 10, 27-29). Afterwards `execProps` is executed and the result saved to `x1` (lines 11-13,17-25). The third statement terminates the program with the exit code specified by `x1` (line 14).

To provide `execProps` with a list of `IO`-actions containing all *EasyCheck* tests we one again filter the list for property based tests (line 18-21). `makeExpr` then transforms each property based test into an *AbstractCurry* representation of the function calls to the generated test functions (cf. section 5.4.2, `createTests`). This generated list in then transformed into its *AbstractCurry* representation itself by `list2ac` (Listing 5.10, line 17).

5.4.3. Phase 3 - execTests

The third and last phase is the easiest to implement. All that remains to be done, to run the tests, is executing `execTests`'s main function. To do this *currycheck* starts a system call running the curry system (Listing 5.11).

Listing 5.11: execTests

```
execTests :: VerbosityMode -> IO Int
execTests m
  = system $ makeCmdQuiet "kics2␣:l␣execTests␣:eval␣main␣:q" m
```

6. Future Work

This chapter tries to point out some improvements that might be useful to implement in the future. We will focus on two different aspects that might be worth a more careful examination: porting *CurryCheck* to *PAKCS* and using type-classes in the implementation of *CurryCheck*.

6.1. PAKCS support

As of now *CurryCheck* is only supported by the *KiCS2* compiler because the underlying *EasyCheck* implementation uses features unique to *KiCS2*. In detail, *EasyCheck* uses *KiCS*'s ability to generate and return a search tree for a given expression. *EasyCheck* uses this as part of its *valuesOf* function that returns a list of all values an expression can assume.

It might be worth to port *CurryCheck* to *PAKCS*. To port *CurryCheck* *KiCS2*'s `SearchTrees` would have to be manually generated. As we know all the used data types when analysing the module and can obtain the data types' definitions, specifically their *AbstractCurry* representations. It should be possible to generate a set of functions that create the necessary `SearchTrees`.

6.2. Type-classes

Currently neither *PAKCS* nor *EasyCheck* support type-classes. Both compilers, on the other hand, have begun implementing them. When they are finished it might be a good idea to use type-classes in the implementation of *CurryCheck*.

It already exists a version of *EasyCheck* named *curry-test* that uses type-classes[1]. Although *curry-test* currently uses some libraries specific for the

Münster Curry Compiler[1], it could then be ported to *KiCS2* and *PAKCS* which share almost all libraries.

curry-test introduces the `Arbitrary` type-class, which introduces one function: `arbitrary :: () -> a`. As the name suggests `arbitrary` non-deterministically yields an "arbitrary" value of type `a`. All types of which *curry-test* has to generate values must provide an instance of *Arbitrary*. Although this seems restrictive at first, it is actually quite useful. It gives the user of the library the chance to specify a more useful implementation than completely arbitrary generation.

Let us look at an example. When designing and testing a data type that implements sets `Set a`, most test will probably depend on valid sets as input. When generating a set with *EasyCheck* there is no way to prevent duplicate values in the generated set. Consequently, we have to use (`(===>)`) in all tests that depend on valid sets as input parameters. With *curry-test*, on the other hand, it is possible to specify our own instance of `Arbitrary` for `Sets`. This instance can then ensure that only valid sets are being generated.

To prevent the user from having to implement instances of `Arbitrary` for all his data types, it is possible to let *CurryCheck* generate default instances if none is defined by the programmer. We can use the data type's definition and *AbstractCurry* to generate these default instances. The actual implementation of `arbitrary` is straight forward. Each of the type's constructors correlates to one alternative and the arguments of the constructors are generated using `arbitrary`. If the data type uses type parameters, they must instantiate `Arbitrary` as well. The following listing shows an example.

```
data Maybe a = Nothing | Just a

instance Arbitrary a => Arbitrary (Maybe a) where
  arbitrary () = Nothing ? Just (arbitrary ())
```

[1]e.g. the *Random* libraries are different

7. Conclusion

To draw a conclusion, will revisit the list at the end of section 1.2 one by one and determine whether the individual goals are met by *CurryCheck*.

1. **simple syntax**

   If the syntax is simple or not is of course a rather subjective point. Let us try to make an objective analysis anyway. Although there are some limitations in the provided syntax, e.g. functional tests can only be used if the test operator is used directly on the right hand side of the test definition (cf. section 4.8), *CurryCheck* tries to provide a concise interface. All operators *CurryCheck* provides to define tests are used in a similar manner, there is no need to distinguish between non-deterministic and deterministic functions, and the operators themselves are nearly identical to *EasyCheck*s operators, which helps people already familiar with *EasyCheck*. In summary it should be fair to assume that *CurryCheck* meets this goal.

2. **tests are valid Curry code**

   As shown in chapter 4 and chapter 5, all *CurryCheck* tests are written directly in Curry.

3. **provides all features of existing frameworks**

   We discussed the porting of *EasyCheck* tests and *CurryTest* tests in detail in section 4.9 and section 4.10, respectively. In these two sections it was shown how the tests can be rewritten using *CurryCheck*.

4. **tests defined inside the same module**

   Part of this goal was it, to use this property to generate more useful error messages in the case of a failure. All *CurryCheck* tests can be written in the same module as the functions they test and in fact is actually encouraged, as *CurryCheck* automatically includes the tests module and line number in the test output like proposed in section 1.2.

Also related to this goal, was the desire to keep the module's interface free of tests in contrast to *CurryTest*. Since no tests have to be exported for *CurryCheck* to find them, this goal is met as well.

5. **automatic test detection and execution**

   We have shown in great detail in chapter 5 how *CurryCheck* analyses and copies the whole module to detect all contained tests without the need to export the test functions. In the same section we have also seen how *CurryCheck* then creates the necessary infrastructure to create all the test previously found.

We can see on this list that the goals set in section 1.2 were met, but there is also still room for improvement. Especially the missing support for *PAKCS* (section 6.1) is something that should be implemented to make *CurryCheck* truly useful.

A. Installation

CurryCheck's source code can be found here[1]. The framework comes with its own *makefile* to install the *currycheck* tool and the libraries. A simple call to `make install` will run the installation process.

When `make install` is called it first compiles the *currycheck* tool and then copies the resulting executable to the compiler's installation directory (as specified by `Distribution.installDir` into the *bin* directory. If, for example, `installDir` is */opt/kics2*, *currycheck* gets copied into */opt/kics2/bin*. The installation then proceeds to copy *TreeSearchTraversal*, *EasyCheck*, and *CurryCheck* to *installDir/tools/currycheck*.

As currently only *KiCS2* is supported, the *makefile* expects *kics2* in the users *PATH*. If this is not the case or multiple versions of *KiCS2* are installed, a specific version can be specified by providing the complete path of the compiler as parameter of make:

```
make install CURRY=/opt/kics2/bin/kics2
```

To use *CurryCheck* in a project execute `currycheck -init` once inside the directory in the tool will be run. This command will copy the installed modules needed by *CurryCheck* to the current working directory and allows *CurryCheck* to find them.

[1]https://git.informatik.uni-kiel.de/jpb/master-thesis

Bibliography

[1] *Automatic Testing of Curry Programs.* https://github.com/sebfisch/curry-test. – 25.11.2015

[2] *CurryTest: A Tool for Testing Curry Programs.* http://www-ps.informatik.uni-kiel.de/currywiki/tools/currytest. – 25.11.2015

[3] *EasyCheck.* http://www-ps.informatik.uni-kiel.de/currywiki/tools/easycheck. – 25.11.2015

[4] *HTF: The Haskell Testing Framework.* https://hackage.haskell.org/package/HTF. – 25.11.2015

[5] *KiCS2: System Libraries.* http://www-ps.informatik.uni-kiel.de/kics2/lib/. – 01.07.2015

[6] *Lenses for Curry.* https://github.com/ichistmeinname/Curry-Lenses. – 25.11.2015

[7] Antoy, S. ; Hanus, M. : *A Tutorial Introduction.* Available at https://web.cecs.pdx.edu/~antoy/Courses/TPFLP/, 2006

[8] Christiansen, J. ; Fischer, S. : EasyCheck – Test Data for Free. In: *Proceedings of the 9th International Symposium on Functional and Logic Programming (FLOPS'08)*, Springer Verlag, 2008. – available at: http://www-ps.informatik.uni-kiel.de/~sebf/data/pub/flops08.pdf

[9] Claessen, K. ; Hughes, J. : QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In: *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming.* ACM (ICFP '00). – ISBN 1–58113–202–6, 268–279

[10] Dylus, S. : *Lenses for Curry*, Christian-Albrechts-University of Kiel, Diplomarbeit, 2014

[11] Fischer, S. : *On Functional Logic Programming and its Application to Testing*, Christian-Albrechts-University of Kiel, Diss., 2010

[12] Gleick, J. : *A Bug and a Crash - Sometimes a Bug is More Than a Nuisance*. `http://www.around.com/ariane.html`. – 25.11.2015

[13] Hanus, M. : *Deklarative Programmiersprachen*. `https://www.informatik.uni-kiel.de/~mh/lehre/dps14/`. Version: 2014

[14] Hanus, M. ; Huch, F. : *Fortgeschrittene Programmierung*. `https://www.informatik.uni-kiel.de/~mh/lehre/fortprog15/`. Version: 2015

[15] Hanus (ed.), M. : *Curry: An Integrated Functional Logic Language (Preliminary Vers. 0.9.0)*. Available at `http://www.curry-language.org`, 2015