A Web-Based Editor for Cloud-Based Programming

Jan Bracker

Master's Thesis
submitted in March 2014

Christian-Albrechts-Universität zu Kiel
Institut für Informatik
Arbeitsgruppe für Programmiersprachen und Übersetzerkonstruktion

Advised by: Prof. Dr. Michael Hanus

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel,

Abstract

Today's programmers and development tools still use concepts and workflows that were introduced with the early operating systems. Programs are still structured in files and to reuse code a developer still has to search and integrate appropriate libraries all by herself. One way to solve these problems is to structure source code using the semantic concepts a language offers and store them in a database to allow structured access and a more advanced search and support for reuse. These ideas lead to the concept of a code cloud. Last year a group of students at the Christian-Albrechts-University developed such a code cloud; they called it Claude. It supports simple Haskell code, up- and download of Haskell packages, and offers search functionality for the code. Though there are Haskell specific features the general structure of Claude is language independent and is supposed to fit as many languages as possible. This thesis extends Claude with a web-based editor to develop new packages in the code cloud and release them. The editor by design is also kept language independent, but offers integration of language specific features.

Contents

List of Abbreviations

ADT	Algebraic Data Type
AJAX	Asynchronous JavaScript and XML
API	Application Programming Interface
BSON	Binary JSON
CSS	Cascading Style Sheets
CRUD	Create, Read, Update and Delete
DOM	Document Object Model
DSL	Domain-Specific Language
GADT	Generalised Algebraic Data Type
GHC	Glasgow Haskell Compiler
GUI	Graphical User Interface
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
ID	Identifier
IDE	Integrated Development Environment
JSON	JavaScript Object Notation
MVC	Model View Controller
NoSQL	Not only SQL
URL	Uniform Resource Locator

List of Figures

List of Listings

Chapter 1

Introduction

Modern programming still uses concepts and workflows that were introduced through early operating systems. Programs are structured in files instead of using the abstractions offered by the underlying programming language. Especially in large software projects the number of files can be overwhelming. Developers can easily loose track of where they put a certain function or what the purpose of a specific file or module was.

Another regular problem when developing software is code reuse. It avoids errors and saves a considerable amount of time to use libraries already developed by others. But it is not always easy to find libraries that provide the functionality needed. Once found, a library has to be downloaded, compiled and integrated into a project and version management. This process is different for each programming language and, depending on the provided infrastructure, can be tedious.

A solution to these problems would be to actually structure software by the concepts offered in the used programming language. Instead of using files, a database would allow structured access and search functionality. Combining this approach with a web interface leads us to the idea of a code cloud. Such a code cloud is a central point that holds libraries and code from different developers and makes them easily accessible to other developers.

Last year a student project at the Christian-Albrechts-University developed such a code cloud. The project's goal was to develop a code cloud that is language independent and offers a way to store sources and documentation. There should also be a way to search the stored objects using different criteria. Sources in the cloud should be enriched with hypercode references that show which other objects in the cloud they rely and depend on. On top of everything a editor was supposed to enable developers to create new programs within the cloud allowing easy access to existing functionality.

The project group achieved to create a code cloud they called Claude [claude]. They concentrated on support for typical functional programming concepts: functions, data types, modules and packages. But it should not be hard to integrate further concepts from other languages into Claude. The code cloud supports importing and exporting Haskell packages using the Cabal package format [cabal-a]. One can also search within Claude to see which objects are available. Due to the limited amount of time it was not possible to develop an editor on top of Claude.

The goal of this thesis is to implement such an editor. As mentioned, it shall enable developers to write new packages on top of those already within a cloud instance. When finished, such a package may be released and reused in other packages of the cloud. The editor should work independent of the underlying cloud implementation. Following the spirit of Claude the editor is web-based and, by that, diminishes the need to install additional software on a users machine.

The remainder of this thesis will be structured as follows:

▻ First of all, Chapter 2 will familiarize us with the used technologies and concepts that form the basis of Claude. To write a web-based editor we need to use ECMAScript [ecma11] (also called JavaScript), as it is the language understood by most browsers. A short introduction will be given in Section 2.1. Haskell and Claude will be highlighted in Section 2.2 and 2.3.

▻ In Chapter 3 we talk about the implementation of the editor and the changes to Claude. The general structure of the editor and how it works with Claude is described in Section 3.1. The required changes to access Claude and its data model will be discussed in Section 3.2. After discussing the changes, we describe the editor's graphical user interface (GUI). Of course, it is not possible to fit all capabilities of each programming language into a general scheme, therefore, Section 3.4 explains how Claude and the editor handle language specific properties.

Section 3.5 describes the development interface, which is used to access and modify objects in Claude. The network communication between the editor and Claude is explained in Section 3.6.

▻ The next chapter discusses limitations and known problems of Claude and the editor. We will highlight limitations of the database system underlying Claude in Section 4.2 and 4.3. A discussion on the management of concurrent changes in the editor will follow up in Section 4.4. We will also present future work in this process.

▻ At last, we discuss related work in Chapter 5 and then conclude in Chapter 6.

Chapter 2

Foundations and Technologies

In order to build a large software system we have to reuse the work others already did. The code cloud project group made the decision to write Claude in Haskell. They used *MongoDB* [mongo] as database system to store information and provide the web-based interface using the Yesod Web Framework [yesod]. The editor also uses Yesod to provide its interface. Main parts of the editor's web-based user interface are written in JavaScript.

The following sections give a brief introduction to all of the mentioned technologies; but we still assume the reader has a basic familiarity with the HyperText Transfer Protocol (HTTP), HyperText Markup Language (HTML), Cascading Style Sheets (CSS), and the Document Object Model (DOM) web browsers use to represent the HTML document they are displaying. Though these technologies may not be essential to understand the contents of this thesis, they are important to comprehend the actual implementation.

2.1 JavaScript

JavaScript is a prototype-based programming language with duck typing [Hei07, p. 68] that offers features from object-oriented, functional and imperative languages. It is standardized as ECMAScript [ecma11] in version 5.1 at the time of writing. As JavaScript is supported across several popular browsers, [browsers] it is the standard script language to manipulate the browsers DOM and interact with the user.

2.1.1 JavaScript Object Notation

There are several primitive types built into JavaScript. Literals for strings, booleans and numbers use the notation known from most C-like languages. There are also inhomogeneous and associative arrays. Values of these primitive types can be written down directly using JavaScript Object Notation (JSON). An example of a possible JSON value for an author [Ada] is illustrated in the following listing:

```
{ "name" : 'Douglas␣Adams',
  'books': [ "The␣Hitchhiker's␣Guide␣to␣the␣Galaxy"
           , 'The␣Restaurant␣at␣the␣End␣of␣the␣Universe' ],
  age    : 49 }
```

We can see that an associative array (also called object or dictionary) in JavaScript is enclosed by curly brackets and each key value pair is separated by a colon. A key can be any string and does not have to be enclosed in quotes if it is a valid identifier. A numerically indexed array is enclosed in square brackets. Strings can be enclosed in single or double quotes. All of these values are handled as objects by JavaScript, which means they have a set of methods that can be called on the value.

2.1.2 Language Description

There is no predefined entry point for JavaScript. It is either executed when it first occurs in the document or when the browser fires an event that triggers it. A common way to execute JavaScript is to embed the code into HTML; this can be done through the `script` tag. It should contain a `type` attribute with the value `text/javascript`. The contents can be any valid JavaScript. It is also possible to store JavaScript in separate files, which is preferable for larger chunks of JavaScript, to reuse them across several pages. To do so, we leave the contents of the `script` tag empty and add the `src` attribute with the value set to the location of the JavaScript file.

Listing 2.1 displays an example of JavaScript. Line 1 and 2 of the listing show the declaration and definition of a variable. A declaration always begins with the keyword `var`. In Line 2 we can also see how the method

```
1 var str = "42,␣is␣the␣answer!";
2 var answer = str.substr(0,2);
3 if(answer == 42) {
4   str = "The␣answer␣is␣there!";
5 } else {
6   str = "No␣answers...";
7 }
```

Listing 2.1. Short example of JavaScript code

`substr` of the string object defined in Line 1 is called. The third line shows how the equality operator implicitly converts the string value in `answer` to a number to compare it with `42`. JavaScript always tries to perform implicit conversions for primitive types if they mismatch. Depending on the outcome of the comparison, a different destructive update on `str` is done. We can also see that the branching control structure looks the same as in other C-like languages. This is also the case for other control structures such as the switch-case or the while- and for-loop.

Functions are first class values in JavaScript. To define a function we use the `function` keyword. The following listing shows two ways to do this:

```
function fun(a,b) { return a + b; };
var fun = function(a,b) { return a + b; };
```

Both variants are equal in that the variant from the first line is translated to the variant in the second line by JavaScript. The second variant uses an anonymous (or lambda) function. We can see that defining a function is nothing else then declaring a variable and setting its value to a function object.

As mentioned earlier, JavaScript is a prototype-based language. First of all, objects in JavaScript are nothing more then associative arrays. That means, we can add methods and attributes to an object as we please. In Line 2 of Listing 2.1 we called the `substr` method of a string object. The method selection `str.substr` is nothing else then a short-cut for `str['substr']` and then calling the function that entry is associated with. For built-in types we cannot manipulate these objects, but for user-defined objects we can just overwrite a single method if we want to by setting the specific entry. If we create a function object and add a `prototype` entry to it, we can use that function as a constructor for new objects. To construct a new object we use the `new` keyword as in the following listing:

```
 1  var Pos = function() {};
 2  Pos.prototype = { x: 0, y: 0 };
 3  var p = new Pos();
 4  p.x;            // ⇝ 0
 5  p.x = 5;
 6  p.x;            // ⇝ 5
 7  Pos.prototype.x; // ⇝ 0
 8  p.y;            // ⇝ 0
 9  Pos.prototype.y = 42;
10  p.y;            // ⇝ 42
```

To lookup an attribute or method in an object created this way, we first look if the object has a matching entry. If this is not the case, we look into its prototype and follow the chain of prototypes until we either find the attribute or have to return `undefined`. JavaScript uses prototypes to represent advanced concepts such as inheritance, that are not part of the language itself.

In our example neither `x` nor `y` are entries of `Pos` objects. This means, the result values in Line 4, 8, and 10 come from the entries of the prototype for `Pos`. As we can see in Line 9 and 10, changing the prototype object affects all objects related to the prototype, if the value was not overridden beforehand.

2.1.3 Utility Libraries

The standard JavaScript API of most browsers is cumbersome to use, especially when performing DOM manipulation. Therefore, there are many JavaScript libraries that provide a more pragmatic interface to access these capabilities. We decided to use two libraries to make programming the editor user interface more convenient.

Also, these libraries guarantee that the written JavaScript behaves the same when executed in different browsers. Though many browsers claim to implement the standard, some may not fully support all details or there may be inaccuracies within the standard [browsers], that lead to different implementations.

jQuery: DOM Manipulation Library

The first library we choose is *jQuery* [jquery]. It is mainly a DOM manipulation library. The library introduces the dollar (`$`) function, which can be used to select and create new nodes in the DOM tree. It can also be used to access predefined objects through the *jQuery* interface. The result of the dollar function is an object that provides a versatile API to manipulate the DOM tree and manage events. A few examples can be seen in the following listing:

```
1 $(document).ready(function() { alert("DOM␣loaded"); });
2 $('.someCssClass').text('New␣content');
3 $('<div></div>').append($('#elementId'));
```

The first line wraps the global `document` object into a *jQuery* object and sets the callback for the event, that is triggered when the complete document is loaded. The second line uses a CSS selector to select all elements with the CSS class `someCssClass` and sets their text content to “New content”. Line 3 creates a new `div` element and appends it to the children of the element with ID `elementId`.

Another important capability that *jQuery* provides access to is asynchronous JavaScript and XML (AJAX). This allows a document to communicate with its server without reloading the complete page. The `ajax` method allows access to this functionality. An example call that would login to the code cloud is displayed in the following listing:

```
$.ajax({
  type: 'POST',
  url: 'http://codecloud.example/login',
  data: { username: 'test', pwd: 'test' },
  dataType: 'json',
  success: function(result) { /* ... */ },
  error: function(result, status, errorMsg) { /* ... */ },
  mimeType: 'application/json' });
```

We can see that the `ajax` method is configured by a JavaScript object. There are a variety of options that can be set for an AJAX request. These options are all listed in the *jQuery* documentation [jquery-a].

Lo-Dash: Functional Utility Library

Another library we use is *Lo-Dash* [lodash]. It introduces the underscore (_) object that provides methods to enable a more functional style of programming, e.g., functions to access or iterate over data uniformly across different types. Short examples of these functions are given in the following listing:

```
1 _.forEach("Text", function(ch) { /* do something */ });
2 _.map([1,2,3,4], function(val) { return val + 1; });
3 _.cloneDeep({ name: "Douglas␣Adams" });
```

The first line calls a function for each character of a text. `map` in Line 2 applies the given function to each element in the given collection and produces a new collection with the result values. Both `map` and `forEach` work on strings, arrays and objects in the same way. The call to `cloneDeep` in the third line creates a deep copy of the given value.

2.1.4 CodeMirror: Editor Component

There are many different text editor components for JavaScript available. The most mature ones we could find were *Ace* [ace] and *CodeMirror* [codemirror]. Both offer about the same set of features. They support standard text editing, syntax highlighting, marking of text, editing history and many more.

Though *Ace* seems more mature and modular when it comes to the API design, we decided to use *CodeMirror*. It offers marking text through actual DOM elements within the editor content, which is important for the way we implement some features. Additionally, it is easier to adapt the history behavior in *CodeMirror* to enable custom undo and redo operations.

A new editor component can be created using the `CodeMirror` function. It expects two arguments. First the element to insert the editor component into and second a JSON object to configure the instance with. The result is a `CodeMirror` object.

The `CodeMirror` object offers a variety of methods to control the behavior of the editor component. For us the document interface is especially useful. The document is responsible to manage the contents displayed by an editor instance. To access the document we need to call the method `getDoc`. The document object offers the `markText` method. It annotates a certain range of text in the editor with a configurable DOM object and by that allows us to link events with contents of the editor, or track the location of a text-mark while editing. `markText` accepts three arguments. The first two are the beginning and end of the text range to mark. The third is a configuration object for the mark.

To register event handlers in *CodeMirror* the `CodeMirror` object offers the `on` method. It takes the name of the event as first argument, and the event handler function as second argument.

Further information about the `CodeMirror` object and its functionality can be found in the manual that is linked on the *CodeMirror* homepage [codemirror].

2.2 Haskell

Haskell is a pure functional programming language. It uses a non-strict evaluation strategy and provides a strong static type system. Typical features of the functional paradigm such as pattern matching, anonymous functions, algebraic data types (ADTs) and type polymorphism are offered. It is standardized in the Haskell 2010 Language Report [Mar10].

We decided to use Haskell, because it was already used in Claude. By also using Haskell, we gain the advantage that we can reuse code and data structures that were already written for Claude.

All developed Haskell code for Claude and the editor was compiled using the Glasgow Haskell Compiler (GHC) in Version 7.4 and 7.6.

We assume that the reader is familiar with standard Haskell as defined by the 2010 Language Report. Some advanced language extensions will be explained here, since we used them to implement certain parts of the editor.

Generalised Algebraic Data Types

The major extension we use is Generalised Algebraic Data Types (GADTs). GADTs extend the data type definition syntax to look similar to that of type classes and allow to determine type parameters of polymorphic data types on construction. As an example, we can look at an `Exp` data type to represent simple expressions in the following listing:

```
data Exp a where
  IntVal  :: Int  → Exp Int
  BoolVal :: Bool → Exp Bool
  Plus       :: Exp Int  → Exp Int → Exp Int
  Equal      :: Eq a     ⇒ Exp a   → Exp a    → Exp Bool
  IfThenElse :: Exp Bool → Exp a   → Exp a    → Exp a
```

All of the constructors have specialized the type parameter of `Exp` in their return value. This enables us to give functions using a GADT more type safety. When pattern matching on one of these constructors, we can be sure that types are bound correctly in each case. Also note the `Eq` constraint on the `Equal` constructor. Setting such a constraint on types involved in a constructor would normally not be possible at this level.

As an example, we look at the `eval` function in the following listing to see how we benefit from this extension:

```
eval :: Exp a → a
eval (IntVal n)  = n
eval (BoolVal b) = b
eval (Plus n m)  = eval n + eval m
eval (Equal a b) = eval a == eval b
eval (IfThenElse b t e) = if eval b then eval t else eval e
```

Note that `eval` is polymorphic over `a`. Though this is the case we can still return a value of the concrete type `Int` or `Bool` in second and third line. This would not work with a normal ADT. Due to the `Eq` constraint on `Equal` we can compare the results of both evaluations in line five, although our polymorphic type `a` does not have to be comparable in general. An extensive use of a GADT can be seen in Section 3.6.2.

Overloaded Strings

The overloaded strings extension generalizes the way string literals are handled by GHC. It has the name `OverloadedStrings`. When activated the `fromString` function from the type class `IsString` is used to convert string literals to the type currently needed.

```
class IsString a where
  fromString :: String → a
```

This behavior is similar to that of numeric literals where the `fromInteger` or `fromRational` functions are used to convert them into the needed type.

This extension is especially useful since we often use the `Text` data type from the package *text* [OSu], as it is a more efficient representation of strings.

2.2.1 http-client: HTTP Network Protocol API

We use HTTP to communicate with Claude. To access the protocol in Haskell we decide to use the *http-client* library [Sno13]. We choose this specific library, because our web framework (Section 2.2.3) also uses it to communicate over HTTP.

The most important types introduced through *http-client* are `Request` and `Response`. A `Request` is a record that allows to set the HTTP request method, host, port, path, headers, cookies and the body. It can be sent using the `httpLbs` function.

```
httpLbs :: Request → Manager → IO (Response ByteString)
```

Sending a `Request` either results in an `IO`-based exception or the received `Response` data is returned. The `Response` type parameter specifies the contents of the response body, which in the most general case is just a sequence of bytes. The `Manager` provides the network connection to use for the request.

As an example we can see a simple GET request in the following listing:

```
main = do
  req <- parseUrl "http://www.loremipsum.de/downloads/original.txt"
  mng <- newManager defaultManagerSettings
  rsp <- httpLbs (req {method = "GET"}) mng
  print (responseBody rsp)
    ⇝ Lorem ipsum dolor sit amet, ...
```

The request is sent to `http://www.loremipsum.de/downloads/original.txt` and prints the result body. In the second line we create the request value from an URL that is given as a string. Next we create a new manager that acquires a network connection for us and then we send the request. Before sending it, we make sure it actually is a GET request. At last we print the body of the response we received.

2.2.2 aeson and bson: Data Serialization

Most AJAX data sent over network by the editor is encoded in JSON. The Haskell library *aeson* [OSu13] is used to do this serialization. For this purpose it provides the two type classes `FromJSON` and `ToJSON`:

```
class FromJSON a where
  parseJSON :: Value → Parser a

class ToJSON a where
  toJSON :: a → Value
```

The `FromJSON` type class is responsible for decoding a JSON encoded value. It uses a `Parser` to provide appropriate errors if decoding fails for some reason. The `ToJSON` type class encodes a value into JSON.

The framework offers a set of utility functions to use the type classes:

```
decode :: FromJSON a ⇒ ByteString → Maybe a
encode :: ToJSON a   ⇒ a          → ByteString

eitherDecode :: FromJSON a ⇒ ByteString → Either String a
```

These functions use `ByteStrings`, that can be sent over the network directly.

The document-based database Claude uses as backend stores documents in a format called Binary JSON (BSON) [bson]. This format is similar to standard JSON, but extends it with support for some other data types and binary data. To convert between the Haskell data types and their BSON representation we have to implement the `Val` type class:

```
class (Typeable a, Show a, Eq a) ⇒ Val a where
    val   :: a     → Value
    cast' :: Value → Maybe a
```

The `val` function creates a BSON `Value` and the `cast'` function decodes it again.

2.2.3 Yesod

Yesod describes itself as "[...] a Haskell web framework for productive development of type-safe, RESTful, high performance web applications" [yesod]. It is based on the model view controller (MVC) pattern. *Yesod* is not only a framework to develop web applications with, but it also offers the `yesod` command [Snoc], which allows to generate basic project infrastructure and run web applications during development.

The model usually consists of a database layer. *Yesod* prefers to use the *persistent* [Snoa] package for this purpose. But the Claude project decided to use another database directly instead of using the abstraction layer provided by *persistent*. We will give more details about this in Section 2.3.

The following subsections will provide an overview of how the view and controller part in *Yesod* are realized. They will also give insight to the general structure of a *Yesod* project as it is produced by the `yesod init` command.

Routing and Type-safe URLs

The reachable paths of a web application in *Yesod* are specified through routes. Routes are specified in a central route file. A possible route consists of one line. That line begins with the route itself followed by a valid Haskell data type constructor name and the HTTP methods it supports to handle. All three parts are separated by whitespace characters. As an example, if we want to display user information depending on the user's name, a route may look as follows:

```
/user/#String UserR GET
```

We can see that the name of the user we want to see can be encoded as a parameter within the route. The `#String` tells *Yesod* that the route `/user/` may be followed by something that can be decoded as a Haskell `String`. Such a decoded value can be passed to the handler or controller of that route. The example route only handles GET requests.

Yesod generates a data type for routes or URLs of the application. Each route is represented by one constructor with the name given in the second entry of the route. The `String` encoded parameter of our example is one entry of the constructor `UserR` in that data type. These constructors enable us to view links to routes in the application as an instance of a constructor in this data type. We can use this type to specify links in a type-safe manner.

Once a user tries to access a certain route *Yesod* calls an appropriate handler (controller). A handler is represented as a function with a standard naming scheme. Each handler begins with the methods of the HTTP request it handles and ends with the name of the associated URL constructor. As an example

```
getUserR :: String → Handler Html
```

would handle requests sent to `/user/#String`, because that entry only handles GET requests and the associated URL constructor is `UserR`. We can see that the route parameters are arguments to the handler and a handler always has to operate in the `Handler` monad. Additionally, a handler needs to return a response to the request. In this case we respond with some HTML. *Yesod* automatically selects the right content type for the returned content and sets up the correct HTTP headers.

Standard Project Structure

Yesod offers the possibility to generate infrastructure for a project. We choose to use this generated infrastructure. It provides the following directories and files:

`config/routes` The `routes` file is responsible to set up the reachable paths of our web application. It contains all routes available.

`Handler/` This directory contains Haskell modules that provide the route handler functions.

`templates/` The `templates` directory provides template files that can be used within the controllers to produce the view. The next section explains the structure of templates and how they can be used.

`Application.hs` This file is generated by *Yesod* and links everything together. We usually only have to modify it to add new modules, that provide handlers, to the list of imports.

`Foundation.hs` The foundations are responsible for reading our configuration and implementing the type classes *Yesod* needs to run our application. Here we can fine tune and configure the mechanisms *Yesod* uses to provide our application.

`static/` This folder is used to provide static files that do not change while the application is running. The *Yesod* binary automatically creates a route for this folder in the `config/routes` file.

Other source directories All other directories that could represent a module can be source directories for further Haskell sources. We would just have to configure the project's cabal file correctly. Claude and the editor use the `Claude` directory to store all Haskell sources aside from the handlers and other *Yesod* generated sources.

Template System

Yesod uses the *Shakespeare*an template system [Snob], which provides three Domain-Specific Languages (DSLs) to generate HTML, CSS and JavaScript. These templates have the file extensions `hamlet`, `lucius` and `julius`. The DSLs are custom tailored languages that are translated to Haskell during compilation using *Template Haskell* [ghc-th; SJ02]. To use a certain template we can use the command `$(widgetFile "myTemplate")`, where `"myTemplate"` gives the name of the template without extension within the `templates` directory. A matching `lucius` and `julius` file is automatically also loaded and correctly embedded within the page.

All template formats offer the `#{expr}` syntax to embed the Haskell expression `expr` into the template contents. They also offer `@{routeConstr}` to embed a link to an application page specified by the route constructor `routeConstr`. The scope of available identifiers in such a template is equal to the scope at which it is embedded within the handler. In other words, a template can only be used at different locations if the same identifiers with the same respect type are in scope.

Within the templates for CSS and JavaScript we can write normal CSS and JavaScript with the addition to embed Haskell expressions and links into the template contents. Templates for HTML use syntax that differs from actual HTML. They use the offside rule to nest HTML elements and remove closing tags unless the offside rule is not applicable, i.e., the closing tag is in the same line as the opening tag. To clarify, we look at a small example:

```
1 <html>
2   <head>
3     <title>A HTML Page</title>
4   <body>
5     <p>
6       #{someText} - Visit the
7       <a href=@{UserR "Jan␣Bracker"}>author</a>.
```

We can see how the different elements are nested by the offside rule. The closing tag of the `title` element in Line 3 is optional, while the closing tag of the `a` element in Line 6 is required. Line 6 shows how the Haskell value of `someText` is embedded within the template. Note that the link to the author's page in Line 7 uses a type-safe URL.

Hamlet templates also offer control structures to express pattern matching, branches, loops over lists and other constructs. To clarify, we look at the short example of a branch:

```
$if null myList
  <p>No entries!
$else
  <p>List has #{length myList} entries.
```

The example shows how the case that a given list `myList` is empty can be handled specially. All control structures in Hamlet templates begin with a dollar sign (`$`) and use the offside rule for nesting. More elusive examples can be found in the official *Yesod* book [Sno12].

2.3 Claude: Code Cloud

The Claude project group [claude] aimed to provide a language independent code cloud to store source code and documentation of software. The semantic objects represented by the code is stored in a structured way instead of using the language syntax. Of course, this is only possible up to a certain level.

2.3.1 Data Model

As a first approximation, Claude supports objects typical for functional languages such as Haskell. It distinguishes between packages, modules, functions and data types. When talking about an object in Claude, we refer to either of these.

By default Claude does not allow objects to be modified. The simple reason for this is, that a modification may break the dependencies that exist between different objects in the database. To give an example: If a function in some package uses a data type in another package, a modification of that data type may render the function broken. It would be even worse, if we

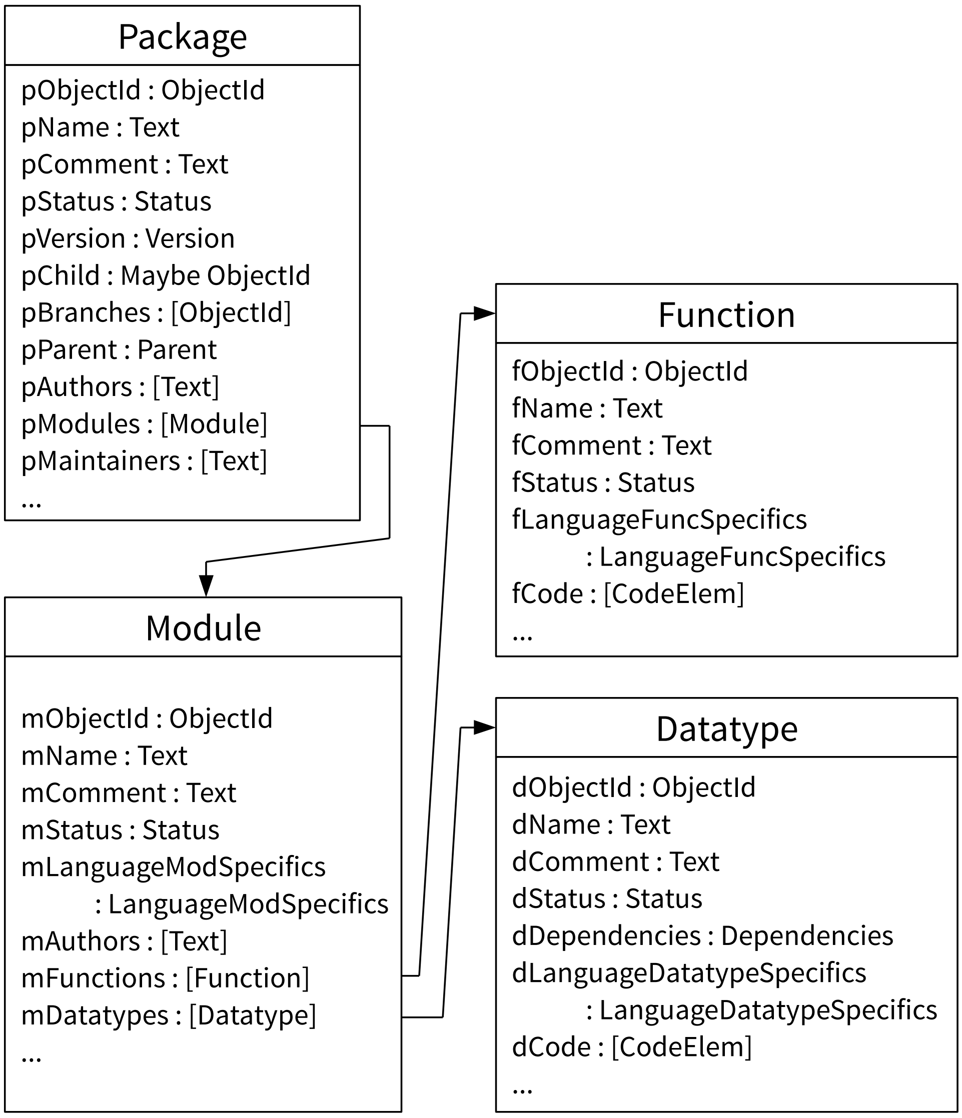


Figure 2.1. Claude's data model

deleted the data type; that would not only break the function referring to it, the deletion would also invalidate the dependency of that function and lead to an inconsistent state.

Figure 2.1 gives an overview of the data types representing the objects. We can see that the types are organized hierarchically. A `Package` contains several `Modules` and a `Module` contains several `Functions` and `Datatypes`. These Haskell data types are implemented as records. They contain fields for meta data, e.g., an object's name, comment or authors. This enables us to access meta data in a structured way. Note that the object records are not absolutely custom tailored for Haskell. They should fit as many functional languages as possible. Specialties of certain languages are supposed to be stored in the language specific fields.

Each object has a unique `ObjectId` that identifies it within the database and allows direct access to that specific object. When talking about a dependency hereafter, we refer to the reference of such an `ObjectId`. We sometimes separate between internal and external `ObjectIds`. A dependency is internal if the referring object is in the same package as the referred object. It is external if the referring and referred object are in different packages.

All objects have a `Status`. This meta information represents the stability or maturity of a given object. Claude currently offers a small set of different states:

```
data Status = Alpha | Beta | Default | Deprecated
```

Listing 2.2. Definition of the `Status` data type

Though the status normally is not reflected by a programming language, we annotate objects with it to give the programmer more information about the objects he is working with. We also use the status to distinguish between code in development and released code (Section 3.2.2).

The version field gives the version of a package. The version number format corresponds to Haskell's package version format. This reflects in the type, which is just a wrapper around a list of numbers:

```
newtype Version = Version [Int]
```

Relationships between packages are modeled using the field `pChild`, `pBranches` and `pParent`. `pParent` is used to model the relationship to a parent package. The `Parent` (Listing 2.3) data type offers three constructors. If the `Parent` is `New`, a package is independent from other packages. The

```
data Parent = ChildOf ObjectId
            | BranchOf ObjectId
            | New
```

Listing 2.3. Definition of the `Parent` data type

`ChildOf` constructor references the parent package of a version update, i.e., if the current package is a version update of some package, it is the child of that package. A package referred to from `ChildOf` contains the child package identifier (ID) in the `pChild` field. There can only be one child of a package. Thus, a hierarchy of child and parent packages is always linear. It is also possible to branch a package. A branch is an experimental or development version of a package. A branch sets its parent to `BranchOf`. All branches of a package are listed in `pBranches`. We add the possibility to set a name for a branch. We take a closer look to this change in Section 3.2.2.

Certain features of Haskell are not yet representable, e.g., type classes are still missing. The same is true for non-functional features that are typical for object-oriented or logical languages. But the data model should be easy to extend with new objects for this kind of features.

The only parts of a program that remain in their original syntactic form is the source code of a function or a data type. Source code stored inside the cloud is enriched with hypercode references that show what a certain syntactic object in the code refers to and depends on. Enriched source code is represented as a list of `CodeElem`s:

```
data CodeElem = CodeText Text
              | CodeRef ObjectId Name (Maybe Qualifier) Text
              | CodeBaseRef Name Qualifier Text
```

Listing 2.4. Definition of the `CodeElem` data type used to represent hypercode references

The `CodeText` constructor just marks regular source code without any annotations. A reference to another object in the cloud is given by a `CodeRef` object. It contains an `ObjectId` that identifies the other object, the `Name` of the object it refers to, the optional `Qualifier` and the `Text` actually representing this reference in the original source code. Note that `Name` and `Qualifier` are just synonyms for `Text`. As an example, if we assume `objId` refers to `Data.Maybe.fromJust`, then a call of `Data.Maybe.fromJust` would be annotated the following way:

```
CodeRef objId "fromJust" (Just "Data.Maybe") "Data.Maybe.fromJust"
```

The last constructor `CodeBaseRef` represents a reference to an object that is predefined in the current language, e.g., the objects from the module `Prelude` in Haskell.

Currently, the language specifics only support Haskell specific features. The different language specifics are shown in Listing 2.5.

```
data LanguageModSpecifics =
    HaskellModuleSpecifics [Pragma]

data LanguageFuncSpecifics =
    HaskellFuncSpecifics { fPragmas  :: [Pragma]
                         , fType     :: [CodeElem]
                         , fDataType :: Maybe ObjectId }

data LanguageDatatypeSpecifics =
    HaskellDatatypeSpecifics [Pragma] [ObjectId]
```

Listing 2.5. Definitions of the language specific data types

We can see that a module only gets the Haskell specific pragmatics as annotation. The type `Pragma` is just a synonym for `Text` right now, but this may change in the future. Functions get more Haskell specific information. Aside of pragmatics they can also be annotated with their type signature (`fType`). The type signature can contain hypercode references to the used types. In case the function actually is a constructor it may contain the ID to the data type it is from in the `fDataType` field. Data types again contain pragmatics and a list of IDs to their constructors. Constructors are inserted into the database as additional objects to allow searching for them.

Function and data type objects also contain an entry for dependencies. `Dependencies` is just a container that collects the IDs of function and data type objects the respective object refers to in its sources:

```
data Dependencies = Dependencies
    { funDeps :: [ObjectId], dtDeps  :: [ObjectId] }
```

The dependency data type is there to decide if a hypercode reference, in the source code, refers to a function or data type.

Packages also contain a list of maintainers. They show who uploaded the package and is responsible for it. Aside of that, the editor uses them to control who is allowed to modify a package and its contents (Section 3.5).

2.3.2 Database

The Claude project group decided to use a document-oriented database. They used *MongoDB* [mongo]. There are several reasons for this decision. First of all, *MongoDB* does not require a predefined schema for the held data. This gives flexibility when changing or extending a representation, because a field can be added or removed on demand, without putting the complete data set into jeopardy. Another reason was that the project group wanted to see how well a NoSQL database meets their requirements and how mature *MongoDB* in particular is. As a consequence Claude does not use the *persistent* library [Snoa] *Yesod* usually advises as abstraction layer of the database.

Claude uses the *MongoDB* Haskell bindings provided by the *mongoDB* package [Han13] to access the database. This interface is generic and does not offer type-safety while using it. All documents and queries in *MongoDB* are expressed as BSON values. Therefore, every data type of the model implements the `Val` type class from Section 2.2.2, which enables storing them in the database. The Claude project group wrote a wrapper that offers a type-safe interface to manipulate the data managed by Claude.

An abstraction over the *MongoDB* query language is provided by Claude. The main abstraction is the `Query` data type that can be seen in Listing 2.6.

```
data Query a = Any           | NotSet
             | Equal a       | NotEqual a
             | ContainsAll a
```

Listing 2.6. Definition of the `Query` data type

The constructor Any puts no constraint on the a field. NotSet requires the field not to be set in the database record. The queries Equal and NotEqual check if the respective value is equal or not equal to the given value. We can use ContainsAll, if the queried field contains a list; it checks, if the queried entry list contains all entries of the list given to ContainsAll.

Querys are used to create query data types for each of the objects stored in the database. Listing 2.7 displays an example for such a query data type.

```haskell
data QModule = QModule
  { mObjectId :: Query ObjectId
  , mName     :: Query Text
  , mComment  :: Query Text
  {- ... -}
  , mFunctions :: QFunction }
```

Listing 2.7. Excerpt of the query data type for Module

We can see that the fields match the entries of the module object, except that the Query data type is applied to them. If we want to search for an object, we just fill the entries of our query data type and pass it to the appropriate interface function. Note, that the mFunctions entry in Line 6 has the type of the query data type for functions QFunction; this is used to query a module that contains a function matching the query of the QFunction value.

To manipulate the contents of the database we use the Modification data type:

```haskell
data Modification query modifier = Modification query modifier
```

Basically, it is a tuple that pairs a query and a modifier data type together. The query then selects the value that needs to be modified and the modifier says how they shall be modified. The modifier data type has the same structure as the respective query data type, but instead applies the Modifier type from the following listing to each field:

```haskell
data Modifier a = Unchanged
                | Set a             | Unset
                | Add a (Maybe Int) | Remove a (Maybe Int)
```

If the entry is set to `Unchanged` it remains unchanged. `Set` and `Unset` replace, add or remove the field from the document. If the field contains a list `Add` and `Remove` can be used to add or remove an entry of that list. The optional `Int` parameter is needed for nested lists and specifies in which sublist to add or remove the given value, if set.

The `Claude.Server.DataInterface` module provides a variety of functions to access the database using queries and modifications. We will not go into further detail on these functions, since we will not use them directly.

Claude also provides a way to apply a full-text search to the contents of the cloud; this full-text search uses an index that *MongoDB* supplies. Using the search facility, we can also find objects that contain a search term in one of their meta information fields, e.g., in their comments or one of their authors.

There are some limitations and problems with *MongoDB* that we will discuss in Section 4.2 and 4.3.

2.3.3 Authentication

To authenticate as a user we have to login using a user name and a password. When logging in, Claude generates a unique token that is associated with the current user in the database. Then this token is saved in an encrypted cookie of the users browser. Once the user accesses Claude again, the token can be read from the cookie and Claude can find the associated user information in the database. Claude originally supported only one token associated with a single user, which prevents her from using different devices at the same time. A token is only valid for a specific amount of time to prevent a user from staying logged in forever. Section 3.2.1 explains all changes made to the authentication process of Claude.

Chapter 3

Implemenation

In this chapter we look at the implementation of the editor. Therefore, we talk about the application structure in Section 3.1 and proceed with the changes required in Claude (Section 3.2). After the foundations, Section 3.3 introduces the user interface. The last three sections look at specific aspects of the implementation. Section 3.4 highlights language specific behavior. After that, we introduce the development interface that is used to communicate with Claude in Section 3.5 and close by looking at the specifics of network communication in Section 3.6.

3.1 Application Structure

When designing the application's general structure, we made the decision to separate the editor from Claude. This allows exchanging either the editor or Claude afterwards in case there is a radical change to one of these components. Thus, the editor can be reused if Claude is rewritten at some point or if a completely different architecture is developed.

The different layers resulting from this decision can be seen in Figure 3.1. To separate the editor and Claude from each other we wrote a development

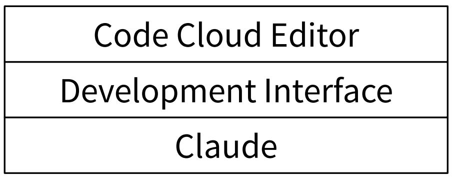


Figure 3.1. Logical structure of Claude and the editor

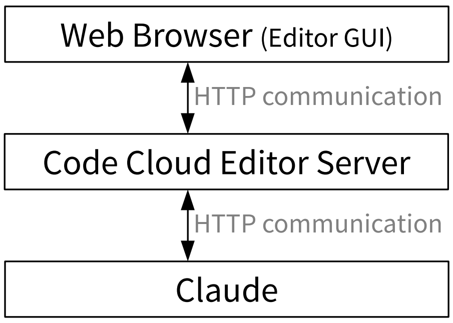


Figure 3.2. General communication structure of Claude and the editor

interface that the editor uses to communicate with Claude. All data and requests are sent through this interface. We will take a closer look at the different operations the interface provides in Section 3.5.

We also do not want to replicate the data management already provided by Claude. Therefore, the editor does not store any persistent data; this reflects in the development interface. Not storing data in the editor also avoids the need to manage inconsistencies between editor and Claude data, though this problem reoccurs with the editor GUI as we will discuss in Section 4.4. Nevertheless, the editor does need to save session information of the current user to log in to Claude. We use cookies to save the session data in the user's web browser. Section 3.5 and 3.5.1 will give an overview of the authentication process.

When running the editor, we typically have three instances communicating with each other using HTTP, which are shown in Figure 3.2. First of all, we have the web browser that displays the GUI to the user. The GUI is initially provided by the editor server. Once the GUI is deployed, it requests information from and sends updates to the editor server via HTTP. The editor server decodes these requests and uses the development interface to send them to Claude. After processing the request, Claude sends an answer back as response. We will highlight further details of this process in Section 3.6.

The described architecture decouples the editor from a specific Claude instance such that several editors can work on one cloud or one editor can work with different instances of Claude.

3.2 Changes to the Code Cloud

Claude was structured as one monolithic package when we began to develop the editor. Of course, we do not want the editor to depend on the Claude package `code-cloud`, because then each instance of an editor would be required to install the complete code cloud. In consequence we restructured the package and created several other packages as can be seen in the dependency graph of Figure 3.3

First of all, we move the shared structures and logic to a separate package we call `code-cloud-data`. The package does not only contain shared data structures; it also provides the code to serialize and deserialize those data structures into JSON or BSON. Serialization is important for network communication (JSON) and to enable storing the data structures in *MongoDB* (BSON). The serialization code is thoroughly tested using *QuickCheck* [CBS13] tests to give confidence that it works correctly. Aside of serialization the package also provides network infrastructure that is used in all other packages of the project. We highlight the provided infrastructure in Section 3.6.2.

The development interface is in the additional package `code-cloud-interface`. This division allows different applications to also use the interface for development access to the cloud and enables exchanging the

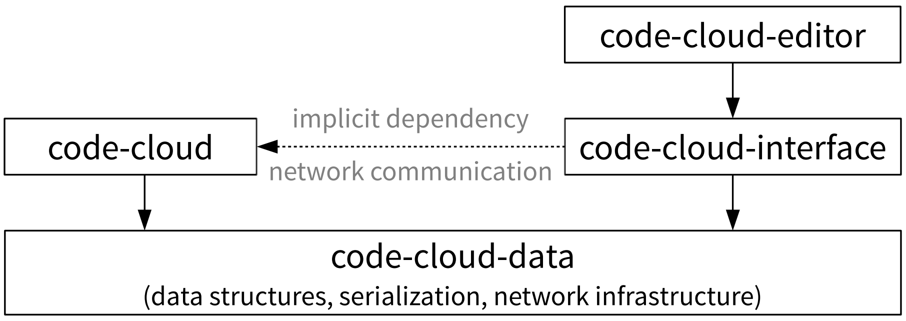


Figure 3.3. Package dependency graph of Claude and the editor

editor without touching other components. The interface package currently only depends on `code-cloud-data` because all communication with Claude is done over network. If required, the interface package can be exchanged with an alternate implementation. Details of the development interface are discussed in Section 3.5.

Last but not least, the editor is provided through the package `code-cloud-editor`. This package only depends on the interface and the data package, but not on `code-cloud` due to the reasons mentioned above.

3.2.1 Authentication

Originally every time a user logged in to Claude, exactly one token was generated and associated with that user to identify him by his cookie. This approach works fine, if a user only uses a single device and browser to access Claude, but it leads to problems once there are several different devices or browsers. If another device logs in, it would replace the token associated with the user, and thus, log out any other device currently logged in. We need the ability to be logged in from two different places at the same time, e.g., a user may access Claude and the editor simultaneously.

To allow several different devices to be logged in with the same user account, we extend the original mechanism. We now allow multiple tokens to be associated with a single user. Each time a new device logs in, a new token is generated for that device and associated with the user that logged in. This way each device can log in and out independently. In case a user does not log out properly, the list of associated tokens is cleaned up each time it is accessed. The cleanup uses time stamps, that are associated with each token and limit its validity.

During the time of writing, we noticed that it may be possible to simplify our enhancement by using only one token per user. Instead of generating a new token each time a user logs in to Claude, we could just lookup an already existing token and hand it out to the new device. A log out could be realized by simply deleting the cookie on client side. This would also guarantee that a user is automatically logged out after a certain amount of time, because of the cookie's time stamp. The only advantage, we see in our approach, is that the server ensures a device is logged out after a certain amount of time, and we do not have to trust the browser to invalidate its cookies correctly. We decided not to change the system again, as it works as is.

3.2.2 Data Model

Claude's data model, as presented in Section 2.3.1, needs to be altered to fit the purposes of the editor.

First of all, we extend the `Parent` data type from Listing 2.3. We add the ability to name branches by adding a `Text` field to the `BranchOf` constructor. This makes sense in context of the editor, since there may be many different development branches of a package at the same time. Without human readable names for each branch it would be hard for a developer to distinguish them from each other. Therefore, we ensure that each branch of a package needs to have a unique name among all other branches of that package. A valid branch name consists of an alphanumeric character at the beginning followed by dash (`-`), underscore (`_`) or other alphanumeric characters. The same restrictions now also apply for package names.

Another important change is to add the `Development` status to the `Status` data type from Listing 2.2. The `Development` status is used to mark objects, which may be modified and changed. When searching for objects in Claude, development objects are ignored to restrict people from referring to code that may change in future. More details about the general requirements to allow modifications of objects in Claude, when using the editor, are given in Section 3.5.

3.2.3 Database Access

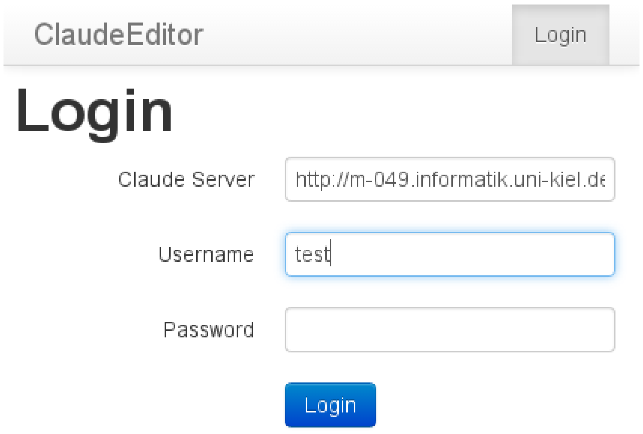
Functions to access the database within Claude were collected in the `Claude.Server.Interface` and `Claude.Server.DataInterface` modules. We refer to these modules as `Interface` and `DataInterface` from now on. The `Interface` module contains low-level database access functions, while the `DataInterface` module offers a more high-level and restricted access.

First, we ensure that all database access in Claude is done through the `DataInterface` functions; there were several places that bypassed this module and used functions from `Interface` directly. Since `DataInterface` contained many functions that only differed by the call of utility functions we remove these duplicates. The interface was still large afterwards, so we created two further modules, `Claude.Server.UserInterface` and `Claude.Server.DevelopmentInterface`. The `UserInterface` contains all database functions related to user data. The `DevelopmentInterface` provides all functions that allow modification of database objects. Note that the `DevelopmentInterface` module does not provide the development interface used by the editor, though that interface mostly relies on functions from the module.

Secondly, we needed to filter objects in development status from Claude's search results, because it should not be possible to reuse a development package until it is released. We need to ignore objects in development status, because the search is used to suggest annotations while developing code. Therefore, development objects would just pollute the search results and not deliver useful information to a user. To do so we had to add the `NotEqual` constructor to the `Query` data from Listing 2.6. Only through this query we can select packages that are not in development status.

Thirdly, we had to implement a simple transaction manager for database queries, because at some point we could not express our queries in one step anymore. This transaction manager is only used for write access, not for read access. It has many limitations and is a work around for the missing capabilities of *MongoDB*. We will give a more detailed explanation of the used techniques and the limitations in Section 4.2.

The last set of changes was adding a HTTP-based JSON interface to the modification functions given in `DevelopmentInterface`. It is accessed by the development interface, we introduce in Section 3.5, and uses the capabilities of the automated JSON transfer infrastructure we developed (Section 3.6.2).



http://m-049.informatik.uni-kiel.de

test

Figure 3.4. Login screen of the editor

3.3 Graphical User Interface

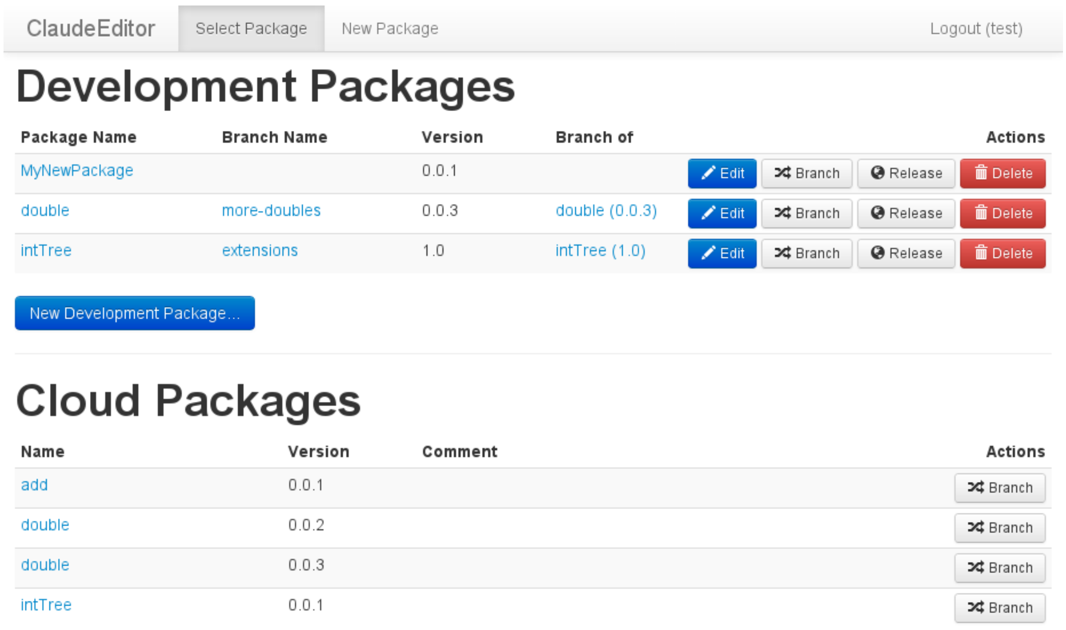
The editor's GUI mainly consists of two parts: the actual editor and the package management.

Before either of these are visible to a user, she has to log in (Figure 3.4). To do so she has to specify the Claude server to work on and credentials of a valid user account on that server.

Every view of the editor displays the main navigation bar at the top of the view port, as seen in Figure 3.4 and 3.5. If the user is not logged in, the navigation bar only contains a link to the login page. Otherwise, there are links to manage the packages, create a new package, or log out.

3.3.1 Package Management

The package management is the landing page after the login and gives an overview of all packages; it separates between those packages that are in development status and those that are already released in Claude. Figure 3.5 shows the package management page. Only packages that are maintained by the currently logged in user are displayed. A user has the possibility to execute the following operations from the package management page:


ClaudeEditor | Select Package | New Package | Logout (test)

Development Packages

Package Name	Branch Name	Version	Branch of	Actions
MyNewPackage		0.0.1		Edit Branch Release Delete
double	more-doubles	0.0.3	double (0.0.3)	Edit Branch Release Delete
intTree	extensions	1.0	intTree (1.0)	Edit Branch Release Delete

New Development Package...

Cloud Packages

Name	Version	Comment	Actions
add	0.0.1		Branch
double	0.0.2		Branch
double	0.0.3		Branch
intTree	0.0.1		Branch

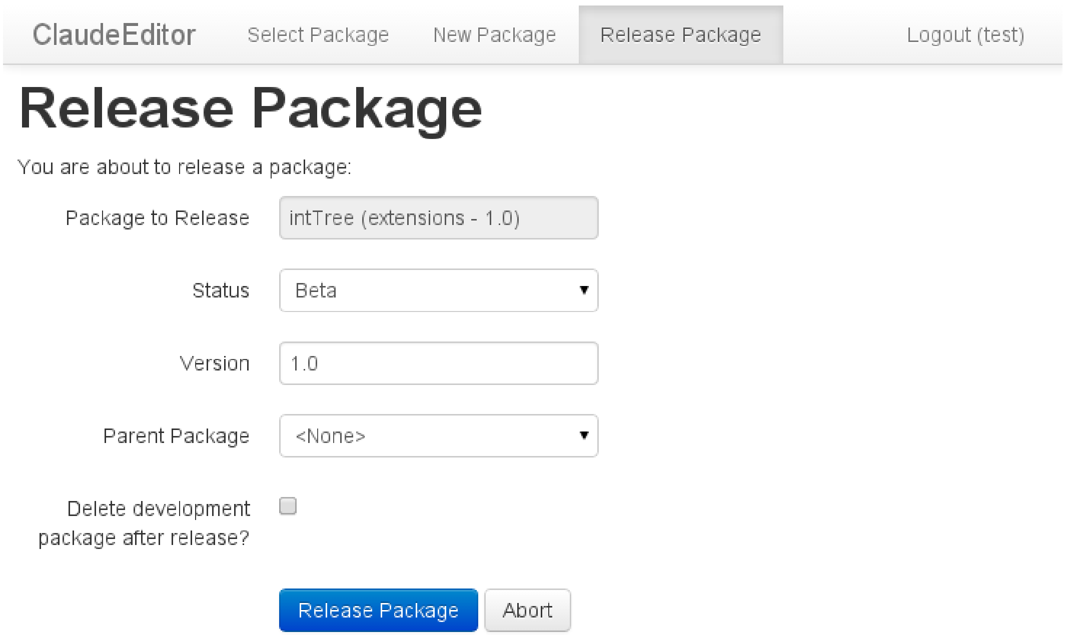
Figure 3.5. Package management landing page of the editor

"Edit" Opens the editor view for the selected development package. The editor view is presented in Section 3.3.2.

"Release" Opens a form to release the selected development package in Claude. The form can be seen in Figure 3.6. We can set the release status, the new version and the parent package in case we want to release the development package as a new version of that package. Since we are ready to release the development package, we are also offered to delete it after the release.

"Branch" Opens a form to create a new development branch of the selected package. The user can supply the branch name. This form looks similar to the form in Figure 3.6.

"Delete" Deletes the selected development package. As this operation is irreversible, there is a security question to ensure that the user really wants to delete the package.



Release Package

You are about to release a package:

Package to Release	intTree (extensions - 1.0)
Status	Beta
Version	1.0
Parent Package	<None>
Delete development package after release?	☐

Figure 3.6. Form to release a package in the editor.

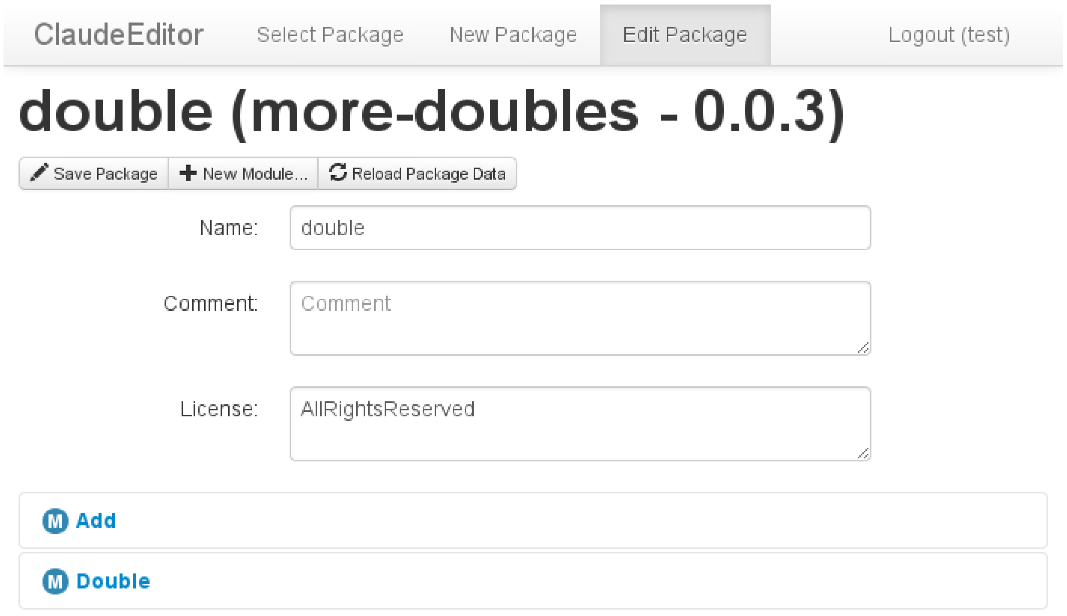
"New Development Package..." Opens a form to create a new empty development package. The form requires to supply a name for the new package. This functionality can also be reached using the entry "New Package" in the navigation bar.

3.3.2 Editor View

When the user decides to edit a development package, by pressing the "Edit" button, he is directed to the editor view, which is displayed in Figure 3.7.

When inside of the editor view, the navigation bar contains an additional entry that leads to the current view. This is mainly there to provide orientation for the user, but can also be used to reload the editor view or open it in another tab.

The editor view is structured hierarchically. At top-level we can modify the package we are working with. First we see a toolbar with operations,



double (more-doubles - 0.0.3)

Name:	double
Comment:	Comment
License:	AllRightsReserved

Figure 3.7. Package-level editor view

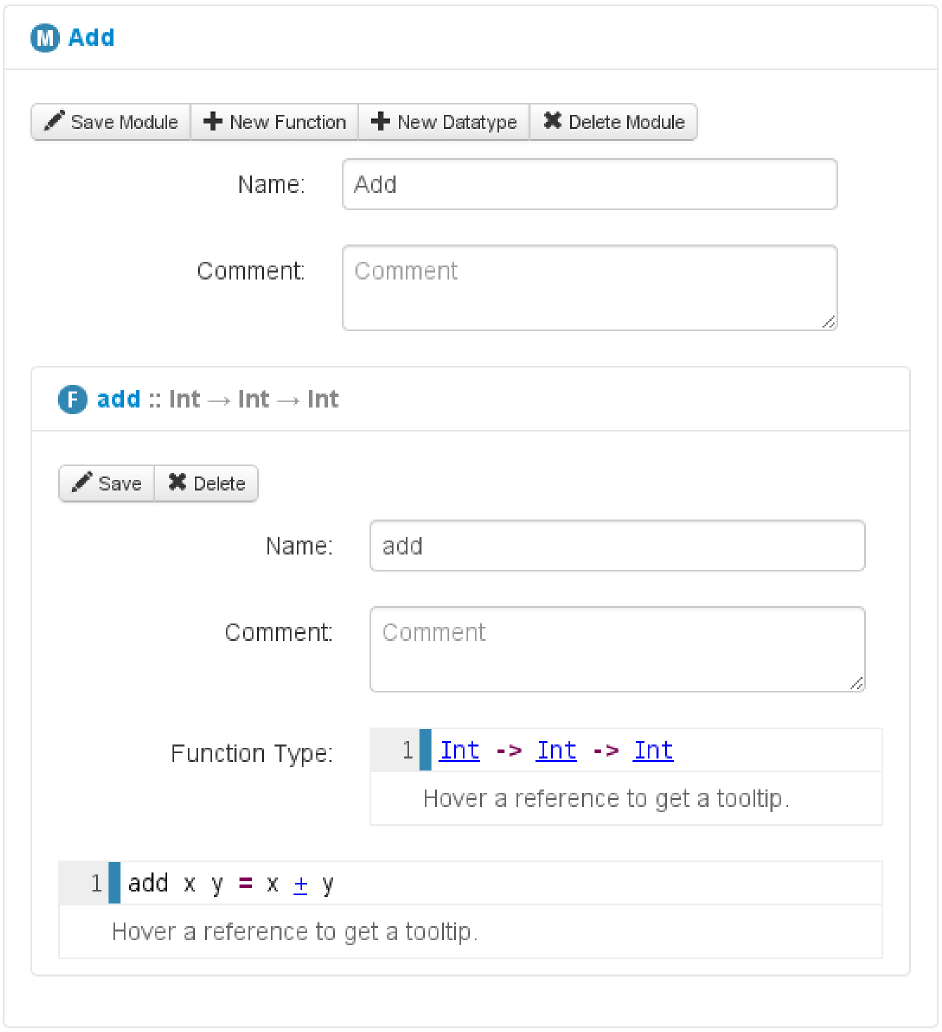
that are available at package-level. The toolbar is followed by form fields to modify package meta information. At last, we see the list of modules contained in the package. Clicking on a module unfolds an editor for that module. The following operations are available at package-level:

"Save Package" Saving a package, hierarchically, saves all changes in the editor; this also includes the modules, functions and data types.

"New Module..." Unfolds an editor to create a new module. More information on this operation will be given in Section 3.3.3.

"Reload Package Data" This operation reloads all information in the editor, i.e., every change or old data is overwritten with the most current package data from Claude.

The unfolded module-level editor is structured similar to the package-level editor, as illustrated in Figure 3.8. There is a toolbar with operations



Add

add

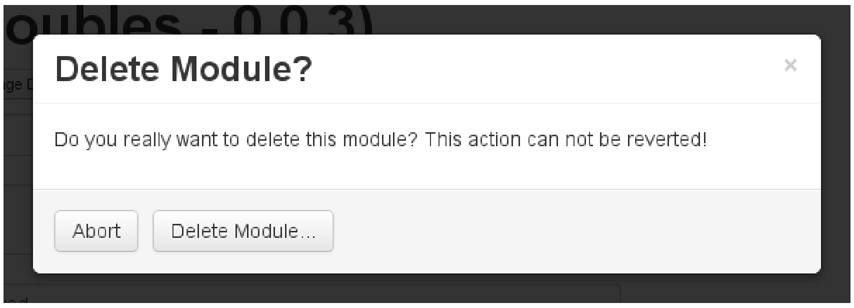
```
Int -> Int -> Int
```

Hover a reference to get a tooltip.

```
add x y = x + y
```

Hover a reference to get a tooltip.

Figure 3.8. Module- and function-level editor view


Delete Module?

Do you really want to delete this module? This action can not be reverted!

Figure 3.9. Dialog to confirm deletion of a module

available for the module, similar to the one at package-level. The toolbar is followed by forms for meta information as well as the list of functions and data types contained in the module. Each of the functions and data types can be unfolded to reveal their editors. Note that the module editor from Figure 3.8 already contains an unfolded function editor. The operations available for modules are:

"Save Module" Saving a module also works hierarchically and saves all contained functions and data types in addition to the changed meta information. Changes to other modules or the package meta information are not saved.

"New Function..." and "New Datatype..." Creates a new entry to create a new function or data type. We will explain this operation in Section 3.3.3.

"Delete Module" Deletes the current module from the package. There is a security question (Figure 3.9) to ensure that the user really wants to do this, since this operation is irreversible.

One major difference when looking at the function or data type editor is that they also contain a source code editor as last element, which is shown in Figure 3.8. We will explain the details of the editor component in Section 3.3.4. The operations available at function- and data type-level are:

"Save" Saves the changes to the current function or data type. This does not save changes of other objects, the parent module, or package.

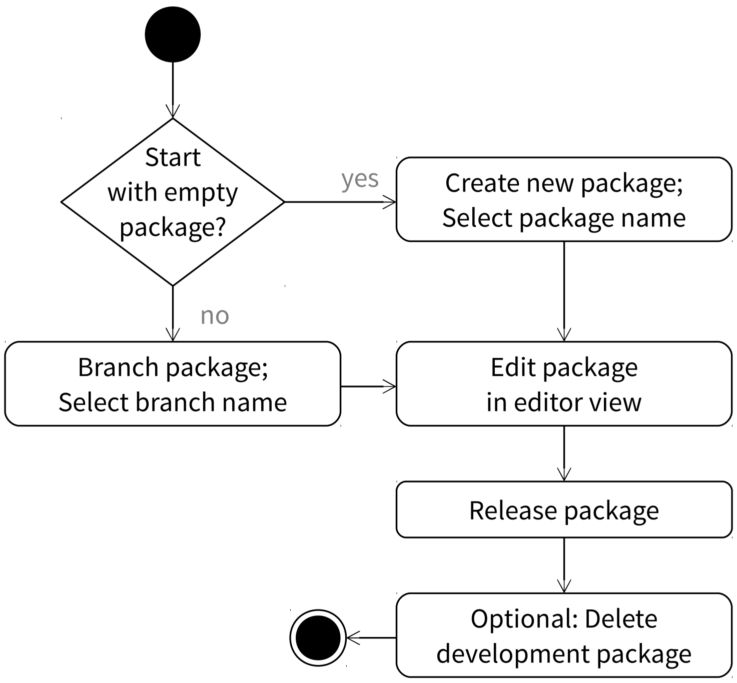

Start with empty package?
yes
no
Create new package; Select package name
Branch package; Select branch name
Edit package in editor view
Release package
Optional: Delete development package

Figure 3.10. Workflow to develop a new package

"Delete" Deletes the function or data type from the module. As for the other delete operations, this one also asks a security question similar to that in Figure 3.9.

The editor view is structured hierarchically, because we think it gives a good view at the semantical structure of a package.

3.3.3 Usage and Workflow

We had several workflows in mind when designing the editors GUI.

First of all, there is a general workflow to develop new packages; it can be seen in Figure 3.10. When developing a package the user first has to decide if her new package shall be based on code of an already existing package, or if she wants to develop a completely new package. Depending on her decision, a branch or package name has to be selected. Once the

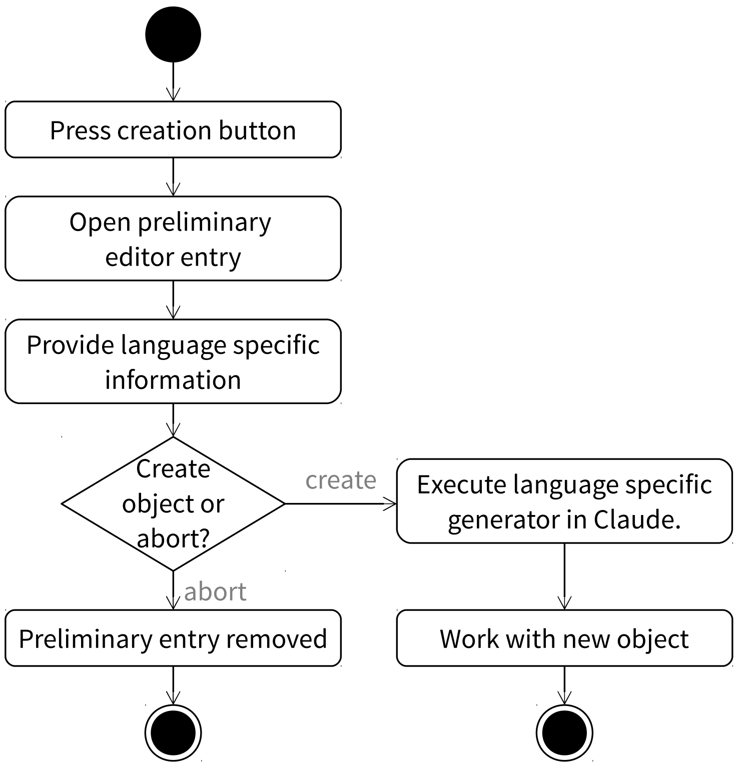

Press creation button
Open preliminary editor entry
Provide language specific information
Create object or abort?
create
Execute language specific generator in Claude.
abort
Preliminary entry removed
Work with new object

Figure 3.11. Workflow to create a new module, function or data type object

new development package was created, the user can change or modify her package in the editor view. When finished with changing or modifying, the user can release her package and then optionally delete the development package afterwards.

Editing a package means to use the editor view. Of course, the editor view has its own workflow. The usage in general should be clear from the previous section, but there are still is a workflow that needs further explanation.

The workflow of creating new objects aside from packages, in general, is the same for modules, functions and data types and can be seen in Figure 3.11. First of all, the user has to press the button to create a new object. This button opens a preliminary entry in the editor view. An example

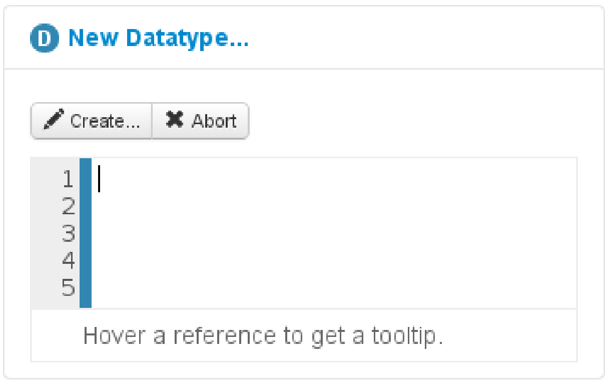


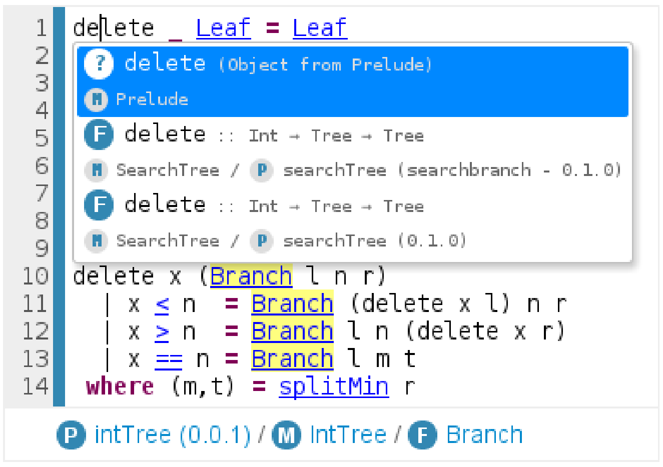
Figure 3.12. Editor view to create a new data type object

of such a preliminary entry for a new data type can be seen in Figure 3.12. The user can provide language specific information to initialize the object with. In case of a module, the development language and name is asked for. For a function or data type the user can enter their source code. The source language is determined by the module that contains the new function or data type. After entering this information, the user can decide if he really wants to create the object or abort the operation. If he selects to abort, the preliminary entry is removed and no changes take place. Otherwise, a language specific generator is called through the development interface, and the object is inserted to the database. Then the user can work with the new object, as he does with any other object in the editor view.

Note that the preliminary entry, used to enter language specific code, is also a hidden dialog to confirm that the user really wants to create a new object.

3.3.4 Hypercode Editor Component

The hypercode editor component is the part of the GUI that allows the user to edit and annotate code with hypercode references. We use *CodeMirror* [codemirror] as a JavaScript based text editor and extend it with the functionality needed. An example of the hypercode editor component in action



```
delete _ Leaf = Leaf
```

```
delete (Object from Prelude)
Prelude
delete :: Int → Tree → Tree
SearchTree / searchTree (searchbranch - 0.1.0)
delete :: Int → Tree → Tree
SearchTree / searchTree (0.1.0)
```

```
delete x (Branch l n r)
  | x < n  = Branch (delete x l) n r
  | x > n  = Branch l n (delete x r)
  | x == n = Branch l m t
 where (m,t) = splitMin r
```

Figure 3.13. The hypercode editor component with activated tooltip and autocompletion

can be seen in Figure 3.13.

In a first step, we add highlighting of existing hypercode references; they are displayed in an underlined blue font. When hovering over a hypercode reference a tooltip for it is displayed. Figure 3.13 shows an example of the hovering behavior. The tooltip links directly to Claude, so a user has additional information about the hovered reference at hand.

In order to implement this behavior, we use the text-marking feature of *CodeMirror* mentioned in the introduction (Section 2.1.4). The hypercode references of our source are stored as an array of objects. Each of these objects represents one reference. We look at an example to show their structure:

```
{ from: { line: 2, ch: 5  }
, to  : { line: 2, ch: 11 }
, ref : { _id       : "5310958f321c6a249f000003"
        , name      : "isJust"
        , text      : "isJust"
        , qualifier: "Data.Maybe" } }
```

The `from` and `to` field delimit the range of the reference within the source code. The actual reference is given in the `ref` entry. In Line 3 to 6 of our example we see a reference as it would be represented by the `CodeRef` constructor from Listing 2.4. A `CodeBaseRef` would just omit the `_id` field.

Assuming `this.refs` is an array that contains all references, we create the text-marks in the following way:

```
 1 _.forEach(this.refs, function(ref) {
 2   ref.marker = this.getDocument().markText(ref.from, ref.to,
 3     { className: 'editor-hypercode-ref'
 4                + '␣' + ref.getClassId()
 5     , addToHistory: false } );
 6   ref.marker.hypercodeRef = ref;
 7 }, this);
 8
 9 _.forEach(this.refs, function(ref) {
10   $('.' + ref.getClassId(), this.getScopeElement())
11     .mouseenter( this.handleMarkHoverInEvent(this, ref) );
12 }, this);
```

The first loop from Line 1 to 8 creates all marks and associates each reference with a created mark. The `getDocument` method returns the *CodeMirror* document object, which we use to create the mark using the `markText` method. Note that we set the CSS class to `editor-hypercode-ref` and the return value of `getClassId`. `editor-hypercode-ref` is used to style the text-mark. The result of `getClassId` identifies the reference's target, which is important for the second loop from Line 10 to 13. The loop selects all elements with this identifying class to associate the event handler, which we call when hovering over the mark.

The `getClassId` method, is a utility added to the reference by the hypercode editor component. The `this` object in our code refers to the object that represents the current hypercode editor component. `getScopeElement` returns the DOM node that contains the editor. To limit the part of the DOM tree the dollar function searches, we pass a DOM node to the function. Limiting the scope is important to ensure that our code does not interact with other editor components, that may use the same class name for their marks.

It is very important that both loops are separated from each other. Due to the creation of marks, the DOM tree of the *CodeMirror* instance may change after each call to `markText`. These changes can invalidate the event handlers associated with the text-mark element. Therefore, we have to associate the event handlers with the text-mark element, after the DOM tree's final structure is known.

In a second step, we implement the removal of hypercode references when editing code. When changing or removing text that is part of a hypercode reference, the reference is automatically removed. The removal is done using the `"change"` event from *CodeMirror*. The event supplies an event handler with the location and range of a change. If one of the reference overlaps with this range, it is removed. All other references are updated correctly according to the change by using their `find` method and updating the reference ranges.

There are some problems with *CodeMirror*s edit history in this context. Since *CodeMirror* does not allow replacing or tapping into its edit history, we have to implement our own history and manually synchronize it with the edit history. Our custom history class is very general, because it just stores two functions for each operation, one to undo and one to redo the operation:

```
1 CustomHistory.prototype.addChange = function(undoOp, redoOp) {
2   this.changes.push({ undo: undoOp, redo: redoOp });
3   if(this.changes.length > this.maxUndoDepth) {
4     this.changes.shift();
5   }
6   this.undoneChanges = [];
7 };
```

The `addChange` method is called every time the *CodeMirror* document changes and appropriate actions to re- and undo the operation are passed in. These functions are stored in our history in Line 2. From Line 3 to 4 we have to check if we have reached the maximum history length; if this is the case, we delete the oldest entry. All undone changes are invalidated after by Line 6, because we only manage a list of changes, not a tree.

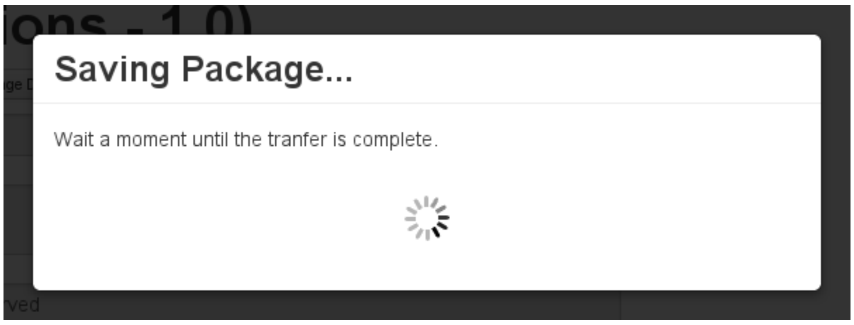
If a change indicates that it is an undo or redo operation, we call the `undo` or `redo` method of our history. Since both are similar, we only look at the `undo` function:

```
1 CustomHistory.prototype.undo = function() {
2   if(_.isEmpty(this.changes) === false) {
3     var change = this.changes.pop();
4     change.undo();
5     this.undoneChanges.push(change);
6   }
7 };
```

Line 2 checks if there are any changes to undo; if so, we get the latest change in Line 3 and call the associate `undo` function. At last, in Line 5, we add the change to the stack of undone changes, so we are able to redo the change if necessary.

Calling our history in sync with all change events makes sure that hypercode references are undone and redone correctly.

At last, we need to add a possibility to add new hypercode references to the source code; this is done using an autocompletion plug-in that *CodeMirror* offers [codemirror-a]. When activated using the keys `ctrl+space`, it uses the current token at cursor position to search the current package and Claude for possibly fitting elements. There also is a possibility to state that code is predefined. Figure 3.13 shows autocompletion. Note that the kind and language of objects depends on the language specific settings currently activated; we will highlight this in Section 3.4. The search interface used for autocompletion will be introduced in Section 3.5.3. There are some limitations when using autocompletion due to the database Claude uses; these will be explained in Chapter 4.


Saving Package...

Wait a moment until the tranfer is complete.

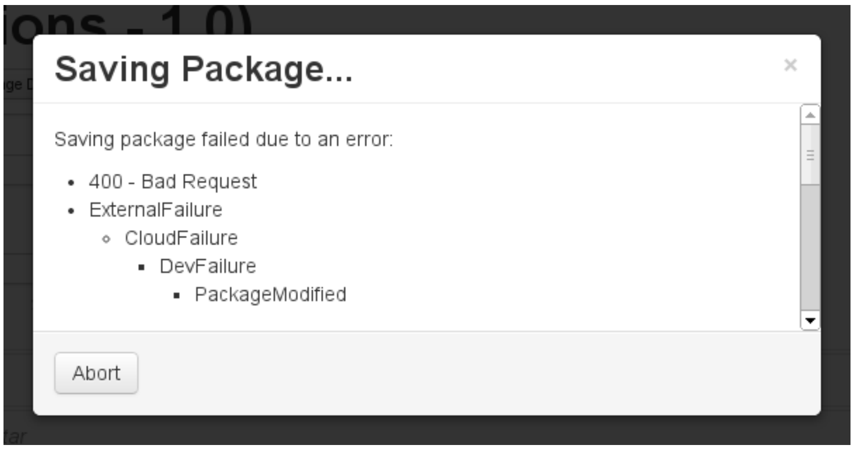
Figure 3.14. Dialog displayed by the editor view when waiting for asynchronous communication

3.3.5 Asynchronous Operations

Each time a change is saved through the editor view network communication with the editor server is required. Since we do not want to reload the editor view for each change, we use JavaScript and AJAX to communicate with the server. This communication is asynchronous, which means there is a time window where an operation has been requested, but the changes resulting from the request have not yet been received and displayed in the view's surface. During this time the user can make changes that lead to an invalid state of the editor. The changes can also be overridden by the response. Hence, we need to keep the user from making those changes.

There are different ways to handle this time window. One would be to make the AJAX communication synchronous. However, this is not desirable and major browsers advise against it [mdn; chrome; msdn; Ros11] since it blocks JavaScript execution and can lead to a "frozen" screen while the communication takes place. The "frozen" screen is due to JavaScript execution being done in the same thread that is responsible for rendering the web page in those browsers.

Therefore, we keep communication asynchronous, but open a modal dialog during the period of time we are waiting for a response. We also disable all forms so the user cannot create an invalid state in between. An example of such a dialog can be seen in Figure 3.14. If the communication is successful and Claude accepts all changes, we just close the dialog again.


Saving Package...

Saving package failed due to an error:

- 400 - Bad Request
- ExternalFailure
  - CloudFailure
    - DevFailure
      - PackageModified

Figure 3.15. Dialog for server-side error messages in the editor view

In case there is an error, it is displayed, as can be see in in Figure 3.15. Currently all information available in a received error object is displayed. Future versions of the editor may choose to render messages differently to give the user a more pleasant experience.

A special case of this asynchronous communication is loading a tooltip for hypercode references in the editor component. Loading such a tooltip should happen as a background operation, because, otherwise, it would interrupt the user's workflow considerably. Therefore, we just display a loading animation in the tooltip area instead of opening a dialog. In case the user wants to see the tooltip of another element before the latest tooltip was loaded, the request is aborted and a new request is started.

We will give an example of an asynchronous JavaScript request when we talk about network communication in Section 3.6.1.

3.4 Language Specific Customization

Although the goal of the editor is to offer a language independent way to develop code in Claude, not every aspect of a language can be captured with the concepts introduced in Claude. Therefore, the language specific types from Listing 2.5 were introduced to the model. This means parts of the editor and also parts of Claude need to handle these language specific aspects.

3.4.1 User Interface Customization

In general the editor view is language independent, but we provide an interface to customize for a language through the JavaScript object `LanguageSpecifics`. It offers the `getLanguageSpecifics` method to get data about language specific customizations that can be applied to the editor. Currently only the language `"Haskell"` is supported. The object returned for customization is guarenteed to have the following methods:

`getLanguage()` Returns the name of the language.

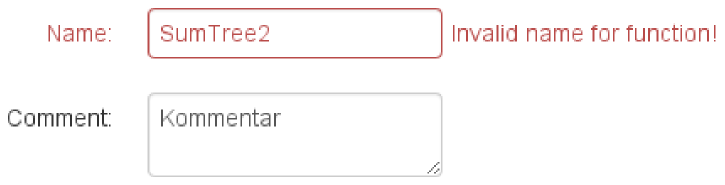
`getCodeMirrorMode()` Returns a mode object [codemirror-b] used to configure the *CodeMirror* instances of the editor view. The mode object configures indentation rules, syntax highlighting and other editor settings.

`getAutocompleteInfo(autocompleteEntry)` Returns a string that is displayed as additional information for a given autocomplete entry. This is responsible for type signatures appearing when autocompleting Haskell code (Figure 3.13).

`getAutocompletePredefined(token)` Returns an object with information if the given token can be annotated as a predefined object and how that reference shall look. Figure 3.13 shows a predefined object generated with this method.

`isAutocompleteToken(token)` Checks if autocompletion should be offered on the given token. This method prevents autocompletion from being offered for keywords or built-in constructs of a language.

`getEditorTitleInfo(editorType , editorData)` Returns an object with additional information to display along with the name of the editor entry. This method is responsible for displaying the type signature and comment in the title line of each editor entry as can be seen in Figure 3.8.



Name:	SumTree2	Invalid name for function!
Comment:	Kommentar	

Figure 3.16. Validation of an editor view form field

`isGeneratedObject(objType, objData)` If the given object is generated by Claude, `isGeneratedObject` returns `true`, otherwise `false`. This method is used to hide generated objects in the editor, because they should not be modified by hand. As an example, generated constructor functions of data types are not available for editing when working with Haskell code, because of this method. We keep this method general, because not yet supported languages may generate other objects.

`getRequestedAutocompletes()` Returns a dictionary to configure which kind of objects shall be offered for auto completion. The dictionary also configures the language of the offered objects. This information is used to feed the autocompletion search offered by the development interface presented in Section 3.5.

`isValidName(type, name)` If the given name is a valid name for an object of the given type, `isValidName` returns `true`, otherwise `false`. This method is used for validation of names before they are sent to Claude. An example can be seen in Figure 3.16.

`createLanguageSpecificForm(objType, objData, formEditor)` Creates the meta data form for language specific data. Figure 3.8 shows a function editor that was enriched with a form for type signatures using this functionality.

More detailed and specific information can be found in the comments of the `LanguageSpecifics` object, which is defined in `static/js/editor/Language-Specifics.js` of the `code-cloud-editor` package.

3.4.2 Object Handling in Claude

As Section 3.3.3 already mentioned, actions in Claude often involve language specific operations. These operations are used to correctly update the model depending on the language. To give an example: each time a Haskell data type is modified, Claude needs to update the constructor functions, which are autogenerated function objects. Though this example is not yet implemented, Claude already offers an interface where this functionality can be implemented easily. The interface is provided through the module `Claude.Server.LanguageSpecific`, which offers the data type `LanguageSpecific`, that is illustrated in Listing 3.1.

```haskell
data LanguageSpecific m = LanguageSpecific
  { lsAddFunction :: ModuleId → [CodeElem] → Package
                  → SessionMonad m (Package, Function)
  , lsAddDatatype :: ModuleId → [CodeElem] → Package
                  → SessionMonad m (Package, Datatype)
  , lsAddModule   :: Name → [Author] → Package
                  → SessionMonad m (Package, Module)
  , lsModifyFunction :: Function → Function → Package
                     → SessionMonad m Package
  , lsModifyDatatype :: Datatype → Datatype → Package
                     → SessionMonad m Package
  , lsModifyModule   :: Module → Module → Package
                     → SessionMonad m Package
  , lsRemoveFunction :: Function → Package → SessionMonad m Package
  , lsRemoveDatatype :: Datatype → Package → SessionMonad m Package
  , lsRemoveModule   :: Module   → Package → SessionMonad m Package
  , lsValidateFunction :: Function → SessionMonad m ()
  , lsValidateDatatype :: Datatype → SessionMonad m ()
  , lsValidateModule   :: Module   → SessionMonad m ()
  , lsValidatePackageModules :: Package → SessionMonad m ()
  }
```

Listing 3.1. Data type to define language specific functionality in Claude

The type parameter `m` specifies the monad transformer stack the functions stored in `LanguageSpecific` work on. We could provide the stack directly, but using a type parameter decouples the data type from the rest of our application and allows us to write more general code.

We can see that there are four groups of functions offered by `Language-Specific`. Most of these functions take a package and return a modified version of that package. Functions to handle packages are not offered, as packages are assumed to be language agnostic. The offered functions all run within the `SessionMonad` to give access to the database in case it is required for the language specific functionality. This access should only be used for read access and not for write access. The changes made are all written back automatically after the language specific functionality has been accessed. Hence, writing to the database from within the interface functions may cause an invalid state or lead to an unnecessary additional update.

First of all, the functions in Line 2 to 7 generate new objects from their parameters. The first two create a function or data type depending on the sources given to them; it is added to the module with the given `ID`. The third function creates a new empty module with the given name and author. Note that the parameters of all three functions reflect exactly what can be entered in the editor view when creating a new object with it, as explained in Section 3.3.3 and displayed in Figure 3.12.

The second group of functions in Line 8 to 13 modifies an existing object. As such a modification may require knowledge about the old version of that object, they always receive the old and the new version of the object to update as their parameters.

At last, the functions in Line 14 to 16 remove objects and the functions in Line 17 to 20 are responsible for performing language specific validation. In the context of Haskell, validation would check if names are unique and valid within their scope. Note, the last function `lsValidatePackageModules` is only there to perform checks that require knowledge of all modules in a package, e.g., if the module name is unique. To check a single module for consistency, we provide the function `lsValidateModule`.

There are separate functions to validate objects, because the validation done within the functions to add, modify or remove objects would always be the same. Therefore, it is more consistent to implement validation at this central point. Validation is always performed before writing back changes. The separate validation is another reason why functions in the interface should not modify the database, because those changes would not be validated properly.

The function `getLanguageSpecific` is used to retrieve a `LanguageSpecific` record for a given language:

```
getLanguageSpecific :: (Sessionable m) ⇒ Language
                    → Maybe (LanguageSpecific m)
```

Currently the only supported `Language` is `Haskell`. If the given language is not found in the map of available language specifics, `Nothing` is returned. Trying to create, update or remove an object for an unsupported language leads to an `UnsupportedLanguageFailure`.

3.5 Cloud Development Interface

The development interface provides the link between the editor and Claude. The editor uses the interface to access the development features of Claude and manipulate the data it stores.

The interface offers the usual create, read, update and delete (CRUD) operations and session management operations. All operations require to run in a monad transformer stack that implements `MonadIO` [hackage-b], because network communication and database access are side-effects and need to be lifted into the `IO` monad.

To indicate failures, all operations return an `Either` value. The return value can either contain an `ExternalFailure` if something went wrong or the actual result of the operation.

All functions also require a `CloudContext` as first parameter. The cloud context supplies the operation with all information needed to contact a specific instance of Claude. The current implementation of `CloudContext` is as follows:

```
data CloudContext = CloudContext
  { cloudHost :: String
  , cloudPort :: Int
  , cloudSession :: Maybe Cookie
  , cloudToken :: Maybe TokenObj
  , cloudHttpLogOutput :: Bool
  , cloudConnManager :: Manager }
```

First of all, it contains the host address and port of the Claude instance we want to work with. When logged in, the entry `cloudSession` contains the cookie Claude uses to identify users. The token associated with the current user is stored in `cloudToken`; it is stored independently of the cookie, because we can not read the contents of the cookie as it is encrypted by Claude. During authentication, the JSON interface of Claude returns the token as result to give us access to it. The `cloudHttpLogOutput` entry tells the interface if it should log debugging information when called. At last, the `Manager` is needed by *http-client* to create new network connections and reuse old ones where possible.

The implementation of `CloudContext` can be exchanged later on if the implementation of the interface changes. The interface provided for `CloudContext` consists of the following functions:

```
mkCloudContext     :: MonadIO io ⇒ (String, Int) → io CloudContext
serializeContext   :: MonadIO io ⇒ CloudContext  → io ByteString
deserializeContext :: MonadIO io
                   ⇒ ByteString → io (Maybe CloudContext)
```

To create a new context we offer the `mkCloudContext` function, which only requires the host and port as parameters; it is the only function that may reflect a change in the underlying implementation of the interface, as it may require different input to create a new context. `serializeContext` and `deserializeContext` can be used to save and load a context; the session management of the editor uses these functions every time the context is stored inside or loaded from the cookie.

The user is required to be logged in for almost all of the operations introduced in the following sections; an exception are the read operations and the authentication functions. Additionally, only maintainers of a package have the right to create new objects in a package, manipulate objects or delete them.

3.5.1 Authentication

Most operations of the development interface require the user to authenticate before he is allowed to modify the objects in the database. As mentioned earlier, the editor does not store any persistent data. Thus, the editor does also not manage its own user accounts. We use user accounts that already exist within Claude for authentication. The three operations from Listing 3.2 manage authentication with Claude.

```
login  :: MonadIO io ⇒ CloudContext → (String, String)
       → io (Either ExternalFailure CloudContext)

logout :: MonadIO io ⇒ CloudContext → io CloudContext

isLoggedIn :: MonadIO io ⇒ CloudContext → io (CloudContext, Bool)
```

Listing 3.2. Development interface functions to authenticate

All of the functions need a `CloudContext` to supply them with all information about the Claude instance to contact; they also may modify the context during their execution and return an updated version of the context.

To authenticate we call `login`. In addition to the context, we have to supply a user name and password in the tuple of the second argument. If authentication was successful the updated context is returned.

We call `logout` if we want to log out. Logging out is always successful, even when we are not logged in.

The last operation `isLoggedIn` checks if the given context is authenticated with the cloud. In case the authentication timed out, `isLoggedIn` eventually updates the context.

3.5.2 Create Operations

The create operations generate new objects within the database.

When we talk about a *package clone* in the following paragraphs, we mean an exact copy of a package, but object IDs are mapped to newly generated ones. Thus, there exists a bijection between the object IDs of the package and the package's clone. References to external IDs are not changed.

The three functions from Listing 3.3 create new packages or release existing ones.

```
createDevelopmentPackage :: MonadIO io
  ⇒ CloudContext → Name → Version
  → io (Either ExternalFailure Package)

createDevelopmentBranch :: MonadIO io
  ⇒ CloudContext → PackageId → BranchName
  → io (Either ExternalFailure Package)

releaseDevelopmentPackage :: MonadIO io
  ⇒ CloudContext → Maybe PackageId
  → Version → Status → PackageId
  → io (Either ExternalFailure Package)
```

Listing 3.3. Development interface functions to create and release packages

The first two functions create new development packages. `createDevelopmentPackage` creates a new empty package with the given initial name and version. `createDevelopmentBranch` creates a branch of an already existing package; it creates a package clone for the package with the given ID. The package or branch name given to either of these functions has to comply with the new rules for package and branch names, that were introduced in Section 3.2.2. These names are not subject to language specific validation, since we assume that packages are language agnostic.

A finished development package can be released with `releaseDevelopmentPackage`; it creates a package clone of the package with the ID given by the last parameter. The clone is updated with the new version given. If a package ID is given in the second argument, it is used as parent package and the clone is the child of that package. This only works if the releasing user also maintains the parent package and the new version is larger than the version of the parent. Of course, the parent may not already have a child package. Releasing a package only works if all objects of the package to release are in development status. All objects of the cloned package receive the new status given in the fourth argument; it may not be the `Development` status.

The next three functions in Listing 3.4 are there to create modules, functions and data types in an existing package.

```
createDevelopmentModule :: MonadIO io
  ⇒ CloudContext → Language → PackageId → Name → [Author]
  → io (Either ExternalFailure (Package, Module))

createDevelopmentFunction :: MonadIO io
  ⇒ CloudContext → Language → PackageId → ModuleId → [CodeElem]
  → io (Either ExternalFailure (Package, Function))

createDevelopmentDatatype :: MonadIO io
  ⇒ CloudContext → Language → PackageId → ModuleId → [CodeElem]
  → io (Either ExternalFailure (Package, Datatype))
```

Listing 3.4. Development interface functions to create modules, functions and data types

All of the functions require the altered package to be in development status; they also return an updated version of the modified package along with the newly created object. Both are returned, because the language specific operations (Section 3.4.2), run to create the new object, may generate further objects that are also inserted into the package. An example of such further objects are the data type constructor functions, that should be inserted to a package every time a data type is created. We also have to supply the language to create the object for as second argument, because otherwise Claude can not decide which language specific generator to use.

New modules can be created with `createDevelopmentModule`. The ID of the package to insert the modules into, the name and the authors are also given. The new module is empty initially.

`createDevelopmentFunction` and `createDevelopmentDatatype` create a new function or data type object respectively. The new function or data type is generated from the annotated source code given in the last argument. We also have to supply the ID of the package and module to insert the new object into.

Note that, the parameters these functions accept exactly match the information that is requested from the user through the editor view. Looking at the signature of these functions, the workflow presented in Section 3.3.3 seems logical.

3.5.3 Read Operations

The read operations just retrieve certain objects from Claude. There are four of them, which are listed in Listing 3.5.

```
getObject :: MonadIO io ⇒ CloudContext
  → ObjectId
  → io (Either ExternalFailure MixedResult)

getPackage :: MonadIO io ⇒ CloudContext
  → PackageId
  → io (Either ExternalFailure Package)

getUserPackages :: MonadIO io ⇒ CloudContext
  → io (Either ExternalFailure [Package])

searchTextAutocomplete :: MonadIO io ⇒ CloudContext
  → Text → Maybe Language → Bool → Bool → Bool → Bool
  → io (Either ExternalFailure [MixedResult])
```

Listing 3.5. Development interface functions to read objects

All functions except `getUserPackages` can be used without authentication.

getObject and getPackage only search for an object with a given ID; the former looks for a package while the later returns any kind of object with the specified ID. The definition of MixedResult is as follows:

```
data MixedResult
  = PackageResult  Package
  | ModuleResult   (PackageDescription, Module)
  | FunctionResult (PackageDescription, ModuleDescription, Function)
  | DatatypeResult (PackageDescription, ModuleDescription, Datatype)
```

It provides one constructor for each type of result object. The tuples are used to ease the usage with other database functions that return exactly these tuples. The PackageDescription and ModuleDescription are just records of the respective ID, name and version.

getUserPackages returns a list of packages that are maintained by the current user.

The last function, searchTextAutocomplete, is used to search for possible hypercode references to annotate with source code in the editor. The first argument is the name of the objects we are searching for. This is supposed to be a prefix search for names, but there are some limitations that will be discussed in Section 4.2. The second argument sets the language of objects to search for; when Nothing is given objects of all languages are returned. The last four boolean arguments say if packages, modules, functions or data types are included in the search results (in that order).

3.5.4 Update Operations

There are four operations to update or modify objects in the interface:

```
updateDevelopmentFunction :: MonadIO io
  ⇒ CloudContext → PackageId → Function → Function
  → io (Either ExternalFailure (Package, Function))

updateDevelopmentDatatype :: MonadIO io
  ⇒ CloudContext → PackageId → Datatype → Datatype
  → io (Either ExternalFailure (Package, Datatype))
```

```

updateDevelopmentModule :: MonadIO io
  ⇒ CloudContext → PackageId → Module → Module
  → io (Either ExternalFailure (Package, Module))

updateDevelopmentPackage :: MonadIO io
  ⇒ CloudContext → Package → Package
  → io (Either ExternalFailure Package)
```

Listing 3.6. Development interface functions to modify objects

All of the functions receive the object containing all changes in the last argument. An object can only be updated if it did not change in the time in between loading and saving it. Therefore, the unchanged original object, from when editing started, is given in the second last argument. It is then checked if the persisted object still matches that unchanged version to ensure no intermediate changes occurred. Only if this check succeeds, the object is updated. It is only possible to update objects that are in development status. Even if the object ID or status is set to a new value in the changed object, the functions do not modify these attributes, since this would invalidate the development status or referential integrity of other hypercode references. All dependencies to other objects in the database are automatically updated according to the changed contents of the updated object.

The first three functions receive the ID of the package, containing the object to change, in the first argument, whereas `updateDevelopmentPackage` extracts the package ID from the unchanged package.

All functions return an updated version of the package containing the changed object to reflect language specific changes, that may have altered the package as a whole. In case of the last function, the changed package and object coincide.

3.5.5 Delete Operations

The last class of operations consists of functions to delete objects (Listing 3.7). All of these functions require the object that shall be deleted to be in

```
deleteDevelopmentFunction :: MonadIO io
  ⇒ CloudContext → PackageId → Function
  → io (Either ExternalFailure Package)

deleteDevelopmentDatatype :: MonadIO io
  ⇒ CloudContext → PackageId → Datatype
  → io (Either ExternalFailure Package)

deleteDevelopmentModule :: MonadIO io
  ⇒ CloudContext → PackageId → Module
  → io (Either ExternalFailure Package)

deleteDevelopmentPackage :: MonadIO io
  ⇒ CloudContext → PackageId
  → io (Either ExternalFailure ())
```

Listing 3.7. Development interface functions to delete objects

development status.

The first three functions take the ID of the package to delete from and the object that shall be deleted. Deletion can only succeed if the object has not changed in the database yet; this, again, is a measure to prevent lost updates. The return value is the modified package after deleting the given object. There may be further, language specific, changes to the package, e.g., the removal of automatically generated constructor functions for data types. All hypercode references to the given object are automatically removed everywhere in the package. It is not necessary to remove the references from other packages, because it is not allowed to set external dependencies to development objects.

The last function, `deleteDevelopmentPackage`, only takes the package ID to delete the package. It returns unit if the package was deleted successfully and an `ExternalFailure` otherwise.

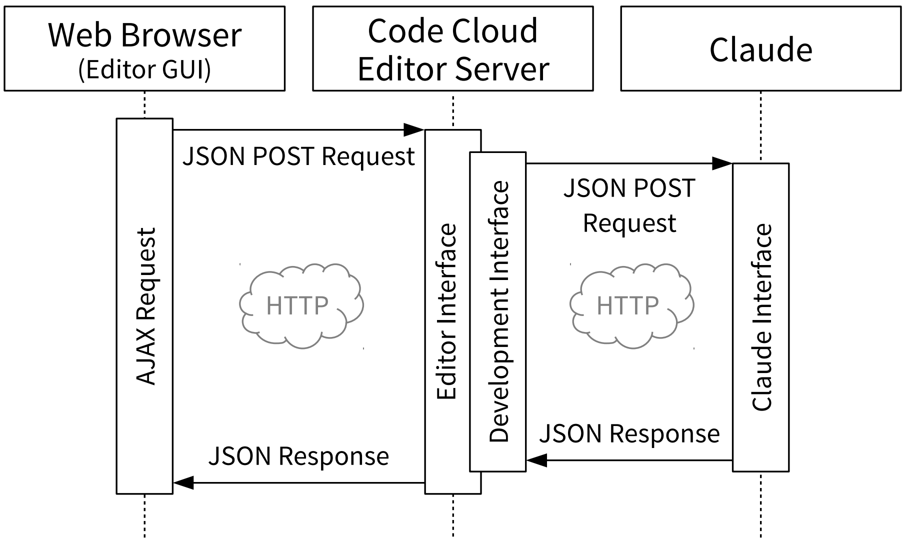


Figure 3.17. AJAX communication of the different editor components

3.6 Network Communication

Figure 3.2 gives us a general idea of which instances are present when the editor is running and which of them communicate with each other. In the following section we will take a closer look at this communication.

3.6.1 HTTP Communication

All communication is done over HTTP. As we do not want the editor view to reload each time an action is selected, it sends an AJAX request using JavaScript to execute an action and update the view depending on the answer. Figure 3.17 displays how these AJAX requests are run through the different instances.

Once the button of the selected action is pressed, a JSON object with the request data is constructed. The JSON object is sent to the editor server using a POST request. We use the `ajax` function from *jQuery* to send such requests. To save the changes of a data type, sending the request looks as follows:

```
$.ajax({
  type: 'POST',
  url: '/editor/update/datatype',
  data: $.toJSON({
    'package-id'       : pkgId,
    'original-datatype': origDtp,
    'updated-datatype' : modDtp
  }),
  dataType: 'json',
  success: function(result) {
    /* Update editor view and hide dialog... */ },
  error: function(result, status, errorMsg) {
    /* Display error in dialog and make it non modal... */ },
  processData: false,
  mimeType: 'application/json'
});
```

Listing 3.8. Sending an AJAX request to save changes of a data type

Of course, this code is part of a function to abstract over the process of sending a request. The function parameters are already substituted and GUI related code is not shown. We assume that `pkgId` contains the ID of the package containing the data type. We also assume that `origDtp` and `modDtp` contain the original and the modified version of the data type respectively. In addition to the standard parameters already explained in the introduction, we also set the flag `processData` to `false`. Setting this flag prevents *jQuery* from automatically converting our data into an URL encoded string. Note that the `data` values exactly match the parameters needed to call the `updateDevelopmentDatatype` function (Listing 3.6) of the development interface. The code above is exactly what the sequence diagram in Figure 3.17 refers to with "AJAX Request".

The editor server decodes the request data to Haskell values and calls the appropriate function of the development interface. The interface then reconstructs a JSON object from the Haskell data and sends it to Claude through another POST request. Claude processes this data and executes the requested operation. When the operation is processed or has failed, the answer is again encoded in JSON and sent back as HTTP response. The development interface decodes the answer and returns it as a Haskell value, which is then reencoded as JSON and sent back to the browser by the editor server. The browser processes the JSON answer by calling the `success` or `error` callbacks on Line 10 and 12 of Listing 3.8. The callbacks update the editor view accordingly.

Answers and errors are likewise encoded in JSON, but their response status differs. While a successful request always responds with a 200 "OK" status, an erroneous response has a 400 "Bad Request" status. The response code is how the `ajax` method knows, if it has to call the `error` or the `success` handler in our example from Listing 3.8.

We notice that the editor server does nothing else but decoding and encoding JSON messages while it uses the development interface. Of course, this could be simplified, but by using the development interface, we decouple the editor from Claude and enable us to exchange the implementation of either the editor or the data management in Claude at some point.

Other parts of the editor such as logging in, creating new packages, releasing packages or deleting packages use classical HTML forms. Once the form is filled with all required data, we can send it using a submit button. The data is then URL encoded inside of a post request. Once these requests arrive at the editor server, they are handled and the development interface is called. The interface then communicates with Claude using the already explained AJAX/JSON mechanism.

The only exception to this behavior within the development interface is the log in. Since we want to reuse the existing infrastructure offered by Claude, we actually send form data instead of JSON when logging in. This way we do not have to handle logging in differently depending on the two different components — the editor and Claude — we can authenticate from. Using the HTTP "Accept" header, we can tell Claude that we want a response encoded as JSON, not as HTML, because the editor can only properly process JSON.

3.6.2 Automated JSON Transfer Infrastructure

We already noticed that the editor server only forwards messages from the editor GUI to Claude and vice versa. Programming this forwarding mechanism by hand each time is error-prone and repetitive. Therefore, we develop infrastructure to ease this process; it allows us to write down the names of fields in request objects and associate them with their corresponding Haskell type. With these definitions (signatures) we can automatically handle the encoding and decoding of JSON requests and call functions that match their structure.

The type to write down signatures is `JsonParams`:

```
data JsonParams f m err v where

  (:→) :: (FromJSON a, ToJSON a)
        ⇒ Text → JsonParams f m err v
        → JsonParams (a → f) m err v

  Res   :: JsonParams (m (Either err v)) m err v
```

The first type parameter `f` contains the complete function signature of a function that can be called if the described JSON object is decoded. It can just as well describe the function that is produced to encode a value into the described JSON object. The second type parameter `m` provides the monad the function result is run in. The `err` type parameter contains the error type used if something fails, and the last parameter `v` represents the value returned by a described function.

The "`:→`" constructor extends a described function by one argument. The constructor sets the name of the JSON object field associated with this argument and takes the rest of the function signature as second argument. Note that the returned function signature is extended by the new argument; this requires the use of GADTs. Also note that every parameter value needs to implement `toJSON` and `fromJSON` from the *aeson* library (Section 2.2.2). These constraints ensure that conversion from and to JSON works for a function involving the type `a`.

The second constructor `Res` determines the end of a signature; it sets the return value to the monad computation, that either returns an error value or the result value.

As an example, we can look at the signature defined for `updateDevelopmentDatatype` from Listing 3.6:

```
updateDevelopmentDatatypeSig
  :: JsonParams (PackageId → Datatype → Datatype
                             → m (Either err (Package, Datatype)))
                m err (Package, Datatype)
updateDevelopmentDatatypeSig =
  "package-id" :→ "original-datatype" :→ "updated-datatype" :→ Res
```

Listing 3.9. JSON transfer infrastructure signature of the `updateDevelopmentDatatype` function

Recall the type of `updateDevelopmentDatatype`:

```
updateDevelopmentDatatype :: MonadIO io
  ⇒ CloudContext → PackageId → Datatype → Datatype
  → io (Either ExternalFailure (Package, Datatype))
```

Listing 3.10. Reminder of the function signature of the `updateDevelopmentDatatype` function

Note the signature's function type is unifiable with the type of `updateDevelopmentDatatype` after the `CloudContext`. This means, with the right tools we can call `updateDevelopmentDatatype` directly from a function that handles a request. Also note that the names given in the signature match those used when sending the AJAX request in Listing 3.8.

First of all, the infrastructure offers a function to handle incoming JSON requests:

```
runJsonRequest :: (MonadHandler m, ToJSON v)
               ⇒ (String → err')
               → (err → err')
               → (err' → m TypedContent)
               → JsonParams f m err v
               → f
               → m TypedContent
```

The first three parameters handle errors; if decoding of JSON fails, the function from `String` to `err'` is called with the decoding error message. The second function converts the error, resulting from the function `f`, to an error that can be handled by the third function. We decouple the error type of the handler function `f` from JSON errors, because of `runJsonRequest`'s use in the editor server. When handling a JSON request in the server, the called handler function is provided by the development interface, which uses its own error type `ExternalFailure`. However, inside of the editor server, we commonly only handle `EditorFailures` and, therefore, need to convert errors. The other two parameters take the signature of the JSON objects to handle and the function that is supplied with the contents of the objects. `runJsonRequest` is already specialized for uses in *Yesod*, because we only handle requests when receiving them in the editor server or in Claude, which both use *Yesod*. All request handling done in *Yesod* uses monads that implement `MonadHandler` and return some `TypedContent`. In this case, the `TypedContent` contains a HTTP response that contains the JSON encoded result value of type `v`.

As an example, we look how the AJAX request from Listing 3.8 is handled using the following function:

```
1 postEditorUpdateDatatypeR :: Handler TypedContent
2 postEditorUpdateDatatypeR = runWithLogin $ \cxt → do
3   runJsonRequest
4     JsonProcessingFailure ExternalFailure
5     sendJsonError
6     updateDevelopmentDatatypeSig
7     (updateDevelopmentDatatype cxt)
```

The displayed source code is all code required to handle the AJAX request and forward it to the development interface. The call of `runWithLogin` ensures that the user is logged in and supplies us with a `CloudContext`.

The first two parameters of `runJsonRequest` in Line 4 are constructors for error values. If we fail to parse the incoming JSON value, we produce a `JsonProcessingFailure` value. In case the call of the development interface fails, an `ExternalFailure` is created and sent back to the editor view. `sendJsonError` is a function, also provided by the infrastructure, that sends an error encoded as JSON. In this case, its signature specializes to `EditorFailure → Handler TypedContent`. Both error constructors, `JsonProcessingFailure` and `ExternalFailure`, are part of `EditorFailure`. The fourth parameter is the signature we want to use (Listing 3.9). The last parameter is the call to the development interface function `updateDevelopmentDatatype` from Listing 3.10.

The second function offered by the infrastructure sends JSON requests:

```
sendJsonRequest :: (Monad m, ToJSON o)
                ⇒ JsonParams f m err o
                → (Value → m (Either err o))
                → f
```

The function is not specialized to *Yesod*, since it is used from the development interface that uses *http-client* to send its requests. Once supplied with the signature and a function that sends the created JSON `Value`, it constructs a function according to the signature. The constructed function can be supplied with the Haskell values given in the signature; it automatically converts the Haskell values into JSON values and sends them as part of the JSON request object. There are no arguments for error handling, because converting some value to a JSON object is always possible, if the `ToJSON` type class is implemented.

An example for the use of this function is the implementation of the `updateDevelopmentDatatype` function:

```
updateDevelopmentDatatype
  :: (MonadIO io) ⇒ CloudContext
  → PackageId → Datatype → Datatype
  → io (Either ExternalFailure (Package, Datatype))
updateDevelopmentDatatype cxt =
  sendJsonRequest
    updateDevelopmentDatatypeSig
    (jsonPostRequest cxt ["dev", "update", "datatype"])
```

We only need to supply `sendJsonRequest` with the signature and the function to send the JSON value. These arguments are enough to automatically generate the rest of the interface function. The `jsonPostRequest` function sets up a HTTP POST request with the given JSON value; it requires a `CloudContext` to log in and the path of Claude to contact. In this case, `jsonPostRequest` would contact the path `/dev/update/datatype`.

As we can see, the infrastructure allows us to write down the code to handle and send AJAX request concisely and supports us with its use of the type system in doing so.

Chapter 4

Problems, Limitations and Future Work

Though we finished a working version of the editor, there are still known problems and limitations. We have some ideas for improvements, which can be done in the future.

4.1 Editor and Claude

Though the editor and Claude can handle simple Haskell packages, they do not support major parts of Haskell yet. There is no support or representation for type classes. Non-standard extensions of the language, such as type families [Sch+08] or functional dependencies [Jon00], are also ignored.

Claude has the ability to up- and download Haskell packages to and from its database respectively. However, this translation is incomplete; it ignores comments, type classes, pragmatics, or extensions.

The language specific functionality to create, modify and delete objects has a central interface, but there is only a basic implementation that lacks expected functionality. Right now, Haskell is the only supported language, and the validation of Haskell specific aspects works correctly, as described in Section 3.5. However, the creation, modification and deletion of functions and data types is incomplete. The correct extraction of meta information from sources is not implemented and automatic generation or handling of constructor functions for data types does not work either.

The language specifics in the editor view miss two checks for annotating predefined tokens. First, tokens can always be annotated as predefined in `Prelude`, although the `Prelude` may not actually offer them. Secondly, it is not possible to annotate tokens as predefined in another base module aside from `Prelude`. All of these problems and limitations need to be solved in future work.

4.2 Database Limitations

Some of Claude's limitations arise from the underlying database *MongoDB*.

One limitation is that the full-text search capability of *MongoDB* is word-based. Therefore, it is not possible to search for prefixes of words in text. [mongo-b; mongo-c] A prefix search would be useful to improve the autocompletion feature of the hypercode editor component introduced in Section 3.3.4. Right now, the autocompletion only suggests names for an exact match. With a prefix search, suggestions for other objects could also be made, which would improve the usefulness of autocompletion considerably. There are different ways to solve this limitation: a future version of *MongoDB* may add this feature; it could be implemented within Claude at some point. An external search engine could also be used.

Another limitation is *MongoDB*'s lack of transactions. Only one single operation on a single document is handled atomically, but there is no way to atomically express several subsequent operations [mongo-d]. However, some of our more elaborate validations require such operations on several documents. To prevent race conditions, we had to develop our own transaction management. The transactions are implemented within `Claude` and integrated into the database interface functions in the modules `DataInterface`, `DevelopmentInterface` and `UserInterface`. Locks are placed on IDs of database objects to keep the system simple; they are reentrant to ensure that different locking functions remain composable. We decided to only lock on write operations, because read operations are more complicated to realize as we do not always know which objects are read. As an example, if we search for objects with a certain property, we cannot know which objects are the result; this means we do not know which IDs to lock. In contrast, all of our writing operations work with objects that are identified by their ID. The interface of our transaction management consists of three functions:

```
mkTransactionManager :: MonadIO io ⇒ io (TransactionManager res)
doTransaction
  :: (MonadError e io, MonadIO io, Ord res)
  ⇒ TransactionManager res → res → io a → io a
doMultiTransaction
  :: (MonadError e io, MonadIO io, Ord res)
  ⇒ TransactionManager res → [res] → io a → io a
```

First of all, we provide a function to create a `TransactionManager`. The `TransactionManager` manages locks on a given type of resource `res`. Only transactions using the same manager can block each other. We abstract over the resources we lock on, but Claude only uses `ObjectIds` as resource. A transaction over a single resource can be performed with `doTransaction`. If we want to lock several resources at once, it is safer to use `doMultiTransaction`, because it ensures that the resources are locked in a global order to avoid possible deadlocks. We also take care of errors that arise in `MonadError` instances [hackage-a] to ensure the lock is released even if such an error occurs.

Note that we abstract over `MonadIO` [hackage-b] for operations that can be run in transactions, because they are later used on operations that take place in a monad transformer stack, e.g., the `Handler` monad. A problem with this abstraction is that we are not able to handle `IO`-based exceptions anymore. To catch such an exception we would need to use `catch` from `Control.Exception`:

```
catch :: Exception e ⇒ IO a → (e → IO a) → IO a
```

However, we cannot catch exceptions from our operation `io a`, since `MonadIO` only offers a lifting operation `IO a → io a`. To catch `IO`-based exceptions with `catch` we would need a lowering operation `io a → IO a`. Providing such an operation may be possible if we know which monads are involved in our transformer stack and how they are implemented, but it is not possible in our case. We do not know how the internals of the `Action` monad from *MongoDB* or the `HandlerT` and `WidgetT` from *Yesod* work; these are an integral part of the transformer stack used in Claude. Therefore, we ignore `IO`-based exceptions in our transactions. We do not think that ignoring these exceptions is a problem in our case, since Claude and the editor use the `ErrorT` transformer [hackage-c; LHJ95] to handle errors. Any `IO`-based exception that might occur comes from *Yesod* or *MongoDB* itself and we do not know if we could recover from such an error anyway.

Edward Z. Yang [Yan12] gives a more elaborate discussion on proper exception handling in `MonadIO` transformer stacks and the problems that arise from `IO`-based exceptions in this context.

In the future, the missing support of transactions can be solved in different ways. We can hope that *MongoDB* will implement them at some point. Using a different database system with support for transactions may also be useful. A relational database system would have advantages over *MongoDB*'s document-based approach. In addition to proper transaction support, it would allow a more granular access to information, where *MongoDB* always requires us to handle the complete document, which may become large for a big package.

4.3 Database Abstraction

The Claude project group decided to implement its own API on top of *MongoDB*. We did not tap into this abstraction layer to concentrate on the editor itself, but there are some limitations we have noticed.

First of all, the function `updateFunction` from `Claude.Server.Interface` to update a function object does not work. When applied, it leads to a failure coming from the database. This problem seems to be independent of the used query and may be related to a known problem with querying nested arrays in *MongoDB*. [mongo-a] As a workaround, we started updating the complete package instead. In the editor's current state, a complete update of the package has to be done anyway, because language specific changes may change several other parts of the package when updating an object. Therefore, this bug is not relevant to the development interface or the editor anymore, but should be noted for future developers. We have not checked if other related functions such as `updateDatatype` or `updateModule` also malfunction.

There was another problem when querying for objects that contain fields with `Maybe` values. When querying for `Nothing` by setting the query to `NotSet`, no results were delivered. We did not investigate this problem far enough to know the reason, but we assume that there is some semantic confusion during the translation of the Claude query to a *MongoDB* query. All `Maybe` fields that were queried had the type `Maybe Text`. Therefore, we solve the problem by replacing them with a field of type `Text`. The `Text` type essentially has the same meaning for each of the involved fields.

The query and modifier data types introduced in Section 2.3.1 are limited in their expressiveness. As an example, there is no way to query for a module that contains several functions with different attributes, because the `mFunctions` field of `QModule` has type `QFunction` (Listing 2.7). If we want a module that contains a function with name "f" and a function with name "g" this would not be possible, because `QFunction` only allows us to query for either of these values. This problem also occurs when trying to query for several modules in a package.

A similar problem arises if we want to find objects of a certain language. The language of an object is determined by the instance of its language specific value (Section 2.3.1, Listing 2.5). When querying for such a value, we only have the possibility to specify a concrete value as a `Query LanguageModSpecifics`, `Query LanguageFuncSpecifics`, or `Query LanguageDatatypeSpecifics`, but we cannot formulate the general query for all Haskell objects. There is no query record for these types that would allow this kind of query.

We do not discuss solutions to the limitations of the query language abstraction here, since that would lead out of scope. These remarks are meant as a hint for future developers of Claude. Currently, both cases are handled by hand in Haskell after the query has taken place.

4.4 Management of Concurrent Changes

Currently, a user cannot save her changes if someone changed the objects she is working on concurrently. The user has to reload the objects from Claude and overwrite her changes. A future version of the editor may be extended with a proper interface to merge changes. *CodeMirror* already offers a plug-in to display differences [codemirror-c] between two documents.

When implementing such a merge view, saving objects should also be made more granular. Currently, we can only save objects as whole, but it would be useful to only save those parts of an object that have actually changed. With this approach, conflicts with concurrently changed objects would occur less frequently. It may be necessary to refactor the data model used by Claude to support these more granular changes.

4.5 Future Work

There is a wide range of other work that can be done in future.

First of all, there still are a lot of features that the editor should support. It should be possible to move objects, e.g., move a function from one module to another. More extensive refactoring operations such as renaming are also useful.

In light of the change management discussed in the previous section a proper version control may be useful. Right now saving a change means overwriting the old version. In the past it has proven useful to maintain a history of changes, as done by modern version control systems.

The language specific features can be extended as well. A package developer is interested if the code contained in the package compiles. Therefore, offering to run the language specific compiler or interpreter to check syntactic integrity would be useful, especially, since this is not checked at all right now. In context of Haskell packages the type system supports the programmer and it would be useful to see its response on code written in Claude, too.

We developed a considerable amount of JavaScript code. During the development, we noticed that JavaScript is not a language to develop large projects with; it is possible when disciplined, but may cause a lot of struggle. JavaScript by itself misses important features to program in the large. There is no module system to structure a large code base. Aside of syntax checks only runtime errors show typos or the wrong use of code. Static code checks are not available by default. It would have been useful to use a system such as *JSLint* [jslint] or *JSHint* [jshint] to statically check code quality from the beginning. The support of a static type system would have also been desirable.

A solution to this problem would be to use a language that compiles down to JavaScript to write the editor view. There are different languages that may be suitable: *TypeScript* [typescript] or *PureScript* [purescript]. From the beginning, both of these languages were designed to compile to JavaScript and are well suited to write large programs in them. The languages offer a type system and a variety of static checks to support the programmer. [typescript; purescript-a] Most importantly, they offer a module system [typescript; purescript-a] to structure large application. *PureScript* is especially interesting, since its syntax is close to that of Haskell. In theory this may allow sharing common code and data types between Haskell and the *PureScript* code, which would be beneficial for maintenance and to ensure communication compatibility.

Chapter 5

Related Work

The ideas behind Claude and the editor were already expressed in a paper from Kaiser and Dossick [Kai+97]. We picked up the expressed ideas and tried to apply them to our work.

Claude and the editor combine several different aspects offered by a variety of applications that already exists. Looking at each of these aspects, we find a number of applications that offer similar functionality. An exhaustive survey of all of these would lead beyond this work. Therefore, we will pick some specific systems at each point. Our focus is set on applications from the Haskell and JavaScript ecosystem, since those are the tools used to develop Claude and the editor.

First of all, Claude offers a way to search the stored packages and objects to view their meta information, documentation and implementation. A similar functionality is offered by the many different code search engines available. The Haskell community offers two: *Hoogle* [hoogle] and *Hayoo* [hayoo]. Their results link with the documentation hosted on *Hackage* [hackage]. Other languages such as JavaScript tend to combine a search directly into their documentation, e.g., the *jQuery* documentation [jquery-b]. Claude is distinct from these systems in that its search works directly on the source objects it stores in its database, instead of indexing or being built separately. Of course, Claude's search is not as sophisticated or elaborate as those provided by *Hoogle* and *Hayoo*. However, creating such an advanced search on top of Claude would be possible and may be done in future work on Claude.

Aside of being a search engine Claude is also a system to host code and packages. A different yet comparable system would be *Hackage*. It also stores Haskell packages, but does not analyze their sources, annotate them with hypercode, or check for their integrity other then looking at the package description. JavaScript does not have a commonly acknowledged system to host and distribute packages. There are several different tools to offer this service: *npm* [npm], *Bower* [bower], *Ender* [ender], *volo* [volo] and some others [comp; jam]. However, none of them has a dominating role and it is also common not to use any of them at all. Again, these systems are meant to store or allow access to packages, but they do not check the packages for integrity other then looking at the package description. Aside of the additional dependency analysis and hypercode generation, another major difference between Claude and the mentioned systems is, that Claude does not offer a way to install packages. *Hackage* uses *Cabal* [cabal-b] for this purpose and the JavaScript tools offer their own mechanisms to install packages. Claude allows exporting a package again, but requires a user to install it himself. Claude has the advantage that it is able to automatically offer all dependencies required for a package, and it can also create a new package already containing all sources of dependencies, which may ease the installation process considerably. The abstract representation of language objects and the hypercode annotations allow this mingling of a package with its dependencies.

Claude can also be seen as a platform to host source code such as GitHub [github], which is popular among Haskell and JavaScript developers, but Claude lacks the proper version control functionality other systems offer.

The idea to offer a web-based integrated development environment (IDE) is not new either, there are several successful projects such as *Cloud9* [cloud9], *Koding* [koding], *FP Haskell Center* [fpcenter], or *Eclipse Orion* [orion]. In contrast to Claude, these IDEs still work in file-based manner and just offer a proper integration of the tool-chain needed to develop in a certain language; they do not use an abstract representation of the language concepts to work with. The major goal of the mentioned IDEs is to shift from desktop-based to web-based development. A major drawback of the editor is that it does not yet support language specific tool-chains. There is no integration to compile, debug or run libraries and applications.

Working on language concepts instead of structuring everything in files is also not a new idea. IDEs such as *Lighttable* [light-a; light-b] or *Code Bubbles*

[Bra+10a; Bra+10b] provide an interface to support working with abstract language concepts. Though the interface of the IDEs is built around this idea, they still use files to store their code. In contrast, our editor works on an abstract representation instead of converting back to a file-based representation behind the scenes. Of course, the interface of the editor is not yet as sophisticated and well designed as that of the two mentioned IDEs, but there is nothing to hinder improving it in future.

Chapter 6

Conclusion

We started by introducing Claude and the used technologies as a foundation. Building on these foundations, we presented the general structure of the editor and discussed the changes made to Claude during its development. The features and usage of the editor were introduced by discussing the GUI. Knowing what features are there, we then talked about other details of the implementation: the language specific functionality, the development interface, and the network communication. We closed our thesis by first discussing the different limitations and problems that still exist, and then talking about the related work.

Most related work only relates to certain aspects of the editor or Claude. None of the work matches the full capabilities of Claude and the editor, but all of it offers more mature capabilities when looking at the matched parts.

Our initial goal was to develop an editor that enables a user to create, modify and release packages in Claude in a language independent manner. We have achieved this goal. As discussed in the GUI section, the editor is capable of working with the packages held by Claude. The package editor is able to annotate source code with dependencies from other packages, and it is also able to modify all contents of a package including its meta information. We also kept the editor decoupled from Claude by introducing the development interface.

Of course, there still is future work. The language specific customization still requires to implement proper handling of language specific changes, integration of language specific tools, and the GUI should not offer predefined annotations for every syntactic object. Aside of language specific features, the database misses support for transactions and its abstraction needs to be extended. A proper support of version management or at least the ability to merge changes in the editor view would also be pleasant.

Bibliography

Literature

[Bra+10a] Andrew Bragdon, Steven P. Reiss, Robert Zeleznik, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola Jr. "Code bubbles: rethinking the user interface paradigm of integrated development environments". In: *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*. ICSE '10. Cape Town, South Africa: ACM, 2010, pp. 455–464. ISBN: 978-1-60558-719-6. DOI: `10.1145/1806799.1806866`. URL: `http://doi.acm.org/10.1145/1806799.1806866`.

[Bra+10b] Andrew Bragdon, Robert Zeleznik, Steven P. Reiss, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola Jr. "Code bubbles: a working set-based interface for code understanding and maintenance". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '10. Atlanta, Georgia, USA: ACM, 2010, pp. 2503–2512. ISBN: 978-1-60558-929-9. DOI: `10.1145/1753326.1753706`. URL: `http://doi.acm.org/10.1145/1753326.1753706`.

[ecma11] Standard ECMA-262. Ecma International. June 2011. URL: `http://www.ecma-international.org/publications/standards/Ecma-262.htm` (visited on Mar. 6, 2014).

[Hei07] M. Heim. Exploring Indiana Highways: Trip Trivia. Travel Organization Network Exchange, 2007. ISBN: 9780974435831.

[Jon00] Mark P Jones. "'Type classes with functional dependencies'". In: *Programming Languages and Systems*. Springer, 2000, pp. 230–244.

[Kai+97] Gail E. Kaiser, Stephen E. Dossick, Wenyu Jiang, and Jack Jingshuang Yang. "An architecture for www-based hypercode environments". In: *Proceedings of the 19th International Conference on Software Engineering*. ICSE '97. Boston, Massachusetts, USA: ACM, 1997, pp. 3–13. ISBN: 0-89791-914-9. DOI: `10.1145/253228.253231`. URL: `http://doi.acm.org/10.1145/253228.253231`.

[LHJ95] Sheng Liang, Paul Hudak, and Mark Jones. "Monad transformers and modular interpreters". In: *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '95. San Francisco, California, USA: ACM, 1995, pp. 333–343. ISBN: 0-89791-692-1. DOI: `10.1145/199448.199528`. URL: `http://doi.acm.org/10.1145/199448.199528`.

[Mar10] Simon Marlow, ed. Haskell 2010 Language Report. 2010. URL: `http://www.haskell.org/onlinereport/haskell2010/` (visited on Mar. 6, 2014).

[Sch+08] Tom Schrijvers, Simon Peyton Jones, Manuel Chakravarty, and Martin Sulzmann. "Type checking with open type functions". In: *ACM Sigplan Notices* 43.9 (2008), pp. 51–62.

[SJ02] Tim Sheard and Simon Peyton Jones. "Template meta-programming for haskell". In: *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*. Haskell '02. Pittsburgh, Pennsylvania: ACM, 2002, pp. 1–16. ISBN: 1-58113-605-6. DOI: `10.1145/581690.581691`. URL: `http://doi.acm.org/10.1145/581690.581691`.

[Sno12] M. Snoyman. Developing Web Applications with Haskell and Yesod. Oreilly and Associate Series. O'Reilly Media, Incorporated, 2012. ISBN: 9781449316976. URL: http://www.yesodweb.com/book-1.2.

Online Resources

[ace] Ace - The High Performance Code Editor for the Web. Ajax.org B.V. URL: http://ace.c9.io/ (visited on Mar. 6, 2014).

[Ada] Douglas Adams. DNA/Biography. The Digital Village, Ltd. URL: http://www.douglasadams.com/dna/bio.html (visited on Mar. 6, 2014).

[bower] Bower. Twitter, Inc. URL: http://bower.io/ (visited on Mar. 6, 2014).

[browsers] Comparison of web browsers – Wikipedia, The free encyclopedia. Wikimedia Foundation, Inc. URL: http://en.wikipedia.org/wiki/Comparison_of_web_browsers#JavaScript_support (visited on Mar. 6, 2014).

[bson] BSON - Binary JSON. URL: http://bsonspec.org/ (visited on Mar. 6, 2014).

[cabal-a] Duncan Coutts and Ian Lynagh. Cabal User Guide. *Developing Cabal packages*. URL: http://www.haskell.org/cabal/users-guide/developing-packages.html#package-descriptions (visited on Mar. 6, 2014).

[cabal-b] Duncan Coutts and Ian Lynagh. The Haskell Cabal. URL: http://www.haskell.org/cabal/ (visited on Mar. 6, 2014).

[cabal-c] The Haskell Cabal. URL: http://www.haskell.org/cabal/download.html (visited on Mar. 19, 2014).

[CBS13] Koen Claessen, Björn Bringert, and Nick Smallbone. Hackage: QuickCheck: Automatic testing of Haskell programs. Version 2.6. Mar. 7, 2013. URL: http://hackage.haskell.org/package/QuickCheck-2.6 (visited on Mar. 6, 2014).

[chrome] Disabled Web Features - Google Chrome. Google Inc. URL: http://developer.chrome.com/apps/app_deprecated (visited on Mar. 6, 2014).

[claude] Bastian Holst, Julia Beck, Karl Balzer, Sandra Dylus, and Timo von Holtz. Wiki - Master-Projekt Sommersemester 2013 - Redmine. 2013. URL: https://redmine.ps.informatik.uni-kiel.de/projects/projekt-13s/wiki (visited on Mar. 6, 2014).

[cloud9] Cloud9 IDE | Your code anywhere, anytime. Cloud9ide. URL: https://c9.io/ (visited on Mar. 6, 2014).

[codemirror] Marijn Haverbeke. CodeMirror. URL: http://codemirror.net (visited on Mar. 6, 2014).

[codemirror-a] Marijn Haverbeke. Code Mirror: User Manual. *hint/show-hint.js*. URL: http://codemirror.net/doc/manual.html#addon_show-hint (visited on Mar. 11, 2014).

[codemirror-b] Marijn Haverbeke. CodeMirror: User Manual. *Writing CodeMirror Modes*. URL: http://codemirror.net/doc/manual.html#modeapi (visited on Mar. 6, 2014).

[codemirror-c] Marijn Haverbeke. CodeMirror: User Manual. *merge/merge.js*. URL: http://codemirror.net/doc/manual.html#addon_merge (visited on Mar. 6, 2014).

[comp] TJ Holowaychuk. component - modular javascript framework. URL: http://component.io/ (visited on Mar. 6, 2014).

[ender] Dustin Diaz, Jacob Thornton, and Rod Vagg. Ender - the no-library JavaScript library. URL: http://ender.var.require.io/ (visited on Mar. 6, 2014).

[fpcenter] FP Haskell Center - FP Complete. FP. URL: https://www.fpcomplete.com/business/fp-haskell-center/ (visited on Mar. 6, 2014).

[ghc] GHC: Download version 7.6.3. Version 7.6.3. URL: `https://www.haskell.org/ghc/download_ghc_7_6_3` (visited on Mar. 19, 2014).

[ghc-th] The GHC Team, ed. The Glorious Glasgow Haskell Compilation System User's Guide. *Template Haskell*. Version 7.6.3. Chap. 7. URL: `http://www.haskell.org/ghc/docs/7.6.3/html/users_guide/template-haskell.html` (visited on Mar. 6, 2014).

[github] GitHub. GitHub, Inc. URL: `https://github.com/` (visited on Mar. 6, 2014).

[hackage] Hackage: Introduction. URL: `http://hackage.haskell.org/` (visited on Mar. 6, 2014).

[hackage-a] Andy Gill and Edward Kmett. Control.Monad.Error.Class. Version 2.1.2. June 23, 2012. URL: `https://hackage.haskell.org/package/mtl-2.1.2/docs/Control-Monad-Error-Class.html#t:MonadError` (visited on Mar. 6, 2014).

[hackage-b] Andy Gill and Ross Paterson. Control.Monad.IO.Class. Version 0.3.0.0. Mar. 22, 2012. URL: `http://hackage.haskell.org/package/transformers-0.3.0.0/docs/Control-Monad-IO-Class.html#t:MonadIO` (visited on Mar. 20, 2014).

[hackage-c] Andy Gill and Ross Paterson. Control.Monad.Trans.Error. Version 0.3.0.0. Mar. 22, 2012. URL: `https://hackage.haskell.org/package/transformers-0.3.0.0/docs/Control-Monad-Trans-Error.html#t:ErrorT` (visited on Mar. 6, 2014).

[Han13] Tony Hannan. Hackage: mongoDB: Driver (client) for MongoDB, a free, scalable, fast, document DBMS. Version 1.4.4. Dec. 21, 2013. URL: `http://hackage.haskell.org/package/mongoDB-1.4.4` (visited on Mar. 10, 2014).

[hayoo] Timo B. Kranz, Sebastian M. Gauck, and Uwe Schmidt. Hayoo! - Haskell API Search. URL: `http://holumbus.fh-wedel.de/hayoo/` (visited on Mar. 6, 2014).

[hoogle]	Neil Mitchell. Hoogle. URL: http://www.haskell.org/hoogle/ (visited on Mar. 6, 2014).
[jam]	Caolan McMahon. Jam - The JavaScript package manager. URL: http://jamjs.org/ (visited on Mar. 6, 2014).
[jquery]	jQuery. The jQuery Foundation. URL: http://jquery.com/ (visited on Mar. 6, 2014).
[jquery-a]	jQuery.ajax() | jQuery API Documentation. The jQuery Foundation. URL: http://api.jquery.com/jQuery.ajax/ (visited on Mar. 10, 2014).
[jquery-b]	jQuery API Documentation. The jQuery Foundation. URL: http://api.jquery.com/ (visited on Mar. 6, 2014).
[jshint]	Anton Kovalyov. JSHint, a JavaScript Code Quality Tool. URL: http://www.jshint.com/ (visited on Mar. 6, 2014).
[jslint]	Douglas Crockford. JSLint: The JavaScript Code Quality Tool. URL: http://www.jslint.com/lint.html (visited on Mar. 6, 2014).
[koding]	Koding | A New Way For Developers To Work. Koding, Inc. URL: https://koding.com/ (visited on Mar. 6, 2014).
[light-a]	Chris Granger. Light Table. URL: http://www.lighttable.com/ (visited on Mar. 6, 2014).
[light-b]	Chris Granger. Chris Granger - Light Table. URL: http://www.chris-granger.com/lighttable/ (visited on Mar. 6, 2014).
[lodash]	John-David Dalton, Blaine Bublitz, Kit Cambridge, and Mathias Bynens. Lo-Dash. URL: http://lodash.com/ (visited on Mar. 6, 2014).
[mdn]	Web API Interface | MDN. *Synchronous and asynchronous requests*. Mozilla Developer Network. URL: https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest/Synchronous_and_Asynchronous_Requests (visited on Mar. 6, 2014).
[mongo]	MongoDB. Version 2.4.8. MongoDB, Inc. URL: http://www.mongodb.org/ (visited on Mar. 6, 2014).

[mongo-a] Durran Jordan. [SERVER-831] Positional Operator Matching Nested Arrays - MongoDB. Mar. 25, 2010. URL: `https://jira.mongodb.org/browse/SERVER-831` (visited on Mar. 18, 2014).

[mongo-b] MongoDB Manual 2.4.9. *Text Indexes*. MongoDB, Inc. URL: `http://docs.mongodb.org/v2.4/core/index-text/` (visited on Mar. 6, 2014).

[mongo-c] MongoDB Manual 2.4.9. *Search String Content for Text*. MongoDB, Inc. URL: `http://docs.mongodb.org/v2.4/tutorial/search-for-text/` (visited on Mar. 6, 2014).

[mongo-d] MongoDB Manual 2.4.9. *FAQ: MongoDB Fundamentals*. MongoDB, Inc. URL: `http://docs.mongodb.org/v2.4/faq/fundamentals/#does-mongodb-support-acid-transactions` (visited on Mar. 6, 2014).

[mongo-e] MongoDB Manual 2.4.9. *Install MongoDB*. MongoDB, Inc. URL: `http://docs.mongodb.org/v2.4/installation/` (visited on Mar. 19, 2014).

[msdn] open method (Internet Explorer). Microsoft Corporation. URL: `http://msdn.microsoft.com/en-us/library/ie/ms536648(v=vs.85).aspx` (visited on Mar. 6, 2014).

[npm] npm. npm, Inc. URL: `https://www.npmjs.org/` (visited on Mar. 6, 2014).

[orion] Orion. The Eclipse Foundation. URL: `http://www.eclipse.org/orion/` (visited on Mar. 6, 2014).

[OSu] Bryan O'Sullivan. Hackage: text: An efficient packed Unicode text type. URL: `http://hackage.haskell.org/package/text` (visited on Mar. 10, 2014).

[OSu13] Bryan O'Sullivan. Hackage: aeson: Fast JSON parsing and encoding. Version 0.6.2.1. Oct. 14, 2013. URL: `http://hackage.haskell.org/package/aeson-0.6.2.1` (visited on Mar. 6, 2014).

[purescript] Phil Freeman. PureScript. URL: `http://functorial.com/purescript/` (visited on Mar. 6, 2014).

[purescript-a] Phil Freeman. Introduction - PureScript 0.4.0 documentation. URL: `http://purescript.readthedocs.org/en/latest/intro.html#introduction` (visited on Mar. 6, 2014).

[Ros11] Dom Roselli. Why You Should Use XMLHttpRequest Asynchronously - WER Service - Site Home - MSDN Blogs. Microsoft Corporation. Aug. 3, 2011. URL: `http://blogs.msdn.com/b/wer/archive/2011/08/03/why-you-should-use-xmlhttprequest-asynchronously.aspx` (visited on Mar. 6, 2014).

[Snoa] Michael Snoyman. Hackage: persistent: Type-safe, multi-backend data serialization. URL: `http://hackage.haskell.org/package/persistent` (visited on Mar. 6, 2014).

[Snob] Michael Snoyman. Hackage: shakespeare: A toolkit for making compile-time interpolated templates. Version 1.0.5.1. URL: `http://hackage.haskell.org/package/shakespeare-1.0.5.1` (visited on Mar. 6, 2014).

[Snoc] Michael Snoyman. Hackage: yesod-bin: The yesod helper executable. URL: `http://hackage.haskell.org/package/yesod-bin` (visited on Mar. 6, 2014).

[Sno13] Michael Snoyman. Hackage: http-client: An HTTP client engine, intended as a base layer for more user-friendly packages. Version 0.2.0.3. Dec. 10, 2013. URL: `http://hackage.haskell.org/package/http-client-0.2.0.3` (visited on Mar. 6, 2014).

[typescript] Welcome to TypeScript. Microsoft Corporation. URL: `http://www.typescriptlang.org/` (visited on Mar. 6, 2014).

[volo] volo. The Dojo Foundation. URL: `http://volojs.org/` (visited on Mar. 6, 2014).

[Yan12] Edward Z. Yang. Modelling IO: MonadIO and beyond. Jan. 24, 2012. URL: `http://blog.ezyang.com/2012/01/modelling-io/` (visited on Mar. 6, 2014).

[yesod] Yesod Web Framework for Haskell. URL: `http://www.yesodweb.com/` (visited on Mar. 6, 2014).

Appendix A

Project Structure

The source code of the four packages introduced in Section 3.2 is stored in the Git repository of the research group Programming Languages and Compiler Construction: `https://git-ps.informatik.uni-kiel.de`. Each of the four packages has its own repository inside of the project `programming-in-the-cloud`:

`code-cloud-data` has the structure of a standard Haskell package. There is a `src` folder for all module sources and a `test` folder with all test code.

`code-cloud` is a standard *Yesod* project as described in Section 2.2.3. The `tests` directory contains some tests.

`code-cloud-interface` has the same structure as `code-cloud-data`, but misses tests.

`code-cloud-editor` is also a standard *Yesod* project. It is important to note that the editor view's JavaScript sources are stored in `static/js/editor`.

All of the repositories contain a cabal package file and a read me file. The original version of Claude in the `code-cloud` repository, is tagged as `masterproject-version`. All repositories are tagged with `masterthesis-jbra-version` to mark the version as it was when this thesis was submitted. The zipped packages are also included on the attached CD.

Appendix B

Installation Guide

This section will give instructions on how to setup an instance of Claude and the editor on a Linux machine.

B.1 Prerequisites

Installing the software requires compiling it from source. It is advised to do so on the target machine to avoid problems with missing libraries. Therefore, an up-to-date version of GHC and `cabal` has to be installed [ghc; cabal-c] on the target system. Instead of `cabal` one can also use `cabal-dev`. We will use `cabal-dev` from now on, though there should be no problems just exchanging it with `cabal`. When using `cabal` you may want to remove the sandbox parameters from the given commands.

We also need to ensure that *MongoDB* is installed [mongo-e]. Version 2.4.8 of *MongoDB* was used during development. To check if *MongoDB* is installed run to following command:

```
mongo --version
```

The output should display the installed *MongoDB* version.

In a next step, we need to clone the repository contents. We assume that everything is located in the directory `~/code-cloud/`. To clone all repositories to that directory execute the following commands:

```
cd ~/code-cloud/
git clone ssh://git@git-ps.informatik.uni-kiel.de:55055/\
programming-in-the-cloud/code-cloud-data.git
git clone ssh://git@git-ps.informatik.uni-kiel.de:55055/\
programming-in-the-cloud/code-cloud.git
```

```
git clone ssh://git@git-ps.informatik.uni-kiel.de:55055/\
programming-in-the-cloud/code-cloud-interface.git
git clone ssh://git@git-ps.informatik.uni-kiel.de:55055/\
programming-in-the-cloud/code-cloud-editor.git
```

Note, that we had to break the lines of the `git` commands, due to the limited width of a page.

Changes and additions to Claude are in the `extensions` branch of the `code-cloud` repository. So switch to that branch before proceeding:

```
cd ~/code-cloud/code-cloud/
git checkout extensions
```

B.2 Compilation

To compile and install the binaries it is advised to create a shell script. The following commands can be executed to compile everything:

```
cd ~/code-cloud/
cabal-dev --sandbox=./cabal-dev --force-reinstall --reinstall \
  install \
  ./code-cloud-data \
  ./code-cloud \
  ./code-cloud-interface \
  ./code-cloud-editor \
  blaze-html-0.6.1.2 cryptohash-0.9.1 shakespeare-js-1.1.4.1 \
  wai-extra-1.3.4.6 warp-1.3.8.4 yesod-1.2.4 random-1.0.1.1 \
  transformers-0.3.0.0 text-0.11.3.1 primitive-0.5.0.1 zlib-0.5.4.1 \
  shakespeare-css-1.0.6.6 semigroups-0.12.1
```

Explicit versions of some dependencies are given to ease installation and avoid cabal hell. It may be necessary to give further restrictions. We provide a complete list of all packages installed in our development sandbox on the attached CD.

If compilation worked ensure that the executables `code-cloud` and `code-cloud-editor` were correctly written to `~/code-cloud/cabal-dev/bin`.

B.3 Configuration

For Claude to work properly, we have to configure *MongoDB* to create a text index for its full-text search. This has to be done manually. The first possibility to activate the text index is by executing the *MongoDB* daemon with additional parameters:

```
mongod --setParameter textSearchEnabled=true
```

Another possibility is to use the *MongoDB* shell and enter the following command:

```
db.adminCommand( { setParameter : 1, textSearchEnabled : true } )
```

When the full-text search index is activated we still have to use the *MongoDB* shell to configure it further. Before we can configure we need to select the right database in the shell:

```
use Claude
```

Currently Claude uses the database called `Claude`. To create the indexes we have to execute the following command:

```
db.cloud.ensureIndex( {"$**": "text"},
    {name: "fts", default_language: "none"} )
```

We also have to ensure that *MongoDB* keeps an index for each object in the database:

```
db.cloud.ensureIndex( {"modules._id": 1} )
db.cloud.ensureIndex( {"modules.functions._id": 1} )
db.cloud.ensureIndex( {"modules.datatypes._id": 1} )
```

Once *MongoDB* is configured correctly we can configure Claude and the editor by editing their configuration files

▷ `~/code-cloud/code-cloud/config/settings.yml` and

▷ `~/code-cloud/code-cloud-editor/config/settings.yml`.

Inside of the `Production` section of these files, we have to modify the `approot` and `port`. The `approot` sets the applications root, that is used as prefix for all generated Uniform Resource Locators (URLs). Be sure that the editor and Claude have different ports set. Also be sure to include the port within the `approot`, if it is not the standard HTTP port. As an example for a correctly setup configuration section we can look at the following:

```
Production:
  approot: "http://m-049.informatik.uni-kiel.de:8080"
  port: 8080
  <<: *defaults
```

This is the configuration that was used for the internal test server.

The following section will assume that Claude is configured to run on port 8080 and the editor on port 8081.

B.4 Execution

Now that everything is set up correctly and the Claude and editor executables are ready we can start Claude:

```
cd ~/code-cloud/code-cloud/
../cabal-dev/bin/code-cloud Production --port 8080
```

The command to start the editor looks similar:

```
cd ~/code-cloud/code-cloud-editor/
../cabal-dev/bin/code-cloud-editor Production --port 8081
```

For both commands it is important to change into the correct working directory, because the servers search for their resources and static files in the working directory.

Both applications should now be accessible by opening their respective `approot` with a web browser.

It may be advisable to set both, the editor and Claude, up as a system service that is automatically started when the target system boots.

Appendix C

Contents of the Attached CD

All data on the attached CD represents the most current version of the software, from when this thesis was submitted. Every directory on the CD contains an `index.html`, which provides links to all important contents.

The attached CD contains the following directories:

`documentation/` The documentation directory contains the generated Haddock documentation of all four packages.

`packages/` The package directory contains a zipped version of each package. These packages contain the source code and project files.

`dependencies.html` Provides a list of all packages that were installed in the development sandbox.

`thesis.pdf` An electronic copy of this thesis.