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, Januar 2013

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.[Wen10]

Inhaltsverzeichnis

Inhaltsverzeichnis i

1 Einführung **1**

1.1 Motivation 1

1.2 Aufgabenstellung 1

1.3 Anmerkungen 1

1.4 1

2 Grundlagen **3**

2.1 HTML5 3

2.1.1 Dokumenobjektmodell 3

2.1.2 Cascading Stylesheets 4

2.1.2.1 LESS 4

2.1.3 JavaScript 4

2.1.3.1 CoffeeScript 4

2.1.4 HTTP 4

2.1.4.1 AJAX 5

2.1.4.2 WebSockets 5

2.1.5 JavaScript-Bibliotheken 5

2.1.5.1 jQuery 5

2.1.5.2 Backbone 5

2.1.5.3 RequireJS 6

2.2 Scala 6

2.2.1 Sprachkonzepte 6

2.2.1.1 Implizite Parameter 7

2.2.1.2 Implizite Konversionen 7

2.2.1.3 Typklassen 7

2.2.1.4 Dynamische Typisierung 8

2.2.2 Akka 8

2.2.3 Play Framework 9

2.2.3.1 Iteratees . . . 9
2.2.3.2 Websockets . . . 9
2.3 Isabelle . . . 9
2.3.1 Asynchrones Beweisen . . . 10
2.3.2 Isabelle/Scala . . . 10
3 Anforderungen und Entwurf **11**
3.1 Server . . . 12
3.1.1 Wahl des Webframeworks . . . 12
3.1.2 Authentifizierung . . . 12
3.1.3 Persistenz . . . 13
3.1.4 Isabelle/Scala Integration . . . 13
3.2 Kommunikation . . . 13
3.2.1 WebSockets . . . 14
3.2.2 Protokoll . . . 14
3.3 Client . . . 15
3.3.1 Browserkompatibilität . . . 15
3.3.2 Benutzeroberfläche . . . 15
3.3.2.1 Die Editor-Komponente . . . 15
3.3.3 Client-Modell . . . 16
3.3.4 . . . 16
4 Implementierung **17**
4.1 ScalaConnector / JSConnector . . . 17
A Appendix **19**
A.1 Abbildungsverzeichnis . . . 19
A.2 Tabellenverzeichnis . . . 19
A.3 Literatur . . . 19
A.4 Liste der Abkürzungen . . . 20

Kapitel 1

Einführung

1.1 Motivation

1.2 Aufgabenstellung

1.3 Anmerkungen

1.4

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 die relevanten Begriffe kurz geklärt werden.

2.1 HTML5

Für die Implementierung der Browseranwendung wird auf den aktuellen Entwurf des zukünfitgen HTML5 Standards zurückgegriffen. [Smi12]

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 (JS) 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

JS 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 JS-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 JS-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 JS 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[Mar11]. 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

Im folgenden sollen die für diese Arbeit relevanten Konzepte der Sprache Scala kurz vorgestellt werden. Dabei verzichten wir allerings auf Grundlagen wie Objektorientierung und Funktionale Programmierung, sowie bereits aus Java bekannte Konzepte, da diese bereits im Gundstudium der Informatik abgedeckt wurden.

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 um eigene Funktionen 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 markieren. Wenn die Typüberprüfung dann bei einem Aufruf auf einem als Dynamisch markierten Objekt fehlschlägt wird der Aufruf auf eine der Funktionen `applyDynamic`, `applyDynamicNamed`, `selectDynamic` und `updateDynamic` abgebildet. Diese Übersetzung geschieht nach dem Muster in **??**.

```
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

[Hal12]

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

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.

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 JS zur Verfügung gestellt.

Die *RequireJS* Bibliothek bietet die Möglichkeit den JS-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.

2.2.3.1 Iteratees

2.2.3.2 Websockets

Websockets werden direkt von Play unterstützt. ...

2.3 Isabelle

Isabelle ist ein generisches System welches vor allem zum interaktiven Beweisen von Theoremen unter der Nutzung von Logik höherer Ordnung (HOL) eingesetzt wird. Isabelle ist in Standard

ML (SML) implementiert und stark davon abhängig. [al12]

2.3.1 Asynchrones Beweisen

Seit Version 2009 von Isabelle wurde es möglich Beweis-Dokumente, bzw. Theorien nebenläufig zu überprüfen. Das macht es realistisch Echtzeitinformationen, so wie sie eine IDE liefern sollte verfügbar zu machen.

2.3.2 Isabelle/Scala

Seit 2010 existiert mit Isabelle/Scala eine neue Schnittstelle zu Isabelle welche auf der Sprache Scala basiert. Isabelle/Scala stellt eine API zur Arbeit mit Isabelle bereit, welche die zur Nutzung relevanten Teile der SML Implementierung in Scala abbilden. [Wen10]

Über statisch typisierte Methoden können die Dokumente modifiziert werden. Dafür wurde ein internes XML-Basiertes Protokoll eingeführt, welches die Scala API mit der SML API verknüpft. ÜDementsprechend sind auch die Informationen welche von Isabelle geliefert werden typisiert. Das macht Isabelle/Scala in der Nutzung recht robust, da ein Großteil der Fehler bereits zur Übersetzungszeit gefunden werden kann.

Isabelle/Scala wurde für und zusammen mit der Anwendung Isabelle/jEdit entwickelt. JEdit wurde unter anderem deswegen gewählt, weil es eine sehr einfache API hat und somit das Projekt nicht zu sehr auf den Editor konzentriert ist.

Kapitel 3

Anforderungen und 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 JS, 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 mittels JS 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,
- Informationen über Beweiszustände bzw. Fehler zu erhalten,
- als auch Informationen über die Typen, bzw. Definitionen von Ausdrücken zu erhalten.

All diese Informationen müssen zuvor Serverseitig aufbereitet und bereitgestellt werden. Dabei ist es aus Sicht der Perfomanz wichtig, unnötige Informationen zu eliminieren und die Daten zu komprimieren.

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* nichts. Das *Play* Webframework 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 zur sicheren Verwaltung von Session-Bezogenen Daten zurückgreifen um die Anmeldung aufrechtzuerhalten.

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

3.1.3 Persistenz

Als Besonderheit bei der Datenpersistenz sind die serverseitig zu verwaltenden Theorien zu nennen. Da jeder Benutzer eine von allen anderen Benutzern unabhängige Menge von Projekten mit Theorien besitzt, also eine hierarchische Struktur besteht, spricht nichts dagegen, die Daten Serverseitig im Dateisystem zu verwalten. Somit ist auch eine eventuelle spätere Integration eines Versionsverwaltungssystems möglich. Da über diese Daten hinaus nur wenige Informationen vom Server verwaltet werden müssen, ist die Einrichtung und Anbindung einer Datenbank nicht von Nöten.

3.1.4 Isabelle/Scala Integration

3.2 Kommunikation

Da viele Daten in hoher Frequenz übertragen werden müssen, (Nach jeder Veränderung des Dokuments muss der Server informiert werden, der dann zu unbestimmten Zeitpunkten in mehreren Schritten die Zustands- Informationen zurücksendet) ist eine normale Datenübertagung wie bei Webapplikationen üblich über Asynchronous JavaScript and XML (AJAX) bzw. Hypertext Transfer Protocol (HTTP)-Anfragen nicht gut geeignet: Bei normalem HTTP ist es zum einen immer nötig auf Browser Seite eine Anfrage zu stellen um Informationen vom Server zu erhalten, zum anderen hat jede Anfrage und jede Antwort zusätzlich einen Header, welcher mindestens einige hundert Bytes groß ist.

Als Worst-Case Beispiel kann das Entfernen von Kommandos aus dem Dokumenten-Modell betrachtet werden: Um das Modell eines Dokuments auf Server und Client synchron zu halten müssen ab und zu Nachrichten vom Server gesendet werden, welche signalisieren, dass ein Kommando aus dem Modell entfernt wurde. Diese Nachricht enthält als Information die eindeutige ID des Kommandos. Bei der ID handelt es sich um eine 64-Bit Zahl. Zusätzlich dazu muss signalisiert werden, um welche Aktion es sich eigentlich handelt. Dafür reichen bei der überschaubaren Anzahl an möglichen Aktionen weitere 4 Byte mehr als aus. Das bedeutet, die eigentlichen Informationen die für diese Aktion relevant sind belaufen sich auf höchstens 12 Byte. Würde diese Aktion über HTTP laufen müsste zunächst eine Anfrage gestellt werden

```
GET /user/project/file.thy/remove-command HTTP/1.1
Host: www.clide.net
```

Diese Anfrage allein ist bereits 70 Zeichen lang ohne dass überhaupt relevante Informationen übertragen wurden. Die minimale Antwort sähe dann in etwa so aus:

```
HTTP/1.1 200 OK
Server: Apache/1.3.29 (Unix) PHP/4.3.4
Content-Length: 12
Content-Language: de
Connection: close
Content-Type: text/html

178
```

Das sind zusammen über 250 Zeichen um zu signalisieren, dass Kommando 178 entfernt werden soll. Damit wurde die Information um den Faktor 30 aufgeblasen. Zusätzlich kommt es zu Verzögerungen durch die zusätzlichen Anfragen.

3.2.1 WebSockets

Die im HTML5-Standard eingeführten WebSockets sind die ideale Lösung für dieses Problem. Bei WebSockets wird eine vollduplex Verbindung über TCP aufgebaut, welche ohne den HTTP-Overhead auskommt und lediglich ein Byte pro Nachricht benötigt um zu signalisiern, dass eine Nachricht endet. Außerdem ist es durch die duplex Verbindung möglich sogenannte Server-Pushes wie in dem vorangegangenen Beispiel ohne vorheriges Polling bzw. eine verzögerte Antwort auf eine Anfrage zu realisieren.

Dadurch, dass WebSockets ein relativ neues Konzept bilden, werden durch deren Anwendung die meisten älteren Browser von der Benutzung der Webapplikation ausgeschlossen. Ein Fallback auf HTTP wäre zwar relativ leicht zu implementieren, aber in der Benutzung kaum akzeptabel, da sich die zusätzlichen Verzögerungen bei teilweise weit über 1000 Nachrichten pro Minute so negativ auf die Ausführungsgeschwindigkeit auswirken würden, dass ein produktives Arbeiten mit dem System nicht mehr möglich wäre. Aktuell werden WebSockets von WebKit-Basierten Browsern (Google Chrome, Safari) sowie dem Internet Explorer in der jeweils aktuellsten Version unterstützt. Mozilla Firefox erwartet einen nicht Standard-Konformen Aufruf über `MozWebsocket` statt `WebSocket`. Da WebSockets allerdings so unabdingbar sind wird auf die Kompatibilität mit älteren Browsern verzichtet. (Siehe 3.3.1)

3.2.2 Protokoll

Für das Protokoll wurde zunächst der Einfachheit halber JavaScript Object Notation (JSON) gewählt mit der Möglichkeit im Hinterkopf, es zu einem späteren Zeitpunkt leicht duch das

Tabelle 3.1 Kompatibilität der gängigsten Browser mit den Verwendeten Standards

	Chrome	**Safari**	**IE**	**Firefox**	**Opera**
WebSockets	14.0	6.0	10.0	11.0	12.1
History API	5.0	5.0	10.0	4.0	11.5
WebSockets	4.0	4.0	10.0	3.5	10.6
CSS Transitions	4.0	3.1	10.0	4.0	10.5

komprimierte Binary JSON (BSON)-Protokoll zu ersetzen. Um das möglich zu machen wird von WebSockets abstrahiert, (Siehe 4.1) damit das Protokoll ausgetauscht werden kann.

3.3 Client

3.3.1 Browserkompatibilität

Auf Grund der in 3.2.1 beschriebenen Notwendigkeit, kann auf WebSockets nicht verzichtet werden. Damit sind die meisten älteren Browser nicht mit der Anwendung kompatibel.

Aus Tabelle 3.1 ist zu entnehmen, dass alle weiteren in der Anwendung benutzten Standards eine geringere oder die gleicher Anforderung an die Aktualität des Browsers haben. Da WebSockets ein sehr neues Konzept sind, welches notwendig ist dienen sie als Orientierung: Alle Features, welche von jedem Browser, der WebSockets unterstützt, auch unterstützt werden, dürfen verwendet werden. Alle anderen schließen wir aus, da sonst die Zahl der potentiellen Nutzer weiter eingeschränkt würde. Die Anwendung sollte damit im Standardbrowser auf allen Systemen mit einem aktuellen Betriebssystem (Windows 8, Ubuntu 12.10, OpenSUSE 12.2, OS X 10.8.2), sowie auf dem iPad, und aktuellen Windows RT Tablets benutzbar. Bei der Entwicklung wurde jedoch besonderes Augenmerk auf WebKit-basierte Browser, insbesondere Google Chrome gelegt und einige der anderen genannten Systeme sind ungetestet und damit ohne Gewähr.

3.3.2 Benutzeroberfläche

3.3.2.1 Die Editor-Komponente

Die wichtigste Benutzerkomponente einer Entwicklungsumgebung ist der Text-Editor. Ein Editor für Isabelle-Code hat hierbei besondere Anforderungen: Während in der Praxis bislang nur rudimentäre Unterstützung für die Darstellung von Isabelle-Sonderzeichen und insbesondere von

Sub und Superskript existierte, hat Isabelle/jEdit bereits eine stärkere Integration dieser eigentlich recht essentiellen Visualisierungen eingeführt. Da bei der HTML-Darstellung kaum Grenzen gesetzt sind und sich CSS-Formatierung sehr leicht dazu benutzen lässt bestimmte Text-Inhalte besonders darzustellen, ist es klar, dass unsere Entwicklungsumgebung an dieser Stelle besonders glänzen soll.

In einem ersten Prototypen war es möglich eine JS-Komponente zu entwickeln, welche es zuließ, Isabelle-Code zu bearbeiten, sodass Sub- und Superskript sowie

3.3.3 Client-Modell

3.3.4

Kapitel 4

Implementierung

4.1 ScalaConnector / JSConnector

Anhang A

Appendix

A.1 Abbildungsverzeichnis

A.2 Tabellenverzeichnis

3.1 Kompatibilität der gängigsten Browser mit den Verwendeten Standards 15

A.3 Literatur

[al12] Makarius Wenzel et al. *The Isabelle/Isar Reference Manual*. http://isabelle.in.tum.de/doc/isar-ref.pdf, 2012.

[Fla97] David Flanagan. *JavaScript: The Definitive Guide*. Second. O'Reilly & Associates, Jan. 1997. ISBN: 1-56592-234-4. URL: *http://www.ora.com/catalog/jscript2/noframes.html*.

[Hal12] Philipp Haller. *Actors in Scala*. 1. Aufl. Artima, 2012. ISBN: 0981531652.

[Hic11] Ian Hickson. *The WebSocket API*. World Wide Web Consortium, Candidate Recommendation CR-websockets-20111208. Dez. 2011.

[Mar11] Bill Venners Martin Odersky Lex Spoon. *Programming In Scala*. 2. Aufl. Artima, 2011. ISBN: 0981531644.

[Smi12] M. K. Smith. *HTML: The Markup Language (an HTML language reference)*. W3C Working Draft (work in progress). http://www.w3.org/TR/2012/WD-html-markup-20121025/. W3C, 2012.

[Wen10] Makarius Wenzel. *Asynchronous Proof Processing with Isabelle/Scala and Isabelle/jEdit*. Techn. Ber. Université Paris Sud 11, LRI, Orsay France, 2010.

A.4 Liste der Abkürzungen

AJAX Asynchronous JavaScript and XML, S. 13

BSON Binary JSON, S. 15

CSS Cascading Style Sheets, S. 4, 16

DOM Document Object Model, S. 3

HTML Hypertext Markup Language, S. 3, 4, 14, 16
HTTP Hypertext Transfer Protocol, S. 13

JS JavaScript, S. 3–5, 9, 11, 16
JSON JavaScript Object Notation, S. 14
JVM Java Virtual Machine, S. 6

SML Standard ML, S. 9