Universität Bremen

Fachbereich 3: Mathematik und Informatik

# Diplomarbeit

## clide

### Eine webbasierte Entwicklungsumgebung für den interaktiven Theorembeweiser Isabelle


Martin Ring

Matrikel-Nr.221 590 8


16. Januar 2013

**1. Gutachter:** Prof. Dr. Christoph Lüth

**2. Gutachter:** Dr. Dennis Krannich

**Betreuer:** Prof. Dr. Christoph Lüth

**Martin Ring**

clide

Eine webbasierte Entwicklungsumgebung für den interaktiven Theorembeweiser Isabelle

Diplomarbeit, Fachbereich 3: Mathematik und Informatik

Universität Bremen, Dezember 2012

## Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig angefertigt, nicht anderweitig zu Prüfungszwecken vorgelegt und keine anderen als die angegebenen Hilfsmittel verwendet habe. Sämtliche wissentlich verwendete Textausschnitte, Zitate oder Inhalte anderer Verfasser wurden ausdrücklich als solche gekennzeichnet.

Bremen, den 16. Januar 2013

_____

Martin Ring

## Zusammenfassung

2010 wurde für den interaktiven Theorembeweiser Isabelle die Isabelle/Scala Schnittstelle eingeführt.**iscala**

# Inhaltsverzeichnis

Kapitel 1

# Einführung

Kapitel 2

# Grundlagen

Da die im Rahmen dieser Diplomarbeit entwickelte Anwendung vor allem die Verknüpfung sehr vieler Techniken, Konzepte und Standards aus normalerweise getrennten Bereichen der Informatik erfordert, ist es umso wichtiger diese für das Verständnis zu kennen. Im Folgenden sollen also die relevanten Begriffe geklärt werden.

## 2.1 HTML5

Für die Implementierung der Browseranwendung wird auf den aktuellen Entwurf des zukünfitgen HTML5 Standards zurückgegriffen.

Hypertext Markup Language (HTML) ist eine Sprache die der strukturierten Beschreibung von Webseiten dient. Die Sprache wurde in ihrer ursprünglichen Form von 1989-1992, lange vor dem sogenannten Web 2.0, von Wissenschaftlern des Europäischen Kernforschungsinstitut CERN entwickelt. Sie war der erste nicht proprietäre globale Standard zur digitalen übertragung von Strukturierten Dokumenten. Die Sprache HTML an sich ist nicht geeignet um dynamische Inhaltem wie sie heute praktisch auf allen modernen Webseiten vorkommen, zu beschreiben.

Der heutige HTML5-Standard geht weit über die Sprache HTML selbst hinaus und umfasst vor allem auch die Scriptsprache JavaScript und die darin verfügbaren Bibliotheken, sowie das sogenannte Document Object Model (DOM), auf welches in Scripten zugegriffen werden kann um den angezeigten Inhalt dynamisch zu verändern.

### 2.1.1 Dokumenobjektmodell

Das DOM dient bla bla...

### 2.1.2 Cascading Stylesheets

Cascading Style Sheets (CSS) ist eine Sprache die der Definition von Stilen bzw. Stilvorlagen für die Anzeige von Webinhalten dient.

Durch die Trennung von HTML und CSS wird erreicht, dass HTML-Dokumente sich auf den Inhalt einer Seite beschränken, während alle die grafische Anzeige belangenden Aspekte in die sogenannten Stylesheets in CSS Dateien ausgelagert werden.

#### 2.1.2.1 LESS

CSS hat einige Einschränkungen welche die Arbeit damit erschweren:

- Es ist nicht möglich Variablen zu definieren um Eigenschaften, welche an vielen Stellen vorkommen nur einmal zu definieren.
- Es fehlen Funktionsdefinitonen um ähnliche oder abhängige Definitionen zusammenzufassen und zu parametrisieren.
- Die Hierarchie einer CSS-Datei ist flach obwohl die Definitionen geschachtelt sind. Das reduziert die lesbarkeit der Dateien.
- Wenn aus Gründen der Übersichtlichkeit CSS in mehrere Dateien aufgeteilt werden, müssen alle Dateien einzeln geladen werden, was zu längeren Ladezeiten führt.

*LESS* ist eine Erweiterung von CSS welche unter anderem Variablen- und Funktionsdefinitionen, verschachtelte Definitionen sowie Dateiimports erlaubt. Damit werden die oben genannten Einschränkungen von CSS aufgehoben.

### 2.1.3 JavaScript

JavaScript ist eine dynamisch typisierte, klassenlose objektorientierte Scriptsprache und

#### 2.1.3.1 CoffeeScript

*CoffeeScript* ist eine

### 2.1.4 HTTP

HTTP (Hypertext Transport Protocol)

#### 2.1.4.1 AJAX

AJAX (Asynchronous JavaScript and XML) ist keine Bibliothek und auch kein Standard sondern ein sehr weit verbreitetes Konzept zur Übertragung von Daten zwischen Browser und Webserver. Hierbei wird das JavaScript Objekt `XMLHttpRequest` verwendet um während der Anzeige einer Webseite

#### 2.1.4.2 WebSockets

Websockets sind ein in HTML5 neu eingeführter Standard zur bidirektionalen Kommunikation zwischen Browser und einem Webserver. Hierbei wird anders als bei AJAX eine direkte TCP-Verbindung hergestellt. Diese Verbindung kann sowohl von Browser- als auch von Serverseite aus gleich verwendet werden. Das macht es unnötig, wie bei AJAX wiederholte Anfragen oder Anfragen ohne Zeitbegrenzung zu stellen um Informationen vom Server zu erhalten wenn diese Verfügbar werden. Ein weiterer Vorteil gegenüber HTTP-Anfragen ist, dass durch die direkte permanente Verbindung kein Nachrichtenkopf mehr nötig ist. Das macht es deutlich effizienter viele kleine Nachrichten zu versenden.

### 2.1.5 JavaScript-Bibliotheken

Über den HTML5 Standard hinaus werden für die strukturierung der Anwendung einige JavaScript-Bibliotheken verwendet, welche im folgenden kurz erläutert werden.

#### 2.1.5.1 jQuery

Die Bibliothek *jQuery* ist ein defacto Standard in der Webentwicklung. In erster Linie erleichtert es den Zugriff auf das *DOM*.

#### 2.1.5.2 Backbone

*Backbone* ist eine Bibliothek, die der Strukturierung von JavaScript Anwendungen dient.

#### 2.1.5.3 RequireJS

Da JavaScript von Haus aus keine Möglichkeit der Modularisierung bietet, komplexe Anwendungen jedoch ohne Modularisierung kaum wartbar bleiben, haben sich unterschiedliche Lösungsansätze für dieses Problem entwickelt. Einer der umfassendsten ist die Bibliothek *RequireJS*.

Mit der Funktion `define` können Module in Form von Funktionsdefinitionen definiert werden. Alle lokalen Variablen in dem Modul sind anders als bei normalen Scripten außerhalb nicht mehr sichtbar, da sie innerhalb einer Funktion definiert wurden. Das Funktionsergebnis ist das was nach außen Sichtbar ist. Das kann ein beliebiges Objekt (also auch eine Funktion) sein.

Die so definierten Module können Abhängigkeiten untereinander spezifizieren indem der `define`-Funktion eine Liste von Modulen übergeben wird, die das aktuelle Modul benötigt. Die *RequireJS*-Bibliothek sorgt dann dafür, dass diese Module geladen wurden bevor das aktuelle Modul

## 2.2 Scala

Die Programmiersprache Scala ist eine an der École polytechnique fédérale de Lausanne von einem Team um Martin Odersky entwickelte statisch typisiserte, objektorientierte, funktionale Sprache**scala** In Scala entwickelte Programme laufen sowohl in der *Java Virtual Machine* (JVM) als auch in der *Common Language Runtime* (CLR). Die Implementierung für die CLR hängt jedoch stark hinterher und ist für diese Arbeit auch nicht von Interesse. Die aktuelle Sprachversion ist Scala 2.10 welche auch im Rahmen dieser Arbeit verwendet wird.

Scala versucht von Anfang an den Spagat zwischen funktionaler und objektorientierter Programmierung herzustellen. Hierbei ist es sowohl möglich rein objektorientierten als auch rein funktionalen Code zu schreiben. Dadurch entstehen für den Programmierer sehr große Freiheitsgrade und es ist beispielsweise auch möglich imperativen und funktionalen Code zu mischen. Diese Freiheit erfordert eine gewisse Verantwortung von Seiten des Programmierers um lesbaren und wartbaren Code zu erstellen.

### 2.2.1 Sprachkonzepte

#### 2.2.1.1 Implizite Parameter

Implizite Parameter werden in Scala verwendet um Parameter die sich aus dem Kontext eines Funktionsaufrufs erschließen können nicht explizit übergeben zu müssen. Eine Funktion `f` besitzt

hierbei zusätzlich zu den normalen Parameterlisten auch eine implizite Parameterliste:

```
f(a: Int)(implicit x: T1, y: T2)
```

In dem Beispiel hat die Funktion einen normalen Parameter `a` und zwei implizite Parameter `x` und `y`. Der Compiler sucht bei einem Funktionsaufruf, welcher die beiden oder einen der impliziten Parameter nicht spezifiziert nach impliziten Definitionen vom Typ `T1` bzw. `T2`. Diese Definitionen werden im aktuell sichtbaren Scope nach bestimmten Prioritäten gesucht. Dabei wird zunächst im aktuellen Objekt, dann im zu den typen `T1` und `T2` gehörenden objekten und dann in den importierten Namensräumen gesucht. Implizite definitionen haben die Form `implicit def/val/var x: T = ...` wobei der name `x` keine Rolle spielt.

#### 2.2.1.2 Implizite Konversionen

In Scala existiert das Konzept der impliziten Konversionen. Hierbei werden bei Typfehlern zur Kompilierzeit Funktionen mit dem Modifizierer `implicit` gesucht, die den gefundenen Typen in den nötigen Typen umwandeln können gesucht. Die Priorisierung ist genauso wie bei impliziten Parametern. Ein Beispiel:

```
implicit def t1tot2(x: T1): T2 = ...
def f(x: T2) = ...

val x: T1 = ...
f(x)
```

In dem Beispiel wird eine implizite konversion von `T1` nach `T2` definiert. Bei dem Aufruf `f(x)` kommt es zu einem Typfehler, weil `T2` erwartet und `T1` übergeben wird. Dieser Typfehler wird gelöst indem vom Compiler die implizite Konversion eingesetzt wird. Der Aufruf wird also intern erweitert zu `f(t1tot2(x))`.

#### 2.2.1.3 Typklassen

Mit Hilfe von impliziten Definitionen ist es möglich das aus der Sprache Haskell bekannte Konzept der Typklassen in Scala nachzubilden.

Eine Typklasse bietet die möglichkeit Ad-hoc-Polymorphie zu implementieren. Damit ist es möglich ähnlich wie bei Schnittstellen Funktionen für eine ganze Menge von Typen bereitzustellen. Diese müssen jedoch nicht direkt von den Typen implementiert sein und können auch Nachträglich beispielsweise für Typen aus fremden Bibliotheken definiert werden.

In Scala werden Typklassen als generische abstrakte Klassen oder Traits implementiert.

Instanzen der Typklassen sind implizite Objektdefinitionen welche für einen spezifischen Typen die Typklasse bzw. die abstrakte Klasse implementieren.

Eine Funktion für eine bestimmte Typklasse kann durch eine generische Funktion realisiert werden. Diese ist dann über einen oder mehrere Typen parametrisiert und erwartet als implizites Argument eine Instanz der Typklasse für diese Typen, also eine implizite Objektdefinition. Wenn diese im Namensraum existiert, wird sie automatisch vom Compiler eingesetzt.

Dieses Konzept ist vor allem sehr Hilfreich wenn es darum geht fremde Bibliotheken zu erweitern.

#### 2.2.1.4 Dynamische Typisierung

Seit Scala 2.10 ist es möglich Funktionsaufrufe bei Typfehlern Dynamisch zur Laufzeit auflösen zu lassen. Damit die Typsicherheit nicht generell verloren geht ist es nötig den Trait `Dynamic` zu importieren um einen Typ als Dynamisch zu deklarieren. Wenn die Typüberprüfung dann bei einem Aufruf fehlschlägt wird der Aufruf auf eine der Funktionen `applyDynamic`, `applyDynamicNamed`, `selectDynamic` und `updateDynamic` abgebildet:

```
x.method(”arg”) => x.applyDynamic(”method”)(”arg”)
x.method(x = y) => x.applyDynamicNamed(”method”)((”x”, y))
x.method(x = 1, 2) => x.applyDynamicNamed(”method”)((”x”, 1), (””, 2))
x.field         => x.selectDynamic(”field”)
x.variable = 10 => x.updateDynamic(”variable”)(10)
x.list(10) = 13 => x.selectDynamic(”list”).update(10, 13)
x.list(10)      => x.applyDynamic(”list”)(10)
```

### 2.2.2 Akka

**actors**

`Akka` ist eine Implementierung des Aktoren-Modells.

### 2.2.3 Play Framework

Das *Play Framework* ist ein Framework zur Entwicklung von Webanwendungen auf der JVM mit einer speziellen API für Scala. Play ist ein sehr effizientes Framework welches auf Akka auf-

baut um hohe Skalierbarkeit zu gewährleisten. Damit wird es leichter verteilte hochperformante Webanwendungen zu realisieren.

#### 2.2.3.1 LESS

Play ermöglicht es die Stylesheet-Sprache *LESS* zu verwenden ohne, dass diese auf Browserseite unterstützt werden muss. Hierfür werden die in *LESS* definierten Stylesheet auf Serverseite in *CSS* übersetzt und dem Browser zur Verfügung gestellt.

Dafür müssen die Dateien an einem vorher konfigurierten Ort liegen. Nach dem übersetzen werden sie an der selben Stelle zur Verfügung gestellt wie normale *CSS* Dateien.

#### 2.2.3.2 CoffeeScript

Genauso wie für *LESS* existiert in Play die Serverseitige Unterstützung für *CoffeeScript*. Die in *CoffeeScript* geschriebenen Dateien werden ebenfalls an gleicher Stelle wie normale *JavaScript*-Dateien dem Browser als *JavaScript* zur Verfügung gestellt.

#### 2.2.3.3 RequireJS

Die *RequireJS* Bibliothek bietet die Möglichkeit den *JavaScript*-Code für den Produktiveinsatz zu optimieren. Dafür gibt es das sogenannte *r.js*-Script welches unter andem alle Abhängigkeiten zusammenfasst und den Code durch das Entfernen von Whitespaces und Kommentaren sowie dem Umbenennen von Variablennamen verkürzt. Zur Entwicklungszeit ist diese nicht mehr lesbare Code nicht erwünscht. Deswegen bietet Play eine integrierte Version von RequireJS, welche automatisch den lesbaren Code zur Entwicklungszeit bereitstellt, im Produktiveisatz jedoch den optimierten.

#### 2.2.3.4 Iteratees

#### 2.2.3.5 Websockets

Websockets werden direkt von Play unterstützt. ...

## 2.3 Isabelle

**isabelle**

Kapitel 3

# Entwurf

Auf Grund der Komplexität und dem unvermeidbaren Rechenaufwand, welcher ein Theorembeweiser mitbringt, ist es aus aktueller Sicht nahezu ausgeschlossen diese Arbeit zu größeren Teilen im Browser zu verwirklichen. Die einzige von allen größeren Browsern unterstützte Scriptsprache ist zur Zeit immernoch *JavaScript*, welche vor allem auf Grund der dynamischen Typisierung und der fehlenden Paralellisierbarkeit um einige Faktoren langsamer ausgeführt wird, als nativer Code.

Ein besonderer Vorteil, den die Anwendung gegenüber bisherigen Lösungen bringen soll, ist die Mobilität. Das bedeutet, dass es von jedem Rechner mit Internetzugang und einem modernen Webbrowser aus möglich sein soll, die Anwendung zu nutzen und auf eventuell bereits zu einem früheren Zeitpunk an einem anderen Ort erstellten Theorien zugreifen zu können. Damit wird klar, dass die Projekte und Theorien nicht lokal auf den einzelnen Rechnern verwaltet werden können, sondern an einer zentralen von überall erreichbaren Stelle gespeichert sein müssen. Die Entscheidung zu einem Client-Server Modell ergibt sich bei einer Webanwendung ohnehin automatisch.

Da es sich bei der Webanwendung um eine Entwicklungsumgebung handelt, welche insbesondere durch Echtzeitinformationen einen Mehrwert bringen soll, ist einer der wichtigsten Aspekte des Entwurfs die effiziente Kommunikation zwischen Server und Browser. Da die Kommunikation bei einer Webanwendung generell sehr Zeitaufwändig ist - abhängig von der Internetanbindung kann es zu großen Verzögerungen kommen - muss abgewogen werden, welche Arbeit im Browser und welche auf dem Server erledigt werden soll. Als illustratives Beispiel kann das Syntax-Highlighting genannt werden: Isabelle verfügt über eine innere und eine äußere Syntax, die sich im analytischen Aufwand stark unterscheiden. Während die äußere Syntax relativ leicht zu parsieren ist und dabei bereits viele Informationen liefert, ist die innere Syntax sehr komplex, flexibel und stark vom jeweiligen Kontext abhängig. Somit liegt es nahe, das Syntaxhighlighting aufzuteilen: Um Übertragungskapazität zu sparen kann das Highlighting der äußeren Syntax bereits im Browser stattfinden. Die feiner granulierten Informationen aus der inneren Syntax können dann auf dem Server ermittelt werden und mit kurzer Verzögerung in die Darstellung integriert werden.

## 3.1 Server

Der Webserver muss neben den normalen Aufgaben eines Webservers, wie der Bereitstellung der Inhalte, der Authentifizierung der Benutzer sowie der Persitierung bzw. Bereitstellung der Nutzerspezifischen Daten (In unserem Fall Sitzungen / Theorien), auch eine besondere Schnittstelle für die Arbeit mit den Theorien bereitstellen. Vom Browser aus muss es möglich sein, die einzelnen Theorien in echtzeit zu bearbeiten sowie Informationen über Beweiszustände bzw. Fehler als auch über die Typen, bzw. Definitionen von Ausdrücken zu erhalten.

### 3.1.1 Wahl des Webframeworks

Für die Realisierung des Webservers wählen wir das *Play Framework*[1] in Version 2.1. Da wir die Isabelle/Scala Schnittstelle nutzen, liegt es nahe ein Webframework in Scala zu nutzen um den Aufwand für die Integration gering zu halten. Als Alternative existiert das *Lift Webframework*[2] welches allerdings auf Grund des Rückzugs von David Pollack aus der Entwicklung seit einiger Zeit nicht mehr geordnet weiter entwickelt wird und zudem für unsere Zwecke überdimensioniert ist. Da die Webanwendung eher unkonventionelle Anforderungen an den Server hat, nutzen die meisten Funktionen von *Lift* uns nicht. *Play* ist hingegen vorallem auf hohe Performance und weniger auf die Lösung möglichst vieler Anwendungsfälle ausgelegt, womit es für unsere Zwecke interessanter bleibt.

### 3.1.2 Authentifizierung

Die Authentifizierung soll in diesem Projekt bewusst einfach gehalten werden, da es sich hierbei um eine Nebensächlichkeit handelt, welche ohne weitere Probleme aufgerüstet werden kann. Wir beschränken uns daher auf die Möglichkeit sich mit einem Benutzernamen sowie einem dazugehörigen Passwort einzuloggen, welche dann mit einer Konfigurationsdatei auf dem Server abgeglichen wird. Wir können dann auf die Möglichkeit des Play Frameworks

### 3.1.3 Persistenz

Auch bei der Datenpersistenz

[1] http://www.playframework.org
[2] http://www.liftweb.org

### 3.1.4 Isabelle/Scala integration

## 3.2 Kommunikation

## 3.3 Client

### 3.3.1 Benutzeroberfläche

### 3.3.2 Client-Modell

### 3.3.3

Kapitel 4

# Implementierung

Anhang A

# Appendix

## A.1 Abbildungsverzeichnis

## A.2 Tabellenverzeichnis