Eberhard Karls Universität Tübingen
Mathematisch-Naturwissenschaftliche Fakultät
Wilhelm-Schickard-Institut für Informatik

Bachelor Thesis

Designing a Haskell Library for Interdependency Testing

Lukas Probst

31.05.2023

Reviewers

Prof. Dr. Klaus Ostermann
Wilhelm-Schickard-Institut für Informatik
Universität Tübingen

Ingo Skupin
Wilhelm-Schickard-Institut für Informatik
Universität Tübingen

Lukas Probst:
Designing a Haskell Library for Interdependency Testing
Bachelor Thesis
Eberhard Karls Universität Tübingen
Thesis period: 01.03.2023 - 31.05.2023

ABSTRACT

This thesis is proposing a new Haskell library for interdependency testing. Compiler test suites or other projects may benefit from the ability to model dependencies in an intuitive way. Two types of dependencies could be identified. Horizontal dependencies exist between different tests. Vertical dependencies exist between different instances of the same test. Disregarding these dependencies may lead to redundant and inefficient testing. A solution to effectively handle this problem is presented in the form of a Haskell library that developers can use to easily implement dependent testing in their own projects. It utilizes the Haskell testing library Hspec, and a custom monad stack designed for testing with dependencies. The resulting library was implemented in the Duo language project of the chair of programming languages at the Wilhelm-Schickard-Institute of the University of Tübingen.

ZUSAMMENFASSUNG

In dieser Arbeit wird eine neue Haskell-Bibliothek für Interdependenztests vorgestellt. Compiler-Testsuiten, oder auch andere Projekte, könnten davon profitieren, Abhängigkeiten zwischen Testergebnissen zu modellieren. Zwei Arten von Abhängigkeiten konnten identifiziert werden. Horizontale Abhängigkeiten existieren zwischen verschiedenen Tests. Vertikale Abhängigkeiten existieren zwischen verschiedenen Instanzen desselben Tests. Das Ignorieren solcher Abhängigkeiten zieht redundante und ineffiziente Test mit sich. Eine Lösung wird in Form einer Haskell-Bibliothek vorgestellt, mit dem Entwickler auf einfache Art und Weise, abhängige Tests in ihre eigenen Projekte implementieren können. Es nutzt die Haskell-Testbibliothek Hspec und einen benutzerdefinierten Monad-Stack, der für Tests mit Abhängigkeiten entwickelt wurde. Die resultierende Bibliothek wurde im Duo Programmiersprachprojekt des Lehrstuhls für Programmiersprachen am Wilhelm-Schickard-Institut der Universität Tübingen erstmals implementiert.

ACKNOWLEDGEMENTS

A very big thank you to Ingo Skupin, my supervisor, who tirelessly answered all my questions and was incredibly helpful along the entirety of the thesis.

Contents

1 INTRODUCTION

"Compilers translate programs from high-level languages into representations executable by the computer", say [Kossatchev and Posypkin 2005]. Most of the time, this translation is either into machine code, or into another programming language, a so-called *meta language*. This translation is called *compiling*. As compilers need to be able to handle every possible program that could be written in their language, they are very big and complex pieces of software, according to [Chen et al. 2020]. To make sure that they compile code correctly, they need to be rigorously and thoroughly tested. As one can imagine, testing such big pieces of software is no easy task, and more often than not a lot of effort is put into it. In fact, most compiler projects have their own subfolders just for testing purposes, a so-called *test suite*. It is in the interest of developers to keep creating new tests that check the functionality of their compiler. As test suites often run hundreds to thousands of tests, it is also essential to optimize their internal processes for minimal runtime, as well as clarity and readability when displaying the tests.

The goal of this thesis is to implement a runtime optimization, while also providing increased control to the user of the library. Consider the following example: In a compiler test suite we want to read files of the compilers programming language. Then, after parsing the content of those files into usable data formats, we perform different tests over them. We could, for example, first test whether the input file could be parsed at all, and in a second test whether that parse can also be type checked. Finally, in a third test, we pretty-print this type checked file and test whether the generated pretty-print can be successfully parsed and type checked again. Before this second type check test can be performed, a parse of the file must be available. And when testing whether a pretty-print can be type checked *again*, the information whether the file could be type checked the first time, has to be acquired beforehand. This means there are *interdependencies* between these tests. In this example, we use a collection of input files and run tests over them. However, no test will use multiple input files at the same time. Rather, we want each test to be executed on each file independently. It can be observed, that the parse, the type check calculated from it, and also it's pretty-print can be traced back to the same source. Namely, the initial input file. We will call such an isolated, independent stream of test results with a common root, a *test instance*.

Regularly, we observe such dependencies between tests. The success of one test might be dependent on the successes of one or even multiple tests that ran before it, like in the example. A naïvely implemented test suite might run all of those tests, regardless of whether those dependencies exist. We, however, want to use this observation to optimize the testing process. After all, if we already know that a test will fail due to unsuccessful dependencies, there is no point in running it. In this thesis we will call this phenomenon, where the success of certain tests depends on successful instances of other tests, *horizontal dependencies*.

Orthogonal to horizontal dependencies, there might be tests which rely on the success of other instances of the same test to be successful. Say, for example, we want to test whether a file type checks and in the first few lines of the file contain import statements. Import statements import other files. The success of this test is dependent on whether the imported files could be type checked as well. Instead of type checking all the imported files again for every instance of this test, it might be more effective to remember which files have been (un)successfully type checked. We can also predict, that the current test instance will fail if one of the imports does not type check in the first place, and immediately abort in this case. We will call this type of dependency, where the success of a test input is dependent on another instance of the same test, *vertical dependencies*.
The different types of dependencies can be seen in fig. 1. On the left, horizontal dependencies

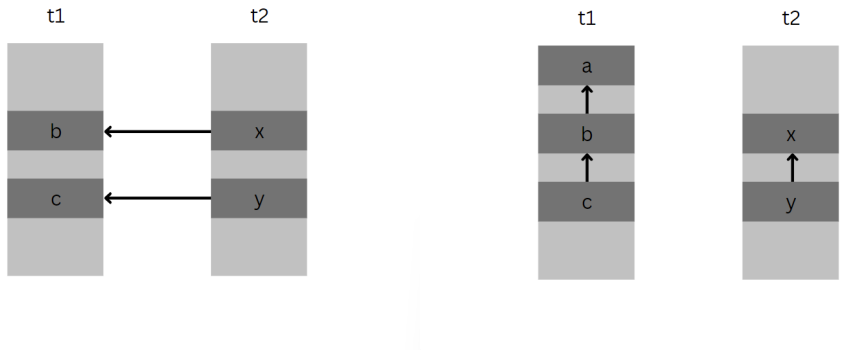

Fig. 1. A visualization of horizontal dependencies on the left and vertical dependencies on the right

between the same instances in the tests `t1` and `t2` can be seen. The test values `x` and `b`, as well as `y` and `c` are of the same instance. The arrows symbolize that test value `b` can only be successful, if `x` was successful. Similarly, `c` can only succeed, if `y` succeeded. On the right side are vertical dependencies between different instances of the tests `t1` and `t2`. Here, the success of `c` is dependent on another instance of `t1`, namely `b`, which is again dependent on `a`. Similarly, `y` is dependent on `x`. If all dependencies for a test value `t` were successful, we will say that the dependencies for `t` were *fulfilled*.

The programming language project Duo[1], in which this library will first be implemented, so far uses a linear compiler test-suite. With that we mean that tests which could be pruned away, due to horizontal or vertical dependencies, are not handled in any meaningful way. These failing test instances are also propagated throughout the testing process, which clutters the terminal output, making it hard to pinpoint the cause of the test failures. In this thesis we will design a Haskell library that allows users to model interdependent tests with intuitive API style functions. To implement the testing functionalities, the Haskell HSpec library is used to perform grouped tests.

Ultimately, the aim is to create a closed Haskell library which can be flexibly used in any project that may require dependent testing.

This thesis is structured as follows:

- In section 2 we will explain the theoretical basics.
- In section 3 we will present the main ideas.
- In section 4 we describe our approach to the implementation.
- In section 5 we conclude and discuss which ideas were not implemented as well as possible future features of the library.

2 PRELIMINARIES

2.1 Monads in Haskell

As the implementation of this library will utilize a custom monad, it is essential to understand how monads work in Haskell. In essence, monads are type constructors, which are used to chain commands in a consistent and predictable way. They also enable the use of side effects, like input

[1] https://github.com/duo-lang/duo-lang

```haskell
sequence :: IO ()
sequence =
    \_ -> 2 + 5 >>= s -> s / 2 >>= half -> print half >>= _ -> print s
```

(a) A sequence of `bind` operations in an IO monad.

```haskell
sequence :: IO ()
sequence = do
    s <- 2 + 5
    half <- s / 2
    print half
    print s
```

(b) The same IO monad sequence in do-notation. Note that all expressions are evaluated in a top-down order.

Fig. 2. A monadic sequence in bind-notation and in do-notation

and output operations, while keeping the pure nature of a functional programming language like Haskell. To create a monad, one needs to define two functions for it: The `pure` function defines how a value can be put inside this monad in the simplest way. The `bind` function, denoted as `>>=` defines how this monad chains operations together.

There are a few monads already implemented in the standard Haskell library, two of which are of importance to this thesis. One of them is the `Reader` monad. The `Reader` monad enables the user to pass an arbitrary value when creating an instance it. This value can then be read out at any point in the monad. In our custom monad we will utilize this to be able to have a global configuration. Inside the monad, at any point, the `ask` function can be used, to get access to the value written in the reader monad.

The second monad that holds significance for us is the `State` monad. Like the `Reader` monad, it accepts an arbitrary value during construction, called the *state*. This value can be accessed inside the monad via the `get` function. Unlike the `Reader` monad though, the state can be manipulated at runtime via the `put` function. We will use the reader monad to keep track of test results.

The arguably most important feature of monads in Haskell is the so-called *do-notation*. It allows programmers to use an almost iterative programming style in a language that originally only supports declarative programming, as seen in fig. 2. It depicts an example of the same sequence of operations, one as a chain of `bind` operations, the other in do-notation. One can observe, that the evaluation follows a strict order, and binds the result of a function as a value to give to the next function. It is also notable, that we can again use the variable `s` later in the sequence, even long after binding it. The iterative style of the do-notation enables us to not only force an evaluation order, but also make it clearly visible, as statements are evaluated from the top down. Binding "variables", which are in reality only partial results of the whole bind-chain, is done in this iterative "one line after the other" fashion.

It is possible to unify the properties of different, already existing monads by utilizing *monad transformers*. Monad transformers define a way in which a monad can be "stacked" on another monad. For this purpose the type signature of a monad transformer expects a monad `m` to be passed and embeds the solution of its operations in the constructor of `m`. This way, we can stack an arbitrary

```
main :: IO ()
main = hspec $ do
    describe "Prelude" $ do
        describe "read" $ do
            it "can parse integers" $ do
                read "10" `shouldBe` (10 :: Int)

        describe "head" $ do
            it "returns the first element of a list" $ do
                head [23 ..] `shouldBe` 23
```

Fig. 3. An example usage of the Hspec library, grouping together 2 tests and running them. Source: [HSpec 2011]

amount of monads on top of each other. Like with monads, many monad transformers are already implemented in Haskell, for example the reader monad transformer `ReaderT` and the state monad transformer `StateT`.

2.2 Hspec

Hspec is a testing framework for Haskell. It works by creating and passing around values of the type `Spec`. With Hspec it is very easy to group tests together. In fig. 3 two tests are grouped under the description "Prelude". Both get their own descriptions, "read" and "head" respectively. With the `it` function, in combination with `shouldBe`, both tests create a value of type `Spec`. Also note how both tests are put in a monadic sequence via the do-notation and therefore will be executed after each other. In the end, the tests are executed by using `hspec` on the resulting `Spec`.

The functions of Hspec all use and return a monadic type `SpecWith a`. However, for developers using the library, the type `Spec` can be used as a substitute without type argument. A `Spec` can be sequenced with other `Spec`, and they contain tests that can be executed. The most important function to create a `Spec` is the function

```
it :: String -> Expectation -> SpecWith (Arg Expectation)
```

that accepts a **String** description and an `Expectation`. It creates a single unit test in conjunction with

```
{shouldBe :: forall a. (HasCallStack, Show a, Eq a) => a -> a -> Expectation}
```

that compares two values and provides such an `Expectation`. Via the function

```
{describe :: forall a. HasCallStack => String -> SpecWith a -> SpecWith a}
```

unit tests can be grouped together under a shared description. There is also a `pending` function, which can be used to implement pending tests. We utilize `pending` to mark tests that did not have their dependencies fulfilled.
The functions `it`, `describe` and `pending` all return `Spec` values. To run them and display the results on the terminal, the function `hspec` may be called inside an IO monad. It is important to underline, that a `Spec` only contains instruction of how a test is executed, and only the `hspec` function creates a visible output for the user. After the tests run, `hspec` returns an empty IO value. This means that we are unable to use any of the results that we might calculate, the Hspec library is solely made to run tests and display their results.

```
testingPipeline :: SomeTestingMonad
testingPipeline = do
    result1 <- runTest "description" testValues [] func1
    result2 <- runTest "descr" result1 [] func2
    result3 <- runTest "descr" testValues [result1, result2] func3
```

Fig. 4. An example test specification running three tests, and displaying the use of horizontal dependencies, in a monadic style.

3 MAIN IDEAS

3.1 A Testing Monad

The central goal of this thesis is to implement a simple way to run dependent tests. We want to model dependencies that enable us to skip certain tests for which we can predict a failure. For horizontal dependencies this means to make sure that previous tests on this instance ran successfully. To enforce that the dependencies of tests are run before them, we decided to take a monadic approach. The basic idea is shown in fig. 4.

In the testing monad, tests should be called by a *test running function*. In fig. 4 this is `runTest`. This function will perform tests over a list of *to-test values*, which are values that will each be tested by a given *testing function*. There are four arguments passed to `runTest`.

Firstly, there is a description in form of a **String**. It will be used to group the single tests together under a common description, i.e. it will be the argument passed to the `describe` function of the Hspec library.

Secondly, a list of to-test values is passed. In the example, this list is called `testValues`. Every to-test value is an instance of this test, meaning that the testing function will be applied to each of the to-test values individually. In section 1 we introduced an example that tested whether files could be parsed. In that case the list of to-test values would be a list of input files, or a list of parses.

Next, the function should accept dependencies. For now, we will focus on horizontal dependencies, and talk about vertical dependencies later on. In fig. 4, they are passed as a list of previous test results. If the list is empty, there are no horizontal dependencies for this test. Only in the third call of `testRun` there are horizontal dependencies, namely `result1` and `result2`.

The last argument passed to `testRun` is the testing function, in the first call that is `func1`. These testing functions are not included in this library and need to be defined by a user. The parsing test and the type check test from section 1 would be examples for such testing functions. They are supposed to accept a to-test value and return an output containing information about result of the test. The plan is, that `runTest` will run a testing function over every element of the list of to-test values.

After the first tests are evaluated, we want to store the results in a variable. In the example, the first results are stored in `result1`. Afterwards, we run a second test with a second testing function `func2`. As to-test values we now pass the results of the first test, `result1`. This means that all tests that failed in `result1` will not be tested in this call of `runTest`. Even though we did not pass any horizontal dependencies explicitly, by reusing test results, we implicitly create horizontal dependencies to the previous test run. In the third call of `runTest` we perform the testing function `func3` over our initial to-test values `testValues`. This time, `result1` and `result2` are used as horizontal dependencies. In other words, the testing function `func3` will test the values of the initial `testValues`. However, only instances that were successful in the first two test calls of `runTest`

will be considered. This means that, if the horizontal dependency for an instance at index `i` was not fulfilled, the to-test value at index `i` is not applied to the testing function.

Especially in do-notation it is visible, how this monadic approach forces the evaluation of dependencies before their dependent. Results can be reused as to-test values or passed as horizontal dependencies to a new test. To reuse them we have to first bind them to a name. Whether the results become to-test values or dependencies for new tests, by doing so, it is guaranteed, that they were evaluated before being used.

Ideally, the test results are not only available locally, where a result variable is created, but at any point inside the monad. So, to continue in our monadic approach, we want to save the results in a state, and retrieve them by an index. This way the results from a call of our test running function will only be a pointer to some results in the state. The `Spec` that are created from running our tests should also be saved in one central place. This will also happen in the state. This way, all results can be retrieved from the state at once, with one call of the `get` function.

We also want to be able to give the developer the ability to decide whether they, for example, want to display tests that were not executed due to failing dependencies, or not. There are different methods to implement this, but with extensibility for more configuration options in mind, a `Reader` monad could be the best option here. As we need the capabilities of a `Reader` monad for configuration, and the capabilities of a state monad for storing results, we use a monad transformer to unify these properties in a single monad. As mentioned, transformers assist in building monads on top of each other. This means that they use a value of a given type and a monad to construct a new monadic value. In the `TestM` Monad, we use two transformers in order to be able to stack yet another monad of the users choosing. This design was picked specifically with IO operations in mind. In the Duo project's test suite, it is necessary to read files for the parse test. Reading files is an **IO** operation, and Haskell allows such operations only inside an IO monad, to remain pure. With this design, testing functions can take the form of `test :: MonadIO m =>a -> m b`. Therefore, **IO** can be used without leading to complications.

In the rest of this section, we will discuss the different aspects that will play together in this testing monad. First, let's talk about the testing functions that we want to use on our to-test values.

3.2 Testing Functions

Like discussed in section 2.2, Hspec does not give us a possibility to continue working with test results in any way. But, as we want to model dependencies between tests, we do not only need to be able to assess that a certain test was successful, but also to use its result in further calculations. Here we have to draw an important line between the results of a test, i.e. the result of the calculations done within a testing function, and the success of a test, i.e. whether it produces the expected result and therefore did not fail. If we read and parse a file for our parsing test, and in the next step try to type check it, it would be a waste of computing resources to do the first two steps again. Likewise, if the parsing test was not successful we do not want to repeat a parse that did not work in the first place.

To achieve this, we need functions that return both, a result with the type **Maybe** `b`, and a Hspec executable `Spec`. This will be in the form of a tuple. As mentioned, we also give the user a possibility to embed an additional monad in the monad stack. This could be an IO monad, but does not have to be. We want the testing functions to be run inside this monad `m`. If they take an argument with type `a`, they have the type signature **Monad** `m =>a -> m (`**Maybe** `b, Spec)`. This type of functions we will call *testing function*. The testing functions are not part of this library, but supposed to be user-defined. To be compatible with this library they need to follow the described type signature. Figure 5 shows a testing function. This particular example tests whether a given

```
biggerThan5 :: Monad m => Int -> m (Maybe Int, Spec)
biggerThan5 n = let success = n > 5
                    spec = it (show n ++ " is bigger than 5") $ success `shouldBe` True
                in if success
                    then return (Just n, spec)
                    else return (Nothing, spec)
```

Fig. 5. An example for a testing function, testing whether a value is bigger than five

Int value is bigger than five. First, it is determined whether the test was successful, by comparing the **Int**. A `Spec` is created by using the `it` function on an `Expectation`. The `Expectation` here is `success `shouldBe` True`. Then, depending on whether the test was successful, either the number wrapped in a **Just** or a **Nothing** is returned, together with the test's `Spec`.

The result of this function is the **Int**. In this case we did not calculate a new value but just returned the input. However, as the design of these functions is left to the user, the test result could be calculated and the success determined by the result of this calculation. The success of the testing function is indicated by giving the result the type **Maybe** `b`. Only if a value wrapped in a **Just** constructor is returned, the test was successful. If a **Nothing** is returned, the test failed. The purpose of the `Spec` is to generate a test output that can be displayed on the terminal.

As one can see, the success of the test was determined on the premise `n > 5`. Because this information is needed to create the expectation, as well as to determine whether we return a success or a failure, to reduce redundancy it is practical to bind it to a name.

3.3 Different Types of Dependencies

Dependencies may exist between instances of different tests. Let's take an example out of the Duo project's test suite. When the test suite is started, a list of file paths is generated. In the parse test, these duo files are then read, and their contents are parsed into a *concrete syntax trees* (CST), a tree structuration of commands. All other tests that follow this, require CSTs, or a product of transforming CSTs, as to-test values. Take for example its immediate successor, the type check test. It takes the CST of a file as an argument and tests whether the tree can be assigned a type, constructing a *typed syntax tree* (TST). Because the type check test requires a CST as argument, without a successfully parsed file this type check can never be successful. The same is true for all other tests, that also require a CST of this instance as input, as well as all tests that require a TST of this instance.

These *horizontal dependencies* allow us to effectively prune away the tests, that depend on the success of a preceding test of the same instance. Our monadic approach already enforces that all horizontal dependencies are run before the current test. By using lists of test results we ensure that the indices of these results are always the same for the same instance. This means, that if the input file of the parse test in our example had index `i`, the type check test result for this instance will also have the index `i` in the list of results of the type check test. Horizontal dependencies are defined in the context of those indices. In general, a certain index in such a list of test results always equals the index of a to-test value in the original list.

In other words, imagine a user introduces a list of to-test values `[1,2,3]` that, after a certain amount of tests, arrived at the list of results `["foo","bar","baz"]`. By our design, it is always true, however many tests ran between these two lists, the `"foo"` value is of the same instance as `1`, i.e. the chain of tests, that had `1` as original input, now arrived at the result `"foo"`. The same is the case for `2` and `"bar"`, and `3` and `"baz"`, for every index of these two lists and for all list of test results that

were intermediate results between them.

Dependencies may also exist between instances of the same test. To explain this phenomenon, we'll use the example from the Duo project's test suite again. When we want to run the type check test on a CST, there might be some imports in the first few lines of code. As we assume that all imports will be used in the file, if one import would fail the type check test, the current file will certainly also fail it. If we collect these *vertical dependencies* before testing the current file, and run the same test on them, it is possible to prune away unnecessary redundancy.

Imports are the only vertical dependency we observed in the test suite of the Duo project. But any algorithm with a *divide and conquer* approach, for example, could use vertical dependencies as verification of its correctness. A simple example for this could be *merge sort*. This may not be a particularly practical example, as merge sort is very well documented, however, it accurately describes a vertical dependency approach to testing. Merge sort recursively splits a list in two until there are only single item lists remaining. It then unifies these partial problems again, producing correct partial solutions for every depth of the recursion.

To now verify that merge sort was implemented correctly, i.e. sorts both parts of a split list in the correct order, we can use vertical dependencies. Say, for example, our list is `[3,2,4,1]`, then we could formulate a vertical dependency stating that merge sort only works correctly if the lists `[3,2]` and `[4,1]` are sorted correctly. Moreover, we could define a function that just splits a list in two halves, and use this function as a vertical dependency every time we test whether merge sort works. Then, during testing, we make sure that all the results of this function, in this example the halves of this list, return successful test results. Only in this case we proceed to run the test over the whole list. This dependency is recursive, meaning that the halved lists also will be halved and their respective halves tested before them, and so on. We want to use vertical dependencies exactly in this way. As functions that calculate one or multiple dependencies from a to-test value. These dependencies are then required to pass this test beforehand.

It is up to the user of this library to determine which dependencies a test has, as well as declaring horizontal dependencies, and defining functions that calculate vertical dependencies. Now that we discussed how dependencies will be implemented, let's take a look on how we treat test results depending on their success.

3.4 Maybe as Results

As we have to assume that some tests fail, we have to think about how we handle successful and failed tests. If a test succeeds, we want to keep the information that it was successful, as well as the result. If it failed, we don't need its result, or there might not even be one. But, we want to keep the information that it failed, because test failures are propagated through dependencies. **Maybe** provides us with exactly this functionality.

Maybe is an implemented monad in Haskell. **Maybe** `a` constructs us a type that is either **Nothing**, which is a constructor with no argument, or **Just** `a`, which is a constructor with one argument that has to have the type `a`. As **Maybe** is a monad, and all monads implement **Functor**, we can apply the function `fmap :: Functor f =>(a -> b) -> f a -> f b` to a value with the type **Maybe** `a`, like in fig. 6. For **Maybe** `a` typed values, `fmap` behaves in a way that it applies a Function only to the values wrapped in a **Just**, and passes a **Nothing** otherwise.

Successful tests produce a **Just** `b` as result, failed tests produce a **Nothing**. This enables us to pattern match whether a test was successful without needing to know the type of the result.

```
fmap (fmap (* 2)) [Just 1, Nothing, Just 3]
:> [Just 2, Nothing, Just 6]
```

Fig. 6. An example usage of `fmap` over a list of **Maybe Int** values. The `fmap` is applied twice, once to the list and once to the **Maybe**s.

```
data TestResult = forall a. TestResult a

hidden :: TestResult
hidden = TestResult 6
```

Fig. 7. Creating an existential type `TestResult`.

```
listOfValues :: [TestResult]
listOfValues = [TestResult 6, TestResult "foo", TestResult True]
```

Fig. 8. The existential wrapper now allows us to create lists, in which we can put values of different types.

3.5 Existential Hiding of Test Results

We can now determine the success of a test without needing the type of its result. To continue working with the result of a test, it needs to be stored. With our state monad approach, we need to define which components the state contains beforehand. It is therefore not possible to store the results in different lists. To do this, we would need a way to change the amount of components of a record during runtime. Our aim is therefore to store all test results in a single list. The problem with this is, test results could have any type, and we can't store values with different types in the same list. This is where existential hiding comes into play.

In Haskell, which is a strictly typed language, every term needs to be given a valid type. In order to create a list which contains values with different types, we can hide the type of values using existential types. Existential type constructors, after taking an argument, do not have a type argument in their type. Figure 7 shows the construction of an existential type. The type constructor `TestResult` can take any argument with type `a`. The type of the variable `hidden`, even though it was constructed with an **Int**, is just `TestResult`. It is visible that the information about the type of the wrapped value has been "forgotten". It has, in fact, really been forgotten. The `TestResult` wrapper still points to a place in memory, meaning that the information is still there in the form of raw bits. But all information which type that value had, is lost. This effectively allows us to store values of different types in a list, without the need to explicitly state their type, by putting them into an existential type wrapper. An example for this can be seen in fig. 8. The values `6`, `"foo"` and **True** are put into an existential wrapper `TestResult`. A list can be constructed from the resulting values, where the information about the types is forgotten, resulting in a list with type `[TestResult]`.

The drawback for this is, that in order to again gain access to the stored value, we need to assign a type again. To do this, we need to coerce the original type. In Haskell we can use the `unsafeCoerce` function for this. The coercion of these hidden types proved to be one of the biggest hurdles of this thesis.

Existential hiding will assist in hiding the type of the test results. The recovering of the hidden types is left to Haskells type inference system. To do this, we create an existential type `TestResult`,

```
f :: Int -> Int
f = id

x = f $ unsafeCoerce (head listOfValues)
```

Fig. 9. The coercion of a type that was lost when constructing a `TestResult` value. The value of `x` is 6

which we use to wrap test results. By passing whichever value is inside the wrapper into a testing function, Haskell can infer its type, as the type signature of the testing function states the type of its input.

To realize this type, `unsafeCoerce` comes into play, shown in fig. 9 To recover the data from a list of existentially hidden values, here `listOfValues`, it is necessary to retrieve information about the type. A developer can either know which type to retrieve and state it explicitly, or use type inference. To showcase this, a dummy function `f` is defined. Here, the function `f` demands an **Int** as its argument. This means, that if Haskell uses `unsafeCoerce` on the list head, the type inference system knows, that the result must be an **Int**, otherwise `f` couldn't be called on it. In other words, the type system infers that the incoming value of a function should have a certain type. Then, `unsafeCoerce` realizes that type, by interpreting the value with the "lost" type, as a value of the inferred type.

With the problem of different type signatures out of the way, it is now possible to construct a custom monad, that will handle saving our test results, recovering them when we need them, and assist in modelling dependencies.

4 IMPLEMENTATION

In this section the details of the implementation are discussed. The code can be found in the GitHub repository.[2]

4.1 The `TestM` Monad

The `TestM` monad is implemented as a monad stack of a reader monad transformer and a state monad transformer. The `TestState` is a record with two components which will serve as the state. The first component is a list with test results `tests :: [TestResults]`. The type `TestResults` is a list of results of individual tests, **type** `TestResults = [Maybe TestResult]`. The second component of our state is `testSpecs :: Spec`. We use this to accumulate `Spec` before running them with `hspec` in the end. `Spec` can be sequenced using the sequence operator >>, which creates a new value with type `Spec` that, when run with the `hspec` function, just executes the two sequenced `Spec` after each other.

To add new tests to the state, the function

```
addTestResult :: MonadState TestState m => ([Maybe b], Spec) -> m ()
```

can be used. This function also takes the incoming `Spec` and just sequences is after the already existing `Spec` in the `TestState`. It also adds the incoming test results with type `[Maybe b]` to the end of the `tests` list. The test running functions, that will be talked about in the next section, then return only the index this test got in the `tests` list. We can use this integer index to retrieve the results of tests that already ran, or when we want to declare horizontal dependencies.

We also want to use a `ReaderT` which will allow us to set a certain configuration. There is one configuration implemented as type `Config` in this thesis. It offers the two options `DefConf :: Config`

[2] https://github.com/LukasProgress/interdependency-testing

```
runTest :: (Eq a) => Monad m =>
    Description                          -- A Test Description
    -> [a]                               -- A List of To-Test Values
    -> (a -> Maybe [a], [Int])           -- The Dependency Tuple
    -> (a -> m (Maybe b, Spec))          -- The Testing Function
    -> TestM m Int                       -- Returns the ID of the Test
```

(a) The type signature of `runTest`

```
runTestFromResult :: (Eq a) => Monad m =>
    Description
    -> Int                               -- An ID for Test Results
    -> (a -> Maybe [a], [Int])
    -> (a -> m (Maybe b, Spec))
    -> TestM m Int
```

(b) The type signature of `runTestFromResult`

Fig. 10. The type signatures of both testing running functions. The `runTest` function introduces new instances, while the `runTestFromResult` reuses instances from existing test results.

and `PendingConf :: Config`. These will allow the developer using this library to decide, whether they want to display only the tests that ran without problems, or include the ones for which the dependencies were not fulfilled. The latter can be done with `PendingConf`, which will display all these tests as *pending*.

In our testing monad we now stack the configuration and the state together in

```
TestM m a = ReaderT Config (StateT TestState m) a
```

An additional, variable monad `m` is included, which allows developers to implement a monad of their choice in their testing functions. This is especially useful for running **IO** operations in the testing functions. For this purpose, the `TestM` monad also implements `MonadIO`.

4.2 How to Run Tests

There are two functions that are able to run tests, i.e. that are test running functions. The `runTest` and the `runTestFromResult` functions can both be used to run testing functions on to-test values, taking dependencies into consideration. They differ in their purpose. Introducing new test instances is done by a call of `runTest`. Whenever the user of this library intends to introduce a new list of to-test values to the testing process, the `runTest` function has to be used. This also means that the first call of a test running function always is `runTest`, because we need to introduce new instances if there are none, to run tests at all. On the other hand, the `runTestFromResult` function builds tests on already existing instances. This means, that it reuses the test results from tests that ran previously.

The complete type signatures for both of the test running functions can be seen in fig. 10. It can be seen that the second argument is the only difference between these type signatures. A call of `runTestFromResult` requires four arguments.

First, a description is given as **String**. The description will give this call of the test running function a name. This name will be displayed in the terminal when we run the collected `Spec` values, and the results of this test will be grouped under this description.

The second argument is an ID, given as an **Int**, that corresponds with an index in the `tests` list of the `TestState`. The test ID is looked up in the list and resolved to a list of test results.

The third argument is the dependency tuple. It carries a function that calculates vertical dependencies, as well as a list of integers which represent the IDs of the tests of the horizontal dependencies. These dependencies are to be defined by the user of the library. In the case that a test has no dependencies, the library offers the constant `noDeps = (const [], [])`.

The fourth argument for `runTestFromResult` is the testing function. Here, the user passes a function that accepts values that have the same type as the values behind the resolved test result ID. This is essential, as we need to infer and coerce the type these values had before they were wrapped in an existential type. Only if the types match, a test will run without error. The result of this testing function is a tuple inside a monad `m`. The other specifications of testing functions were described in section 3.2.

The technical difference between `runTestFromResult` and `runTest` lies in the second argument. Because `runTestFromResult` reuses test results, it accepts an index to retrieve them from the test state. The test results from the state are wrapped into the existential type `TestResult`. Before we can pass them to the testing function, their type has to be coerced. In contrast, `runTest` was designed to introduce a new list of instances to the process, so it accepts a list of to-test values that has the type `[`**Maybe** `a]`. The to-test values of that list can directly be passed into the testing function. This difference requires us to write two different testing algorithm functions, one that works with `TestResult` values, and one that works with "normal" values.

After taking dependencies into account and running the testing function with every to-test value, by calling a testing algorithm function described in section 4.3, the results come back in form of a list `[(a, (`**Maybe** `b, Spec))]`. The first value in this tuple, with type `a`, is the to-test value over which the testing function was executed, the second value of this tuple is the result that the testing function returned for this specific to-test value. The reason for that is, that we might run more tests than entries in our list of to-test values, due to vertical dependencies. By attaching the original value that was tested to the result, we can afterwards filter the results to match the original list of to-test values.

The right side of the tuple is another tuple, containing a **Maybe** `b` and a `Spec` value. The first of those two is the result of the test. The `Spec` contains a single Hspec test of this specific to-test value with this specific testing function. As we get a list of these tuples, where the original to-test value is attached to its result, we may speak of a *result dictionary*. It can be used to look up the result to a specific to-test value. As mentioned, the results are filtered with the original list of to-test values. To do this we look up all the elements of our original to-test list in the result dictionary, sorting the results in the order of the to-test values in the same step. Depending on the configuration, the `Spec` of tests, for which the dependencies failed, might also be dropped. In the end, the results and `Spec` are written into the corresponding variables in the `TestState`.

Ultimately, both `runTest` and `runTestFromResult` return an integer ID, which can be used for future tests. We can then use these IDs to give to-test values to `runTestFromResult`, or use them as horizontal dependencies.

4.3 Testing with Dependency

We only want to run tests for instances where all dependencies are fulfilled. The dependencies of an instance are fulfilled, if none of them have a **Nothing** as result. We first check horizontal dependencies, and then vertical dependencies. Horizontal dependencies are checked in a method we will call *horizontal filtering*. This means, that we set all to-test values, for which a horizontal dependency was not fulfilled, to **Nothing**.

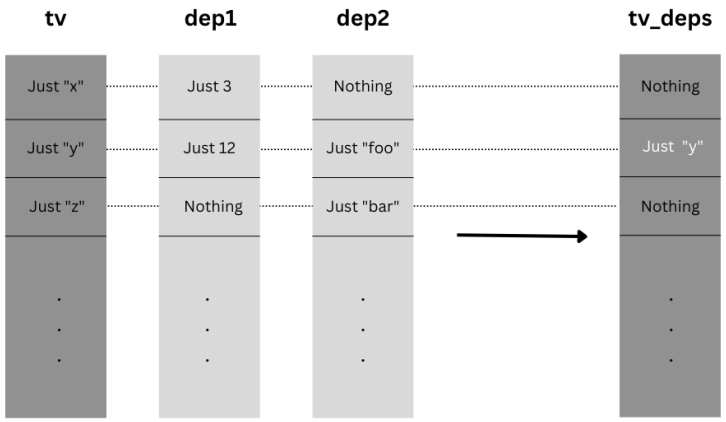

Fig. 11. The list of to-test values (tv) and the two horizontal dependencies dep1 and dep2 at index produce a list of to-test values with successful dependencies (tv_deps), by comparing their elements at every index `i`.

Figure 11 provides a graphical idea of how this is done. The `tv` column is the list of to-test values, that was given to the function, while `tv_deps` on the right-hand side is the list of to-test values after horizontal filtering, that will actually be tested. The two columns `dep1` and `dep2` represent the horizontal dependencies declared by the user. The cells inside the columns all represent the test results of single instances of a single test. In case that this would be a call of `runTest` however, the `tv` column would be a list given by the user, instead of reused test results. Only the first 3 elements of each list are shown, the dots in the lower half suggest that these lists may be of arbitrary, but identical, lengths.

Here, a list of to-test values `tv` was given to the test running function, along with the horizontal dependencies `dep1` and `dep2`. The dotted lines suggest, that each instance corresponds to exactly one index, and all lists that occur are arranged in a way so that the value at the index corresponds to this one instance. If any entry in a dependency is **Nothing**, the dependency is not fulfilled. In this case the index i in the resulting list of filtered to-test values `tv_deps` is set to **Nothing**. If all values x of an index in the to-test value list, as well as in all dependencies, are a **Just** `x`, the to-test value is kept in the filtered list.

Since indices correspond to instances, we can use a rather easy approach of implementing the horizontal filtering. In essence, it is a matrix transposition. As a matrix, imagine the columns of fig. 11 next to each other. The leftmost matrix column are the to-test values, the others are the horizontal dependencies. The rows of the matrix are then the ith element of every test, i.e. they are all the same instance. For the transformations, the lists are aggregated at their ith entries into a new list, and then check whether any entry at this index was **Nothing**. In that case either a dependency was not fulfilled, or the entry in the to-test value list was already **Nothing**. Of lists that had no **Nothing** we keep the head, i.e. the to-test value. This way, all to-test values, for which one or more horizontal dependencies were not fulfilled, are set to **Nothing**, and will be skipped while running the tests.

The resulting list is now passed into one of the functions that actually run tests, the *testing algorithm functions*. There are `depTesting` and `depTestingWrapped` which are used by `runTest` and `runTestFromResult` respectively. As mentioned, because of the difference of `runTest` and

```
depTesting :: Eq a => Monad m =>
    [(a, (Maybe b, Spec))]                -- A Result Dictionary
    -> [Maybe a]                          -- The To-Test Values
    -> (a -> [a])                         -- A Dependency Function
    -> (a -> m (Maybe b, Spec))           -- A Testing Function
    -> TestM m [(a, (Maybe b, Spec))]
```

(a) The type signature of `depTesting`

```
depTestingWrapped :: Eq a => Monad m =>
    [(a, (Maybe b, Spec))]                -- A Result Dictionary
    -> [Maybe TestResult]                 -- The To-Test Values
    -> (a -> [a])                         -- A Dependency Function
    -> (a -> m (Maybe b, Spec))           -- A Testing Function
    -> TestM m [(a, (Maybe b, Spec))]
```

(b) The type signature of `depTestingWrapped`

Fig. 12. The type signatures of both testing algorithm functions. The `depTestingWrapped` function handles values wrapped in `TestResult`, while `depTesting` accepts a list of unwrapped to-test values.

`runTestFromResult`, we need two different testing algorithm functions. One that handles tests that are wrapped in an existential type, and one that handles unwrapped values.

The purpose of both functions is running a testing function over every element of the list of to-test values. At the same time, for every element the vertical dependencies are calculated and tested beforehand. These two steps were not isolated for two reasons. First we want to keep track over which values were tested, and reuse these gathered results, in case of duplicates. Secondly the calculated vertical dependencies might have vertical dependencies again. So we use the very same function to test all the to-test values, as well as their vertical dependencies. To implement this "testing before testing" approach of vertical dependencies, we use recursion.

The type signature of both testing algorithm functions can be seen in fig. 12.

The first argument serves as an accumulator. It is a dictionary mapping to-test values to their test results, a *result dictionary*. We utilize this to collect the results, but also to look up before testing, whether we already know the result for a given to-test value, to prevent redundancy. When the algorithm ends, i.e. when the end of the list of to-test values is reached, this result dictionary contains all to-test values mapping to their respective results, and is returned.

The last argument of both functions is a testing function. This is a function defined by the user of the library. In case of parse testing, this would be a function which takes the file name, reads its content, and tries to parse that into a syntax tree. Then it checks whether the parsing was successful, and returns this information along with the parsed tree. The details of what these functions look like were discussed in section 3.2.

The third argument is the dependency function, which is used to calculate vertical dependencies and also defined by the user of the library. It has the type `a -> [a]`, which means the dependencies for a particular to-test value are calculated as a list. If the result of the function is empty, the input has no vertical dependencies.

The second argument of both functions is the list of to-test values, that was horizontally filtered. For `depTestingWrapped` they all have the type **Maybe** `TestResult`, while for `depTesting` they have the type **Maybe** `b`. The head element of the list, we call the *current value*. We recurse over this list, looking at the current value and running our algorithm on it, before removing the current value from the list, and running the function again on the tail. If the list is already empty, the result dictionary is returned. If the current value is a **Nothing**, then the dependencies for this particular instance already were not fulfilled. In that case, the testing algorithm function calls itself with the rest of the list, which means that we simply skip this instance. If the current value is a **Just** `x` for any `x`, the actual testing algorithm is used.

First, the current value is looked up in the result dictionary to determine whether the result has already been calculated. This could happen either because the current value was a horizontal dependency for another instance, or because the current value is a horizontal dependency that was already evaluated for another to-test value. As we assume that duplicate to-test values result in duplicate results, if we find an existing result, we can simply drop the current value and call the testing algorithm function again with the rest of the list of to-test values.

If no result is found, the current value still needs to be tested. To do this, we first determine its vertical dependencies by passing the current value as an argument to the dependency function. If this function returns an empty list, it means there were no dependencies. In that case we can directly run the testing function on the current value. After this, the current value and its result can be attached to the result dictionary, and we recurse with the rest of the list.

If there are dependencies however, we want to check whether they are successful before executing the test on the current value. In that case, the test algorithm function calls itself recursively. This is the second way in which recursion is possible. We pass the already collected result dictionary, and the dependencies as new to-test values. By recursing over the dependencies, we make sure that every vertical dependency is tested before running the testing function with the current value. When this recursive call returns, the results for all dependencies will have been collected in the result array. We can then check the returned result dictionary, if all dependencies also were successful, i.e. whether any of them returned a **Nothing** as test result. If they were not, the current to-test values test result is also **Nothing**. If they were successful, we run the testing function with it. As we want to keep the information about test results that we acquired during recursion, after running this testing function we append the tested value and its result to the returned result dictionary. We then pass this dictionary when calling the testing algorithm function again with the rest of the list of to-test values.

As mentioned, the two testing algorithms handle different types of to-test values. Therefore, the way they call the testing function is slightly different. The `depTesting` function works with to-test values that have the type **Maybe** `a`. It can therefore pass a current value `x :: a` directly into a testing function `f :: Monad m =>a -> m (Maybe b, Spec)`.

In contrast, `depTestingWrapped` works with to-test values that have the type **Maybe** `TestResult`. Before being able to apply a testing function to a current value `x :: TestResult` the to-test value needs to be unwrapped from the existential type wrapper. We use `unsafeCorce` to extract the values hidden in a `TestResult`, and infer their type with help of the type inference system. By using a testing function `f :: Monad m =>a -> m (Maybe b, Spec)` directly on the coerced value, Haskell is able to infer, that the result of a call of `unsafeCoerce` should have the type `a`. The same unwrapping takes place when vertical dependencies are calculated. Here, the to-test value is coerced when passing it into the dependency functions. Mirroring this, if vertical dependencies are found, they are all wrapped into the `TestResult` existential type constructor, before they can be passed as the new list of to-test values in the recursion.

Both functions return a result dictionary with all values that were tested. This does not include information whether these results stem from to-test values or from their vertical dependencies. For this purpose the function that called this algorithm again filters out the results that we originally wanted. This process was described in section 4.2.

5 DISCUSSION AND OUTLOOK

During the development of this library the biggest problem that arose was that of typing. It was intended to extend the functionality to be able to use all horizontal dependencies as arguments for the testing function, not just a single list of to-test values. But, as we hide the types of the test results, when trying to retrieve them from the `TestState`, we cannot predict which types they will have. To do this, we rely on the testing function, and coerce the type inside `TestResult` at exactly the moment we call the testing function on it. As Haskell's type inference system knows which type the argument for the testing function has, it can infer which type to give to the value inside the `TestResult` wrapper. At the same time there was no obvious way to abstract the type for the testing function that is passed into `runTest` and `runDependentTest` enough, so the amount of arguments was variable. This means that it was not possible to declare a type for the testing function that was always correct. We have, however, in this thesis already found a way to circumvent such problems. When the `TestResults` list had to deal with different types, we used an existential type to hide them. It would therefore be necessary to also hide the type of the testing function to implement this feature in our current framework. The problem with this arises, when trying to apply the hidden function to hidden arguments. The type inference would need to infer both the function's and the arguments' type. As one relies on the knowledge of the other, this was not possible to implement in this way.

So, what could be done to circumvent this problem? It is not possible to store a type, and always associate a list of test result with its types. An idea was to put the test results in an existential wrapper and tuple them with their own coercion function. This, however, was also doomed to failure, as Haskell expects us to coerce the type inside the wrapper before putting it in the function. The functionality of being able to use values of the horizontal dependencies was not observed as necessary, however. If this necessity should arise, it can be circumvented by, for example, returning tuples as results in the testing functions.

There are more features that were not implemented. One of them was more configuration possibilities. Specifically, the ability to not only display the results of the to-test values that were an input, but all the tests that ran. That means that also all the values that were tested as vertical dependencies would be shown on the terminal for every test. Another feature that was not implemented but may be very interesting for the future would be adding instances to an already existing list of test results. Right now, it is only possible to introduce an entirely new list of to-test values, using `runTest`. All of these to-test values form their own test instances. With this feature, we may add an arbitrary amount of instances to an already existing list of test results. Ideally, this would mean, that we add results, that this list of to-test values already calculated in previous tests. This could only be achieved, if we add to-test values of the "root type", i.e. to-test values, that have the same type as the instances of this list had, when they were added in a call of `runTest`. Then, we would need to keep track of the test functions that were used on these instances. When adding new instances, they need to also run all of these tests we kept track of. Only that way all instances have test results in every previous test. Alternatively one could think about how to link instances more efficiently than by indices in a list. This could enable us to add instances at any point in the testing process, without the need to run all previous tests on them. This is because we would not have out of bounds indices when using horizontal dependencies.

Future developers of this library may then think about how to handle horizontal dependencies for values that did not exist when that dependency was evaluated. Maybe, this would also be a way to extend the configuration with multiple possibilities of how to handle these cases.

The library was integrated in the Duo language projects' test suite, and made it possible to prevent redundant testing. It also helped to better isolate errors, as failed dependencies were not propagated throughout the testing process anymore. In the future, developers of the Duo language, or of any other project that use this library, might benefit from being able to model interdependencies between tests.

REFERENCES

Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. 2020. A Survey of Compiler Testing. *ACM Comput. Surv.* 53, 1, Article 4 (feb 2020), 36 pages. https://doi.org/10.1145/3363562

HSpec. 2011. Writing Tests with HSpec. https://hspec.github.io/writing-specs.html [Online; accessed 14-May-2023].

Alexander S. Kossatchev and Mikhail Posypkin. 2005. Survey of compiler testing methods. *Program. Comput. Softw.* 31, 1 (2005), 10–19. http://dblp.uni-trier.de/db/journals/pcs/pcs31.html#KossatchevP05

SELBSTÄNDIGKEITSERKLÄRUNG

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbständig und nur mit den angegebenen Hilfsmitteln angefertigt habe und dass alle Stellen, die dem Wortlaut oder dem Sinne nach anderen Werken entnommen sind, durch Angaben von Quellen als Entlehnung kenntlich gemacht worden sind. Diese Bachelorarbeit wurde in gleicher oder ähnlicher Form in keinem anderen Studiengang als Prüfungsleistung vorgelegt.

Ort, Datum Unterschrift