CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

A Front-End for Daison

Development of an Interactive Haskell Environment Using the Glasgow Haskell Compiler as a Library

Bachelor's Thesis in Computer Science and Engineering

CHRISTOFFER KALTENBRUNNER
ALEXANDER NELDEFORS
HUGO STEGRELL
PHILIP VEDIN

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2021

Bachelor's Thesis 2021

A Front-End for Daison

Development of an Interactive Haskell Environment
Using the Glasgow Haskell Compiler as a Library

CHRISTOFFER KALTENBRUNNER
ALEXANDER NELDEFORS
HUGO STEGRELL
PHILIP VEDIN

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2021

A Front-End for Daison
Development of an Interactive Haskell Environment Using the Glasgow Haskell Compiler as a Library
CHRISTOFFER KALTENBRUNNER
ALEXANDER NELDEFORS
HUGO STEGRELL
PHILIP VEDIN

Supervisor: Krasimir Angelov, Department of Computer Science and Engineering
Examiner: Mary Sheeran, Department of Computer Science and Engineering

Bachelor's Thesis 2021
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2021

A Front-End for Daison
Development of an Interactive Haskell Environment Using the Glasgow Haskell Compiler as a Library

CHRISTOFFER KALTENBRUNNER
ALEXANDER NELDEFORS
HUGO STEGRELL
PHILIP VEDIN

Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Daison is a database written in the functional programming language Haskell. It has no built-in visual representation of data and using it in an interactive Haskell environment such as GHCi is inconvenient since it requires boilerplate code. Our work presents an application that provides an alternative interactive environment that both removes the boilerplate code the user currently needs to write and displays query results in a table format. The application aims to be user-friendly and provide an experience similar to GHCi. It is written in Haskell and uses the Glasgow Haskell Compiler as a library. Automated tests are in place for ease of maintainability, though in a limited capacity. Manual testing was used to verify that requirements were fulfilled. Findings from usability testing, carried out in two iterations with a total of nine participants, suggest the application was well received. While fully supporting functionality provided by Daison, there remain a range of improvements that can further enhance the user experience. Examples of such improvements include better table formatting and the addition of a graphical user interface.

Keywords: Daison, Database, Glasgow Haskell Compiler, Haskell

Sammandrag

Daison är en databas skriven i det funktionella programmeringsspråket Haskell. Den har ingen inbyggd visuell representation av data och att använda den i en interaktiv Haskell-miljö som GHCi är opraktiskt eftersom det kräver mycket *boilerplate*-kod. Vårt arbete introducerar en applikation som erbjuder en alternativ interaktiv miljö som både tar bort *boilerplate*-kod som användaren behöver skriva samt visar resultatet från databasfrågorna i en formaterad tabell. Applikationen avser att vara användarvänlig och erbjuda en liknande upplevelse som GHCi. Den är skriven i Haskell och använder Glasgow Haskell Compiler som ett bibliotek. Automatiska tester finns för att underlätta underhållningen av mjukvaran, dock i begränsad omfattning. Manuella tester genomfördes för att verifiera att alla kraven var uppfyllda. Användarvänlighetstester, vilka genomfördes i två omgångar med totalt nio deltagare, antydde att applikationen är lättanvänd. Applikationen har fullt stöd för Daison, men det finns ytterligare förbättringar för att optimera användarupplevelsen. Exempel på sådana förbättringar inkluderar bättre formaterade tabeller samt ett grafiskt användargränssnitt.

Nyckelord: Daison, Databas, Glasgow Haskell Compiler, Haskell

Acknowledgements

We are grateful to our supervisor, Krasimir Angelov, who has given us feedback and helped us in all stages of the project, from the planning all the way to the final presentation. We also thank Mary Sheeran, our examiner. She has provided good answers when asked questions. We are thankful for all the volunteers who participated in the usability tests.

The Authors, Gothenburg, May 2021

Contents

1

Introduction

Very few modern applications are created without a database back-end. Although using a ready-made database instead of a custom-made storage gives a lot for free, it comes at a cost. Many database concepts are fundamentally different from those used in general-purpose programming languages, such as C, Java and Haskell, resulting in an impedance mismatch between the application and database layers. The impedance mismatch in turn leads to programs that are hard to maintain. Additionally, many databases come with their own programming language requiring the developer to use multiple languages for the application.

A common solution is to use an embedded language as a high-level interface in which programmers can describe the database queries. Thus, the queries can be written at a higher level of abstraction, are statically safe and can be type-checked at compile time. This approach is supported by many programming languages, including Language Integrated Query (LINQ) for C# [1], ScalaQL for Scala [2], and Selda for Haskell [3].

One can also bypass the impedance mismatch by storing data as native data types of the application language. For Haskell programmers, this can be done by using *Daison* (a word play on DAta lISt comprehensiON) developed by Angelov [4], the database which is the topic for this project. Daison is a NoSQL database which replaces the traditional Structured Query Language (SQL) with a Haskell Application Programming Interface (API). It stores ordinary Haskell values and uses Haskell data types for table definitions.

However, there is no front-end available by Daison's original developer. This means that in order to modify a database, the programmer has to either write Haskell code and compile it, or modify the database directly in the interactive Haskell environment GHCi (Glasgow Haskell Compiler interactive). The former is not very convenient if the user wants to load test data. The latter is not user-friendly either, since the user must explicitly start a transaction every time and pass a reference to the database for each query. Furthermore, the output is not printed in a readable way.

1.1 Purpose

The purpose of the project is to build a user-friendly and well-documented front-end for Daison by using the Haskell Glasgow Compiler (GHC) as a library. The front-end should be a command line program that interprets and executes Daison functions in the same interactive way that GHCi interprets and executes Haskell code. The target group for the application is programmers with experience in Haskell.

Table 1.1 shows the purpose broken down into a requirement specification for the project. Requirements one through eight were specified in the beginning of the project and requirements nine through twelve were added during the project.

Table 1.1: Requirements for the application.

ID	Description
R1	The Daison library should be loaded by default.
R2	A transaction should be started automatically for every database query.
R3	It should be possible to write queries interactively at the prompt.
R4	It should be possible to write Haskell code interactively at the prompt.
R5	The front-end should accept the Daison monad.
R6	The output from queries should be printed as a formatted table.
R7	It should be possible to navigate the printed output (up/down).
R8	The front-end should accept the IO monad.
R9	It should be possible to navigate the file system at the prompt.
R10	It should be possible to load Haskell modules at runtime and startup.
R11	It should be possible to set flags at runtime and startup.
R12	It should be possible for the user to open and close databases at runtime. Databases should also be possible to set at startup.

1.2 Delimitations

As stated above, this project aims to build a front-end which interactively interprets and executes Daison queries in a GHCi-like manner by using GHC as a library. We will not expand upon the set of queries that are currently supported by Daison nor will we implement an embedded language for database interaction as this is beyond the scope of the project.

2

Theory

Haskell is a general-purpose, statically typed, purely functional programming language developed to be suitable for teaching, research and industrial applications. It is specified in the Haskell Language Report [5] and the most popular and widely used implementation of Haskell is the Glasgow Haskell Compiler (GHC).

In this chapter, we begin by giving a short introduction to Haskell, presenting some core concepts as well as more advanced concepts. The experienced Haskell programmer might want to skip this section. Next, we introduce Daison, the database which is the topic of this project. This is followed by an overview of GHC and its Application Programming Interface (API).

2.1 General Knowledge of Haskell

The Haskell language has many features, for example *pure functions*, *lazy evaluation*, *pattern matching*, *list comprehensions*, *type classes* and *type polymorphism*. In this section we will explain the language features that are particularly important for this project.

2.1.1 Functions and Types

A Haskell program consists of *pure functions*; that is, functions which do not have any side-effects. A function in Haskell will always return the same value given the same input, whenever it is being evaluated in the program. One of the benefits with this approach is that functions are easy to test for correctness. For those not familiar with functional programming, a pure function can be compared to a mathematical function. For instance, consider a function, $f : \mathbb{Z} \to \mathbb{Z}$, which increments any integer by one,

$$f(n) = n + 1.$$

For any $n \in \mathbb{Z}$ the function $f(n)$ will always return the same value. In Haskell, this function could be written as:

```
inc :: Int -> Int
inc n = n + 1
```

where the first line is a *type declaration* and the second line is the *implementation* of the function. Looking at the type declaration, one can see that `inc` is a function that takes something of type `Int` as an argument and evaluates to something of the same type.

Because of Haskell's strong static type system, trivial errors like `"Hello, world" + 12` is found at compile-time. Haskell has predefined types for truth values (`Bool`), integers (`Int` and `Integer`), floating-point numbers (`Float` and `Double`), characters (`Char`), and strings (`String`). But, we can also make our own types. For instance, we could represent a person which has a name of type `String` and an age of type `Int` as `data Person = Person String Int`. However, if we would like to get a person's name or age, we would need to define functions that return them. Since this is a common use case, we can simply use *record syntax* instead:

```
data Person = Person {name :: String, age :: Int}
```

By using record syntax, Haskell automatically creates the functions `name :: Person -> String` and `age :: Person -> Int`. Thus, if we now define a person `p = Person "Alice" 25`, we could get her name and age:

```
nameOfPerson = name p  -- "Alice"
ageOfPerson  = age p   -- 25
```

2.1.2 List Comprehensions

Lists in Haskell are similar to sets in mathematics, except that lists can contain duplicate values and can only contain one type. They can be written as a comprehension of the form `[ e | q1, ..., qn ]`, $n \geq 1$, where each q is either a generator, a local declaration or a boolean condition [5, Ch. 5]. For instance, the set of even positive integers less than or equal to 100 could in Haskell be expressed as `[ 2*x | x <- [1..50] ]` where `[1..50]` is the list of integers from one to fifty.

Consider another example: We would like to calculate all positive integers less than or equal to 250 that are divisible by three and seventeen. With a list comprehension we could express this set concisely:

```
[x | x <- [1..250], x `mod` 3 == 0, x `mod` 17 == 0]
```

2.1.3 Type Classes

Let us begin with a simple example of a type class. Below is the definition of the `Num` class from the *base* package [5, Ch. 6]:

```
class Num a where
    (+), (*), (-) :: a -> a -> a
    negate, abs, signum :: a -> a
    fromInteger :: Integer -> a
```

The class defines the type signatures for a set of functions. It is possible to define default implementations, but often there are only type signatures in a class. This feature is similar to *interfaces* in Java or *traits* in Scala, although there are small differences between the languages.

A class can be derived in Haskell by specifying that a type is an *instance* of that class. Since there are no implementations in the `Num` class, any type that derives `Num` must implement these functions, which are often called the *minimal complete definition*.

Aside from the minimal complete definition, type classes can also have laws that the implementations need to follow. An example of a type class that has laws is the `Num` type class which was just presented. As specified in the Haskell Language Report [5, Ch. 6.4], any number must have a magnitude, a sign and should follow the law that the absolute value multiplied by the sign is equal to the original number. The report even gives the implementation for real numbers:

```
abs x    | x >= 0  = x
         | x <  0  = -x

signum x | x >  0  = 1
         | x == 0  = 0
         | x <  0  = -1
```

These functions use pattern matching with Haskell guards. The guards are indicated by pipes and decide how the function should evaluate. If a guard is evaluated to true, the body to the right will be used. Otherwise it moves on to the next guard until a match has been found.

Classes are sometimes expected to have certain properties, which are often described in mathematical terms. The documentation of the `Num` class states properties such as associativity, commutativity and distributivity that the functions should follow.

The benefit of type classes is to guarantee the compiler and the programmer that a set of functions are defined for a data type. They can then safely be used for ad-hoc polymorphism, which is similar to overloading in object-oriented languages.

Two standard Haskell types, `Integer` and `Double` are instances of `Num` [5, Ch. 6]. Without type classes every type that is an instance of `Num` would need a separate function for $(+)$, $(*)$, $(-)$, etc. It could be written like this:

```
addDouble :: Double -> Double -> Double
mulDouble :: Double -> Double -> Double
subDouble :: Double -> Double -> Double
-- etc.

addInteger :: Integer -> Integer -> Integer
mulInteger :: Integer -> Integer -> Integer
subInteger :: Integer -> Integer -> Integer
-- etc.
```

But with type classes they only need to implement the functions

```
(+) :: Num a => a -> a -> a
(*) :: Num a => a -> a -> a
(-) :: Num a => a -> a -> a
-- etc.
```

which is much simpler to write and easier to maintain.

2.1.4 Monads

Haskell is a purely functional language and pure functions can not have side-effects. But without side-effects, a program is almost useless. Common side-effects include input and output actions such as reading and printing. Haskell's solutions to safely provide effect-based computations are *monads* and *monadic actions*.

Monads are data types which are instances of the *Monad* type class. The type class has the following explicit definitions [5, Ch. 13]:

```
class Monad m where
  (>>=)  :: m a -> (a -> m b) -> m b
  (>>)   :: m a -> m b        -> m b
  return :: a                 -> m a
```

The `(>>=)` function, often referred to as *bind*, takes two parameters. They are composed and the value produced by the first is passed to the second, which in turn evaluates to a new monadic computation (`m b`). The `(>>)` function also composes two actions, but discards the value produced by the first action. Lastly, the `return` function takes a pure value and wraps it inside the monad.

Every Haskell program is run within the `IO` monad so as to allow input and output operations during the program's execution [5, Ch. 5]. To ease the implementation of other monads that permit the execution of `IO` actions, there is a type class called `MonadIO`. This type class requires an implementation of `liftIO :: IO a -> m a` which allows any `IO` action to be performed in any monad that is an instance of `MonadIO` [6].

A monad highly important for this project is the `Daison` monad defined in the

Daison library. It uses `IO` internally but is not an instance of `MonadIO` in order to keep the database operations pure. Furthermore, it maintains a database state and is what the Daison back-end is largely based on to perform database computations [4].

Haskell has a syntactic sugar called *do-notation*. The notation allows monadic actions to be written in a style which is similar to programs in imperative languages, with each row containing a single statement. An example of chaining multiple monadic actions and the difference between using *do-notation* or not is the following:

```
-- Without do-notation
f >>= (\result -> g result)

-- With do-notation
do
  result <- f
  g result
```

As seen in the example, `<-` is used to inspect the result of a monadic computation, allowing the unwrapped value to be used for the next computation. A way to visualise it is by seeing `f` as a box, which the value is taken out of, and then put in a new box, `g`. It can be translated to the bind syntax `(>>=)` and the effect will be the same.

The `State` monad is used to replicate a mutable state such as objects in object oriented languages [7]. Even though Haskell does not keep track of states like an imperative language does, it is possible to get similar behaviour while keeping the code functionally pure. The first intuition might be to pass the information along each function, so that the current state is always accessible and can be altered in each function. This however creates a large overhead. With the use of a state monad the process becomes simpler and more clear.

The `MonadState` class in *Control.Monad.State* is defined in the following way [8]:

```
class Monad m => MonadState s m | m -> s where
    get :: m s
    put :: s -> m ()
    state :: (s -> (a, s)) -> m a
```

The type class provides a useful function called `modify`. This function allows for modification of the state by giving it a function that specifies how the state should be modified.

One data type that is an instance of the `MonadState` type class is the type `State`. Because Haskell is immutable, the most intuitive way to implement stateful computations would be a function that takes a state and then returns a tuple with the result of the computation and the modified state (`state -> (result, modifiedState)`).

With such an implementation the state returned from the function can be used as an argument to the next function, replicating stateful computation. This is in fact how the `State` type is implemented [8].

In theory, the `State` data type does not need to be a monad to replicate the actions of stateful computations as specified in the `MonadState` type class. Being a monad however gives access to the *do-notation* which we saw before and removes much boilerplate code.

2.2 An Introduction to Daison

Daison (a word play on DAta lISt comprehensiON) is a NoSQL database for storing ordinary Haskell values [4]. It uses a stripped version of SQLite to store data [9], and the Structured Query Language (SQL) is replaced by a Haskell Application Programming Interface (API) which allows database queries to be written in Haskell. Daison can be installed as a Haskell package and then be used in any Haskell program.

Since not many readers are assumed to have experience with Daison we will present some SQL queries and how they can be expressed in Daison. As in most database languages, Daison requires queries to run within a transaction. They can be initialised with `runDaison :: Database -> AccessMode -> Daison a -> IO a`, and if there were no exceptions the transaction will be commited, otherwise the transaction will be rolled back. With this format a transaction could be written as:

```
do
    -- Open the database
    db <- openDB "myDatabase.db"

    -- Create a transaction and execute the database queries
    runDaison db ReadWriteMode (do
        -- Database queries are put inside this do-block
    )
    -- Close the database
    closeDB db
```

Following this we will only present the actual queries which are supposed to replace the comment in the inner do-block, so as to not repeat the transaction setup for each query.

Table 2.1 provides an overview of the most important Daison functions. Some of Daison's functionality will be explained in the following sections but if the reader wishes to learn more about Daison there is a tutorial in its GitHub repository [10] which goes into more detail.

Table 2.1: Overview of Daison functions.

Function	Description
`createTable`	Add a table to the database.
`tryCreateTable`	Safe version of `createTable`.
`dropTable`	Remove a table from the database.
`tryDropTable`	Safe version of `dropTable`.
`insert`	Add entries to a database table.
`select`	Extracts data from a database given a `Query` argument.
`from`	Can be used to create `Query` values.
`update`	Change an existing table entry.
`store`	Can act as either `insert` or `update`.
`delete`	Remove entries from a database table.

2.2.1 Creating Tables

In SQL a simple table for storing people can be created with the following code:

```
CREATE TABLE People (
    id      INT PRIMARY KEY,
    name    TEXT,
    age     INT
);
```

In Daison a table is created by using `createTable :: Table a -> Daison ()` or `tryCreateTable :: Table a -> Daison ()`. The former version will fail if a table already exists with the name that is passed to the function while the latter version can be used to safely create a table only if it does not already exist in the database. As we see, both `createTable` and `tryCreateTable` take a parameter of type *Table a* which in turn could be produced by the function `table :: String -> Table a`. In addition, the data type for the rows has to be defined, all in all resulting in the following:

```
-- Data type for rows
data Person = Person
              { name :: String
              , age  :: Int
              } deriving (Data, Show)

-- Table definition
people :: Table Person
people = table "people"
```

We can now create a table by running `createTable people` in a transaction.

2.2.2 Inserting into a Table

Consider the following scenario: We would like to insert a 25 year old person named Alice into a table. A SQL insertion could be written as:

```
INSERT INTO people VALUES (1,'Alice',25);
```

In Daison, the same insertion could be written:

```
insert people (return (Person "Alice" 25))
```

2.2.3 Selecting from a Table

A simple selection in SQL is the following:

```
SELECT name FROM people WHERE id=1;
```

In Daison the query can be expressed as:

```
select [name p | p <- from people (at 1)]
```

The keyword `at` is used to specify the primary key of the row that we want to search for. Since we defined the `Person` data type in record syntax, names can easily be accessed by the function `name :: Person -> String`.

A common select statement often returns all columns with some restriction, as in the following example:

```
SELECT * FROM people WHERE name='Alice';
```

Select statements in Daison can use ordinary guards as well as zero or more of the `(^<)`, `(^<=)`, `(^>)`, and `(^>=)` operators, which adjusts the `everything` restriction. To write a query that returns all columns, with the restriction that the name is Alice, could in Daison be written as:

```
select [person | (pkey,person) <- from people everything, name
↪  person == "Alice"]
```

As seen in the query, *from* combined with *everything* returns both the primary key and the entry, which is useful if the user does not already know the key.

2.2.4 Updating an Entry

Updating an already existing entry is done with the `update` statement. Setting the age to 26 for the person at index one can in SQL be written in the following way:

```
UPDATE people SET age=26 WHERE id=1;
```

Updates in Daison use key-value pairs, therefore the list comprehension needs to return the primary keys of the rows that should be updated. The SQL query above

is then written in the following manner in Daison:

```
update people [(1,person{age=26}) | person <- from people (at 1)]
```

2.2.5 Deleting from a Table

In SQL, deleting the row with index one from the table `people` is written as:

```
DELETE FROM people WHERE id=1;
```

Deletions in Daison take a query argument which returns the keys to the rows which are to be deleted. Thus, the SQL query written above could in Daison be expressed as:

```
delete people (return 1)
```

2.2.6 Indices in a Table

Table indices can be specified in Daison to allow queries directly on something other than the primary key. The `withIndex` primitive produces a table containing the index.

In Daison, there are three functions that produce indices:

- `index` which takes a table, a string and an arbitrary function which decides how the value of the rows should be indexed,
- `listIndex` which takes the same arguments except that the function uses every value in the list to produce the indices, and
- `maybeIndex` which can be used to produce `Nothing` and skip those rows, similar to nullable columns in SQL.

An easier way to create a select statement can then be to use an `Index` and some operators. A query which returns all people who has reached the age of majority in Sweden could then be expressed in the following way:

```
peopleAge :: Index Person Int
peopleAge = index people "age" age

people :: Table Person -- Table definition
people = table "people"
           `withIndex` peopleAge

select [row | (id,row,age) <- fromIndex peopleAge (everything ^>=
↪  18)]
```

2.3 The Glasgow Haskell Compiler

The Glasgow Haskell Compiler (GHC) is open-source software that can be used as both a standalone tool and as a library for compiling Haskell source code. It supports the entire Haskell 2010 language, has good support for concurrency and parallelism, and the compiled code is generally fast [11].

The first step GHC performs when compiling Haskell code is parsing the source code [12]. The Parser produces abstract syntax which is passed to the Renamer. The main job of the Renamer is to resolve all of the identifiers in the Haskell source code into names, that is, a reference to a particular entity. The next step is type-checking, the process of verifying that the Haskell program is type-correct. All of these three parts of the compiler pipeline can discover and report programmer errors. Once the program is type-checked, it gets passed to the Desugarer which removes all syntactic sugar and translates the Haskell syntax into a smaller language called *Core*. It is then passed to the Simplifier which applies a number of different local optimizations to the program. At this stage, the optimized Core program is passed to the back-end which converts it to C--. Finally, depending on what the target is, GHC will generate pretty-printed C code, machine code, or Low Level Virtual Machine (LLVM) code.

One of the advantages of the GHC architecture is that it is built as a library rather than a monolithic program. On top of this, the GHC team built an API that exposes GHC's functionality. The API has been used for building GHCi, a command-line application which makes it possible to interactively evaluate Haskell expressions and interpret Haskell programs.

GHC functionality provided by the API can be run from any monad with a `GhcMonad` instance. To further ease access for programmers, the API exposes a minimal implementation in a monad called `Ghc` [13]. A subset of the functions exposed by the API can be seen in Table 2.2, together with short descriptions. These have been selected as they are highly relevant for this project.

Table 2.2: A subset of the functions exposed by the GHC API [13].

Function	**Description**
`execStmt`	Run a statement in the current interactive context.
`exprType`	Get the type of an expression.
`runGhc`	Initialises a new GHC session each time it is called.
`reflectGhc`	Takes an action done in the `Ghc` monad and returns an `IO` action instead.
`getContext`	Get the modules in scope in the current session.
`setContext`	Set the current context of the session.
`getSessionDynFlags`	Get the current flags in use.
`setSessionDynFlags`	Set the current flags in use.

The `Ghc` monad manages the state of the current session through the `Session` data type. `Session` maintains the current compiler flags and loaded modules. It also provides functionality to compile and load modules directly from files containing Haskell source code.

3

Methods

This chapter provides an overview of the processes used during the project, including the general approach for the code development and verification that the requirements have been fulfilled.

3.1 Software Development

The first step was to study how the Glasgow Haskell Compiler (GHC) could be used as a library. Therefore, we studied the documentation and tutorials on the official Haskell website. However, since GHC is always evolving, new updates bring code restructuring as well as additions and deletions of code. Unfortunately, the tutorials on the official Haskell website has not been updated together with the updates of GHC. The GHC API, however, is documented and updated regularly, but it is not sufficient for newcomers trying to work with GHC for the first time. Therefore, we decided to study the source code of GHCi, trying to understand how the GHC team uses GHC as a library and then do something similar on our own. Additionally, we studied the library *hint* [14], which is a simple wrapper around a huge subset of the GHC API which helped us understand how GHC could be used as a library.

Once it was understood how to use GHC as a library, we began to implement the requirements. First, we implemented the requirements for loading the Daison library by default and making it possible to write Haskell code at the prompt. Second, we implemented the requirement for displaying the output from queries in a formatted table. Finally, we implemented the requirements for navigating the file system, loading Haskell modules, and setting GHC flags.

3.1.1 Version Control and Continuous Integration

Version control was handled through Git, and the code was hosted in a GitHub repository [15]. The code was maintained according to *Gitflow* [16]; in short, this means new features were developed in their own temporary branches before being merged into a *develop* branch, while the *master* branch only contained fully tested and functional code. Pull requests to *master* were reviewed and approved by at least one other project member before being accepted and merged. Bugs or other issues were reported by creating *GitHub Issues*, which were later discussed during

meetings if the problem or solution was unclear.

During the project we used *Continuous Integration*, a software development method where developers – as the name suggests – integrate their code continuously to the main code repository. Since we pushed code as often as possible to the main repository, we avoided costly code merges by addressing merge conflicts early. Additionally, with *GitHub Actions* we enabled automated testing. The workflow we set up automatically built the project and ran the test suites which contained all automated tests (see Section 3.2.1 and Section 3.2.2). As a result, if one of the group members pushed code which meant the project could not be built or the automated tests failed, GitHub would alert us that the workflow had failed.

3.2 Testing and Verification

A number of different testing methods were used to evaluate the software. *Unit tests* were used to evaluate parts of the code, *functional tests* were used to evaluate the functionality of the application with respect to the requirements specification given in Table 1.1, and *usability tests* were used to evaluate the overall usability of the application.

3.2.1 Unit Tests

Unit tests were used to test specific parts of the code, to ensure that the functions work exactly as expected. We used a package called HUnit, which is inspired by the JUnit framework for Java [17]. The package enabled us to easily create test cases in the form of assertions such as `assertEqual "Label describing test" <expected_value> <function_call>`. It should be noted that unit tests do not cover much of the code base. The problems which were encountered while creating them are discussed in Section 5.2.1.

3.2.2 Functional Tests

We created at least one functional test for each requirement, depending on the scope of the requirement. A functional test is a type of black-box test which tests a software component based on its specification [18]. In this case, we tested a function of the application (not to be confused with functions in Haskell) based on its requirement specification. Each test case specification included a description of what the test was supposed to validate, the steps the tester should follow and the expected result (see Appendix A). The functional tests were executed manually, but most of them could be automated.

The big advantage of using this testing approach is that the software were tested at a high level with respect to the software specification given by the requirements. As seen in Table 3.1, there was a full requirement coverage for the application, and therefore, we could conclude that all requirements were fulfilled if all functional tests were executed successfully. Although this approach does not necessarily lead

to error-free code, any remaining errors should be of minor character.

Table 3.1: Overview of the functional tests for validating the correctness of the final version of the application. Each test case is mapped to one or more requirements, except for F7 which is independent.

Test Case ID	Requirement ID	Purpose
F1	R1	Verify that the Daison library is loaded at startup.
F2	R2, R3 and R5	Verify that it is possible to write queries at the prompt and that a transaction is started automatically.
F3	R4	Verify that it is possible to interactively write Haskell code at the prompt.
F4	R9	Verify that it is possible to navigate the file system at the prompt.
F5	R10	Verify that it is possible to load Haskell modules at runtime.
F6	R11	Verify that it is possible to set flags at runtime.
F7	-	Verify that an unknown command shows an error message.
F8	R6 and R7	Verify that non-small query results are displayed to the user as a formatted, navigable table.
F9	R11	Verify that it is possible to set flags at startup.
F10	R10	Verify that it is possible to load Haskell files at runtime.
F11	R8	Verify that IO expressions are accepted at the prompt.
F12	R12	Verify that databases can be opened and closed.
F13	R10	Verify that it is possible to load Haskell files at startup.

In addition to the functional tests in Table 3.1, we made automated tests for validating requirements one through five using the package QuickCheck [19]. The package enables automated tests with randomly generated data. Data of the types `Person` (see Section 2.2.1), `[Double]`, `(Bool, Int)` and `Char` were generated and then written to and read from a database using both the front-end and the back-end.

3.2.3 Usability Tests

Any front-end application should reasonably have usability testing for improving the user experience (UX). Nielsen and Landauer have estimated that for a medium-sized software project the maximum benefit/cost ratio for usability evaluation is at four evaluations [20], after which the marginal value that another evaluation would provide is smaller than the cost. Therefore, we decided to use an iterative design process where we performed two usability studies with four and five participants respectively. Before the second usability study, we fixed the issues that arose in the first one. By using this approach we could quickly find the biggest problems in the design of the application, implement a solution, and verify that our solution solved the problem.

Since the target group for our application is programmers with experience in Haskell, the participants should have taken a course in functional programming at university level or equivalent. However, since we used a small sample, the results of the usability studies should not be seen as absolute truths but rather as guidelines for what works in the design and what does not.

The usability studies were designed to have as efficient tests as possible, instead of a large amount of tests. The studies were conducted remotely by using video-conferencing software and the spoken language was Swedish since this would allow the participants to express themselves more naturally. Each session was led by a moderator who gave the participant a number of tasks to complete while a note-taker recorded how the participant interacted with the application. The participant was encouraged to think out loud during the completion of the tasks (the so-called Think-aloud Protocol [21]). By explicitly stating their thoughts we received a better understanding of how the participant experienced the design of the application. To ensure that all tests were performed as similarly as possible, the moderator followed a script.

The purpose of the studies was to validate the product's overall usability. Before carrying out the usability tests, we made the following hypothesis: *the application is easy to use, but there need to be improvements to the user feedback messages.* Additionally, we formulated the following questions that the study should provide answers to:

1. Is the application easy to use?
2. Is the visual feedback provided by the application enough for the user to understand the program?
3. Does the consequence of any interaction with the application match the user's expectations?

The participants were asked to perform the following tasks: loading external modules, changing the working directory, creating a database, changing the active database, creating, reading, updating and deleting entries, as well as navigating the

output (see Appendix B). To make sure that we tested the UX of the actual front-end rather than the UX of the Daison back-end, we provided the participants with a predefined definition for the database table as well as a list with a subset of the available Daison functions that could be used for solving the tasks (see Appendix C). For every task performed, the participants were asked to rate the difficulty of the task on a scale from one to five (where one is equivalent to *very easy* and five is equivalent to *very difficult*) and provide a motivation. Finally, we asked the participants to rate the overall usability of the product. It should be noted that the participants did not study the Daison library nor the application beforehand.

4

Results

This chapter presents the results of the project. We begin by going into detail of how the application works as well as discussing the software architecture. Next, we address requirement changes. Finally, we present the test results.

4.1 An Overview of the Application

As shown in Figure 4.1, once the program launches, it initialises a GHC session with `runGhc` within which the Daison library is loaded along with a few other useful libraries such as *Prelude* and *Data.Data*, where the latter is included since Daison requires custom data types to have instances for the `Data` type class. The first steps are setting flags and extensions that are needed for the Daison library. To fix a bug that occurs for Unix users when the program is run from GHCi rather than from an executable file, the signal handler for the *UserInterrupt* signal also needs to be overridden in order to prevent the program from getting stuck in a seemingly half-active state after CTRL+C is pressed after the first iteration of the main loop; that is, the prompt alternates between GHCi's and Daison's as the user presses the Enter key.

After initialisation is completed, the program prints a welcome message which shows the version of the program and that there is a help command. Then the program starts the loop and waits for the users input. Once the user has sent some input, it is first checked against the defined commands listed in Table 4.1 and if the input matches any of them the program executes the corresponding commands. If there is no match the program attempts to run the input as an expression, which could either be a database query or a normal Haskell expression. As in the usual GHCi environment, errors are printed if there are any. Finally, upon exiting the program, all the open connections to databases will be closed.

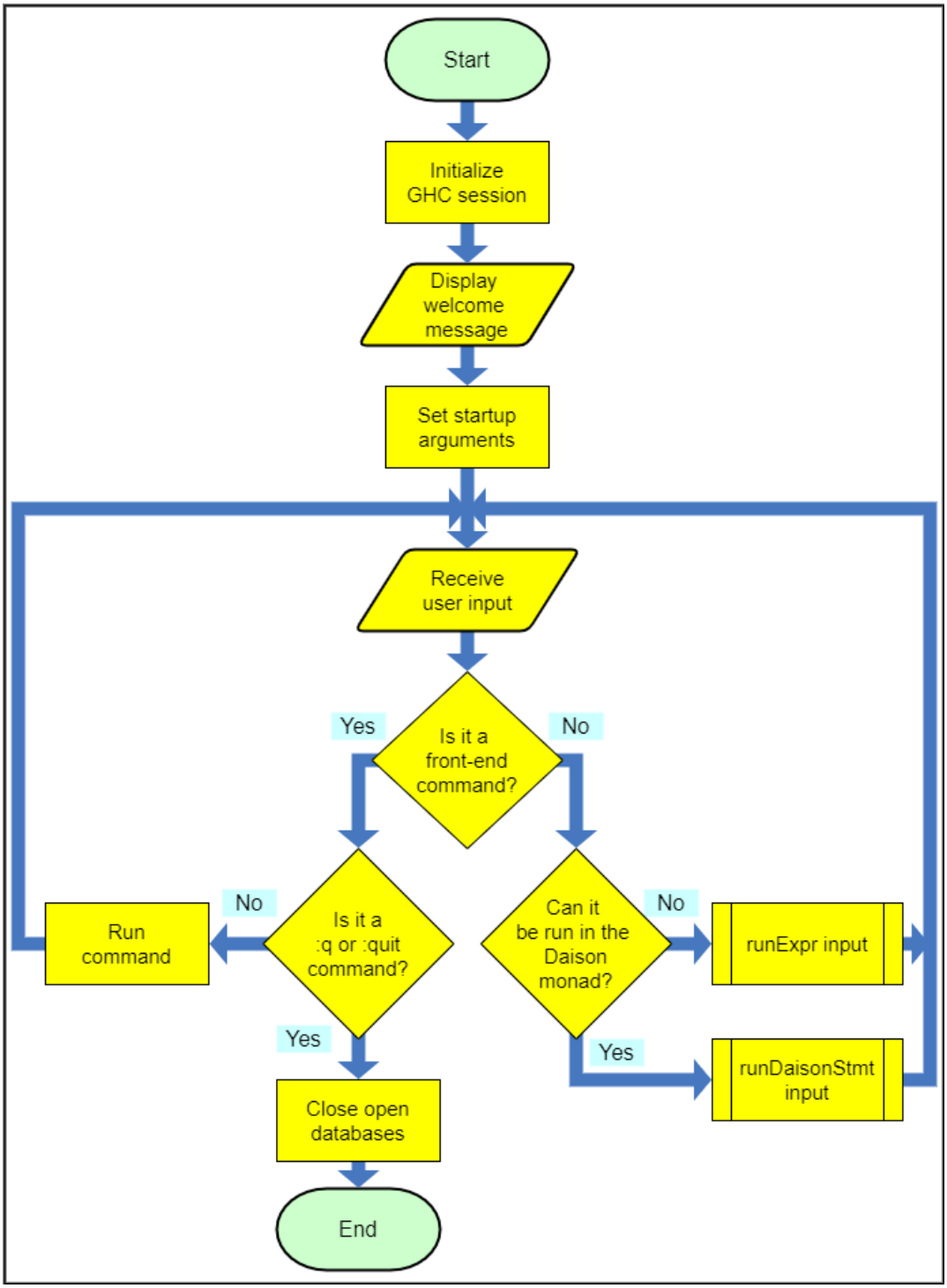

Figure 4.1: An overview of the read-eval-print loop of the program.

Table 4.1: Custom commands defined in the front-end.

Command	Description
`:help, :?`	Prints a help message containing the list of commands and their descriptions.
`:log path`	Display the log file's path.
`:log show`	Display the log file's contents.
`:log toggle`	Enable/disable logging.
`:log wipe`	Attempt to wipe the log file's contents.
`:type <expr>`	Print the type of the expression.
`:! <cmd>`	Run the command in the user's shell environment.
`:quit`	Exit the program.
`:dbs`	Print the list of databases that are open.
`:open <name>`	Open a database with the specified name. Creates a new database with said name if it does not already exist.
`:close <name>`	Close the connection to a database with the specified name.
`:mode [mode]`	Set access mode (`ReadWrite` or `ReadOnly`). Displays the current access mode if no argument is given.
`:cd <dir>`	Change current directory.
`:module <module>`	Import a module to the session.
`:load <filepath>`	Load modules from a Haskell file on the file system.
`:set <option>`	Set a GHC option for the session.

4.2 Handling Databases

There is support for using multiple databases during a session of the program. The user can switch the database against which queries will be run at any time using the `:open <name>` command. Closing a connection can be done by using the `:close <name>` command. A list of the current open databases can be seen by calling the command `:dbs`. The application also parses any arguments ending with ".db" as databases and runs the `:open` command on those during startup, setting the last given database argument as the active one.

In order to allow for multiple databases to be opened simultaneously, these need to be kept track of both within and outside of the GHC session. Within the GHC session, this is done by storing the results from Daison's `openDB` function in a list of tuples of type `(String, Database)`; outside, this is done by storing the names of the opened databases in the loop's state (see Section 4.6.1). The currently active database also needs to be stored on both sides, with its name stored in the loop's state and the corresponding `Database` within the session. To avoid collision with user-defined variables, the within-session variables are prepended with underscores, for example `_openDBs`. When a user opens or closes a database using a front-end command, statements that interact with the variables in the GHC session are generated as strings and run as if they were user input.

4.3 Running Queries

Executing queries can be done directly in the program. There is no need to load pre-written Haskell files to manage the session and execute queries, requiring reloads for each new query. This is all handled within the application.

4.3.1 Parsing Queries

The first step of the program is to parse the query. We used the same method as GHC, where we accept most input and filter it depending on errors further down the line. At this stage, we check if the input is a command defined by the application or a command to be executed.

4.3.2 Type-Checking Queries

When the user enters an input which is considered an expression the application checks whether it is a Daison query or not. In the first case, it runs it in a database transaction and in the second case it runs the expression as either a statement or a declaration. While most Haskell expressions are statements, expressions starting with terms such as `data`, `type` and `instance`, as well as expressions of the form `a = b` that do not start with `let`, are referred to as declarations [5, Ch. 4] and need to be executed using a different GHC API function than is used to execute statements. An expression is considered to be a Daison query if it either belongs to the Daison monad or if it can be run in any monad `m`, disregarding type constraints.

4.3.3 Executing Queries

Once the program has checked that the query is indeed in the Daison monad, the query can be executed. This is done by the `Eval.runStmt` function which makes required initiations and calls `GHC.execStmt`.

In order to execute the query, it needs to be given as an argument to the function `runDaison` (see Section 2.2). Furthermore, the query results need to be accessible to the user; this is achieved by binding them to the variable `it`, which the GHC API uses to store output from the most recent statement evaluation that was not a variable assignment. If the query `dropTable testScores` is entered, the code executed by `runStmt` is `it <- runDaison _activeDB accessMode (dropTable testScores)`, where `_activeDB` is a variable stored within the GHC session (see Section 4.2) and `accessMode` is a variable denoting if write access is desired, whose value was retrieved from the current state (see Table 4.2).

4.4 Displaying Query Results

Results from queries are shown as two-dimensional tables when appropriate. The formatted results are sent to the Unix program *less* in order to provide features such as arrow key navigation, search functionality and the ability to save the output as a text file. Figure 4.2 shows the output of a query that returns every value from a small table.

```
select [x | (_key, x) <- from testScores everything]
it :: [(String, String, Float, Bool)]

it !! | String                                | String  | Float | Bool
------+---------------------------------------+---------+-------+------
0     | "Introduction to Functional Programming" | "Alice" | 82.5  | True
1     | "Linear Algebra"                      | "Bob"   | 65.0  | True
2     | "Introduction to Functional Programming" | "Carl"  | 47.3  | False
3     | "Linear Algebra"                      | "Denise" | 0.0   | False
4     | "Introduction to Functional Programming" | "Eve"   | 100.0 | True
(END)
```

Figure 4.2: Output from a `select` query, viewed through the Unix program *less*. The list comprehension collects all key-value pairs in the table *testScores* and returns the values, which are 4-tuples.

4.4.1 Formatting a Table

The result, typically a list of tuples, is extracted from the GHC session as a string. This string is then split into rows and columns based on the presence of commas and their surrounding context. Context in this case refers to the purpose of a comma and how deeply nested the expression containing it is; in other words, whether the comma is part of a string or used to separate values in a list or tuple, as well as how many round and square brackets it is enclosed by. At present, the number of columns in the resulting table varies only if the stored value is a tuple, in which case the number grows linearly to the tuple size.

If the result is found to only contain a single row, then no formatting is applied. Otherwise, a header row is added that displays the types of the values in each column. Additionally, an extra column denoting the index that can be used to access the row using the `it` variable is added to the start of every row: entering `it !! 2` in the prompt after exiting from *less* in Figure 4.2 returns the 4-tuple containing the string `"Carl"`. The lengths of each table cell are then checked and additional spaces appended such that every cell in a column occupies the same amount of space. Lastly, horizontal and vertical separator lines are inserted before joining the resulting rows and columns together using the *Pretty* library included in the GHC API [22].

4.4.2 Choosing Display Method

The application uses two different methods for displaying output: by directly printing it to the terminal and by sending it to *less*. However, while the latter method makes the output more readable – especially if it is large – it can be inconvenient when making multiple smaller queries such as `insert` as the user needs to exit *less* before inputting another query. Therefore, smaller output – currently defined to be less than 80 characters in length – is printed to the terminal rather than sent to *less*.

In the event that the application fails to send the output to *less*, for example if the application is run from the Windows command line on a standard machine, the output is printed to the terminal instead.

4.5 Loading Files and Modules

When working with databases there will always be a need to create custom tables to fit the domain. Also there might be a need to sort and handle the results of database queries. A good way of representing data in Haskell is by using data types. Defining data types directly at the prompt is not very efficient and very time consuming, especially if there is a need to define helper functions to manipulate the data type in various ways. That is why the application allows the user to load custom Haskell files where all this can be implemented. Outside of that there might be a need to use functions in existing libraries. Because the result of the queries are lists, a library for list manipulation might be useful. In Haskell there is a library called *Data.List* that has various functions that are useful for handling lists. This is why the application also allows the user to load existing Haskell modules.

4.5.1 Loading Modules

The way to load modules in the Daison front-end is the same way it is done in GHCi, by using the module command. Running `:module <name>` will attempt to import an existing Haskell module with the specified name. This is done is by getting the existing list of modules from the DaisonI state, then adding the new module to the list. The way modules are set is by setting the context in the current GHC Session and giving the `setContext` function the list of modules as argument.

4.5.2 Loading Haskell Files

The way to load Haskell files in the application is the same way it is done in GHCi, by using the `:load <path>` command. This is done in a similar way to loading modules, but with one extra step. The GHC API gives the ability to load a target (the path to the file) which then compiles the file (as `ghc --make` does). The next step is to get the module name from the Haskell file and load it as done with regular Haskell modules.

4.6 Software Architecture

The architecture of the software relies on modularisation. The different modules and how they interact with each other are illustrated in Figure 4.3.

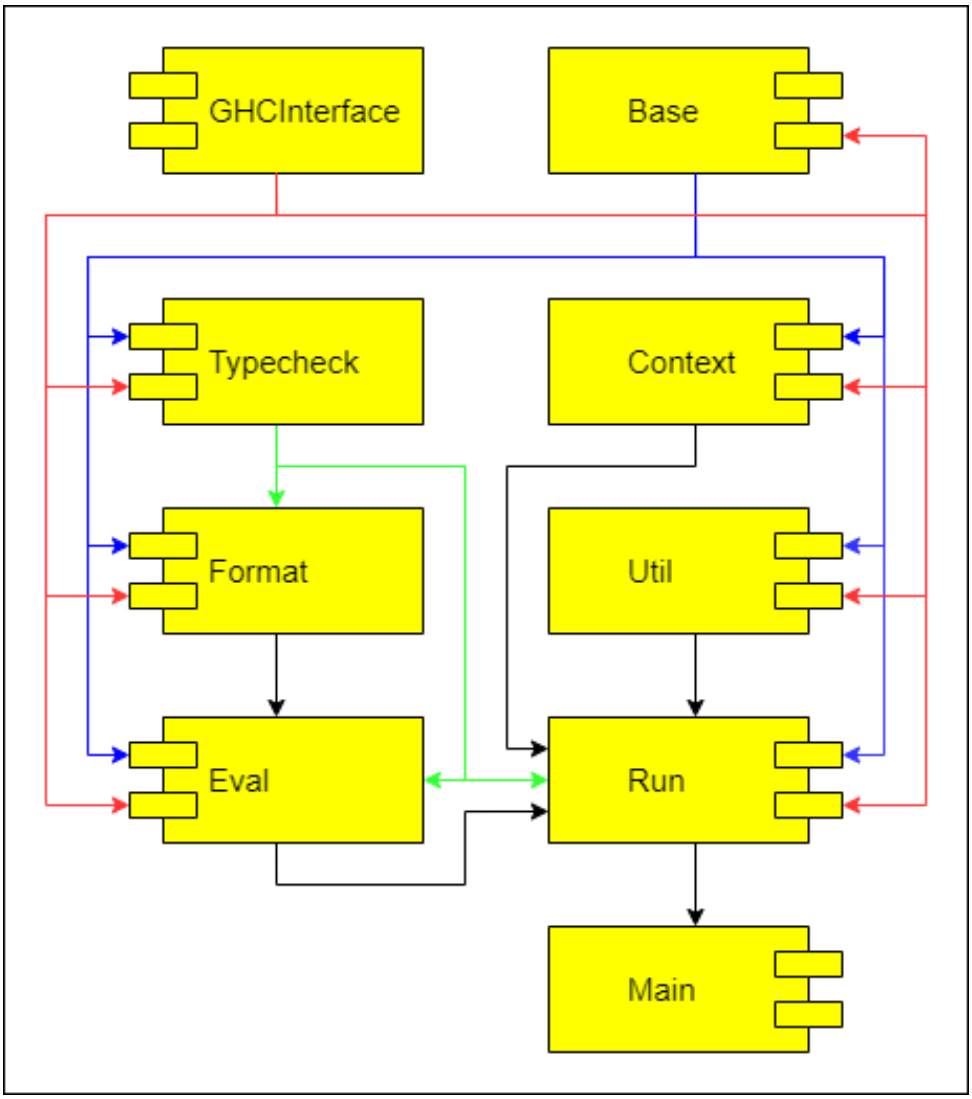

Figure 4.3: An overview of the relations between the front-end's modules. Modules that are depended upon by more than one other module have their export lines colored.

The module *GHCInterface* is, as the name implies, an interface towards the GHC library. Imports from GHC should be made in this module only. It should in turn be imported as *qualified* with the alias `GHC`. This means that whenever a function from the GHC library is used it must be denoted by `GHC.functionName`. The big advantage is that it is clear which functions come from GHC and which functions do not. The *Base* module defines the `DaisonI` monad and the `DaisonState`.

Context management should be handled in the *Context* module. This includes loading modules and setting language extensions. Type checking is handled in the *Typecheck* module and evaluation of statements in the *Eval* module. Finally, the program is located in the *Run* module, with the *Main* module containing the function used to create a line-reading application.

4.6.1 The DaisonI Monad

The `DaisonI` monad is used to replicate the behaviour of a state transformer monad for the `DaisonState`, by also wrapping and transforming `GhcMonad.Ghc` we get a minimal implementation. The state contains nine different values, listed in Table 4.2.

Table 4.2: Fields defined in `DaisonState`.

Field	**Type**	**Description**
`mode`	`AccessMode`	Mode for accessing the database. Can either be `ReadWriteMode` or `ReadOnlyMode`.
`activeDB`	`Maybe String`	Current active database.
`openDBs`	`[String]`	List of databases that are currently open.
`modules`	`[InteractiveImport]`	List of imported modules.
`flags`	`Maybe DynFlags`	Extra GHC flags.
`input`	`Bool -> String -> IO (Maybe String)`	Function used to receive user input, where the boolean value indicates if the input should be logged and the string gives the prompt message.
`logInput`	`Bool`	Log input to allow arrow key navigation.
`logPath`	`Maybe FilePath`	Path to log file.
`currentDirectory`	`String`	Current working directory.

A transformer monad is used to encapsulate functionalities of other monads into the current monad [23]. Common usages would be to handle `IO` actions and error handling within other monads. Because the GHC API is essential to the project the ability to work and handle computations in the `Ghc` monad is essential as well. With the replication of the state transformer, replication of state manipulation and

the ability to perform GHC computations in the same monad is now possible, which allows the ability to keep track of the `DaisonState` while using the GHC API.

Using the already existing state transformer monad is not sufficient since the ability to perform `IO` computations is necessary. Being able to use GHC computations in the `DaisonI` monad allows the ability to lift `IO` computations in the `Ghc` monad and thereby making it possible to lift `IO` computations in the `DaisonI` monad since the GHC monad is an instance of `MonadIO` [6], [13]. These `IO` computations are required for reading user input and to print results of queries.

4.7 Requirement Changes

As work on the project progressed, it became apparent that some of the requirements initially created needed rephrasing. The fifth requirement used to state that the front-end was to accept the Daison monad only; this would render the application nigh unusable as it implied users would not be able to for instance store data in variables for later use. Requirement eight was considered optional for some time, but was implemented together with requirement four. Furthermore, some requirements were ambiguous and have been revised to more clearly specify their intent: this includes the second, third, fourth and sixth requirements.

Some additional features were found to be necessary to provide a better user experience and were added after the initial eight requirements. These include being able to navigate the file system, similar to the `:cd` command in Windows and Unix command prompts; to load external modules as can be done through `:load` and `:module` in GHCi; to set GHC options at runtime and startup; to open databases at runtime and startup as well as closing them at runtime.

4.8 Test Results

As mentioned in Section 3.2, three types of tests were carried out: functional tests, unit tests and usability tests. Manual functional tests were carried out in order to verify whether the application fulfils the requirements, while unit tests and automated functional tests were used during development to lower the risk of bugs being introduced by new changes. Usability tests aimed to investigate the application's ease of use from a user perspective.

4.8.1 Manual Tests

Thirteen functional test case specifications were made to check whether the requirements were fulfilled. Table 3.1 provides a brief description of each test while Appendix A contains the specifications in their entirety. All manual tests passed in the most recent version of the software. Since we have full requirements coverage, we come to the conclusion that all requirements are fulfilled. Many of the functional tests could potentially be automated; however, this was not attempted due to time

constraints.

4.8.2 Automated Tests

Unit tests comprise the majority of the currently automated tests; the exception being two functional tests. Tests of the former type primarily focus on ensuring that the functions for categorising Haskell code and executing it work as expected, while tests of the latter type simulate user input to validate certain functionality. All automated tests pass in the most recent version of the software.

The current unit tests fall into four categories: code execution, expression categorisation, type checking and query detection. Requirement four is addressed by the first two categories: tests in the former category verify that the GHC API functions used to execute code work as expected, while tests in the latter category check that input code is correctly identified as being either a statement or a declaration so that it can be sent to the correct GHC API function. The third and fourth categories examine requirement three: tests in the third category verify that GHC's type checker works as expected, which is a precondition for the tests in the fourth category to work as they check that Daison queries are correctly identified as such.

Automated functional tests – of which there are currently only two – use a different input method which allows input to be read sequentially from a list of strings rather than from standard input, thereby removing the need for manual input. The currently defined tests check requirements one through five by checking whether data read from the front-end's interface matches what is written through the back-end's interface and vice versa. Requirements one through three as well as requirement five are tested through `select` and `insert` queries, while requirement four is tested through the input method used. Passed tests suggest that the tested functionality works as expected even with random inputs.

4.8.3 Usability Tests

The usability of the application was evaluated by two groups of four and five participants each. All participants were students at either Chalmers University of Technology or the University of Gothenburg. They had either taken at least one course in functional programming or were doing their Bachelor's thesis project in a functional language. Six out of the nine participants had previous experience working with databases, and all participants except one said they were comfortable working in a command line.

Each session contained nine tasks for the participants to complete. After the completion of each task, the participant was asked to rate the difficulty of the task from one to five, where one corresponds to *very easy* and five corresponds to *very difficult*. The rating is summarised in Table 4.3, whilst Appendix D shows the numerical values in their entirety. However, since the participants correspond to a relatively small sample, we can not draw any general conclusions from their rating of the difficulty of the tasks. It is worth mentioning that some participants found a task difficult if

they could not solve it right away. This does not necessarily mean that the usability of the application has shortcomings. Part of good usability is that the system gives appropriate feedback once an error occurs to help the user recover and complete the task. It should be noted that all participants managed to solve all tasks in the end. Overall, the participants rated the usability of the application as *good* or *excellent*, as shown in Table 4.4.

Table 4.3: The difficulty of the tasks in the usability test according to the participants. The mean and median ratings include all nine participants. Ratings were between one and five, with one meaning the task was *very easy* and five being *very difficult*.

Task	**Mean**	**Median**
1	1.9	1
2	1.9	1.5
3	2.9	3
4	2.2	2
5	1.2	1
6	1.9	2
7	1.0	1
8	1.2	1
9	2.0	2

Table 4.4: Overall impression of the usability of the application. The possible ratings were *Worst-imaginable, Awful, Poor, OK, Good, Excellent* and *Best imaginable*.

Participant	Overall Impression
1	Excellent
2	Good/Excellent
3	Good
4	Good/Excellent
5	Excellent
6	Good
7	Excellent
8	Excellent
9	Excellent

During the first four evaluations of the application we observed the following design problems:

- The description of the `:open` command was confusing. It was not clear that the command will create a new database if the database does not already exist.
- The message shown when trying to interact with a database when no database is open did not tell the user how to recover from the error.
- The success message from the `:cd` command was confusing. It was not clear whether the directory change was successful or not.

To fix the first issue, we simply rephrased the description of the `:open` command. The second issue was solved by changing the error message to `No open database found. Try :open <name>.` and the third issue was solved by changing the success message of the `:cd` command to `Working directory set to <path>`.

After fixing the issues we did another five evaluations of the application to validate that our solutions solved the problems. After the second round of usability tests we observed that the issues found in the first round was solved. However, we found the following issues:

- Having two different commands (`:open` and `:db`) for the same operation is confusing.
- Having a command for showing the contents of the current directory would be helpful.
- Entering a command without passing any parameters gives the error message `Unknown command` even though the command exists.
- It is not clear that the `:set` command sets GHC flags.

In the final version of the application we removed the `:db` command, added a command that allows the user to run normal shell commands (`:! <cmd>`), fixed the bug when not passing any parameters to valid commands, and specified that the `:set` command sets GHC flags. We are confident that these changes solves the problems, even though we did not validate it with a third group of users.

Additionally, during the usability evaluations we found that the available commands of our application are similar to the commands of GHCi which is helpful for Haskell programmers. Because of this, programmers with experience with GHCi could easily get started using our application.

As mentioned in Section 3.2.3, we formulated three questions that we would like the usability study to provide answers to:

1. Is the application easy to use?

2. Is the visual feedback provided by the application enough for the user to understand the program?

3. Does the consequence of any interaction with the application match the user's expectations?

Based on the overall rating by the participants and the fact that the participants managed to solve all tasks we gave them, we come to the conclusions that the application is easy to use and the visual feedback is enough for the user to fully understand the program. Additionally, we found no situations where the consequence of interaction did not match the user's expectations.

5

Discussion

This chapter discusses problems that arose during implementation and their solutions. It also comments on some of the reasoning behind the results. First, we address how newcomers can understand the GHC API. Second, we discuss the testing methods used. Third, we present ethics that have to be considered. Finally, we present new features and functionality, which we believe are the most beneficial to the application.

5.1 Understanding the Glasgow Haskell Compiler

As mentioned in Section 3.1, the official documentation and tutorials on using GHC as a library were outdated and not as helpful as expected. This lead to the first task, getting the GHC API into scope, taking more resources than expected. The API is updated regularly, and together with the updates, the source code comments are updated. Unfortunately, these do not transition over to the tutorials. Only the code comments are not simple enough for newcomers to understand. The source code comments and code from other packages were eventually pieced together until a good enough understanding was found. Having to study the source code of other packages to see how they implement the API, instead of following the documentation, should not be necessary for programmers using the API for the first time.

5.2 Usage of Test Methods

In hindsight, we are uncertain whether we focused on the correct testing methods. Perhaps an excessive amount of time was spent creating unit tests and manual functional testing compared to automated property-based testing and automated functional testing. We are nonetheless confident that a high focus on usability tests was worthwhile. It gave a substantial amount of suggestions on issues, which may have gone unnoticed if there were no external testers. The usability testing did not reveal any broken features, but it became apparent where the user got confused and needed more guidance from the application.

5.2.1 Unit Testing

As much of the initial codebase was related to altering the `DaisonState`, creating unit tests for these functions was not trivial. For example, we wanted to test that the `addImport` function correctly adds a module to the session. The first idea that came to mind was to use the function and see whether the `InteractiveImport` was in the resulting list from `GHC.getContext`. There is however no easy way to check the contents of the list because the data type `InteractiveImport` does not implement the `Eq` type class. The same argument follows for much of the code tightly related to the GHC API and state management.

Running expressions and declarations within `DaisonI`, as well as type-checking, was more easily testable. We created a function that runs an action within `DaisonI` and then returns the result as a string. This way we could compare that the output was equal to the result from running the expression in a normal GHC context.

5.2.2 Property-Based Testing

Creating property-based tests for specific functionality would encounter the same problems as unit tests when testing single functions. On the other hand, QuickCheck can generate large sample sizes of inputs; which as a result would cover most edge cases that can be hard and tedious to catch by hand.

Since Daison can store any Haskell data type, even types that the user define themselves, testing all possible inputs would become an infinite task. The current tests thus only test a few specific data types which we have defined. The entries are still randomly generated by the library, which is better than manual definitions.

5.2.3 Test Automation

Automated testing is currently employed to a smaller extent than initially planned; writing these tests proved to be more difficult and time-consuming than expected. As the project ran during a limited time period prioritisation had to be put elsewhere.

Expanding the suite of automated tests – both unit tests and functional tests – would have a positive impact on the application as a whole. From a developer's perspective, the code would become easier to maintain as they would need to spend less effort on manually verifying that a change to the code does not have unintended consequences on already established functionality. From a user's perspective, the Continuous Integration status badge shown in the GitHub repository's *README* file would provide better and more comprehensive assurance of quality.

5.2.4 Further Testing

In addition to automating the manual tests – possibly excluding F8 as it checks whether tables are correctly formatted – it would be beneficial to include tests that do not necessarily relate directly to specific requirements. For example, tests that

validate the functionality of front-end commands such as `:dbs` and `:log`; while these cannot be linked to any requirements, they are nevertheless useful features.

5.3 Ethical Considerations

In order to allow the user to navigate through and reuse previous input, a local log of recent input needs to be kept, which the user is informed of at startup. This log is called *.daison_history* and is stored in *%APPDATA%/Daison-Frontend/* on Windows systems and *∼/.Daison-Frontend/* on Unix systems. If the user for example uses the `insert` query to insert sensitive data into a table, the log will contain this query. To address this potential source for data leaks, a `:log` command is included with which the user can clear the log and/or temporarily stop the front-end from writing to it. While the command makes an attempt to render the erased contents unrecoverable by first overwriting it, this may not be effective depending on the storage medium used [24].

An important consideration is that our application does not take extra steps to ensure the confidentiality of the contents of the database. Since encryption is not supported by the base SQLite library used to implement Daison [25], database files are not currently encrypted. However, we argue that it is the responsibility of the back-end to store data securely. Therefore, we do not find this to be a vulnerability of our application.

One could argue that since the application is used for interacting with a database, all ethical questions regarding databases have to be considered as well. Let us discuss the ethical aspects by using three ethical frameworks: the Consequentialist Framework, the Duty Framework, and the Virtue Framework. According to the Consequentialist Framework, it is only the consequences of an action that determine if the action is morally right. We do not see any negative consequences of *developing* a tool for database interaction. However, we see some negative consequences if the tool is *used* to do harm, for instance, if someone uses the application to store personal information without consent. This raises an important question: Do programmers have to take moral responsibility for how other people use their software? If that is the case, we find the Consequential Framework to not be helpful for developers seeking guidance on how to act morally correct since one cannot know the actual consequences of an action in advance. The Duty Framework, on the other hand, determines whether an action is morally right or wrong based on a set of rules. Based on the assumption that the rules we should follow is the Swedish law, we find that we have acted morally correct since the development of this application does not violate any Swedish laws. In addition, the framework states that each individual is responsible only for his or her own actions, and thus, developers do not have any moral responsibility for how other people use their software. Finally, the Virtue Framework states that moral virtues, i.e. qualities that are deemed to be morally good, are central to ethics. Since our intent with the application is to make it easier for Haskell developers to use Daison, we find that we have acted virtuously. In addition, similar to the Duty Framework, the Virtue Framework

states that individuals are responsible only for his or her own actions.

In conclusion, after considering the ethical aspects of developing an application for database interaction, we find that there are no ethical issues of *developing* the application. Additionally, since we have informed the user that a log is kept for arrow key navigation and has given the ability to turn it off, we find that there are no ethical issues regarding the application itself either.

5.4 Future Work

The application has the potential to be improved by adding new functionalities. We believe that the parts presented in this section are the ones that would provide the most benefit to a user.

5.4.1 Table Formatting

One important feature of the front-end is to present output from Daison queries in a more readable manner. It might be worthwhile exploring how this could be done more effectively for a larger variety of data types. Furthermore, as there may be multiple ways to display the same data – for example, different column labels – adding a command that lets the user manage their preferences regarding output formatting may improve the user experience.

While all query results except for single-row entries are displayed as tables, the number of columns in a table only changes if the rows contain tuples. Tables are generated with two columns in all other cases, with the second column containing the data with minimal additional formatting. This may suffice for many smaller data types; however, there are at least two types where separating data would likely improve readability: lists and data types with a single constructor. In both of these cases, introducing an additional header row may enhance readability.

For tables containing lists, assigning a column for every element would make it easier for the user to distinguish between different values and determine the index for a particular element. However, this will in most cases lead to rows with different numbers of columns. One solution could be to pad smaller rows with additional columns, but whether this padding will distract from the data might depend on the user; therefore, multiple different formats may be desired. Figure 5.1 displays a comparison between the current implementation and possible improvements.

For tables containing data types with a single constructor, it might be desirable to have sub-columns for each record, with each sub-column containing one field. A header for each sub-column could indicate the type signature of the field, the name of the function used to access it, or both. Figure 5.2 shows a comparison between the current implementation and possible improvements. However, these improvements will not be enough for data types with multiple constructors. Although an additional sub-column showing the constructor can solve the case where all constructor

arguments have the same types, a separate way of formatting would be needed for the case where they do not. This is because a column could otherwise contain values of different types than is indicated by the second header row.

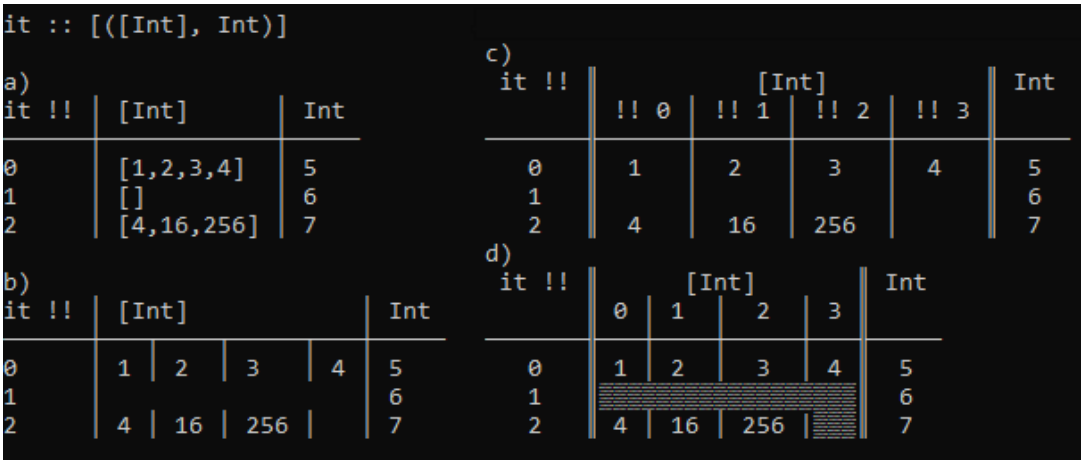

Figure 5.1: The contents of a table shown through *less*, where **a)** shows the current table formatting and **b)** through **d)** show possible improvements. **b)** and **c)** adds sub-columns, while **c)** furthermore adds another header row, uses double vertical lines to separate columns with sub-columns, uses centred text alignment and fills in missing sub-column separators. **d)** is similar to **c)** but omits the double exclamation marks from the second header row and shades empty sub-cells.

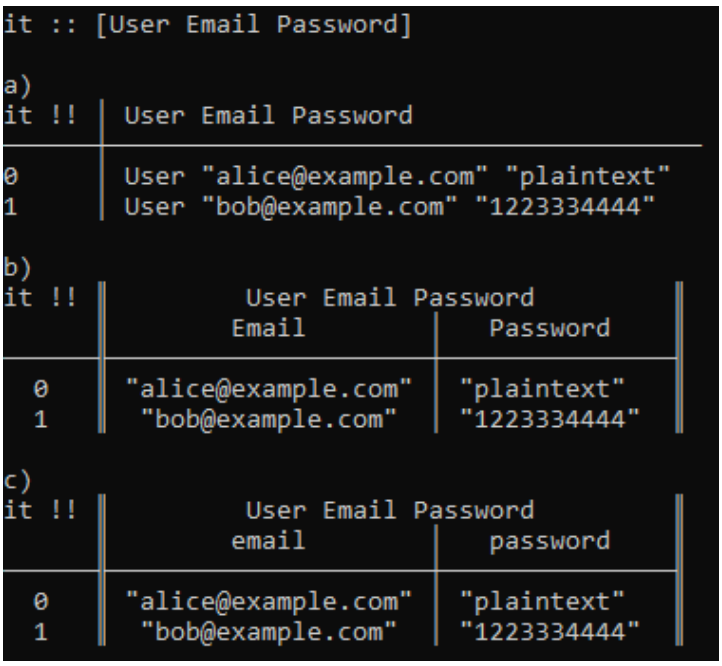

Figure 5.2: The contents of a table shown through *less*, where **a)** shows the current table formatting while **b)** and **c)** show possible improvements. Both format single-constructor data types similarly to Figure 5.1.c), with **c)** replacing the types in the second header row with functions created using the record syntax (or otherwise), assuming the GHC API can be used to detect these.

5.4.2 Security of User Data

The Daison back-end could provide built-in methods for hashing, encrypting and decrypting using the latest algorithms, which the user can use at demand. An extension to make the logging safer and not risk exposing data is encrypting the logs. One potential problem is that consistently encrypting and decrypting the log file might worsen the performance of the application. Therefore an option to toggle encryption on and off would be helpful, giving the user the option to add security but lower performance.

5.4.3 Input Interface

The function used to obtain user input is provided by the library *Haskeline* [26]. It provides customisation options such as whether a history file should be used, whether user input should be added to said file automatically and what function should be used to provide tab completion. Out of the options mentioned, the tab completion function is the only one kept at default; this function fills in file names based on files in the directory the user launched the application from. Substituting the default function for a custom one would likely improve the user experience.

A custom tab completion function could have several improvements compared to the default implementation that would make it more suitable for the application's interface. Changing directory using the `:cd` command does currently not change which file names can be autocompleted; a function for which this is the case would make the interface more intuitive to the user. Tab completion for the commands provided by the application would also make it more user-friendly: pressing *TAB* after entering `:` would display all available commands; `:log p` would autocomplete to `:log path`. The command completion feature could also be used to avoid the need to explicitly define short-hand commands such as `:m` for `:module`.

Additional commands, such as GHCi's *multiline* command could also be implemented. The command allows for user input to span across multiple lines similarly to when writing compiled code. This would make defining tables with accompanying indices through the application more convenient, as shown in Figure 5.3.

```
a)
Daison> (people, people_name, people_age) = (table "people" `wi
thIndex` people_name `withIndex` people_age :: Table (String, I
nt), index people "name" fst, index people "age" snd)

b)
Prelude Database.Daison> :{
Prelude Database.Daison| people :: Table (String, Int)
Prelude Database.Daison| people = table "people"
Prelude Database.Daison|          `withIndex` people_name
Prelude Database.Daison|          `withIndex` people_age
Prelude Database.Daison| people_name = index people "name" fst
Prelude Database.Daison| people_age = index people "age" snd
Prelude Database.Daison| :}
```

Figure 5.3: A definition of a table that stores `(String, Int)` tuples along with indices for each element. **a)** writes this definition in the application using a single line, while **b)** uses the GHCi *multiline* command.

5.4.4 Supporting Multiple Back-Ends

The front-end is only compatible with Daison as a back-end to communicate with databases. A developer who does not think Daison suits the style of database management which they need would therefore find it irrelevant. This can be solved by producing a more generic front-end, which works with multiple back-ends that handle databases in Haskell.

We suggest a solution where the front-end defines a type class, which database handling monads (such as the `Daison` monad) could be an instance of, to mark their compatibility with the front-end. The front-end would then use the type class throughout the application instead, making it modular and less coupled to a specific library. This would in turn yield a product that any developer who is producing a database manager in Haskell could use for more convenience.

5.4.5 A Graphical User Interface

A Graphical User Interface (GUI) as an extension of the front-end would improve the user experience and make the front-end accessible to programmers outside of Haskell since it could remove the need to write Haskell code completely. Even though the command line interface presented here has removed much boilerplate code and the need to keep track of a database reference there is still some inconvenience. For instance, to see all open database connections the user currently need to use the `:dbs` command every time. A GUI could display a list of all known connections, which would be more convenient. Clicking on a database in the list could then provide options to open, close or switch the active database to the selected one. Double-clicking on a connection in the list could also take the user to a screen showing more detailed information like existing tables and their data. Finally, it could add a query builder where the user can build queries with buttons instead of writing Haskell code.

6

Conclusions

Using Daison interactively in its current state was inconvenient. Interactively writing queries required the user to explicitly bind a database reference and keep track of the bound variable. The user also needed to write wrapping boilerplate code to run single queries. Our front-end manages these parts using the GHC API and provides easy access to databases using the Daison library.

The purpose of the project was to build a front-end for Daison by using GHC as a library. This purpose was broken down into requirements, all of which have been fulfilled to a satisfactory degree. Thus, we deem the project successful.

The base of the front-end uses a state where it keeps track of open databases and settings which the user has set. Most of this functionality is either related to the GHC API or the Daison back-end. By keeping this information and evaluating statements entered in the prompt, the usability of Daison has increased. As the front-end also formats output, the readability of query results is greatly improved.

As mentioned previously, the application improves the usability of Daison substantially. Further additions can improve it a great deal more and make the program usable for people who do not have experience in Haskell.

Bibliography

[1] E. Meijer, B. Beckman, and G. Bierman, "LINQ: Reconciling Object, Relations and XML in the .NET Framework," in *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '06, Chicago, IL, USA: Association for Computing Machinery, 2006, p. 706, ISBN: 1595934340. DOI: 10.1145/1142473.1142552. [Online]. Available: https://doi.org/10.1145/1142473.1142552.

[2] D. Spiewak and T. Zhao, "ScalaQL: Language-Integrated Database Queries for Scala," in *Software Language Engineering*, M. van den Brand, D. Gašević, and J. Gray, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 154–163, ISBN: 978-3-642-12107-4.

[3] A. Ekblad, "Scoping monadic relational database queries," in *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell*, ser. Haskell 2019, Berlin, Germany: Association for Computing Machinery, 2019, pp. 114–124, ISBN: 9781450368131. DOI: 10.1145/3331545.3342598. [Online]. Available: https://doi.org/10.1145/3331545.3342598.

[4] K. Angelov. (Mar. 12, 2021). "A NoSQL database in Haskell." Commit: 57849f9, [Online]. Available: https://github.com/krangelov/daison (visited on Apr. 18, 2021).

[5] S. Marlow. (Apr. 2010). "Haskell 2010 Language Report," [Online]. Available: https://www.haskell.org/definition/haskell2010.pdf (visited on Apr. 2, 2021).

[6] The GHC Team. (2021). "MonadUtils," [Online]. Available: https://hackage.haskell.org/package/ghc-8.10.2/docs/MonadUtils.html#t:MonadIO (visited on Apr. 2, 2021).

[7] J. Launchbury and S. L. Peyton Jones, "State in Haskell," *LISP and Symbolic Computation*, vol. 8, no. 4, pp. 293–341, Dec. 1995, ISSN: 1573-0557. DOI: 10.1007/BF01018827. [Online]. Available: https://doi.org/10.1007/BF01018827.

[8] A. Gill. (2001). "Control.Monad.State.Lazy," [Online]. Available: https://hackage.haskell.org/package/mtl-2.2.2/docs/Control-Monad-State-Lazy.html (visited on Apr. 2, 2021).

[9] (2021). "SQLite Home Page," [Online]. Available: https://www.sqlite.org/ (visited on May 13, 2021).

[10] K. Angelov. (Mar. 12, 2021). "daison/tutorial.md at master · krangelov/daison." Commit: 57849f9, [Online]. Available: https://github.com/krangelov/daison/blob/master/doc/tutorial.md (visited on Apr. 18, 2021).

[11] B. Gamari. (2021). "Home — The Glasgow Haskell Compiler," [Online]. Available: https://www.haskell.org/ghc/ (visited on Apr. 8, 2021).

[12] S. Marlow and S. Peyton Jones, "The Glasgow Haskell Compiler," in *The Architecture of Open Source Applications, Volume 2*, The Architecture of Open Source Applications, Volume 2. Lulu, Jan. 2012. [Online]. Available: https://www.microsoft.com/en-us/research/publication/the-glasgow-haskell-compiler/.

[13] The GHC Team. (2021). "GHC," [Online]. Available: https://hackage.haskell.org/package/ghc-8.10.2/docs/GHC.html (visited on Apr. 2, 2021).

[14] The Hint Authors. (2021). "hint," [Online]. Available: http://hackage.haskell.org/package/hint (visited on Apr. 2, 2021).

[15] C. Kaltenbrunner, A. Neldefors, H. Stegrell, and P. Vedin. (May 7, 2021). "Daison Frontend." Commit: c2b4d0a, [Online]. Available: https://github.com/PUGzera/DATX02-DIT561 (visited on May 8, 2021).

[16] Atlassian. (2021). "Gitflow Workflow | Atlassian Git Tutorial," [Online]. Available: https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow (visited on Jan. 25, 2021).

[17] D. Herington. (Jan. 2021). "HUnit: A unit testing framework for Haskell," [Online]. Available: https://hackage.haskell.org/package/HUnit (visited on Apr. 11, 2021).

[18] "ISO/IEC/IEEE International Standard - Systems and software engineering–Vocabulary," *ISO/IEC/IEEE 24765:2017(E)*, pp. 1–541, Sep. 28, 2017. DOI: 10.1109/IEEESTD.2017.8016712. (visited on May 8, 2021).

[19] N. Smallbone. (Jan. 2020). "QuickCheck: Automatic testing of Haskell programs," [Online]. Available: https://hackage.haskell.org/package/QuickCheck (visited on Apr. 11, 2021).

[20] J. Nielsen and T. K. Landauer, "A mathematical model of the finding of usability problems," in *Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems*, ser. CHI '93, Amsterdam, The Netherlands: Association for Computing Machinery, 1993, pp. 206–213, ISBN: 0897915755. DOI: 10.1145/169059.169166. [Online]. Available: https://doi.org/10.1145/169059.169166.

[21] B. Hanington and B. Martin, *Universal Methods of Design: 100 Ways to Research Complex Problems, Develop Innovative Ideas, and Design Effective Solutions*. Rockport Publishers, 2012, ISBN: 9781610581998.

[22] (2021). "Pretty," [Online]. Available: https://hackage.haskell.org/package/ghc-8.10.2/docs/Pretty.html (visited on Apr. 15, 2021).

[23] S. Liang, P. Hudak, and M. Jones, "Monad transformers and modular interpreters," in *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '95, San Francisco, California, USA: Association for Computing Machinery, 1995, pp. 333–343, ISBN:

0897916921. DOI: 10.1145/199448.199528. [Online]. Available: https://doi.org/10.1145/199448.199528.

[24] M. Wei, L. M. Grupp, F. E. Spada, and S. Swanson, "Reliably Erasing Data from Flash-Based Solid State Drives," in *Proceedings of the 9th USENIX Conference on File and Storage Technologies*, ser. FAST'11, San Jose, California: USENIX Association, 2011, p. 7, ISBN: 9781931971829. DOI: 10.5555/1960475.1960483. [Online]. Available: https://dl.acm.org/doi/10.5555/1960475.1960483.

[25] (2021). "How to Compile and Use SEE," [Online]. Available: https://www.sqlite.org/see/doc/release/www/readme.wiki (visited on May 3, 2021).

[26] J. Jacobson. (2021). "haskeline: A command-line interface for user input, written in Haskell.," [Online]. Available: https://hackage.haskell.org/package/haskeline (visited on May 6, 2021).

A

Test Case Specifications

This appendix contains the test case specifications that were carried out *manually* in order to verify that the application fulfilled the requirements.

A.1 Functional Test #1

ID

F1

Version

2021-04-01

Description

Requirement: R1

Purpose: The purpose of the test is to verify that the Daison library is loaded at startup.

Preconditions

None

Test Steps

1. Start the program
2. Verify that the following Daison functions are in scope by looking up their types (`:t <command>`):
   - `createTable`
   - `tryCreateTable`

- `dropTable`
- `tryDropTable`
- `insert`
- `insert_`
- `select`
- `from`
- `update`
- `update_`
- `store`
- `delete`
- `delete_`

Expected Result

The Daison functions should be in scope.

- `createTable :: Table a -> Daison ()`
- `tryCreateTable :: Table a -> Daison ()`
- `dropTable :: Table a -> Daison ()`
- `tryDropTable :: Table a -> Daison ()`
- `insert :: Data a => Table a -> Query a -> Daison (Key a, Key a)`
- `insert_ :: Data a => Table a -> a -> Daison (Key a)`
- `select :: QueryMonad m => Query a -> m [a]`
- `from :: From s r => s -> r (K s) -> Query (V s r)`
- `update :: Data a => Table a -> Query (Key a, a) -> Daison [(Key a, a)]`
- `update_ :: Data a => Table a -> Query (Key a, a) -> Daison ()`
- `store :: Data a => Table a -> Maybe (Key a) -> a -> Daison (Key a)`
- `delete :: Data a => Table a -> Query (Key a) -> Daison [Key a]`

- `delete_ :: Data a => Table a -> Query (Key a) -> Daison ()`

A.2 Functional Test #2

ID

F2

Version

2021-05-14

Description

Requirements: R2, R3 and R5

Purpose: The purpose of the test is to verify that it is possible to write queries at the prompt, a transaction is started automatically and that the front-end accepts the Daison monad.

Preconditions

A database connection should be open and a table should be created.

Table definition example:

```
data People = People { name :: String, age :: Int } deriving Data

let (people, people_name, people_age) = (table "people"
↪  `withIndex` people_name `withIndex` people_age :: Table
↪  People, index people "name" name :: Index People String,
↪  index people "age" age :: Index People Int)
```

Test Steps

1. Execute Daison queries

Example Daison queries:

```
tryCreateTable people

insert people (return (People "Alice" 23))
insert people (return (People "Bob" 25))

select [name x | x <- from people (at 1)]
```

Expected Result

The Daison queries should be executed successfully. Example queries should yield result `["Alice"]`.

A.3 Functional Test #3

ID

F3

Version

2021-03-26

Description

Requirement: R4

Purpose: The purpose of the test is to verify that it's possible to interactively write Haskell code at the prompt.

Preconditions

None

Test Steps

1. Write Haskell code at the prompt.
2. Verify that the code works as expected.

Expected Result

The code should be working as expected.

A.4 Functional Test #4

ID

F4

Version

2021-05-13

Description

Requirement: R9

Purpose: The purpose of the test is to verify that it is possible to navigate the file system at the prompt.

Preconditions

None

Test Steps

1. Navigate to a new directory using the `:cd` command.
2. Open a database using the `:open` command.
3. Check that the database is created in the current working directory.

Expected Result

The navigation of the filesystem is successful with a database in the current directory.

A.5 Functional Test #5

ID

F5

Version

2021-03-26

Description

Requirement: R10

Purpose: The purpose of the test is to verify that it is possible to load Haskell modules at runtime.

Preconditions

A Haskell module should be created.

Test Steps

1. Import the Haskell module using the `:module` command.
2. Verify that the module is loaded properly by calling its functions.

Expected Result

The module should be loaded successfully.

A.6 Functional Test #6

ID

F6

Version

2021-04-15

Description

Requirement: R11

Purpose: The purpose of the test is to verify that it is possible to set flags at runtime.

Preconditions

None

Test Steps

1. Set a flag at runtime using the `:set` command.
2. Verify that the flag is set successfully.

Example input:

```
[ (x+y) | x <- [1..10] | y <- [11..20] ] -- Should not work
:set -XParallelListComp
[ (x+y) | x <- [1..10] | y <- [11..20] ] -- Should work
```

Expected Result

Flags should be set successfully.

A.7 Functional Test #7

ID

F7

Version

2021-04-08

Description

Requirement: None

Purpose: The purpose of the test is to verify that an unknown command shows an error message.

Preconditions

None

Test Steps

1. Enter an invalid command at the prompt, for instance `:hej "hejhej"`.
2. Verify that an error message is shown.

Expected Result

An error message should be shown successfully.

A.8 Functional Test #8

ID

F8

Version

2021-04-15

Description

Requirements: R6 and R7

Purpose: The purpose of the test is to verify that non-small query results are displayed to the user as a formatted, navigable table.

Preconditions

1. The test database *test.db* and its table definitions *TestDefs.hs*.
2. A terminal that supports the command `echo string | less`, i.e. supports the programs *echo* and *less*, as well as the pipe symbol `|` to transfer the output of *echo* to *less*.

Test Steps

1. Load the test database and its table definitions.
2. Enter the query `select [x | x <- from TABLE everything]` for any table in *TestDefs* except for `emptyTuple` and `empty`.
3. Verify that it is formatted in a table with three columns.
4. Enter the query `select [snd x | x <- from TABLE everything]` for the tables `either3I`, `emptyTuples`, `ssfb`, `tupleInTuple` and `listOfNumbers`.
5. Verify that output from all queries are formatted in tables with appropriate numbers of columns.
6. Enter the query `select [snd x | x <- from TABLE everything]` for the table `numbers`.
7. Verify that the arrow keys can be used to navigate in all cardinal directions.
8. Enter the query `select [snd x | x <- from TABLE everything]` for the tables `empty` and `emptyTuple`.
9. Insert a value into any table, and then remove it.

10. Verify that the outputs in steps 8 and 9 were printed directly to the terminal rather than as a formatted table.

Expected Result

The table in step 2 should have three columns: element index, key, data.

The tables in step 4 should have numbers of columns equal to one plus the number of elements in the top-level tuple as shown in the type signature for it, and two columns if `it` is not a list of tuples.

Below are examples of the correct number of columns given the type signature for `it`:

- `[(String, String, Float, Bool)]` => 5 columns
- `[String]` => 2 columns
- `[[Bool]]` => 2 columns
- `[(Int, (Int, Float, Bool))]` => 3 columns
- `[Maybe (Int, Int, Int)]` => 2 columns

Large tables should be navigable using the arrow keys.

Queries that produce small output (e.g. `insert`, `delete`, `createTable`, `dropTable`, `select` on a table that outputs at most one small row) should be printed directly to the terminal, unformatted (i.e. not as a table). The output for `insert` should be a 2-tuple of `Key`s and the output for a successful `delete` should be a list containing a single `Key`.

A.9 Functional Test #9

ID

F9

Version

2021-04-15

Description

Requirement: R11

Purpose: The purpose of the test is to verify that it's possible to set flags at startup

Preconditions

None

Test Steps

1. Set a flag at startup using *ghc-options*.
2. Verify that the flag is set successfully.

Example commands:

```
foo@bar:~$ daison-frontend -XNumericUnderscores
Daison> let x = 5_12
```

Expected Result

Flags should be set successfully.

A.10 Functional Test #10

ID

F10

Version

2021-04-15

Description

Requirement: R10

Purpose: The purpose of the test is to verify that it is possible to load Haskell files at runtime.

Preconditions

There should be a Haskell module with at least one function. For example:

```
module F10Example(
    exampleFunction
) where

exampleFunction :: Int -> Int -> Int
exampleFunction a b = a*b
```

Test Steps

1. Load the Haskell module at runtime using the `load` command.
2. Verify that the module is loaded properly by calling its functions.

Expected Result

The module should be loaded successfully.

A.11 Functional Test #11

ID

F11

Version

2021-04-29

Description

Requirement: R8

Purpose: The purpose of the test is to verify that `IO` expressions are accepted at the prompt.

Preconditions

None

Test Steps

1. Use `print`, or any other function that returns an IO expression, at the prompt.

Expected Result

If `print` is used, the argument should be printed to the console if it has a `Show` instance. In particular, strings should be surrounded with double-quotes: `print "test"` should print `"test"` rather than `test`.

If a different function is used, it should work as expected.

A.12 Functional Test #12

ID

F12

Version

2021-05-10

Description

Requirement: R12

Purpose: The purpose of the test is to verify that databases can be opened and closed. Databases should also be possible to supply as an argument to the program which should then be set on start-up.

Preconditions

None

Test Steps

1. Start the program with multiple databases as arguments.
2. Check the list of open databases with `:dbs`
3. Close the currently active database (marked at the prompt message)
4. Open a new database
5. Check the list of open databases again

Example commands:

```
foo@bar:~$ daison-frontend firstDb.db secondDb.db
Daison (secondDb.db)> :dbs
Daison (secondDb.db)> :close secondDb.db
Daison (firstDb.db)> :open thirdDb.db
Daison (thirdDb.db)> :dbs
```

Expected Result

1. Upon launching the program it should respond that the arguments have been handled.
2. It should print the supplied database arguments.

3. The active database should be closed and the second-last database argument should be set to the active.

4. The active database should be set to `thirdDb.db`.

5. It should print the open databases, the one that was closed in step 3 should not be in the list.

A.13 Functional Test #13

ID

F13

Version

2021-05-27

Description

Requirement: R10

Purpose: The purpose of the test is to verify that it is possible to load Haskell files at startup

Preconditions

There should be a Haskell file with at least one function. For example:

```
module F13Example(
    exampleFunction
) where

exampleFunction :: Int -> Int -> Int
exampleFunction a b = a*b
```

Test Steps

1. Supply the path to a Haskell file at startup as an argument.
2. Verify that the file is loaded properly by calling its functions.

Example commands:

```
foo@bar:~$ daison-frontend F13Example.hs
Daison> exampleFunction 3 5
```

Expected Result

The file should be loaded successfully.

Example commands should result in `15` being printed.

B

Tasks for Evaluating the Usability

This appendix contains the tasks that were carried out in the usability tests. It should be noted that the usability study was conducted in Swedish and the tasks in this appendix have been translated to English.

B.1 Task 1: Loading Modules

Scenario: Assume that you would like to create a database for storing people. Because of Daison's use of Haskell datatypes for defining rows you have defined these in a Haskell module named *People.hs* in your current working directory.

Task: Try to load the Haskell module.

What we expect them to do: Use the `:load` command for loading a Haskell module.

B.2 Task 2: Changing Working Directory

Scenario: Assume that you would like to create a new database. Therefore, you would like to change working directory.

Task: Try to change working directory to an already created directory called *workspace*.

What we expect them to do: Use the `:cd` command for navigating the file system.

B.3 Task 3: Opening a Database and Creating a Table

Scenario: Assume that you would like to create a database for storing people. You have loaded the table definition as a variable called `people`.

Task: Try to create a database and the table `people`.

What we expect them to do: Use the `:open` command for creating a database and `tryCreateTable people` for creating the table.

B.4 Task 4: Creating an Entry

Scenario: Assume that you would like to create a database for storing peope. You have created a table `people`.

Task: Try to insert a person into the table.

What we expect them to do: Use `insert people (return (People <Name> <Age>))` to insert a person.

B.5 Task 5: Reading an Entry

Scenario: Assume that you have a database with a table for storing people and you would like to know the people which are added to the table.

Task: Try to read what people are added to the table.

What we expect them to do: Use `select [x | x <- from people everything]` to select all people.

B.6 Task 6: Updating an Entry

Scenario: Assume that you have a database with a table for storing people and you would like to update the age of the person at index one.

Task: Try to update the age of the person at index one.

What we expect them to do: Use `update people $ return (1, People { name=<Name>, age=<New Age> })` to update the entry.

B.7 Task 7: Deleting an Entry

Scenario: Assume that you have a database with a table for storing people and you would like to remove the person at index one.

Task: Try to remove the person at index one.

What we expect them to do: Use `delete people (return 1)` to delete the entry.

B.8 Task 8: Changing Database and Creating a Table

Scenario: Assume that you would like to create a new database for storing people.

Task: Try to create a new database and create the table `people`.

What we expect them to do: Use the `:open` command to change database and `tryCreateTable people` to create the table.

B.9 Task 9: Navigating Output

Scenario: Assume that you add more entries than could be displayed in the console.

Task: Add about 25 entries to the table (it could be the same person). Verify that all entries where added correctly.

What we expect them to do: Use `insert` to insert entries and `select` everything to print the contents of the table.

C

Quick Reference Used During the Usability Tests

This appendix includes the source code of a module for representing people as well as a Quick Reference for Daison used by the participants during the usability tests.

Source Code of *People.hs*

```
module People where

import Database.Daison
import Data.Data

data People = People { name :: String, age :: Int } deriving Data

people_name :: Index People String
people_name = index people "name" name

people_age :: Index People Int
people_age = index people "age" age

people :: Table People
people = table "people"
           `withIndex` people_name
           `withIndex` people_age
```

Daison Quick Reference

Create Table in Database

```
tryCreateTable <table_name>
```

Insert a Person into a Table

```
insert <table_name> (return (People <person_name> <person_age>))
```

Select All Rows from a Table

```
select [x | x <- from <table_name> everything]
```

Update Row in a Table

```
update <table_name> (return (<index_of_row>, People {name=<person_name>,
age=<person_age>}))
```

Delete Row in a Table

```
delete <table_name> (return <index_of_row>)
```

D

Usability Test Results

This appendix contains the raw numerical results from the usability testing of the application.

Table D.1: The difficulty of the tasks in the usability test according to the participants.

Task\Participant	1	2	3	4	5	6	7	8	9
Task 1	2	1	1	1	2	1	4	4	1
Task 2	3	1	3	3	1.5	3	1	1	1
Task 3	4	2.5	4	2	1	4	3	3	3
Task 4	1	2	4	2	1	4	1	4	1
Task 5	1	1	1	1	1	2	2	1	1
Task 6	1	3	2	2	1	3	1	2	2
Task 7	1	1	1	1	1	1	1	1	1
Task 8	1	2	1	1	1	1	1	2	1
Task 9	1	2	3	1	3	3	2	2	1