



Universität Bremen

Fachbereich 3: Mathematik und Informatik

Diplomarbeit

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

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 wesentlich 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 sowie die Beispielanwendung Isabelle/jEdit als Entwicklungsumgebung für Isabelle entwickelt. Dabei fiel die Wahl auf jEdit, weil der Aufwand für die Integration gering gehalten werden sollte. In diesem Projekt konzentrieren wir uns nun auf die Ausgestaltung einer neuartigen Entwicklungsumgebung. Dabei werden modernste Web-Techniken miteinander verknüpft und dadurch eine ganz neue Art des Umgangs mit Beweisdokumenten geschaffen. Durch die Kombination des Webframeworks Play, dem aktuellen Entwurf des HTML5-Standards und dem Isabelle/Scala-Layer als Schnittstelle zur mächtigen Isabelle-Plattform entsteht ein Werkzeug, das verwendet werden kann, um ohne jeglichen Konfigurationsaufwand für den Nutzer von einem beliebigen Rechner mit einem aktuellen Browser Beweisdokumente bearbeiten zu können, ohne dass auf dem Rechner die gesamte Isabelle-Plattform installiert sein muss. Es entstehen neue Konzepte wie serverseitiges Syntax-Highlighting und die verzögerte Einbindung von auf dem Server ermittelten Beweiszuständen im Browser.

Inhaltsverzeichnis

Inhaltsverzeichnis	i
1 Einführung	1
1.1 Motivation	1
1.1.1 Isabelle/Scala	1
1.1.2 HTML5	2
1.2 Aufgabenstellung	2
1.3 Anmerkungen	3
2 Grundlagen	5
2.1 Scala	5
2.1.1 Sprachkonzepte	6
2.1.1.1 Traits	6
2.1.1.2 Implizite Parameter	6
2.1.1.3 Implizite Konvertierungen	7
2.1.1.4 Typklassen	7
2.1.1.5 Dynamische Typisierung	8
2.1.2 SBT	9
2.1.3 Akka	9
2.1.3.1 Aktoren	9
2.1.3.2 Iteratees	10
2.1.3.3 Futures	10
2.1.4 Play Framework	10
2.2 Isabelle	12
2.2.1 Proof General	12
2.2.2 Asynchrones Beweisen	14
2.2.3 Isabelle/Scala	14
2.3 HTML5	15
2.3.1 Dokumentenobjektmodell	15
2.3.2 Cascading Styles Sheets	15

2.3.2.1	CSS3	16
2.3.2.2	LESS	16
2.3.3	JavaScript	17
2.3.3.1	CoffeeScript	17
2.3.4	HTTP	18
2.3.4.1	AJAX	18
2.3.5	WebSockets	18
2.3.6	JavaScript-Bibliotheken	20
2.3.6.1	jQuery	20
2.3.6.2	Backbone und Underscore	20
2.3.6.3	RequireJS	20
3	Anforderungen und Entwurf	23
3.1	Server	24
3.1.1	Wahl des Webframeworks	24
3.1.2	Authentifizierung	25
3.1.3	Persistenz	25
3.1.4	Bereitstellung von Ressourcen	25
3.1.5	Isabelle/Scala-Integration	26
3.2	Kommunikation	28
3.2.1	Google SPDY	29
3.2.2	WebSockets	29
3.2.3	Protokoll	29
3.3	Client	30
3.3.1	Browserkompatibilität	30
3.3.2	Benutzeroberfläche	31
3.3.2.1	Login	32
3.3.2.2	Projektübersicht	33
3.3.2.3	Die Sidebar	33
3.3.2.4	Webfonts	33
3.3.2.5	Die Editor-Komponente	34
3.3.2.6	Beweiszustände	36
3.3.3	Modell auf dem Client	37
4	Implementierung	39
4.1	Layout	39
4.2	Abstraktion vom Protokoll	40
4.2.1	ScalaConnector	40
4.2.2	JSConnector	42
4.3	Synchrone Repräsentation von Dokumenten	46

4.3.1	LineBuffer	46
4.3.2	RemoteDocumentModel	47
4.4	Clientseitiges Syntax-Highlighting	48
4.5	Serverseitiges Syntax-Highlighting	49
4.6	Substitution von Symbolen	52
4.7	Modale Dialoge	54
5	Bewertung	57
5.1	Performanz	57
5.2	Funktionalität	58
5.3	Zuverlässigkeit	58
5.4	Benutzbarkeit	59
5.5	Visualisierung	59
5.6	Übertragbarkeit	60
6	Zusammenfassung und Ausblick	63
A	Appendix	65
A.1	Abbildungsverzeichnis	65
A.2	Tabellenverzeichnis	65
A.3	Literatur	66
A.4	Liste der Abkürzungen	68
B	Installationsanweisungen	69

Kapitel 1

Einführung

1.1 Motivation

Die Benutzerschnittstellen interaktiver Theorembeweiser sind seit längerem ein wichtiger Teil der Forschung im Gebiet der formalen Beweissysteme. Lange Zeit wurde hier die Interaktion durch ein sequenzielles *read-eval-print-loop* (*REPL*)-Modell dominiert. Für das verbreitete Isabelle-System ist das Standard-Werkzeug der *Proof General*, welcher auf *emacs* basiert und damit in einer mächtigen Entwicklungsumgebung lebt und große Verbesserungen gegenüber den bisherigen Tools bietet. Das grundsätzliche Interaktionsmodell des Proof General wird in [Wen10, S. 2] jedoch immer noch als sequenziell beschrieben („*one command after another*“).

1.1.1 Isabelle/Scala

2010 wurde von Makarius Wenzel die *Isabelle/Scala* [Wen11] Schnittstelle eingeführt, die es erlaubt, in der Sprache Scala über ein einfaches Dokumentenmodell auf die Isabelle-Plattform zuzugreifen. Scala wird als flexibel genug beschrieben, um die relevanten Teile von Isabelle, das in Standard ML implementiert wurde, abzubilden. Da Scala darüber hinaus in der Java Virtual Machine (JVM) lebt, bieten sich vielfältige Möglichkeiten, die Schnittstelle zu integrieren. Unter anderem existieren für die JVM viele ausgereifte Entwicklungsumgebungen.

Zusammen mit der Schnittstelle wurde zunächst die Isabelle/jEdit Umgebung eingeführt [Wen10]. Die Wahl fiel dabei auf jEdit, da es über eine einfache Plugin-Architektur verfügt und die Integration das Projekt nicht zu sehr ablenken sollte. Dadurch wurden auch die Einschränkungen von jEdit gegenüber mächtigen Entwicklungsumgebungen wie Eclipse, Netbeans oder IntelliJ akzeptiert. Isabelle/Scala wurde dabei aber bewusst so konzipiert, dass es möglichst universell in eine beliebige Umgebung integriert werden kann.

Eine Integration der Plattform in eine der großen Entwicklungsumgebungen ist zweifelsohne ein

wichtiges Projekt für ambitionierte Nutzer, birgt aber weiterhin den Nachteil, dass die komplexe, große und ressourcenhungrige Isabelle-Plattform zunächst lokal auf dem Rechner installiert und vor allem konfiguriert werden muss, was für Einsteiger eine Hürde darstellt und damit besonders für die Lehre nicht optimal ist.

1.1.2 HTML5

Durch die Entwicklung der letzten Jahre im Bereich der Webprogrammierung, insbesondere der Bemühungen des W3C um eine drastische Erweiterung der Möglichkeiten von HTML im zukünftigen HTML5-Standard, ist es heute möglich Anwendungen, wie sie früher nur auf dem Desktop denkbar waren, im Browser zu implementieren. Durch immer effizientere JavaScript-Engines und wachsende Unterstützung für hardwarebeschleunigtes Rendering von Webseiten, wird die Webprogrammierung immer populärer und so hat beispielsweise Microsoft in der aktuellen Version des hauseigenen Betriebssystems sogar eine HTML/JS-basierte API zur Programmierung von Desktopanwendungen integriert.

Ein Thema, das in Zeiten von Netbooks, Smartphones und Tablet PCs immer mehr an Bedeutung gewinnt, ist die Portabilität und Mobilität von Daten und Anwendungen. Der Hype des *Cloud Computing* flaut nicht ab und der Erfolg der einschlägigen Webdienste (*Dropbox*, *Google Docs*, *Instagram*, etc.) bestätigt den Bedarf. Durch neue Techniken wie WebSockets werden die Möglichkeiten solche Dienste zu verwirklichen immer geeigneter.

1.2 Aufgabenstellung

In diesem Projekt soll durch Kombination der Isabelle/Scala Schnittstelle und den neusten Webtechniken eine mobile, plattformunabhängige Entwicklungsumgebung, speziell für die Arbeit mit Isabelle entstehen. Die Umgebung ist dabei der Browser. Die IDE soll ohne Konfigurationsaufwand für den Endnutzer auskommen und dabei trotzdem signifikante Verbesserungen gegenüber Isabelle/jEdit bieten, die im Folgenden aufgeführt werden:

- Geringer Ressourcenbedarf auf Clientseite,
- verbesserte Visualisierung schon während der Bearbeitung,
- verbesserte Integration der Ausgaben in die Visualisierung,
- volle Mobilität (Unabhängigkeit vom Client) sowie
- Mehrbenutzerfähigkeit.

Das Projekt versteht sich dabei als Machbarkeitsstudie, da der volle Funktionsumfang einer ausgereiften Entwicklungsumgebung hunderte Personenjahre Entwicklungszeit benötigen würde.

1.3 Anmerkungen

Die beiliegende Software kann unter Windows, OS X und Linux installiert werden. Genauere Anweisungen sind in Anhang B zu finden.

Die Quellen sind sowohl auf dem beiliegenden USB-Stick, als auch im Internet zu finden:

<http://www.github.com/martinring/clide>

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 geklärt werden sowie jeweils ein kurzer Einblick in wichtige Teilbereiche geliefert werden, um den Leser auf die folgenden Kapitel vorzubereiten.

2.1 Scala

Die Programmiersprache Scala ist eine an der École polytechnique fédérale de Lausanne (EPFL) von einem Team um Martin Odersky entwickelte statisch typisierte, objektorientierte, funktionale Sprache. In Scala entwickelte Programme laufen sowohl in der JVM als auch in der Common Language Runtime (CLR). Die Implementierung für die CLR hinkt 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 Anwendung findet.

Scala versucht von Anfang an den Spagat zwischen funktionaler und objektorientierter Programmierung. 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 seitens des Programmierers um lesbaren und wartbaren Code zu erstellen.

Seit 2011 wird Scala von der durch Martin Odersky ins Leben gerufenen Firma *Typesafe*¹ zusammen mit den Frameworks *Akka* (Abschnitt 2.1.3) und *Play* (Abschnitt 2.1.4) sowie des Buildtools *sbt* (Abschnitt 2.1.2) kommerziell im sogenannten *Typesafe Stack* weiterentwickelt und unterstützt. Dadurch wird Scala zu einer ernstzunehmenden Sprache, die auch in Zukunft noch schnell

¹<http://www.typesafe.com>

weiterentwickelt wird. Scala eignet sich nicht zuletzt durch die Aktoren- Bibliothek Akka dafür hochskalierbare verteilte Systeme zu entwickeln. Firmen wie Twitter, LinkedIn, Siemens, Tom-Tom, Sony, Amazon und die NASA treiben die Entwicklung immer schneller voran und sorgen dafür, dass eine große Infrastruktur um Scala herum entsteht. (Siehe auch [Lex11])

2.1.1 Sprachkonzepte

Im Folgenden sollen die für diese Arbeit relevanten Konzepte der Sprache Scala kurz vorgestellt werden. Dabei wird allerdings auf Grundlagen der objektorientierten und funktionalen Programmierung sowie auf bereits aus Java bekannte Konzepte verzichtet.

2.1.1.1 Traits

Traits sind ein besonders wertvoller und wichtiger Bestandteil von Scala. Sie sind ein Mittelweg zwischen einer abstrakten Klasse und einem Interface. Dabei ermöglichen sie wie Interfaces in Java Mehrfachvererbung, können jedoch auch genau wie abstrakte Klassen schon implementierte Funktionen enthalten. Zudem können Traits in beliebige Klassen mit dem Konstruktor eingemischt werden (Sogenannte *Mixins*).

So können Teile der Funktionalität, die immer wieder verwendet werden, aber nicht von konkreten Klassen abhängig sind, getrennt implementiert werden. Das fördert einen aspektorientierten Programmierstil und schafft zudem Übersichtlichkeit im Code.

2.1.1.2 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, der die beiden oder einen der impliziten Parameter nicht spezifiziert nach impliziten Definitionen vom Typ $T1$ bzw. $T2$. Diese Definitionen werden im aktuellen Sichtbarkeitsbereich nach bestimmten Prioritäten gesucht. Dabei wird zunächst im aktuellen Objekt, dann in den 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.

2.1.1.3 Implizite Konvertierungen

Des Weiteren existiert das Konzept der impliziten Konvertierungen (*implicit conversions*). Hierbei werden bei Typfehlern zur Kompilierzeit Funktionen mit dem Modifizierer **implicit** gesucht, die den gefundenen Typen in den nötigen Typen umwandeln können. Die Priorisierung geschieht hierbei genauso wie bei impliziten Parametern. Ein Beispiel:

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

Hier wird eine implizite Konvertierung 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 Konvertierung eingesetzt wird. Der Aufruf wird also intern zu **f(t1tot2(x))** erweitert.

2.1.1.4 Typklassen

Mit Hilfe von impliziten Definitionen ist es möglich, die aus der Sprache *Haskell* bekannten Typklassen in Scala nachzubilden.

Typklassen erlauben es, Ad-hoc-Polymorphie zu implementieren. Damit ist es ähnlich wie bei Schnittstellen möglich, Funktionen für eine Menge von Typen bereitzustellen. Diese müssen jedoch nicht direkt von den Typen implementiert sein und können so 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, die 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 Sichtbarkeitsbereich existiert, wird sie automatisch vom Compiler eingesetzt.

Als Beispiel betrachten wir die Ordnung eines Typs. Zunächst definieren wir einen generischen Trait **Ord**, der über eine **compare**-Funktion zum Vergleich zweier Werte verfügt.

```
trait Ord[T] {
  def compare: (a: T, b: T): Int
}
```

Wollen wir nun für einen beliebigen bestehenden Typ eine Ordnung definieren, müssen wir lediglich ein implizites Objekt bereitstellen, das `Ord` implementiert.

```
implicit object FileOrd extends Ord[File] {  
  def compare: (a: File, b: File): Int = ...  
}
```

Eine generische Funktion, welche die Funktion `compare` aus der Typklasse `Ord` verwendet, kann nun definiert werden, indem eine Instanz von `Ord` als impliziter Parameter erwartet wird.

```
def sort[T](elems: List[T])(implicit ord: Ord[T]): List[T] = ...
```

Die Funktion `sort` kann nun verwendet werden, um Dateien zu sortieren solange das implizite Objekt `FileOrd` beim Aufruf sichtbar ist.

Das Konzept der Typklassen ist vor allem dort sehr hilfreich, wo es darum geht fremde Bibliotheken um eigene Funktionen zu erweitern.

2.1.1.5 Dynamische Typisierung

Seit Scala 2.9 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 implementieren 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 folgendem Muster:

```
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)
```

Die dynamische Typisierung ist ein Sprachkonstrukt, das in Scala nur in Ausnahmefällen verwendet werden sollte. Es erweist sich aber als sehr praktisch in der Interaktion mit dynamischen Sprachen wie JavaScript sowie bei der Arbeit mit externen Daten, bei denen keine Typinformationen vorliegen. (JSON, SQL, usw.)

2.1.2 SBT

Das Simple Build Tool (sbt)² wird seit 2011 von Typesafe weiterentwickelt und ist das Standard-Werkzeug zur automatischen Projekt- und Abhängigkeitsverwaltung in der Scala- Programmierung. Da es selbst in Scala geschrieben wurde und auch die Konfiguration in Scala-Objekten stattfindet, ist es leicht, Erweiterungen dafür zu entwickeln. Sbt ist mit Maven, einem sehr verbreiteten Build Tool für Java kompatibel, sodass auch Javabibliotheken, welche in einem Maven-Repository liegen als Abhängigkeiten definiert werden können. Typesafe stellt zudem eine sehr große Auswahl von Scala-Bibliotheken in einem eigenen Repository zur Verfügung. Es können aber auch beispielsweise öffentliche Git-Repositories als Abhängigkeit definiert werden.

2.1.3 Akka

*Akka*³ war ursprünglich eine Implementierung des aus *Erlang* bekannten Aktoren Modells, ist mittlerweile jedoch zu einem umfangreichen Framework zur Entwicklung von hochperformanten verteilten Systemen gewachsen. Die Grundlage bilden nach wie vor Aktoren, wenn auch in, gegenüber anderen Implementierungen, leicht veränderter Form. [Hal12]

2.1.3.1 Aktoren

Aktoren haben sich als eine sehr wertvolle Abstraktion zur Modellierung von nebenläufigen Systemen herausgestellt. Dabei wird die Software in mehrere, parallel agierende Aktoren aufgeteilt, die sich untereinander Nachrichten senden. Um ungewollte Nebenläufigkeitseffekte auszuschließen, müssen die Nachrichten unveränderbar sein, was in Scala bislang leider noch nicht überprüfbar ist und somit in der Verantwortung des Entwicklers liegt.

In Akka sind Aktoren ortsunabhängig und können an einer beliebigen Stelle ausgeführt werden. Ein Akteur kann auf dem selben Prozessor, einem anderen Prozessor im selben Rechner, auf einem anderen Rechner im lokalen Netz oder auch auf einem beliebigen über das Internet erreichbaren Rechner irgendwo auf der Welt ausgeführt werden. In dieser Eigenschaft liegt der Schlüssel zur Skalierbarkeit: In Akka entwickelte Systeme können ohne Veränderungen auf einem oder tausenden Rechnern gleichzeitig ausgeführt werden, ganz im Sinne des *Cloud Computing*.

²<http://www.scala-sbt.org/>

³<http://www.akka.io/>

2.1.3.2 Iteratees

Akka bietet über die Aktoren und deren Verwaltung hinaus noch viele weitere hilfreiche Abstraktionen. Besonders erwähnenswert sind hier noch die sogenannten *Iteratees*. Iteratees sind eine Möglichkeit einen Datenstrom zu verarbeiten, ohne dass alle Daten verfügbar sind. Das ist dort besonders wichtig, wo es um nicht-blockierende kommunizierende Prozesse geht, wie es bei Webanwendungen üblich ist. Eine gute Einführung in Iteratees ist in [Sur12] zu finden.

2.1.3.3 Futures

Ein *Future* ist ein Proxy für das Ergebnis einer Berechnung welches nicht unmittelbar bekannt ist. Futures finden dann Anwendung, wenn Rechnungen nicht blockierend ausgeführt werden sollen und die Ergebnisse erst zu einem späteren Zeitpunkt benötigt werden. Die Rechnung wird dabei nebenläufig ausgeführt. Da Futures in Scala 2.10 einen Monaden bilden, können sie komponiert werden.

```
val a = future[Int](expensiveComputation1)
val b = future[Int](expensiveComputation2)
val c = for { aResult <- a, bResult <- b } yield aResult + bResult
```

Bislang existierten Futures in zwei Ausprägungen. Zum Einen gab es eine Implementierung in der Scala Standardbibliothek zum anderen eine in Akka. Der Grund dafür war, dass die Futures in Scala einige Schwächen hatten, welche erst nach Veröffentlichung erkannt wurden. So war es nicht leicht möglich Futures zu kombinieren. In Scala 2.10 wurde eine überarbeitete Implementierung der Futures eingeführt und das Akka Team hat sich daraufhin entschieden in der aktuellen Version 2.1 des Akka Frameworks auf die Futures der Standardbibliothek zurückzugreifen.

2.1.4 Play Framework

Das *Play Framework*⁴ ist ein Rahmenwerk zur Entwicklung von Webanwendungen auf der JVM in Java mit einer speziellen API für Scala. Play ist ein sehr effizientes zustandsfreies Framework welches auf Akka aufbaut um hohe Skalierbarkeit zu gewährleisten. Damit wird es leichter verteilte hochperformante Webanwendungen zu realisieren.

Die Struktur einer Play Anwendung ähnelt der bewährten Struktur von *Ruby on Rails*⁵. Es existieren *Modelle*, *Views* und *Controller*. Auch der Workflow ist ein ähnlicher. So ist es möglich

⁴<http://www.playframework.org>

⁵<http://rubyonrails.org/>

während der Entwicklung der Anwendung durch auffrischen der Seite im Browser immer die neueste Version zu sehen. Play nutzt dafür sbt als Build System.

Views werden als spezielle HTML-Templates, die auch Scala Code zulassen, definiert. Durch die Ausdrucksorientiertheit von Scala ist es damit möglich, die Views typischer zu halten und somit schon zur Kompilierzeit über sehr viele Klassen von Fehlern informiert zu werden. Das ist ein klarer Vorteil gegenüber den meisten andern modernen Web-Frameworks.

Modelle können natürlicherweise als beliebige Scala-Klassen bzw. Datenstrukturen repräsentiert werden. Play beschränkt den Entwickler auch nicht auf eine bestimmte Möglichkeit der Datenpersistenz.

Die Controller gruppieren Aktionen, die als Antworten auf Anfragen agieren und sind prinzipiell als Unterklassen von `play.mvc.controller`, welche in der Akka Infrastruktur leben, realisiert. Aktionen sind nicht mehr als Scala Funktionen, die die Parameter einer Anfrage verarbeiten und daraus eine Antwort produzieren, welche an den Browser zurückgesendet wird.

Das Routing geschieht über die Konfigurationsdatei `conf/routes`, in welcher von URLs auf Aktionen abgebildet wird. Es ist möglich, sogenanntes *reverse routing* zu betreiben um von einer Scala-Funktion zu einer URL zu gelangen. Des weiteren besteht die Möglichkeit, JavaScript-Code zu generieren, der im Browser zum reverse routing verwendet werden kann.

(Siehe auch Abschnitte 2.3.2.2, 2.3.3.1 sowie 2.3.6.3)

2.2 Isabelle

Isabelle ist ein generisches System, das vor allem zum interaktiven Beweisen von Theoremen unter der Nutzung von Logiken höherer Ordnung eingesetzt wird. Isabelle ist in Standard ML (SML) implementiert und stark davon abhängig. In Beweisen kann die volle Mächtigkeit von SML an jeder Stelle benutzt werden. Dadurch ist es schwer, eine Echtzeitverarbeitung wie sie für eine Entwicklungsumgebung nötig wäre zu realisieren.

Die Intelligible semi-automated reasoning (Isar)-Plattform (siehe dazu [Wen07]) bietet eine zusätzliche Abstraktion vom nackten SML Code, die dem Benutzer eine komfortablere Umgebung zur Formulierung von *Beweisdokumenten* liefert. Darüber hinaus ermöglichen die strukturierten Dokumente eine menschenlesbare Veröffentlichung der Beweise. Das ist ein klarer Vorteil gegenüber Beweisen in SML-Skripten, welche eher maschinenbezogen sind.

Isabelle/Isar erlaubt die Veröffentlichung in verschiedene Formate, wie HTML und \LaTeX . Dabei werden bestimmte Konstrukte besonders dargestellt. Solche Symbole werden in der Form `\<...>` im Code repräsentiert. Es gibt theoretisch unendlich viele dieser Symbole. Allerdings wird nur eine kleine Menge von Symbolen in [Wen12, S. 265-270] genau spezifiziert. Desweiteren existieren Kontrollzeichen in der Form `\<^...>`, die benutzt werden können, um Sub- und Superskript zu repräsentieren bzw. Zeichen fett darzustellen. Da die konkrete Benutzung der Isabelle-Plattform selbst für diese Arbeit eine eher untergeordnete Relevanz hat, wird an dieser Stelle für weitere Informationen auf die Isabelle Referenz in [Wen12] verwiesen.

2.2.1 Proof General

Das bisherige Standardwerkzeug für die Erstellung von Beweisdokumenten ist der generische *Proof General*, der auf der *emacs* Plattform lebt. Der Proof General bietet sogenanntes *script management*. (Vgl. [BT98, S. 161-194]) Gegenüber bisherigen Ansätzen entsteht hier ein neues Interaktionsmodell, bei dem Beweisskripte erstellt werden, die vom Proof General so verwaltet werden, dass es möglich ist, in Beweisen Schritte zurück zu gehen bzw. Beweiserzustände an den Kommandos automatisch gespeichert und wiederhergestellt werden.

Problematisch ist dabei, dass keine explizite Nebenläufigkeit besteht und damit auch keine direkte Kontrolle über die Optimierung für Mehrprozessorsystem, wie sie heute üblich sind.

In [Wen09, S. 2] wird folgendes Beispielskript zur Illustration des Problems aufgeführt:

```

theory C imports A B
begin
inductive path for rel ::  $\alpha \Rightarrow \alpha \Rightarrow \text{bool}$ 
where
    base : path rel x x
  |   step : rel x y  $\Longrightarrow$  path rel y z  $\Longrightarrow$  path rel x z

theorem example :
    fixes x z ::  $\alpha$ 
    assumes path rel x z
    shows P x z
using assms
proof induct
    fix x
    show P x x  $\langle$ proof $\rangle$ 
next
    fix x y z
    assume rel x y and path rel y z
    moreover
    assume P y z
    ultimately
    show P x z  $\langle$ proof $\rangle$ 
qed

end
    
```

Anhand dieses Beispiels werden die verschiedenen Schichten von Isabelle/Isar im Bezug auf die Nebenläufigkeit erläutert (hier zusammengefasst).

1. Theorien - Es existiert ein azyklischer Graph von Theorien, der die äußere modulare Struktur der Anwendung abbildet. Im Beispiel Modul C, das von den Modulen A und B abhängt.
2. Definitionen und Kommandos - Diese müssen streng sequenziell betrachtet werden. Im Beispiel sind das die Definition **inductive** und das Kommando **theorem**. Hier besteht kein Optimierungspotential
3. Toplevel-Beweise - Hier wird der meiste Rechenaufwand benötigt.
4. Verschachtelte Beweise - \langle *proof* \rangle - Beweise können hierarchisch strukturiert werden.

In Proof General werden die Toplevel-Beweise sequenziell betrachtet. Lediglich die einzelnen Module werden bereits parallel überprüft.

Da Beweise allerdings in dem Sinne unwichtig sind, dass es für einen abhängigen Beweis nicht

unbedingt nötig ist abzuwarten ob eine Vorbedingung erfolgreich bewiesen wurde, könnte diese sequenzielle Struktur aufgebrochen werden.

2.2.2 Asynchrones Beweisen

Mit Erscheinen der Version 2009 von Isabelle wurde es dann möglich, Beweisdokumente bzw. Theorien nebenläufig zu überprüfen. Also die Toplevel Beweise gleichzeitig zu überprüfen [Wen09]. Das hat die Voraussetzung für effiziente Implementierungen von interaktiven Benutzeroberflächen geschaffen.

2.2.3 Isabelle/Scala

Seit 2010 existiert mit *Isabelle/Scala* eine neue Schnittstelle zur Isabelle-Plattform, die auf 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] (Siehe auch Abbildung 2.1). Dabei werden die in [Wen09] erarbeiteten Konzepte des asynchronen Beweisens angewendet.

Über statisch typisierte Methoden können die Dokumente modifiziert werden. Dafür wurde ein internes XML-basiertes Protokoll eingeführt, das die Scala API mit der SML API verknüpft. Dementsprechend 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. Die Schnittstelle basiert zu großen Teilen auf einfachen Aktoren aus der Scala Standardbibliothek, es wird jedoch auch eine aktorenunabhängige API mit Callback-Funktionen bereitgestellt.



Abbildung 2.1 Konzept des Document Model in Isabelle/Scala

Vgl. [Wen10]

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

2.3 HTML5

Für die Implementierung der Browser-Anwendung wird auf den aktuellen Entwurf des zukünftigen 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 allein ist nicht geeignet, um dynamische Inhalte 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 Skriptsprache JavaScript (JS) und die darin verfügbaren Bibliotheken sowie das Document Object Model (DOM), auf das in Scripten zugegriffen werden kann, um den angezeigten Inhalt dynamisch zu verändern. [Smi12]

2.3.1 Dokumentenobjektmodell

Das *DOM* ist eine Schnittstelle, die es erlaubt HTML- bzw. Extensible Markup Language (XML)-Dokumente zu modifizieren und bildet damit die Grundlage für die Realisierung dynamischer Webseiten.

Der Einstiegspunkt in das DOM ist der **document**-Knoten, welcher in JS global verfügbar ist und von welchem aus die gesamte Baumstruktur erreichbar ist. Jeder HTML-Tag, jedes Attribut und jeder Text wird als ein Objekt, bzw. ein Knoten im Baum repräsentiert. Über verschiedene Methoden ist es möglich die Kinder-, Geschwister- und Elternknoten zu erhalten. Durch Funktionen wie **appendChild** ist es schließlich möglich, neue Elemente in das DOM zu integrieren bzw. bestehende zu modifizieren. [Kes12]

2.3.2 Cascading Styles Sheets

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 betreffenden Aspekte in die Stilvorlagen in CSS-Dateien ausgelagert werden.

2.3.2.1 CSS3

In der kommenden CSS Version 3.0, die bereits zu großen Teilen von den meisten aktuellen Browsern unterstützt wird, kommen einige interessante Neuerungen hinzu, die es vor allem ermöglichen, Anwendungen, welche man Bisher in Frameworks wie *Flash* oder *Silverlight* implementiert hat nun in reinem HTML+CSS zu verwirklichen. Die für die Realisierung der Browser-basierten Entwicklungsumgebung relevanten Neuerungen umfassen im speziellen:

- Einbetten von Schriftarten,
- Animationen und Übergänge,
- Verhindern von Textmarkierungen und
- Festlegen der Sichtbarkeit von Elementen für den Mauszeiger.

2.3.2.2 LESS

Auch CSS3 hat immer noch einige konzeptionelle Einschränkungen, welche die Benutzung erschweren:

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

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

Das Play Framework (Siehe Abschnitt 2.1.4) ermöglicht es, die Stylesheet-Sprache LESS zu verwenden, ohne dass diese seitens des Browsers 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 der Übersetzung werden sie an derselben Stelle zur Verfügung gestellt wie normale CSS-Dateien.

2.3.3 JavaScript

JavaScript ist eine dynamisch typisierte, klassenlose, objektorientierte Skriptsprache, die aktuell der Standard in der Entwicklung von clientseitigem Code für dynamische Webinhalte ist. Der Kern von JS wurde von der *Ecma International* als *ECMAScript* normiert. Durch JS ist es möglich Webseiten dynamisch zu verändern. Für eine gute Einführung in die Konzepte von JS sei an dieser Stelle auf [Fla97] verwiesen.

2.3.3.1 CoffeeScript

JavaScript wurde in Eile entwickelt und normiert, da zur Zeit der Entstehung ein schneller Bedarf an einer normierten Sprache für das Web bestand. Dadurch sind jedoch auch einige Unschöne Konstrukte in den Sprachkern gedrungen. So ist beispielsweise die C-artige Syntax für eine eher funktional angehauchte Sprache wie JS ungeeignet, da Funktionsdefinitionen durch geschweifte Klammern, das Schlüsselwort `function` und ein `return` Statement unnötig aufgeblasen werden und damit unleserlicheren Code erzeugen. Darüber hinaus fehlt in JS jede Möglichkeit der Modularisierung. Da noch nicht einmal Klassen existieren, führt das bei reinem JavaScript schnell zu nicht mehr wartbarem Code. Ein weiterer Stolperstein ist das Schlüsselwort `this`, das nicht immer klar zu verstehen ist, da es in Funktionsaufrufen seine Bedeutung wechseln kann.

*CoffeeScript*⁶ ist eine neue Skriptsprache mit dem Ziel diese „Problemzonen“ von JS auszumerzen. CoffeeScript hat eine eher an funktionale Sprachen wie Haskell erinnernde Syntax mit signifikantem Whitespace und einem `->`-Operator zur Funktionsdefinition. Darüber hinaus bietet CoffeeScript die Möglichkeit Klassen zu definieren und führt das Konzept der Vererbung ein. Das `this`-Schlüsselwort kann an Instanzen von Klassen gebunden werden. CoffeeScript Dateien werden zu optimiertem JS Code kompiliert, der den Vorgaben des *JS Linter*⁷ entspricht. Im Einzelnen sind die Verbesserungen gegenüber JS:

- Vereinfachte Syntax für Funktionsdefinitionen, Arrays, Blöcke,
- automatisches Initialisieren von Variablen (*lexical scoping*),
- variable Parameterlisten (*splats*),
- universellere Iterationsschleifen (`for`),
- vereinfachtes *slicing* und *splicing* (Arrayoperationen),
- Ausdrucksorientiertheit,
- Klassen und Vererbung,
- ein Existenzoperator,
- destrukturierende Zuweisungen (z.B. `[a,b] = [c,d]`),
- Bindung von `this` an Klasseninstanzen sowie

⁶<http://www.coffeescript.org>

⁷<http://www.javascriptlint.com/>

- mehrzeilige Strings und reguläre Ausdrücke mit Kommentaren.

Genauso wie für LESS existiert im Play Framework (Siehe Abschnitt 2.1.4) eine serverseitige Unterstützung für CoffeeScript. Die in CoffeeScript geschriebenen Dateien werden ebenfalls an gleicher Stelle wie normale JS-Dateien dem Browser als JS zur Verfügung gestellt.

2.3.4 HTTP

Das Hypertext Transfer Protocol (HTTP) ist das im Internet verwendete Standardprotokoll zur Übertragung von Daten. (Siehe auch [Fie99]) Leider ist HTTP für die Implementierung von hochdynamischen Webapplikationen nur bedingt geeignet, da Anfragen immer vom Browser gestellt werden, aber keine Möglichkeit vorgesehen ist, in die andere Richtung initiativ zu kommunizieren. Auf Grund der Einschränkungen des Protokolls, kam es in der Vergangenheit zur Entwicklung einiger Tricks um die sogenannten *Server-Pushes* zu realisieren. Die bekanntesten sind das *Polling*, bei dem in kleinen Abständen Anfragen an den Server gestellt werden, und *Comet* welches auf verzögerten Antworten vom Server basiert. Beide Lösungen werfen neue Probleme auf. Zum einen eine deutlich erhöhte Nutzung von Verbindungskapazitäten beim Polling und zum anderen das häufig vollständige Ausbleiben von Serverantworten und damit die fehlende Freigabe von Threads bei Comet-Anfragen.

2.3.4.1 AJAX

Asynchronous JavaScript and XML (AJAX) ist keine Bibliothek und auch kein Standard sondern ein Konzept zur Übertragung von Daten zwischen Browser und Webserver per HTTP auf dynamischen Webseiten. Hierbei wird das JS-Objekt `XMLHttpRequest` verwendet, um während der Anzeige einer Webseite, Daten vom Server nachzuladen bzw. dem Server Daten zu senden, ohne dass die Webseite neu geladen werden muss wie es bei klassischen Webseiten der Fall war. Ursprünglich und namensgebend wurde für die Übertragung der Daten XML verwendet. Mittlerweile ist es nach dem Abflauen des XML-Hypes wegen der guten Unterstützung in JS und den meisten Web-Frameworks auch üblich JavaScript Object Notation (JSON) zur Übertragung zu nutzen. [Jäg07]

2.3.5 WebSockets

WebSockets sind ein in HTML5 neu eingeführter Standard zur bidirektionalen Kommunikation zwischen Browser und Webserver. Hierbei wird anders als bei AJAX eine direkte TCP-Verbindung hergestellt. Diese Verbindung kann sowohl von Browser-, als auch von Serverseite

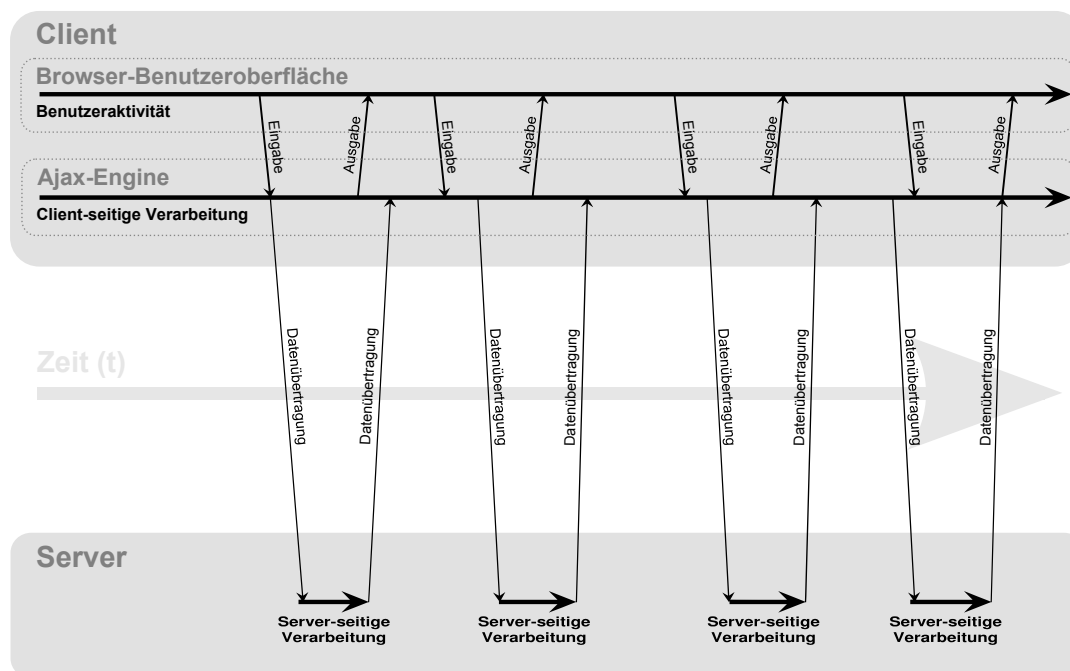


Abbildung 2.2 Ajax Modell einer Web-Anwendung (asynchrone Datenübertragung)

Quelle: Wikipedia

aus gleichartig 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.

Zum Aufbau einer WebSocket Verbindung wird einmalig zu Beginn eine HTTP Anfrage vom Browser gestellt, die der Server dann im Erfolgsfall mit der Eröffnung des WebSockets beantwortet. [Hic12]

2.3.6 JavaScript-Bibliotheken

Über den HTML5 Standard hinaus werden für die Strukturierung der Anwendung einige JS-Bibliotheken benötigt, die im Folgenden kurz erläutert werden.

2.3.6.1 jQuery

Die Bibliothek *jQuery*⁸ ist heute ein defacto-Standard in der Webentwicklung. In erster Linie erleichtert es den Zugriff und die Manipulation des DOM, bietet aber darüber hinaus zahlreiche Vereinfachungen im Umgang mit alltäglichen Aufgaben der Web-Entwicklung. Besonders erwähnenswert ist hierbei auch die AJAX-Abstraktion. Eine gute Einführung in jQuery bietet [Fra11].

2.3.6.2 Backbone und Underscore

*Backbone*⁹ ist eine Bibliothek, die der Strukturierung von sogenannten *single-page*-JS-Anwendungen dient. Backbone baut auf der allgemeinen *Underscore*¹⁰-Bibliothek auf, die einige Erleichterungen im Umgang mit Daten in JS bietet. Backbone führt das Model-View-Controller Konzept in Browseranwendungen ein und bietet hierfür einige Prototypen, von denen abgeleitet werden kann um eigene Modelle und Views zu implementieren, die dann über Events deklarativ miteinander verknüpft werden können.

Besonders interessant ist die Möglichkeit, die im HTML5 Standard neu eingeführte *History API* in Backbone zur Navigation zu verwenden. Dadurch ist es möglich in einer JS-Anwendung, welche die Seite nicht neu aufbaut, sondern immer nur Teile verändert, Navigation einzuführen, sodass der Benutzer zwischen den Zuständen Navigieren kann. Hierfür bietet Backbone die sogenannten **Router** in denen von URLs auf Zustände der Anwendung und umgekehrt abgebildet werden kann. Eine gute Einführung in die Bibliothek findet sich in [Osm12].

2.3.6.3 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

⁸<http://www.jquery.com>

⁹backbonejs.org

¹⁰underscorejs.org

sichtbar, da sie innerhalb einer Funktion definiert wurden. Das Funktionsergebnis ist das was nach außen sichtbar ist. Dies 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 werden bevor das aktuelle Modul ausgeführt wird.

RequireJS erlaubt es, den JS-Code für den Produktiveinsatz zu optimieren. Dafür gibt es das sogenannte *r.js*-Script, das unter anderem alle Abhängigkeiten in eine Datei zusammenfasst und den Code durch Entfernen von Whitespaces und Kommentaren sowie umbenennen von Variablennamen verkürzt. Zur Entwicklungszeit ist dieser nicht mehr lesbare Code nicht erwünscht. Deswegen bietet das Play Framework (Siehe Abschnitt 2.1.4) eine integrierte Version von RequireJS, die automatisch den lesbaren Code zur Entwicklungszeit bereitstellt, im Produktiveinsatz jedoch den optimierten.

Kapitel 3

Anforderungen und Entwurf

Auf Grund der Komplexität und des unvermeidbaren Rechenaufwands, welche ein Theorembeweiser mit sich bringt, ist es aus aktueller Sicht ausgeschlossen diese Arbeit zu größeren Teilen im Browser zu verwirklichen. Die einzige von allen größeren Browsern unterstützte Skriptsprache ist zur Zeit immer noch JS, die vor allem auf Grund der dynamischen Typisierung und der fehlenden Parallelisierbarkeit um einige Faktoren langsamer ausgeführt wird als nativer Code.

Ein besonderer Vorteil, den die Anwendung gegenüber bisherigen Lösungen bieten 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 Zeitpunkt 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 aus erreichbaren Stelle gespeichert sein müssen. Die Entscheidung zu einem Client- Server-Modell ergibt sich bei einer Webanwendung ohnehin automatisch.

Weil es sich bei der Webanwendung um eine Entwicklungsumgebung handelt, die insbesondere durch Echtzeitinformationen einen Mehrwert bieten 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 verarbeiten 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 Syntax-Highlighting 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 Webserver, wie der Bereitstellung der Inhalte, der Authentifizierung der Benutzer oder der Persistierung bzw. Bereitstellung der nutzerspezifischen Daten (in diesem Fall Projekte / 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 und
- 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 Performanz wichtig, unnötige Informationen zu eliminieren und die Daten zu komprimieren.

Der Server stellt zum einen eine normale HTTP API zur Authentifizierung und zur Verwaltung der Projekte zur Verfügung, zum anderen eine WebSocket-Schnittstelle zur Arbeit mit den Theorien. Während bei der HTTP API auf bewährte Methoden aus der Literatur zurückgegriffen werden kann, gibt es für die WebSocket-Schnittstelle keine nennenswerten Erfahrungen, auf die hier aufgebaut werden könnte.

3.1.1 Wahl des Webframeworks

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. Dafür existieren momentan zwei ausgereifte, bekannte Alternativen:

- Das *Lift Webframework*¹ bietet viele neue Ansätze in der Webprogrammierung und kann als „Experimentierkasten“ verstanden werden. Für jeden Anwendungsfall gibt es gleich mehrere Lösungen. Lift wird allerdings, auf Grund des Rückzugs von David Pollack aus der Entwicklung, seit einiger Zeit nicht mehr geordnet weiterentwickelt wird und ist zudem für unsere Zwecke überdimensioniert. Da die Webanwendung eher unkonventionelle Anforderungen an den Server hat, nützen die meisten Funktionen von Lift nicht. Lift wurde in der Vergangenheit schnell in verschiedenste Richtungen weiterentwickelt, dabei ist die Dokumentation jedoch stets vernachlässigt worden.
- Das *Play Framework* (Siehe auch Abschnitt 2.1.4) ist dagegen bewusst leichtgewichtig gehalten und eher auf hohe Performance ausgelegt, als auf die Lösung möglichst vieler Anwendungsfälle in verschiedenen, ausgefallenen Weisen. Darüber hinaus wird Play mittlerweile kommerziell von Typesafe unterstützt und weiterentwickelt und verfügt über eine detaillierte und professionell gestaltete Dokumentation [Typ12].

¹<http://www.liftweb.org>

Erfahrungen aus früheren Projekten mit Lift scheinen an dieser Stelle nicht weiter zu helfen, weil die größte Hürde die Implementierung der WebSocket Schnittstelle bildet und diese in Lift nur sehr spärlich dokumentiert sind und auch kein eindeutig beschriebener Weg dafür existiert.

3.1.2 Authentifizierung

Die Authentifizierung soll in diesem Projekt bewusst einfach gehalten werden, da es sich hierbei um eine Nebensächlichkeit handelt, die ohne weitere Probleme aufgerüstet werden kann. Wir beschränken uns daher auf die Möglichkeit, sich mit einem Benutzernamen sowie einem dazugehörigen Passwort anzumelden. Die Nutzerdaten werden dabei mit einer Konfigurationsdatei auf dem Server abgeglichen. Wir können dann auf die Fähigkeit des Play Frameworks zur sicheren Verwaltung von sessionbezogenen Daten zurückgreifen um die Anmeldung aufrechtzuerhalten.

Es ist zu erwarten, dass in einer zukünftigen Version von Play ein eigenes Modul zur Authentifizierung eingeführt wird, welches dann als Ersatz für die momentane Implementierung verwendet werden kann.

3.1.3 Persistenz

Als Besonderheit bei der Datenpersistenz sind die serverseitig zu verwaltenden Theorien hervorzuheben. 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 wie *Mercurial* oder *Git* möglich. Da über diese Daten hinaus nur wenige Informationen (Passwörter und Projektkonfigurationen) vom Server verwaltet werden müssen, ist die Einrichtung und Anbindung einer Datenbank nicht notwendig.

3.1.4 Bereitstellung von Ressourcen

Das Play-Framework bietet ausgefeilte Möglichkeiten, sowohl statische, als auch dynamische Ressourcen bereitzustellen. Ein Hauptaugenmerk liegt hierbei auf der Bereitstellung der nötigen JS-, CSS- sowie HTML-Dateien.

Während der Entwicklung dieser Arbeit, wurde ein Modul für das Play Framework entwickelt, um **CoffeeScript-Dateien** automatisch zu JS zu übersetzen, Abhängigkeiten mit RequireJS aufzulösen und eine einzelne, optimierte JS-Datei bereitzustellen. Da in der in Kürze erscheinenden Version 2.1 des Play Framework genau diese Funktionalität zu finden sein, liegt die Entscheidung

nahe, diese neue Funktionalität zu nutzen und das eigene Modul wegfallen zu lassen, da so eine Weiterentwicklung bzw. die Kompatibilität mit zukünftigen Versionen des Frameworks gesichert ist (Siehe Abschnitt 2.3.3.1 sowie 2.3.6.3).

Ebenfalls von Play unterstützt wird die Möglichkeit, **Less CSS (LESS)-Dateien** mit ihren Abhängigkeiten zu einer CSS-Datei zu übersetzen. Da diese genau wie bei der CoffeeScript-Übersetzung zur Entwicklungszeit in lesbare und im Produktiveinsatz in optimierte Dateien übersetzt werden. Weil die Oberfläche im Fall der Entwicklungsumgebung sehr vielschichtig und komplex ist, ist eine Modularisierung der Stilvorlagen eine willkommene Erleichterung und im Sinne der Wartbarkeit.

Statische Ressourcen, in unserem Fall z.B. Font-Dateien oder fremde JS-Bibliotheken, werden durch einen sogenannten *Asset-Controller* aus dem Play-Framework bereitgestellt. Dieser bietet die Möglichkeit, alle Dateien in einem Ordner statisch zur Verfügung zu stellen. In unserem Fall sind das die Dateien im Ordner `"/public"` welche unter der Uniform Resource Locator (URL) `"/assets"` bereitgestellt werden.

Die Bereitstellung der einzelnen Dokumente bzw. Theorien findet direkt über die WebSocket API statt (Siehe Abschnitt 3.2).

3.1.5 Isabelle/Scala-Integration

Weil die Isabelle/Scala Schnittstelle ständig weiter entwickelt wird, muss zunächst abgewogen werden, welche Version hier verwendet werden soll. Da es sich noch um ein junges Projekt handelt, das im Moment noch vielen größeren Änderungen unterliegt, entscheiden wir uns dafür die aktuellste stabile Version (die in Isabelle 2012 enthaltene) zu verwenden und damit deren Einschränkungen gegenüber aktuellen Entwicklungsversionen zu akzeptieren, da ein dauerndes „Hinterherlaufen“ hier zu großen Aufwand bedeuten würde.

Isabelle/Scala wurde in Isabelle 2012 für Scala 2.9 übersetzt. Deswegen kann es nicht direkt in diesem Projekt verwendet werden (Scala 2.9 und Scala 2.10 sind nicht binärkompatibel). Deswegen musste die Schnittstelle für Scala 2.10 neu übersetzt werden. Die neu gebaute Version von **Pure.jar** ist im Anwendungsverzeichnis unter `lib/` zu finden. Bei Verwendung älterer Versionen kommt es zu unverständlichen Fehlermeldungen.

Isabelle/Scala arbeitet intern mit Offset-basierten Textpositionen. Da das auf dem Client ineffizient wäre und der verwendete Editor (Abschnitt 3.3.2.5) mit zeilenbasierten Positionen arbeitet, ist es notwendig, eine effiziente Repräsentation der Dateien auf dem Server zu entwickeln, auf welche sowohl über die Zeile/Spalte als auch über absolute Offsets zugegriffen werden kann (Die Implementierung wird in Abschnitt 4.3 beschrieben).

Um die kumulierten Änderungen am Dokument, welche vom Client regelmäßig gesendet werden

nicht nur in die eigene Repräsentation einzuarbeiten, sondern auch an Isabelle/Scala weiterzuleiten, muss eine Umrechnung der Daten in die von der Schnittstelle verwendeten Datentypen geschehen.

Nach jeder Veränderung an einzelnen Dokumenten leitet Isabelle/Scala die Daten an die Isabelle Plattform weiter, welche die Dokumente dann überprüft. Über einen Nachrichtenkanal (`Session.commandsChanged`) kommen nach erfolgreichem Abschluss dann die Ergebnisse zurück welche wiederum in ein für den Client verständliches Format (JSON) mit den Zeilen/Spaltenbasierten Positionen übertragen wird.

Für das Syntax-Highlighting auf dem Client ist es nötig, eine Liste der gültigen Schlüsselwörter in der Theorie an den Browser zu übertragen und aktuell zu halten (Siehe auch 4.4).

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 Datenübertragung wie bei Webapplikationen üblich über AJAX bzw. 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 Google SPDY

Google experimentiert zu Zeit mit einem neuen Protokoll für schnellere Webanwendungen, bei dem die Header komprimiert werden und vor allem unbegrenzt viele parallele Anfragen gestellt werden können. SPDY führt zu einem drastischen Geschwindigkeitsgewinn bei der Arbeit mit vielen kleinen Nachrichten. Das Protokoll ist jedoch zur Zeit noch experimentell und wird lediglich von einem experimentellen Build des *Google Chrome* Browsers unterstützt. Eine Unterstützung in Play fehlt vollständig. Eine Implementierung, welche auf SPDY aufbaut wäre damit nicht von einer größeren Nutzerzahl verwendbar. Darüber hinaus ist der Erfolg des Protokolls ungewiss.

3.2.2 WebSockets

Die im HTML5-Standard eingeführten WebSockets sind die ideale Lösung für das 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 signalisieren, 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 allen relevanten Browsern in der neusten Version unterstützt. Browser, die seit einem Jahr nicht mehr aktualisiert wurden können mit dem Aufruf allerdings zu großen Teilen nichts anfangen (Siehe Abschnitt 3.3.1). Das Play Webframework unterstützt serverseitige WebSocket-Verbindungen. Allerdings muss bei der Benutzung auf viel Funktionalität verzichtet werden. Weil WebSockets allerdings so unabdingbar sind werden diese Einschränkungen akzeptiert.

3.2.3 Protokoll

Für das Protokoll wird zunächst der Einfachheit halber JSON gewählt mit der Möglichkeit im Hinterkopf, es zu einem späteren Zeitpunkt leicht durch das komprimierte Binary JSON (BSON)-Protokoll [] zu ersetzen. Um das möglich zu machen, wird unter Verwendung von dynamischer Typisierung in Scala (Siehe Abschnitt 2.1.1.5) vom Übertragungsprotokoll abstrahiert (Siehe Abschnitt 4.2), damit es ohne größeren Aufwand ausgetauscht werden kann.

3.3 Client

Zur Realisierung des Clients wird auf die etwas komfortablere Skriptsprache CoffeeScript (Siehe Abschnitt 2.3.3.1) zurückgegriffen. CoffeeScript-Code ist gegenüber JS-Code deutlich kürzer (Laut eigenen Angaben etwa 30%) und hat eine an funktionale Sprachen wie Haskell erinnernde Syntax. Da, mit den neuen Möglichkeiten in Play 2.1, CoffeeScript problemlos verwendet werden kann (Der Browser erhält kompiliertes JS), entstehen hierdurch keine Nachteile.

Zur Strukturierung wird RequireJS sowie BackboneJS (und damit implizit auch UnderscoreJS. Siehe Abschnitt 2.3.6.2) verwendet. Durch die vielfältigen Funktionen dieser Bibliotheken ist es möglich den Code klar zu modularisieren.

3.3.1 Browserkompatibilität

Auf Grund der in Abschnitt 3.2.2 beschriebenen Notwendigkeit, kann auf WebSockets nicht verzichtet werden. Damit sind die meisten älteren Browser nicht mit der Anwendung kompatibel.

	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
WebWorkers	4.0	4.0	10.0	3.5	10.6
Webfonts	4.0	3.1	9.0	3.5	10.0
CSS Transitions	4.0	3.1	10.0	4.0	10.5

Tabelle 3.1 Kompatibilität der gängigsten Browser mit den Verwendeten Standards

Daten von caniuse.com

Aus Tabelle 3.1 ist zu entnehmen, dass alle weiteren in der Anwendung benutzten Standards eine geringere oder die gleiche Anforderung an die Aktualität des Browsers haben. Da WebSockets ein sehr neues Konzept sind, 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 potenziellen 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 sein. Bei der Entwicklung wird jedoch besonderes Augenmerk auf WebKit-basierte Browser, insbesondere Google Chrome gelegt und einige der anderen genannten Systeme bleiben ungetestet. Für den Internet Explorer und Firefox ist bekannt,

dass Probleme mit dem Rechtemanagement bei einem lokal ausgeführten Server bestehen und beispielsweise Webfonts teilweise nicht ordnungsgemäß funktionieren.

3.3.2 Benutzeroberfläche

Inspiziert wurde das Design des User Interface (UI) durch die von Microsoft in Windows Phone 7 bzw. Windows 8 eingeführte Design-Sprache *Metro UI* (bzw. seit 2012 *Microsoft Design Language*) [Mic12]. Dabei wird auf unnötige Grafiken verzichtet und die Typographie in den Vordergrund gestellt. Durch die Reduktion auf das Wesentliche, entsteht eine neue Ästhetik, welche eine willkommene Auffrischung der Fensterzentrierten Oberflächen wie sie seit langem bestehen, bietet.

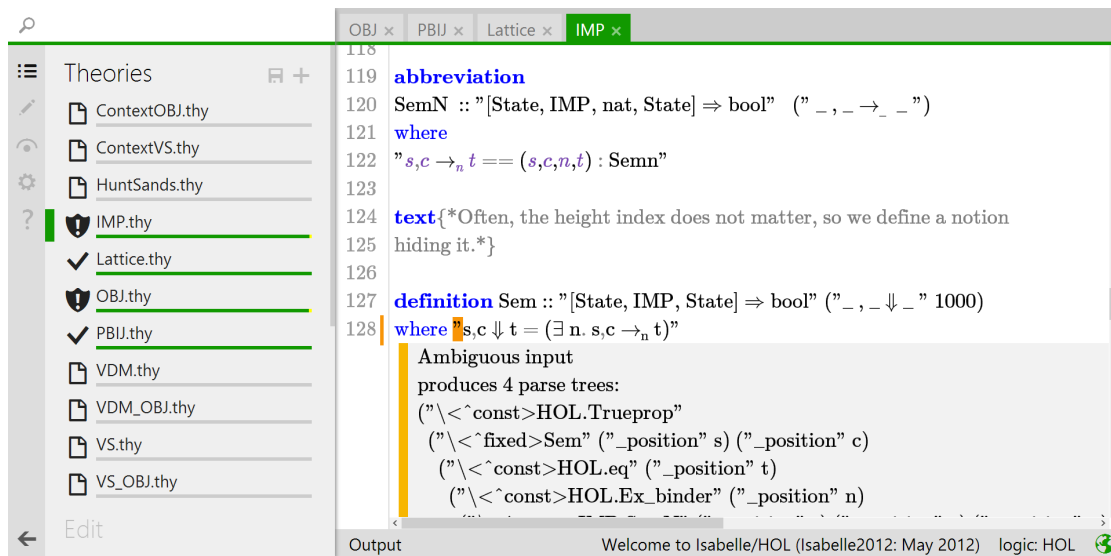


Abbildung 3.1 Die clide-Oberfläche

Für die Gestaltung von HTML UIs gibt es bereits viele ausgereifte Frameworks wie *Twitter Bootstrap* oder *jQuery UI*. Leider sind diese Frameworks vor allem für klassische Browseranwendungen konzipiert, in denen es um einfaches Realisieren von Formularen und Fließtext-Elementen geht. Weil in dieser Anwendung jedoch nur wenige Elemente klassischer Seiten zum Einsatz kommen, entscheiden wir uns, die UI Komponenten selbst zu entwickeln. In diesem Zusammenhang ist es nötig, eine kleine Sammlung von *UI-Controls* zu realisieren:

- Registerkarten,
- Kontextmenüs,
- die Sidebar mit
 - gliedernden Abschnitten,

- einer Navigationsleiste zur Auswahl der Abschnitten und
- *Commands*, welche von den Abschnitten gegliedert werden,
- Fortschrittsbalken,
- Dialogboxen, etc.

Der Vorteil der eigenen Entwicklung ist darüber hinaus, dass die Komponenten als Backbone-Views implementiert werden können und damit direkt und einfach mit den Backbone-Modellen verbunden werden können.

Besondere Herausforderungen im Zusammenhang mit der UI sind außerdem das sinnvolle Unterbringen der Beweiszustände und Fehlerinformationen in der Darstellung der Dokumente sowie die Darstellung des Fortschritts, der einzelnen Theorien, welche nicht zwingend vom Nutzer geöffnet werden müssen, sondern auch implizit als Abhängigkeiten anderer Beweisdokumente verarbeitet werden können, und die Darstellung der Dokumente selbst in einer möglichst L^AT_EX-nahen Form.

3.3.2.1 Login

Der Anmeldebildschirm (Login) ist entsprechend einfach gehalten. Dem Nutzer wird lediglich ein Formular zur Eingabe von Benutzernamen und Passwort präsentiert (Siehe Abbildung 3.2).

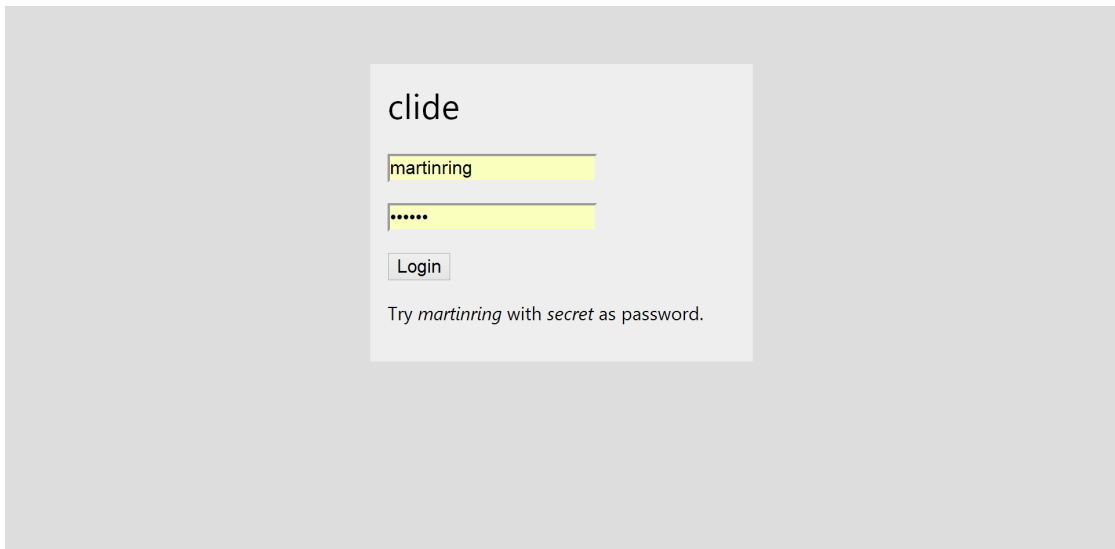


Abbildung 3.2 Das Anmeldeformular

Die Kommunikation erfolgt an dieser Stelle noch über normales HTTP und bei fehlerhaften Eingaben wird auf einer neu geladenen Seite eine Fehlermeldung über dem Formular angezeigt. Das stört an dieser Stelle nicht und eine Nutzung von WebSockets würde die Sache hier nur verkomplizieren, da so nicht die bereits ausgereiften Verfahren zur Anmeldung über HTTP genutzt werden könnten.

3.3.2.2 Projektübersicht

Die Projektübersicht dient dem Benutzer als Startpunkt. Hier kann er ein Projekt zur Bearbeitung auswählen, Projekt anlegen und löschen sowie die Konfiguration ändern. Dem Nutzer wird hierfür in einer Spalte neben jedem Projekt eine Dropdown-Box präsentiert mit einer Liste der auf dem Server verfügbaren Logik-Images. Änderungen werden direkt per AJAX an den Server gesendet und übernommen.

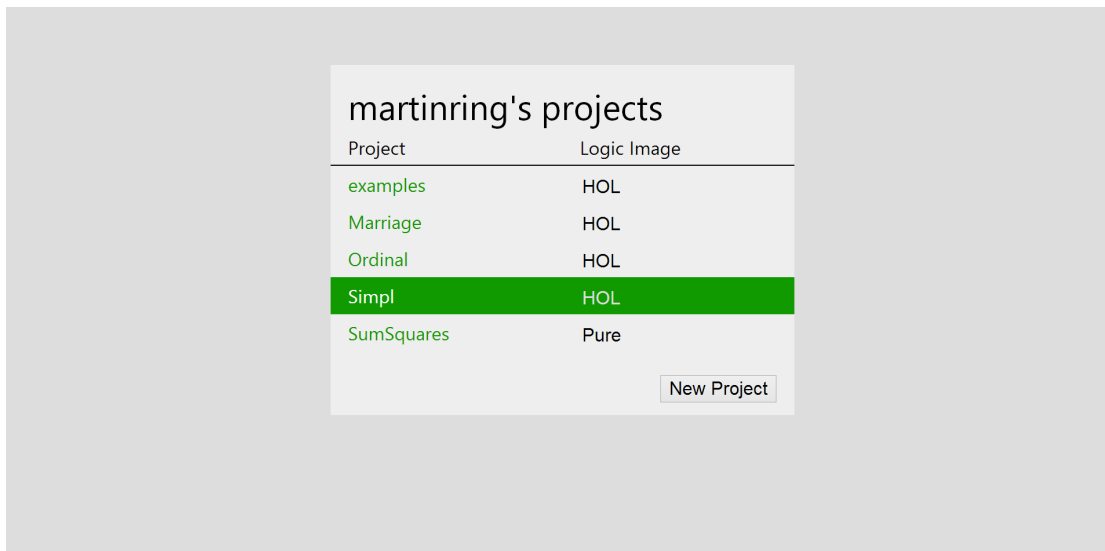


Abbildung 3.3 Die Projektübersicht

3.3.2.3 Die Sidebar

Sobald der Nutzer auf der Übersichtsseite ein Projekt auswählt, indem er es anklickt, wird ihm ein Ladebildschirm präsentiert, welcher dann nach erfolgreichem Aufbau der Sitzung mit dem Server verschwindet. Darunter kommt die eigentliche IDE zum Vorschein, welche zunächst nur die Sidebar anzeigt.

Die Sidebar ist die „Kommandozentrale“ der Entwicklungsumgebung. Hier werden dem Nutzer die Liste der Theorien, Möglichkeiten zur Bearbeitung der Beweisdokumente sowie zur Anpassung der Einstellungen präsentiert.

3.3.2.4 Webfonts

Weil ein formuliertes Ziel ist, dem Nutzer eine möglichst nah an den \LaTeX -Veröffentlichungen orientierte Visualisierung der Beweisdokumente zu präsentieren, müssen hierfür spezielle Fonts

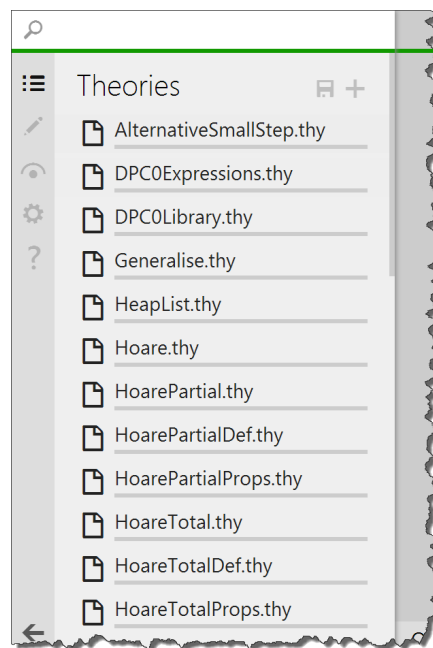


Abbildung 3.4 Die Sidebar

verwendet werden. Glücklicherweise wurden im CSS3-Standard die Webfonts eingeführt (Siehe Abschnitt 2.3.2.1), mit denen es möglich ist, beliebige OTF- und TTF-Schriftarten seitens des Servers bereitzustellen.

In ersten Designprototypen wurde zunächst *Cambria Math* verwendet, da es sich dabei um einen der umfangreichsten Fonts mit mathematischen Symbolen handelt. Dieser Font ist allerdings nicht frei und kann daher in diesem Projekt keine Verwendung finden. Das OpenSource Projekt *MathJax*, welches sich zur Aufgabe gemacht hat MathML- und \LaTeX -Formeln per JavaScript in HTML-Seiten korrekt anzuzeigen, hat als glückliches Nebenprodukt die Standard-Fonts der \LaTeX -Plattform (*Computer Modern* Serie) in die verschiedensten Formate übertragen. Unter anderem auch OTF. Die Fonts sind unter der *Apache License 2.0* lizenziert und damit nutzbar.

Da diese Fonts getrennt wurden in *Main*, *Math*, *AMS*, *Caligraphic*, *Fraktur* und *Typewriter*, muss wie in Abschnitt 4.4 beschrieben, beim Syntax-Highlighting immer auch entschieden werden, welcher Font an welcher Stelle benutzt werden soll.

3.3.2.5 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 [Wen10]. Da bei der HTML-Darstellung kaum Grenzen gesetzt sind und sich CSS-Formatierung sehr leicht dazu benutzen lässt bestimmte Textinhalte besonders darzustellen, ist 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 die Sonderzeichen korrekt dargestellt wurden und bearbeitet werden konnten. Die besondere Anforderung ist hierbei nicht die Darstellung, sondern vor allem der Umgang mit den variablen Breiten. Selbst wenn ein Monospace-Font verwendet werden würde, bestünde das Problem, dass z.b. bei Sub- und Superskript nach Typographischen Standards nur 66% der Textgröße verwendet wird und somit auch die Zeichenbreite geringer wird. Da aber eben die Visualisierung eine besondere Stärke der Anwendung sein soll, wollen wir zusätzlich auch nicht darauf verzichten, ähnliche Fonts zu verwenden, wie in der Ausgabe der L^AT_EX-Dateien, also auch mathematische Sonderzeichen nicht in ein Raster zu quetschen.

Eine weitere besondere Anforderung, welche bislang relativ einmalig zu sein scheint, ist die Tatsache, dass das Syntax-Highlighting zu Teilen auf dem Server stattfinden muss und somit eine Möglichkeit bestehen muss, diese zusätzlichen Informationen in die Darstellung zu integrieren.

Zusammenfassend können folgende besondere Anforderungen an die Editor-Komponente formuliert werden: Der Editor muss in der Lage sein

- Syntaxhighlighting zu betreiben,
- Externes Syntax-Highlighting verzögert zu integrieren,
- Schriftarten mit variabler Zeichenbreite anzuzeigen,
- Tooltips für Typinformationen o.ä. anzuzeigen und
- Isabelle-Sonderzeichen zu substituieren.

Weil der hauptsächliche Aufwand bei einer Editor-Komponente nicht darin liegt, Text zu bearbeiten und darzustellen, sondern vor allem in der Infrastruktur darum (Copy/Paste, Suche, Selektieren, Drag'n'Drop, etc.) ist es verlockend, eine fertige Komponente zu verwenden. Hier existieren mehrere ausgereifte Alternativen. Bei genauerer Betrachtung gibt es jedoch keine, welche optimal für unsere Zwecke geeignet ist.

Der **MDK-Editor**² bietet viele Features, wird aber seit 2008 nicht mehr weiterentwickelt und scheidet damit sofort aus.

Der **AJAX.org Cloud9 Editor (ACE)**³ (ehemals Mozilla SkyWriter) wird momentan sehr stark weiterentwickelt. ACE bietet bereits ein raffiniertes Framework für das Syntax-Highlighting, das sich in einem Prototyp relativ leicht an das serverseitige Syntax-Highlighting anbinden ließ. ACE liefert alle Funktionen, welche man von einem Modernen

²<http://www.mdk-photo.com/Editor/>

³<http://ace.ajax.org/>

Text-Editor erwartet, hat jedoch einen entscheidenden Nachteil: Zur Darstellung wird aus Performance-Gründen intern ein festes Raster verwendet. Dabei wird davon ausgegangen, dass ein Monospace-Font verwendet wird. Von diesem wird einmalig eine Zeichenbreite ermittelt und diese feste Metrik wird dann für alle internen Operationen verwendet. Da diese Designentscheidung so tiefgreifend ist, scheint es nicht realistisch, in akzeptabler Zeit die Komponente so zu modifizieren, dass variable Breiten unterstützt werden können. Außerdem ist es nicht möglich, Textstellen durch Sonderzeichen zu substituieren. Somit scheidet auch ACE für die Verwendung in der Anwendung aus.

CodeMirror⁴ ist ebenfalls eine weit entwickelte Editor-Komponente, welche nicht ganz so umfangreich ist wie ACE, jedoch um einiges flexibler erscheint. In einem Prototyp war es möglich, einige eigene Modifikationen für die Darstellung (Sub- Superskript, Tooltips, Hyperlinks) zu integrieren. CodeMirror verwendet kein festes Raster, darunter leidet die Performanz. Da wir jedoch darauf angewiesen sind, müssen diese Geschwindigkeitseinbußen akzeptiert werden. Seit der am 10.12.2012 erschienenen Version 3.0, ist es möglich Textteile zu durch HTML-Widgets zu substituieren. Dadurch können Isabelle Sonderzeichen durch ASCII Sequenzen wie beispielsweise `\<rightarrow>` für das Zeichen \rightarrow repräsentiert werden direkt im Editor esetzt werden, sodass der bearbeitete Text valider Isabelle- Code bleibt, die Darstellung hingegen der eines \LaTeX -Dokuments entspricht.

Weitere Editoren existieren zwar, scheiden aber alle aus, da sie nicht einmal die Hälfte der oben formulierten Anforderungen erfüllen. Es wird klar, dass CodeMirror der bestgeeignete Editor ist. Trotzdem müssen einige eigene Anpassungen in den Kern des Editors integriert werden. Diese sind unter anderem die Unterstützung von

- Sub- und Superskript (sowie angepassten Cursorpositionen und -höhen),
- Tooltips für einzelne Textabschnitte sowie
- die Darstellung von Hyperlinks im Text.

3.3.2.6 Beweiszustände

Die Integration der Beweiszustände wurde in bisherigen Werkzeugen meist so gelöst, dass ein eigenes Fenster die Information über das Kommando unter dem Cursor visualisiert. Diese Lösung übernehmen wir (erreichbar über den Button „Output“ am unteren Rand), integrieren jedoch zusätzlich einen neuen Ansatz. Um während der Entwicklung einfacher Beweise wie sie in der Lehre am Anfang häufiger vorkommen eine Übersicht zu behalten, wird die Möglichkeit geboten, eine Inline-Anzeige aller Beweiszustände zu integrieren (wie in Abbildung 3.5). Das ist vor allem für das Verständnis der Zusammenhänge hilfreich. Da bei komplexeren Beweisen die Ausgaben jedoch sehr lang sein können, ist diese Option konfigurierbar. Ansonsten würden die Zustände zwischen den Zeilen in diesen Fällen die Lesbarkeit erschweren.

⁴<http://codemirror.net/>

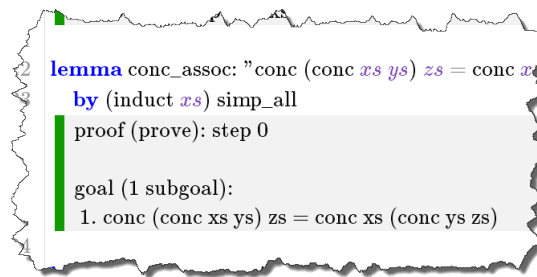


Abbildung 3.5 Inline-Anzeige von Beweiszuständen

3.3.3 Modell auf dem Client

Der Client muss zu jeder Zeit über ein konsistentes Abbild der relevanten Informationen verfügen. Das stellt sich als schwierige Herausforderung dar. Insbesondere muss der Inhalt des Texteditors mit dem Server synchron gehalten werden. Abbildung 3.6 zeigt ein vereinfachtes Abbild des Datenflusses in der Anwendung.

Es kann natürlich nicht jeder einzelne Tastendruck übertragen werden kann. Das würde auch wenig Sinn machen, da es nicht realistisch ist, jede einzelne Veränderung zu überprüfen. Den Nutzer würden Fehlermeldungen zu Zwischenzuständen stören und der Server wäre absolut überlastet. Also wird nach jedem Tastendruck ein *Timeout* von 700 Millisekunden gestartet. Wenn es abläuft, ohne dass eine Taste gedrückt wird, werden die Veränderungen im Dokument an den Server übertragen. In dem häufigen Fall eines weiteren Tastendrucks innerhalb der Zeitspanne, wird das Timeout neu gestartet (*Reset*), so lange, bis der Nutzer über den Zeitraum von 700 ms keine Veränderung mehr vornimmt.

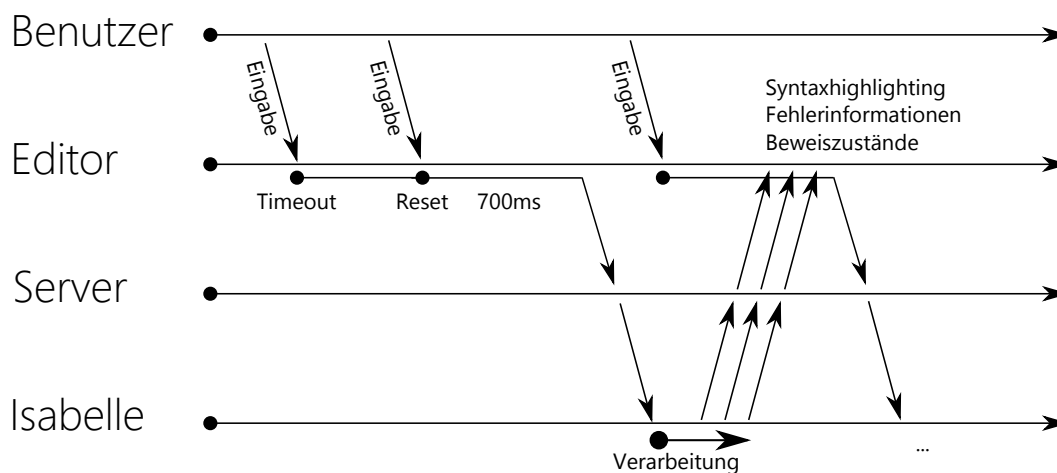


Abbildung 3.6 Datenfluss in clide

Die Zeitspanne von 700 Millisekunden hat sich in eigenen Experimenten als guter Wert herausgestellt und ähnliche Werte wurden auch bei anderen Entwicklungsumgebungen und Editoren beobachtet (Eclipse, Visual Studio, etc.).

Kapitel 4

Implementierung

Im Folgenden werden einige interessante Gesichtspunkte der Implementierung, die in Kapitel 3 nur kurz angesprochen wurden, genauer herausgearbeitet.

4.1 Layout

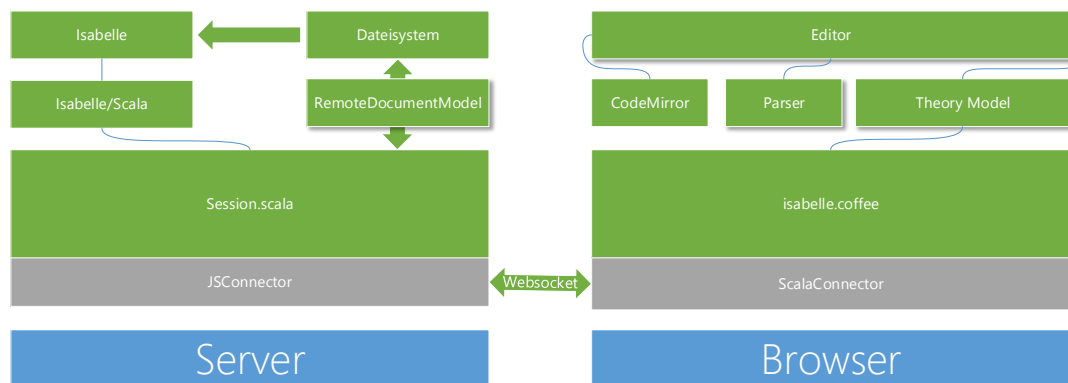


Abbildung 4.1 Architekturlayout

Abbildung 4.1 zeigt ein grobes Architekturlayout. Server und Browser kommunizieren über die serverseitige Schnittstelle **JSConnector** bzw. clientseitig über den **ScalaConnector** (Abschnitt 4.2). Die verknüpften Objekte sind dabei **Session** aus **Session.scala**, wovon für jede Clientseitige Sitzung eine Instanz besteht sowie im Browser das Objekt **Session** aus **isabelle.coffee**.

Im Browser wird ein Modell aller Theorien verwaltet, welches sowohl von der **Session** (z.B. Serverseitiges Syntax-Highlighting in Abschnitt 4.5), als auch vom Editor verändert werden kann. Die Kommunikation erfolgt über eventbasierte Callbacks. Jeder **Editor** erzeugt eine angepasste

CodeMirror Instanz zur Visualisierung mit einem Parser, welcher die äußere Syntax erkennt und direkt in die Darstellung integriert (Abschnitt 4.4).

Auf dem Server wird das **RemoteDocumentModel** mit den **LineBuffern** (Siehe Abschnitt 4.3) verwendet, um den Zustand der Theorien synchron zum Client zu verwalten.

4.2 Abstraktion vom Protokoll

Um das Protokoll austauschbar zu gestalten, wurde eine Abstraktion über WebSockets implementiert. Hierfür wurde seitens des Browsers der **ScalaConnector** und serverseitig der **JSConnector** entwickelt.

4.2.1 ScalaConnector

Das **ScalaConnector** wurde in CoffeeScript als RequireJS-Modul implementiert und dient zur Kommunikation mit dem Webserver. Ein beliebiges Objekt kann sich über den **ScalaConnector** mit dem Server verbinden, sodass dieser dann direkt die Funktionen dieses Objekts aufrufen kann.

```
define -> class ScalaConnector
  constructor: (@url,@object,init) -> ...
```

Dem Konstruktor wird zum einen die url (**url**) des WebSockets mit dem verbunden werden soll, zum anderen das Objekt (**object**), welches dem Server als Schnittstelle zur Verfügung gestellt werden soll übergeben. Als optionales Argument kann eine **init**-Funktion übergeben werden, welche aufgerufen wird, sobald die Verbindung hergestellt worden ist.

Intern wird im Konstruktor zunächst eine Verbindung aufgebaut und an das **onmessage**-Callback des WebSockets verbunden:

```
@socket = new WebSocket(@url) # Connect to server
@socket.onmessage = (e) =>
  @bytesDown += e.data.length # Downstream traffic measure
  recieve(JSON.parse(e.data)) # Interpret messages as JSON
```

Die **recieve**-Funktion unterscheidet hierbei zwischen Antworten auf eigene Anfragen und Anfragen bzw. Kommandos vom Server:

```
recieve = (e) =>
  if e.action # if an action is defined this is a command from the server
    f = @object[e.action] # we try to find the action on the connected object
    if f?
      result = f.apply(f,e.args) or null # execute the function and save the result
      # if an id for the message is defined the server awaits an answer with the result of the
      # function execution. otherwise this is just a command (or server-push)
      if e.id then @socket.send JSON.stringify
        resultFor: e.id
        success: true
        data: result
      else # if the action does not exist we send an error message to the server
        if e.id then @socket.send JSON.stringify
          resultFor: e.id
          success: false
          message: "action '#{e.action}' does not exist"
        else console.error "action '#{e.action}' does not exist"
    else # if no action is defined the message must be an answer to a request
      callback = @results[e.resultFor] # retrieve the callback function for this action
      if callback
        callback(e.data) # answer the callback
        @results[e.resultFor] = null # delete the callback function
```

Hierbei werden drei verschiedene Fälle unterschieden:

- Die Nachricht enthält das Feld **action**: Es handelt sich um eine Anfrage oder ein Kommando vom Server.
 - Wenn zusätzlich das Feld **id** definiert ist, erwartet der Server eine Antwort mit dieser id als Referenz, es handelt sich also um eine Abfrage.
 - Wenn das Feld nicht enthalten ist, handelt es sich um eine einfache Nachricht vom Server auf die nicht geantwortet wird.
- Wenn die Nachricht das Feld **action** nicht enthält, handelt es sich um eine Antwort auf eine vorherige Anfrage des Clients. In diesem Fall wird die entsprechende Callback-Funktion mit dem Ergebnis des Servers aufgerufen.

Der Vorteil der Repräsentation über fehlende Felder ist die Kompaktheit der Daten. Da der WebSocket konfiguriert ist, die Daten zusätzlich komprimiert zu übertragen, sind die Nachrichten sehr klein. Noch kleiner können sie gemacht werden, indem ein eigenes Protokoll implementiert wird. Diese abstrakte Implementierung macht das sehr einfach, da nur an zwei Stellen etwas ausgetauscht werden muss.

Das Gegenstück ist natürlich das Versenden von Daten. Antworten auf Serveranfragen werden wie oben beschrieben automatisch verschickt. Anfragen können über die `call`-Funktion versendet werden.

```
call: (options) =>
  if options and options.action
    if options.callback
      @results[@id] = options.callback
      options.id = @id
      @id += 1
      @socket.send JSON.stringify(options)
    else
      console.error 'no action defined'
```

Die Funktion erwartet ähnlich wie die `ajax`-Funktion aus jQuery ein Konfigurationsobjekt `options`, in welchem mindestens das Feld `action` definiert sein muss. (Andernfalls wird ein Fehler geworfen) Darüber wird definiert welche Funktion auf dem Server aufgerufen werden soll.

Wenn zusätzlich das Feld `callback` mit einer Callback-Funktion belegt wird, wird eine `id` generiert, welche dem Server gesendet wird um unter dieser `id` zu antworten (Siehe oben). Das gesamte Objekt wird nun (natürlich ohne die Callback-Funktion) an den Server als JSON übertragen. Wenn keine Callback-Funktion definiert wurde, wird auch keine Antwort vom Server gesendet.

4.2.2 JSConnector

Das Pendant zum `ScalaConnector` auf der Serverseite ist der `JSConnector`. Dieser wurde mit Hilfe dynamischer Typisierung verwirklicht und ist unter `/app/js/JSConnector` zu finden.

Für die Kommunikation verwenden wir die Akka-Iteratees und Enumeratees (Siehe auch Abschnitt 2.1.3.2). Iteratees werden dabei für den Eingehenden Datenstrom über den Websocket und Enumeratees für die zu sendenden Daten verwendet. Da wir hier ein imperatives Modell entwickeln wollen, können wir zur Erzeugung des Enumeratee auf die `Concurrent.broadcast`-Funktion aus der Play API zurückgreifen (Für genauere Informationen verweisen wir hier auf die Dokumentation in [Typ12])

```
trait JSConnector {
  val (out, channel) = Concurrent.broadcast[JsValue]
  val in = Iteratee.foreach[JsValue] { json =>
    ...
  }
```

Über den `channel` ist es später möglich, Nachrichten an den Client zu senden (welcher per WebSocket an den Enumerator `out` verbunden ist). Die eigentlich interessanten Punkte sind jedoch wie beim Client das Empfangen sowie das Versenden von Nachrichten.

Beim Empfang wird genau wie im Browser zwischen den drei beschriebenen Fällen unterschieden:

```
val in = Iteratee.foreach[JsValue] { json =>
  (json \ "action").asOpt[String] match {
    case Some(a) => // an action is defined
      require(actions.isDefinedAt(a))
      // create a future for the action
      scala.concurrent.future(actions(a)(json \ "data")).onComplete {
        case Success(result) => // future succeeded
          (json \ "id").asOpt[Long].map(id => channel.push(Json.obj{
            "resultFor" -> JsNumber(id), // request id
            "data" -> js.convert(result), // result of invocation))
          case Failure(msg) => debug(msg) // future failed
      }
    // no action defined: this is a result message
    case None => (json \ "resultFor").asOpt[Long] match {
      case Some(id) =>
        val p: Promise[JsValue] = js.requests(id)
        if (!(json \ "success").as[Boolean]) // request failed
          p.failure(new Exception((json \ "message").as[String]))
        else // request succeeded
          p.complete(Try(json \ "data"))
          js.requests.remove(id)
      case None => debug('invalid message from websocket', json)
    }
  }
}.mapDone(_ => onClose)
```

Um die Ausführung nicht zu blockieren, verwenden wir Futures (Abschnitt 2.1.3.3), die Anfragen des Clients in eigenen Threads ausführen. Wenn der Client eine Anfrage stellt, wird der entsprechenden Funktion das Datenobjekt aus dem empfangenen JSON-Code übergeben und diese in einem Future ausgeführt. Wenn das Future die Ausführung erfolgreich beendet, wird das Ergebnis im Bedarfsfall (`id` ist in der Anfrage definiert) an den Client zurückgesandt. Im Fehlerfall wird das Ereignis geloggt. Weil Fehlerfälle für den Client uninteressant sind, da das Debugging auf dem Server stattfinden soll, werden sie in diese Richtung nicht übertragen. Wenn signalisiert werden soll, dass eine Aktion aus einem bestimmten Grund nicht ausgeführt werden kann, sollte mit einem geeigneten Datentyp ein wohldefiniertes Ergebnis von der entsprechenden Funktion

zurückgegeben werden (z.B: `Option[T]`).

Zu beachten ist, dass für ein erfolgreiches Versenden das Funktionsergebnis einen Typ haben muss, welcher die Typklasse `json.Writes` implementiert, damit die Daten konvertiert werden können. Ist dies nicht der Fall, kann der Code nicht übersetzt werden, da ein Typfehler vorliegt.

Für die Kommunikation nach „unten“ nutzen wir dynamische Typisierung. In einer Klasse, die den `JsConnector` einmischt werden drei Objekte zur Verfügung gestellt:

- `js.ignore` kann verwendet werden, um Kommandos oder Informationen an den Client zu senden, ohne dass eine Antwort erwartet wird.
- Über `js.async` können Anfragen versendet werden, für die man ein Future erhält, das bei Erhalt der Nachricht erfüllt wird.
- In manchen Fällen kann es notwendig sein, innerhalb eines Threads zu blockieren bis die Antwort vom Browser erhalten wurde. Dafür kann `js.sync` verwendet werden.

Der Aufruf `js.async.moveCursor(4,1)` könnte beispielsweise dazu führen, dass folgendes JSON-Objekt an den Browser gesendet wird:

```
{
  action: "moveCursor",
  id: 12803,
  data: [4,1]
}
```

Auf dem Client würde dadurch die Funktion `moveCursor` mit den beiden Argumenten `4` und `1` aufgerufen und das Ergebnis mit der Referenz-Id `12803` zurück an den Server gesendet, wo dann das Future mit dem Funktionsergebnis erfüllt würde und ein Callback auslösen würde.

Am Beispiel des `async`-Objekts wollen wir die Funktionsweise der Objekte kurz illustrieren. Es werden die drei Funktionen `selectDynamic`, `applyDynamic` sowie `applyDynamicNamed` implementiert, sodass Aufrufe der Form `js.async.foo`, `js.async.foo(1, bar)` sowie `js.async.foo(bar = 3, doo = 5)` möglich sind (Siehe dazu auch Abschnitt 2.1.1.5).

```
object async extends Dynamic {
  def selectDynamic(action: String): Future[JsValue] =
    applyDynamicNamed(action)()

  def applyDynamicNamed(action: String)(args: (String, Any)*): Future[JsValue] = {
    channel.push(JsObject(
      "action" -> JsString(action) ::
      "id" -> JsNumber(id) ::
      "args" -> JsArray(JsObject(args.map { case (n, a) => (n, convert(a)) }) :: Nil) ::
      Nil
    ))
  }
```

```
    val result = Promise[JsValue]()
    requests(id) = result
    id += 1
    result.future
  }

def applyDynamic(action: String)(args: Any*): Future[JsValue] = {
  channel.push(JsObject(
    "action" -> JsString(action) ::
    "id" -> JsNumber(id) ::
    "args" -> JsArray(args map convert) ::
    Nil
  ))
  val result = Promise[JsValue]()
  requests(id) = result
  id += 1
  result.future
}
}
```

4.3 Synchrone Repräsentation von Dokumenten

Da Isabelle/Scala intern mit absoluten Text-Offsets, die Browseranwendung dagegen mit Zeilen/Spaltennummern arbeitet, ist es notwendig, auf dem Server eine effiziente Umrechnung bereitzustellen. Bei langen Dokumenten wäre ein einfaches Durchlaufen des Dokuments, um die Positionen der Zeilenumbrüche zu erkennen bei jeder Änderung nicht performant genug. Um das Problem zu lösen, wurde der **LineBuffer** auf dem Server implementiert. Darüber hinaus wurde zur Synchronisierung die Klasse **RemoteDocumentModel** entwickelt.

4.3.1 LineBuffer

Der **LineBuffer** ist ein Textpuffer, welcher über zwei Zugriffsmethoden verfügt:

- Über **LineBuffer.chars** kann effizient über Offsets auf den Text zugegriffen werden. **chars** implementiert den Trait **IndexedSeq[Char]** aus der Scala-Standardbibliothek und ermöglicht so alle Funktionen, die von normalen Scala-Collections bekannt sind.
- Über **LineBuffer.lines** kann auf den Text zeilenweise zugegriffen werden. Es können außerdem Zeilen verändert, eingefügt und gelöscht werden. Dafür implementiert **lines** den Trait **Buffer[String]**.

Intern verwaltet der **LineBuffer** dafür zum einen einen effizienten **CharBuffer** mit dem Inhalt des Dokuments, zum anderen wird parallel ein **Buffer[(Int,Int)]** mit den Offsets der einzelnen Zeilen verwaltet.

```
class LineBuffer {  
  private var rngs = Buffer[(Int,Int)]()  
  private val buffer = Buffer[Char]()  
  
  ...  
  
  object lines extends Buffer[String] { ... }  
  
  object chars extends IndexedSeq[Char] { ... }  
}
```

Bei jeder Modifikation werden nun auf der einen Seite der Inhalt des Dokuments, auf der anderen Seite die Offsets aktualisiert. So zum Beispiel bei der Update-Funktion:

```
def update(n: Int, c: String): Unit = {
  require(!c.contains(newline), "updated line may not contain newlines")
  if (n == rngs.length) this += c
  else {
    val (of,ot) = rngs(n)
    val len = ot - of
    val diff = c.length - len
    buffer.remove(of, len)
    buffer.insertAll(of, c)
    rngs = (rngs.take(n) :+ (of,of + c.length)) ++
      rngs.drop(n + 1).map{ case (from,to) => (from + diff, to + diff) }
  }
}
```

Zunächst wird der Trivialfall überprüft, dass es sich um die Zeile hinter der letzten bisherigen handelt. In diesem Fall wird die Zeile über die `+=` Funktion angehängt und dort auch die Offsets eingearbeitet. In jedem anderen Fall wird der `buffer` aktualisiert und dann die bisherigen Offsets bis zur veränderten Zeile übernommen. Das der veränderten Zeile wird verkürzt bzw. verlängert und auf alle weiteren wird die Differenz zwischen neuer und alter Zeile aufaddiert. So ist es nie nötig, den Text tatsächlich zu durchlaufen. Der Aufwand ist damit deutlich reduziert.

Ein Aufruf `myLineBuffer.lines(5) = "Hallo"` würde so den bisherigen Text der sechsten Zeile mit dem Text `"Hallo"` ersetzen und die Offsets der sechsten bis letzten Zeile aktualisieren.

4.3.2 RemoteDocumentModel

Um Isabelle/Scala unter anderem mit den benötigten Datentypen zur Signalisierung von Textmodifikationen zu füttern, wird in der Klasse `RemoteDokumentModel` implementiert, die einen `LineBuffer` verwaltet und auf welche direkt die vom Browser erhaltenen `Changes` angewandt werden können. Dafür existiert die Funktion `change`, welche eine Liste von `isabelle.Text.Edit` zurückgibt, die dann an Isabelle/Scala übergeben werden kann.

Darüber hinaus wird hier die Versionsnummer des Dokuments, die auf selbe Art und Weise (nach jedem `ChangeSet` wird um eins hochgezählt) vom Client geführt wird verwaltet sodass, dem Client immer mitgeteilt werden kann, auf welche Version des Dokuments in einer Nachricht Bezug genommen wird. Dies ist zur Verhinderung von Inkonsistenzen auf dem Client notwendig.

4.4 Clientseitiges Syntax-Highlighting

Auf Grund der in Kapitel 3 beschriebenen Notwendigkeit, Verbindungskapazität einzusparen muss ein Teil des Syntax-Hightlightings, welches von Isabelle/Scala betrieben wird auf Browserseite nachgebildet werden. Dafür wird ein **CodeMirror-Mode** in CoffeeScript implementiert (zu finden unter `/app/assets/javascripts/mode/isabelle.coffee`).

Hier kann die glücklicherweise in [Wen12] detailliert beschriebene äußere Syntax einfach in reguläre Ausdrücke übersetzt werden. Die Möglichkeit der Stringinterpolation in CoffeeScript stellt sich hierbei als hilfreich heraus:

```
# extracted from the isabelle reference manual
greek    = "(?:\\|\\|\\|<(?:alpha|beta|gamma|delta|epsilon|zeta|eta|theta|iota|kappa|' +
'mu|nu|xi|pi|rho|sigma|tau|upsilon|phi|chi|psi|omega|Gamma|Delta|Theta|Lambda|Xi|' +
'Pi|Sigma|Upsilon|Phi|Psi|Omega)>)"
digit    = "[0-9]"
latin    = "[a-zA-Z]"
sym      = "[\\|!|\\|\\#|\\|\\$|\\|\\%|\\|\\&|\\|\\*|\\|\\+|\\|\\-|\\|\\|/|\\|\\|<|\\|\\|=|\\|\\>|\\|\\?|\\|\\@|\\|\\^|\\|\\_|\\|\\|/|\\|\\|~]"
letter   = "(?:#{latin}|\\|\\|\\|<#{latin}{1,2}>|#{greek}|\\|\\|\\|<^isu[bp]>)"
quasiletter = "(?:#{letter}|#{digit}|\\|\\_|\\|\\'|)"
ident    = "(?:#{letter}#{quasiletter}*)"
longident = "(?:#{ident})(?:\\.#{ident})+"
symident = "(?:#{sym}+|\\|\\|\\|<#{ident}>)"
nat      = "(?:#{digit}+)"
floating = "-?#{nat}\\|\\.#{nat}"
variable = "\\|\\|?#{ident}(?:\\.#{nat})?"
typefree = " #{ident}"
typevar  = "\\|\\|?#{typefree}(?:\\.#{nat})"
string   = "\\|\\|\\|\".*\\|\\|\\|\""
altstring = "\".*\""
verbatim = "\\|\\|\\|. *\\|\\|\\|"
```

Als besonders schwierig stellte sich die Erkennung von Kontrollsymbolen für die korrekte Darstellung von Sub- und Superskript bzw. fettgedruckten Zeichen sowie der Spezialsymbole, welche als entsprechende L^AT_EX-Symbole dargestellt werden sollen heraus, da diese an jeder beliebigen Stelle in der Syntax vorkommen können und so nicht leicht gleichzeitig in einem regulären Parser mit entsprechenden Klassen markiert werden können. Um dieses Problem zu lösen wird ein Trick angewandt: Es werden zwei Grammatiken implementiert, die den Zeichenstrom zeilenweise simultan verarbeiten. Die Ergebnisse werden dann kombiniert, sodass für jedes Token jeweils die Vereinigung der Ergebnisse beider Parser zurückgegeben wird.

Bei der Parsierung werden jeweils nur die sichtbaren Zeilen verarbeitet, deswegen ist es notwendig,

am Ende jeder Zeile einen eindeutigen Zustand zurückzugeben, mit dem die Verarbeitung der nächsten Zeile ohne weitere kontextuelle Informationen fortgesetzt werden kann. Da Isabelle wie ML verschachtelte Kommentare zulässt, muss dort unter anderem auch die Kommentarebene verwaltet werden.

Um der Variabilität der äußeren Syntax gerecht zu werden (Die Schlüsselwörter können sich verändern), werden die Schlüsselwörter als Parameter an den Parser übergeben, so dass dieser bei Bedarf neu initialisiert werden kann.

4.5 Serverseitiges Syntax-Highlighting

Für das Highlighting auf dem Server wird die Isabelle/Scala-Schnittstelle verwendet. Die Herausforderung besteht nun darin, die Daten in geeigneter Form an den Client zu übermitteln, damit dieser das verzögerte Highlighting der inneren Syntax vornehmen kann. Zur Illustration ist in Abbildung 4.2 der Serverseitig gehighlightete Teil der Syntax (innere Syntax) grün unterstrichen.

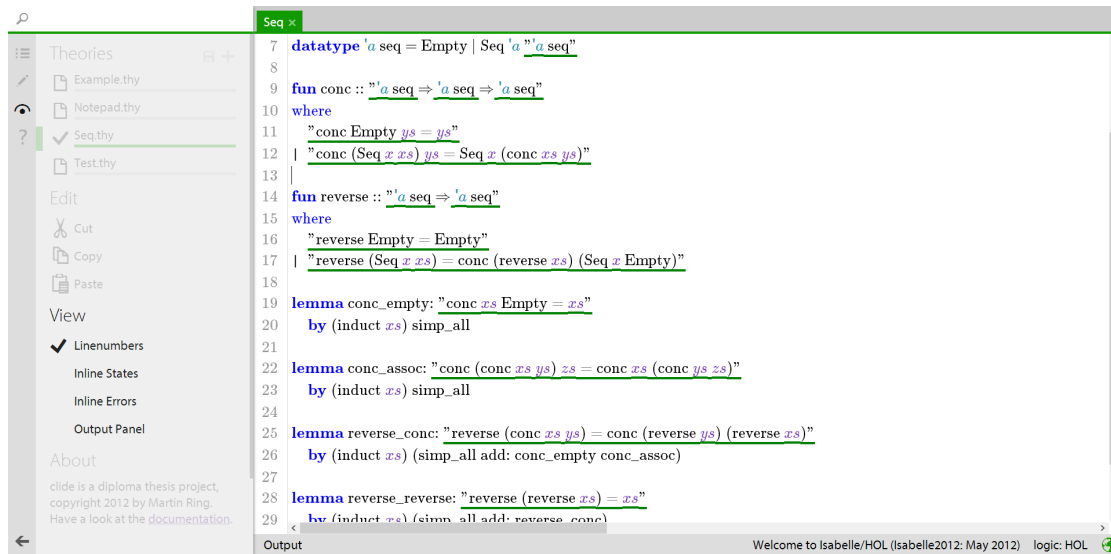


Abbildung 4.2 Serverseitiges Syntax-Highlighting

Im Browser wird eine Liste der Kommandos im Quelltext verwaltet, welche der Server ständig durch Nachrichten aktualisiert. Dafür wurde auf dem Client in `isabelle.coffee` ein Backbone-Modell für die Repräsentation der relevanten Daten von Kommandos (`Command`) entworfen.

```
class Command extends Backbone.Model
  ...
```

```
class Commands extends Backbone.Collection
  model: Command
  getCommandAt: (line) =>
    ...
  getTokenAt: (line, column) =>
    ...
```

Das **Commands**-Modell dient der Verwaltung der Liste und ist eine Backbone-Collection, die den Vorteil bietet, dass sie über Callbacks für Modifikationen (on 'remove', on 'add', ...) verfügt und somit das Markup im Editor bei Bedarf aufgefrischt werden kann.

Die Klasse **Session** in **isabelle.coffee** dient hier als Schnittstelle für den Server und wird über einen **ScalaConnector** (Abschnitt 4.2) mit dem **WebSocket** verbunden.

Auf Serverseite existiert ebenfalls eine Klasse **Session** in **Session.scala** welche wiederum das Gegenstück darstellt und über den **JSConnector** verbunden ist.

Wenn nach einer Änderung am Dokument neue Kommandos erkannt werden bzw. sich Kommandos verändert haben oder Kommandos wegfallen, wird dies über den Nachrichtenkanal **Session.commandsChanged** in Erfahrung gebracht und die Information aufbereitet.

```
session.commands_changed += { change =>
  change.nodes.foreach { node =>
    delayedLoad(node)
    val snap = session.snapshot(node, Nil)
    val status = Protocol.node_status(snap.state, snap.version, snap.node)
    js.ignore.status( ... )
    for {
      doc <- docs.get(node)
    } {
      js.ignore.states(node.theory, MarkupTree.getStates(snap, doc.buffer.ranges))
      val cmds = snap.node.commands.map(_.id)
      doc.commands.keys.foreach { id =>
        if (!cmds.contains(id)) {
          doc.commands.remove(id)
          js.ignore.removeCommand(node.toString, id)
        }
      }
    }
  }
  change.commands.foreach(pushCommand)
}
```

Im `RemoteDocumentModel` wird dafür eine synchrone Repräsentation der Kommandos verwaltet. Sollte ein Kommando hier nicht mehr existieren, wird es über die Nachricht

```
js.ignore.removeCommand(...)
```

im Browser entfernt, damit die Ressourcen dort freigegeben werden können bzw. Syntaxmarkierungen aus dem Dokument entfernt werden können.

Über `pushCommand` werden alle veränderten Kommandos verarbeitet. Wenn sich dann die gefilterten, relevanten Informationen von denen im `RemoteDocumentModel` unterscheiden, werden die neuen Informationen dort an den Client weitergeleitet. In diesen Informationen enthalten sind:

- Token aus der Inneren Syntax,
- Token mit Typinformationen,
- Fehlerhafte Token mit den zugehörigen Fehlermeldungen sowie
- der Beweiszustand des Kommandos.

Weil die verschiedenen Tokenmengen nicht zwingender Weise disjunkt sind, werden die Vereinigungen der jeweiligen Informationen betrachtet und Überschneidungen gegebenenfalls aufgebrochen (in `MarkupTree.scala`).

Auf dem Client werden von den Editoren (`Editor.coffee`) die Callbacks der Kommandoliste der jeweils angeschlossenen Theorie verarbeitet und dann die Informationen in die Darstellung integriert.

```
includeCommand: (cmd) =>
  if cmd.get('version') is @model.get('currentVersion') then @cm.operation =>
    unless cmd.get('registered')
      cmd.on 'remove', (cmd) => if cmd?
        for m in cmd.get 'markup'
          m.clear()
        wid = cmd.get('widget')
        if wid?
          @cm.removeLineWidget(wid)
        cmd.set registered: true

    # add line widget
    @addCommandWidget(cmd)

    # mark Stuff
    old = cmd.get('markup')
    if old?
      for m in old
        m.clear()
```

```
range = cmd.get 'range'
length = range.end - range.start
marks = []
for line, i in cmd.get 'tokens'
  l = i + range.start
  p = 0
  for tk in line
    from =
      line: l
      ch: p
    p += tk.value.length
    unless (tk.type is "text" or tk.type is "'")
      to =
        line: l
        ch: p
      marks.push(@cm.markText from,to,
        className: "cm-#{tk.type.replace(/\.\/g,' cm-')}"
        tooltip: tk.tooltip
        __isabelle: true)
  cmd.set((markup: marks),(silent: true))
```

4.6 Substitution von Symbolen

Die in den vorherigen Abschnitten beschriebenen Tokenklassen werden verwendet, um Symbole direkt im Quelltext zu substituieren. Da CodeMirror glücklicherweise seit Version 3.0, welche gegen Ende der Bearbeitungszeit dieser Diplomarbeit veröffentlicht wurde die Möglichkeit bietet, Textstellen durch HTML-Widgets zu ersetzen, konnte ein vorheriger serverseitiger Ansatz, der von Natur aus recht fehleranfällig war, verworfen werden.

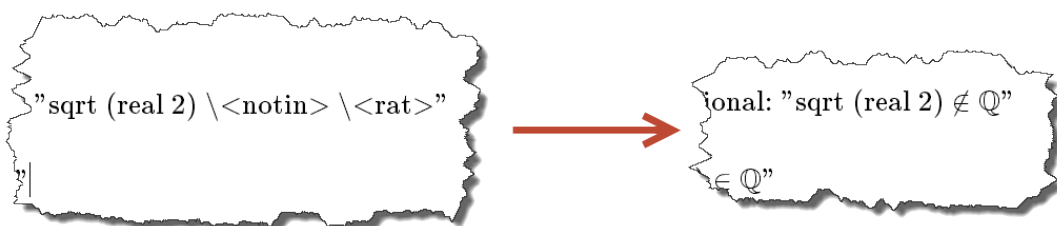


Abbildung 4.3 Substitution von Symbolen

Die grundsätzliche Idee bei der Symbol-Substitution ist es, den eigentlichen Isabelle-Quelltext

auf Clientseite unverändert zu lassen und nur die Visualisierung anzupassen. Die Substitution findet in `/app/assets/javascripts/rjs/editor.coffee` statt.

Hierfür werden während des Ladens eines Dokuments zunächst alle Vorkommen von Spezialsymbolen ersetzt. Dafür bedienen wir uns des CodeMirror Plugins `SearchCursor`, das es erlaubt den Text mit regulären Ausdrücken effizient zu durchsuchen.

```
cursor = @cm.getSearchCursor(special)
while cursor.findNext()
  sym = symbols[cursor.pos.match[0]]
  if sym?
    from = cursor.from()
    to = cursor.to()
    @cm.markText from, to,
      replacedWith: sym(),
      clearOnEnter: false
```

Im laufenden Betrieb werden dann bei jeder Veränderung die Tokenklassen an den veränderten Positionen betrachtet, um zu entscheiden, ob es sich um Spezialsymbole handelt.

Dafür wird das `onchange`-Callback der CodeMirror-Instanz implementiert. Wir verwenden an dieser Stelle `.operation` um die gesamte Operation auszuführen, bevor die Visualisierung angepasst wird, dadurch wird die Ausführungsgeschwindigkeit drastisch erhöht. Zudem stellen wir sicher, dass keine Selektion vorliegt, da in diesem Fall zunächst keine Ersetzungen stattfinden sollen bis die Selektion wieder aufgehoben wurde.

```
@cm.on 'change', (editor, change) => editor.operation => unless editor.somethingSelected()
  ...
```

Nun löschen wir alle an dieser Stelle zuvor eingeführten Substitutionen (durch die Eingabe kann sich das zu substituierende Zeichen verändert haben). Da wir `CodeMirror.operation` verwenden, ist diese Aktion relativ performant.

```
pos = change.to
token = editor.getTokenAt(pos)
marks = editor.findMarksAt(pos)
mark.clear() if mark.__special for mark in marks
```

Wenn es sich bei dem aktuellen Token nun um ein spezielles Zeichen handelt, dann wird es mit Hilfe des CoffeeScript Moduls `symbols`, welches über RequireJS importiert wurde zu einem Widget übersetzt, das dann als Textsubstitution eingesetzt werden kann.

```
if token.type? and (token.type.match(/special|symbol|control|sub|sup|bold/))
```

```
wid = symbols[token.string]
if wid?
  @cm.markText from,to,
    replacedWith: wid(token.type)
    clearOnEnter: false
    __special: true
```

Das `symbols`-Modul wurde aus der Datei `/etc/symbols` in der Isabelle Plattform abgeleitet. Zusätzlich wurden Informationen über den zu verwendenden \LaTeX -Font (Caligraphic, Fraktur, AMS, ...) für jedes Symbol manuell eingearbeitet.

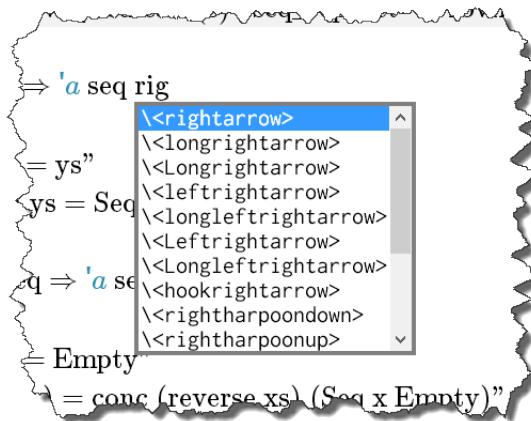


Abbildung 4.4 Vervollständigung von Symbolsubstitutionen

4.7 Modale Dialoge

Um die Funktionsweise von `Backbone.View` und das Zusammenspiel mit dem `ScalaConnector` zu illustrieren, betrachten wir an dieser Stelle ein einfaches Beispiel. An manchen Stellen in der Anwendung ist es notwendig, den Benutzer mit modalen Dialogen zur Beantwortung einer Frage aufzufordern. Hierfür wurde das Modul `Dialog.coffee` entworfen.

```
define ->
  class Dialog extends Backbone.View
```

Beim Initialisieren wird abhängig von den übergebenen Optionen die UI aufgebaut. Mögliche Optionen sind dabei

- ein String `title` um eine Überschrift zu erzeugen,

- ein String **message** um die eigentliche Nachricht zu spezifizieren,
- ein Array **buttons** um festzulegen, welche Buttons angezeigt werden soll,
- **defaultAction** um den Standard Button festzulegen, welche ausgelöst wird, wenn der Nutzer einfach Enter drückt,
- ein String **defaultText**, welcher zum Festlegen eines **input[type=text]** Controls sowie eines eventuellen vorgegebenen Inhalts dient sowie
- ein Callback **done** um das Ergebnis der Nutzereingabe zu verarbeiten.

Die Optionen können weggelassen werden, sodass auf Standardwerte zurückgegriffen wird, bzw. der Titel oder die Nachricht weggelassen werden.

Wenn ein Button geklickt wird oder Enter gedrückt wird, wird die **exit**-Funktion aufgerufen, welche den Dialog aus dem DOM entfernt und das **done**-Callback aufruft. **text** ist dabei **undefined**, wenn kein Input-Control erzeugt wurde.

```
exit: (action) => () =>
  text = @$('#input').val()
  @$el.remove()
  if @options.done then @options.done
    action: action
    text: text
```

Nun können wir beispielsweise als Aktion eines Kontextmenüeintrages „Löschen“ zunächst einen Modalen Dialog erstellen um sicherzugehen, dass die Aktion vom Nutzer gewollt ist.

```
menu.show(e.pageX,e.pageY,[
  text: 'Delete'
  command: ->
    new Dialog
      title: "Delete Project"
      message: "Do you really want to delete project '#{$(elem).data('name')}'?"
      buttons: ['Yes','No']
      defaultAction: 'Yes'
      done: (e) => if e.action is 'Yes' then deleteProject(user,$(elem).data('name'))
])
```

Das Spannende daran ist nun, dass es möglich ist, diese Dialoge über den **JsConnector** bzw. den **ScalaConnector** vom Server aus zu erzeugen und zu nutzen. So kann der Server bei Bedarf über **js.sync** eine blockierende Frage auf dem Client stellen und erst nach der Antwort des Nutzers fortfahren, genauso wie es in Desktopanwendungen über Messageboxen üblich ist.

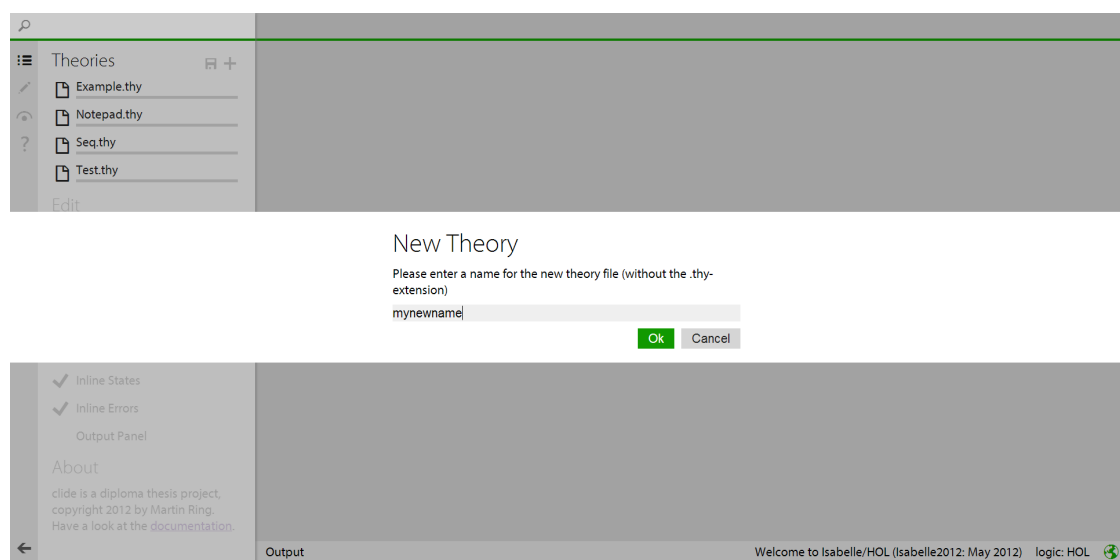


Abbildung 4.5 Ein modaler Dialog

Kapitel 5

Bewertung

Das Ergebnis dieser Arbeit ist ein innovatives, funktionstüchtiges Werkzeug (*clide*), das in der Lage ist dem Nutzer bei der Arbeit mit Beweisdokumenten eine sinnvolle und produktive Unterstützung zu gewähren. Als Maßstab für die Einordnung der Ergebnisse bietet sich ein Vergleich mit dem bisherigen Ansatz Isabelle/jEdit an.

Dabei ist allerdings zu beachten, dass es sich bei diesem Projekt auch um eine Machbarkeitsstudie für die Umsetzung einer webbasierten IDE handelt, da es nicht möglich ist eine sogenannte *full featured* IDE im Rahmen einer Diplomarbeit zu entwickeln und dabei zusätzlich die zahlreichen neuen Aspekte im Zusammenhang mit der Kommunikation einer speziellen Webanwendung zu beachten.

5.1 Performanz

Da die selbe Bibliothek (Isabelle/Scala) wie in der jEdit Version zu Grunde liegt, gibt es keine Unterschiede in der Geschwindigkeit des Beweisers selbst. Bei der lokalen Ausführung auf einem Rechner mit Server und Browser ist zu erwarten, dass es durch die Kommunikation über serialisierte Nachrichten und deren Umrechnung zu leichten Geschwindigkeitseinbußen gegenüber Isabelle/jEdit kommt. Die Performanz auf diese Weise zu vergleichen ist jedoch nicht sinnvoll, da in einem gewöhnlichen Szenario der Server auf einem zentralen Rechner ausgeführt wird, der über die nötigen Ressourcen verfügt, um die Isabelle Plattform tragen zu können. Der Zugriff geschieht dann über mehrere Clients, die keine besonders hohen Ansprüche erfüllen müssen. Hier liegt ein Vorteil gegenüber der jEdit Implementierung. Mit *clide* ist es theoretisch möglich, einen Theorembeweiser von einem Netbook oder einem Tablet PC aus zu verwenden.

5.2 Funktionalität

Die geforderten Funktionen konnten alle implementiert werden. Es existieren vielfältige Möglichkeiten für den Nutzer.

Ein Nachteil gegenüber dem Proof General ist die fehlende Möglichkeit der **semantischen Suche** in Isabelle Quellen. Diese Funktionalität wird von der Isabelle Plattform bereit gestellt und ist bis jetzt leider von der Isabelle/Scala Schnittstelle noch nicht unterstützt (und daher auch in Isabelle/jEdit nicht vorhanden).

Darüber hinaus fehlt natürlich die Infrastruktur an Plugins welche für jEdit existiert, da es sich bei der Anwendung notwendiger Weise um eine Neuentwicklung handelt. Das macht es natürlich mühsamer zusätzliche Funktionalität wie Versionsverwaltungssystem oder andere Erweiterungen, zu integrieren, da diese dann speziell für diese neue Plattform entwickelt werden müssen.

Eine große Zahl von Funktionen ist nur über Tastenkürzel verfügbar, jedoch nicht über Mausaktionen, diese Nachzurüsten ist eher Fleißarbeit, da die Infrastruktur für Kommandos in der Sidebar bereits fertiggestellt wurde. Dieser Umstand betrifft damit eher die Usability in Abschnitt 5.4.

5.3 Zuverlässigkeit

Da die Browseranwendung zwingend in JS, also einer dynamischen Sprache entwickelt werden musste ist es schwieriger, Fehler zu entdecken, da Typfehler in sprachen mit dynamischer Typisierung logischerweise nicht existieren und der Compiler so keine große Hilfe bei der Fehlerfindung darstellt. Während der Entwicklung wurden sogenannte *cross checks* im Code geführt (Welche nun auskommentiert sind (Siehe z.B. `isabelle.coffee`)) um die Datenkonsistenz regelmäßig zu überprüfen. Das reicht aber natürlich nicht aus um die Fehlerfreiheit zu garantieren und da darüber hinaus das systematische Testen von Benutzeroberflächen immer ein Problem darstellt, kann nicht ausgeschlossen werden, dass sich noch Fehler in der Benutzerschnittstelle befinden.

Serverseitig mussten vor allem der `LineBuffer` und das `RemoteDocumentModel` getestet werden. Hierfür wurde zum einen eine zuschaltbare Visualisierung der Serverseitigen Daten implementiert sowie einige in `ScalaTest` implementierte Testfälle (`/test/scala/...`). Die meiste andere Funktionalität stammt aus Isabelle/Scala bzw. Play und wurde nur verknüpft.

5.4 Benutzbarkeit

Da es sich um eine Benutzerschnittstelle handelt ist Usability ein wichtiges Thema. Da es sich hier um eine Machbarkeitsstudie handelt, wurden einige Aspekte der Usability zunächst hinten angestellt. Insbesondere auch die Barrierefreiheit, da diese in einer *single-page*-HTML-Anwendung bislang schwer zu erreichen ist, nicht zuletzt auch weil viele unkonventionelle Elemente verwendet werden mussten. Anstelle von ausgiebigen Usability Tests wurde die Anwendung einigen Probanden (Kommilitonen) zur Benutzung vorgelegt, und deren Verhalten beobachtet. Dabei wurden einige Lücken gewahrt, die noch geschlossen werden konnten. So wurden beispielsweise nach den Tests die meisten Kommandos auf Standard Tastenkombinationen gelegt oder eigene modale Dialoge anstelle der Browser-Dialoge eingeführt.

Isabelle/jEdit hat hier natürlich den Vorteil, dass es in einer ausgereiften Umgebung (jEdit) lebt, die über einige hundert Personenjahre Entwicklungszeit entstanden ist und damit vielfältig ist.

Die Grundlagen für eine benutzerfreundliche Anwendung wurden aber dennoch geschaffen. Insbesondere wurde von Anfang an viel Wert auf die *User Experience* gelegt. In den oben erwähnten Tests wurde diese häufig als gut gelungen bewertet und es konnte beobachtet werden, dass die Probanden Spaß an der Nutzung der Anwendung hatten, selbst bei denen, die das eigentliche Thema des Theorembeweisens nicht interessierte. Das ist eine gute wichtige Voraussetzung für den erfolgreichen Einsatz in der Lehre, da für das Lernverhalten Emotionen eine wichtige Rolle spielen [Nor03].

Ein wichtiger Punkt der Benutzbarkeit ist auch der Konfigurationsaufwand. Dieser ist hier für die Endnutzer welche den von einem Administrator einmal eingerichteten Server verwenden praktisch nicht vorhanden, so dass direkt mit der Inhaltlichen Arbeit gestartet werden kann. Auch dies ist besonders für die Lehre von Interesse, da es nicht zuträglich ist, wenn sich Studierende mit der Installation und Konfiguration der Plattform länger Aufhalten als mit den ersten Versuchen darauf.

5.5 Visualisierung

Die gesamte Anwendung im Browser nutzt keine gerasterten Grafiken, so dass es möglich ist hochqualitative Ergebnisse auf Geräten mit geeigneten Bildschirmen zu erhalten (Abbildung 5.1).

Bei der eigentlichen Visualisierung der Dokumente ist eine klare Verbesserung gegenüber Isabelle/jEdit zu erkennen. Während Isabelle/jEdit sich am festen Textraster, das in jEdit vorgegeben ist, orientieren musste und nur geringe Abweichungen implementieren konnte, ähnelt die Darstellung in der Bearbeitung bei der Webanwendung schon sehr stark der endgültigen Darstellung in veröffentlichten L^AT_EX-Dokumenten (Siehe Abbildung 5.2)

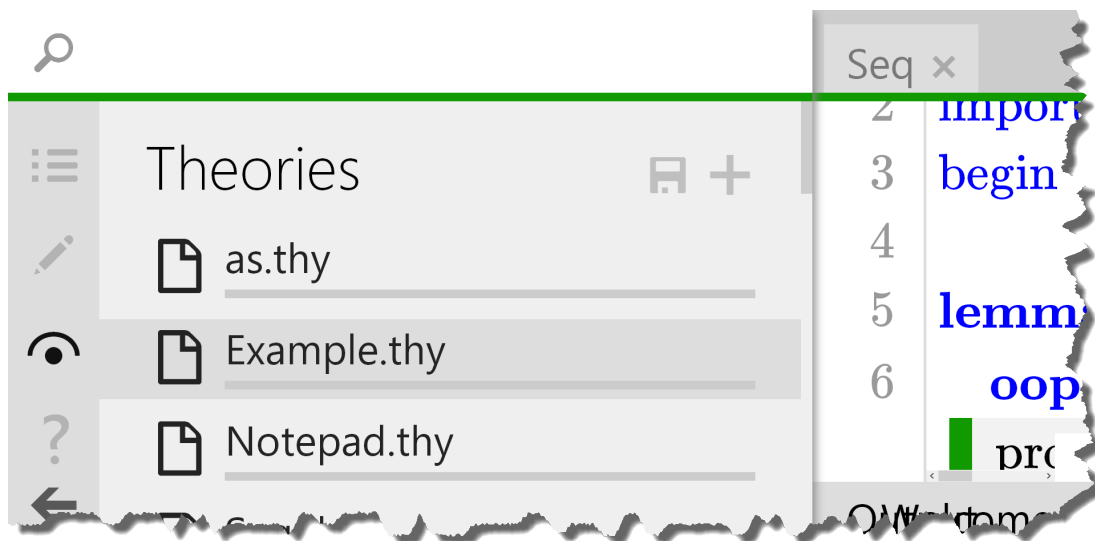


Abbildung 5.1 jEdit vs. clide

Darüber hinaus ist die Integration der Beweiser-Ausgaben flexibler gestaltet und es ist auch möglich diese teilweise oder vollständig *inline* anzuzeigen.

5.6 Übertragbarkeit

clide ist auch deswegen so interessant, weil es eine ganz neue Form der Veröffentlichungen ermöglicht. Durch die gute Visualisierung ist es denkbar, Dokumente direkt mit anderen über die Webanwendung zu teilen. Diese Visualisierung könnte in Zukunft auch als statische *read-only*-Variante zur Verfügung gestellt werden.

Die Webanwendung kann als Grundlage für die Integration weiterer Sprachen im Zusammenhang mit Theorembeweisen genutzt werden. Für die Lehre wäre es denkbar Aufgaben direkt online zu stellen und lösen zu lassen. Während Isabelle/jEdit auf einem lokalen Rechner „gefangen“ ist, können hier ganz neue Optionen der Integration gewählt werden.

Nachdem die Hürden der Kommunikation und der grundsätzlichen Infrastruktur genommen wurden, kann *clide* durch seine skalierbare Architektur in alle denkbaren Richtungen erweitert werden und damit ein mächtiges Werkzeug geschaffen werden.

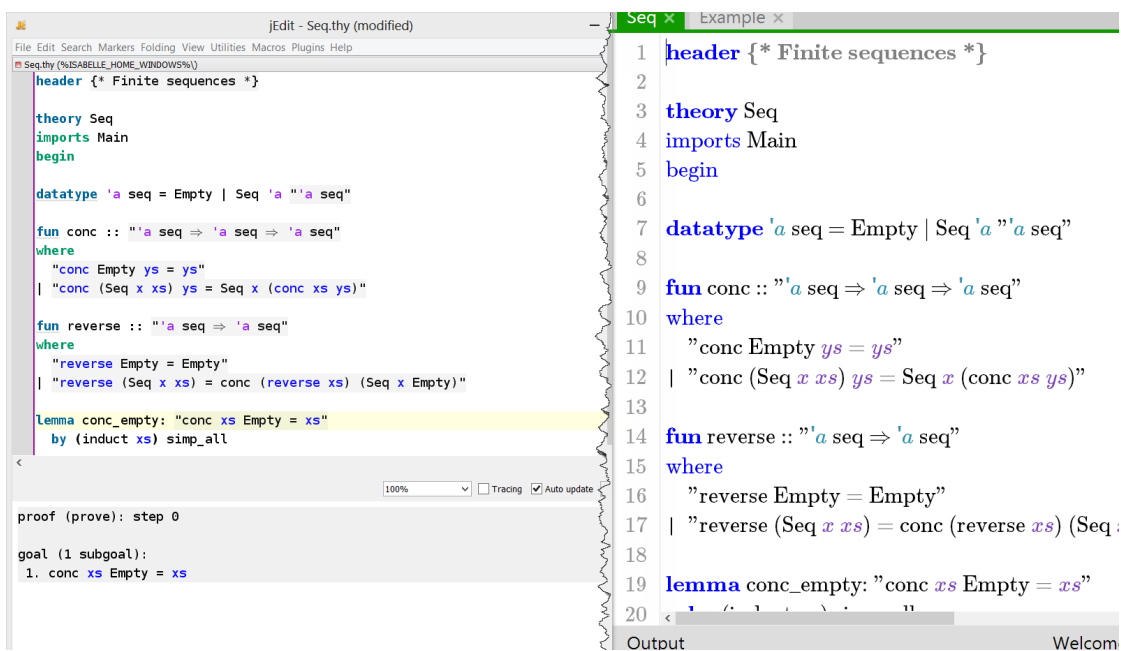


Abbildung 5.2 jEdit vs. clide

Kapitel 6

Zusammenfassung und Ausblick

Die aus dieser Arbeit resultierende Anwendung ist ein ganz neuer Ansatz und es handelt sich um weitestgehend unbetretenes Terrain. Während der Zeit der Entwicklung wurden nach und nach immer mehr webbasierte Entwicklungsumgebungen für Mainstream-Sprachen bekannt. Am meisten Aufsehen hat dabei wohl *koding.com* erregt, was im Moment in eine nicht-öffentliche Beta-Phase eingetreten ist zu der für uns leider kein Zugang besteht. Diese neuen Web-IDEs sind zwar nicht ganz vergleichbar, da für Standard-Programmiersprachen, im Gegensatz zu Isabelle, beispielsweise das Syntax- Highlighting vollständig im Browser stattfinden kann. Jedoch zeigt dieser Trend, dass sehr wohl ein Bedarf an IDEs im Browser besteht. Weil diese Entwicklungen parallel stattfanden und keine quelloffenen Vorbilder bekannt sind, musste das gesamte Konzept der Anwendung neu entwickelt werden. Da in der momentan langsam endenden Findungsphase für den neuen HTML5-Standard eine sehr hohe Fluktuation im Bereich der Webentwicklung besteht, fiel es schwer, fundierte Designentscheidungen zu treffen. Neue Webframeworks sprießen ständig aus dem Boden und bewährte Entwurfsmuster wie sie für klassische Desktopanwendungen bestehen, sucht man vergebens. Alleine für das Problem der Modularisierung von JS-Code gibt es unzählige Varianten, die sich alle als „die eine Lösung“ verkaufen. Dankbarer Weise wurden durch die Gründung der Firma *Typesafe* durch Martin Odersky selbst einige Frameworks für Scala, darunter Play, zu Mitgliedern des *Typesafe Stack* gekrönt und damit deren Weiterentwicklung sichergestellt bzw. auch deren Dokumentation vorangetrieben. Dennoch erschienen im Laufe der Entwicklung zwei neue Scala Versionen, welche die jeweils Binärkompatibilität gebrochen haben, sowie eine neue Version von Play (2.1), die viele Probleme gelöst hat, aber auch dazu führte, dass viele halbfertige eigene Lösungen für die in Play 2.0 bestehende Probleme wieder verworfen werden mussten.

Trotzdem muss das Ergebnis einen Vergleich mit den bisherigen Lösungen nicht scheuen. Es gilt noch einiges an der Benutzbarkeit und der Browserkompatibilität zu erweitern, die Infrastruktur dafür ist aber nun vorhanden und für ein größeres Team sollte es kein Problem darstellen, diese Lösung zu einem vollwertigen Ersatz für bisherige Werkzeuge auszubauen. Darüber hinaus wurden einige revolutionäre Neuerungen im Bereich der interaktiven Theorembeweiser implemen-

tiert, wie die Aufteilung in Server und Client sowie die neuen Möglichkeiten der Visualisierung. Für eine produktive Arbeit mit den Daten wäre es sinnvoll, das in der *Proover Community* verbreitete Versionsverwaltungssystem *Mercurial* zu integrieren und damit den notwendigen Austausch von Daten zu gewährleisten.

Die entwickelte IDE kann als Ansatz für die Implementierung einer universellen Entwicklungsumgebung dienen. So ist es sehr gut denkbar, den neuen *Presentation Compiler* für Scala in die Anwendung zu integrieren und damit auf einfache Weise eine Entwicklungsumgebung für Scala zu erstellen. Vor allem aber wäre es interessant, die für die im Bereich der Theorembeweise anderen relevanten Sprachen wie SML, Lisp oder Haskell zu unterstützen und damit eine abgerundete Arbeitsumgebung bereitzustellen.

Anhang A

Appendix

A.1 Abbildungsverzeichnis

2.1	Konzept des Document Model in Isabelle/Scala	14
2.2	Ajax Modell einer Web-Anwendung (asynchrone Datenübertragung)	19
3.1	Die clide-Oberfläche	31
3.2	Das Anmeldeformular	32
3.3	Die Projektübersicht	33
3.4	Die Sidebar	34
3.5	Inline-Anzeige von Beweiszuständen	37
3.6	Datenfluss in clide	37
4.1	Architekturlayout	39
4.2	Serverseitiges Syntax-Highlighting	49
4.3	Substitution von Symbolen	52
4.4	Vervollständigung von Symbolsubstitutionen	54
4.5	Ein modaler Dialog	56
5.1	jEdit vs. clide	60
5.2	jEdit vs. clide	61

A.2 Tabellenverzeichnis

3.1	Kompatibilität der gängigsten Browser mit den Verwendeten Standards	30
-----	---	----

A.3 Literatur

- [] *BSON 1.0 Specification*. URL: <http://bsonspec.org/#/specification>.
- [BT98] Yves Bertot und Laurent Théry. »A generic approach to building user interfaces for theorem provers«. In: *Journal of Symbolic Computation* 25.7 (Feb. 1998).
- [Fie99] R. Fielding. *Hypertext Transfer Protocol 1.1*. Techn. Ber. IETF, Juni 1999.
- [Fla97] David Flanagan. *JavaScript: The Definitive Guide*. Second. O'Reilly & Associates, Jan. 1997. ISBN: 1-56592-234-4.
- [Fra11] Maximilian Vollendorf Frank Bongers. *jQuery: Das Praxisbuch*. 2. Aufl. Galileo Computing, 2011. ISBN: 3836218100.
- [Hal12] Philipp Haller. *Actors in Scala*. 1. Aufl. Artima, 2012. ISBN: 0981531652.
- [Hic12] Ian Hickson. *The WebSocket API*. W3C Working Draft (work in progress). W3C, Sep. 2012.
- [Jäg07] Kai Jäger. *Ajax in der Praxis: Grundlagen, Konzepte, Lösungen*. 1. Aufl. Springer, 2007. ISBN: 3540693335.
- [Kes12] Anne van Kesteren. *DOM*. W3C Working Draft (work in progress). W3C, Dez. 2012.
- [Lex11] Martin Odersky Lex Spoon Bill Venners. *Programming In Scala*. 2. Aufl. Artima, 2011. ISBN: 0981531644.
- [Mic12] Microsoft. *UX guidelines for Windows Store apps*. Nov. 2012. URL: <http://msdn.microsoft.com/en-us/library/windows/apps/hh465424.aspx>.
- [Nor03] Donald A. Norman. *Emotional Design: Why We Love (or Hate) Everyday Things*. 1. Aufl. Basic Books, 24. Dez. 2003. ISBN: 0465051359.
- [Osm12] Addy Osmani. *Developing Backbone.js Applications*. 1. Aufl. O'Reilly Media, 2012. ISBN: 1449328253.
- [Smi12] M. K. Smith. *HTML: The Markup Language (an HTML language reference)*. W3C Working Draft (work in progress). W3C, Okt. 2012.
- [Sur12] Josh Sureth. *Iteratees*. Feb. 2012. URL: <http://jsuereth.com/scala/2012/02/29/iteratees.html>.
- [Typ12] Typesafe. *Play 2.0 documentation*. 2012. URL: <http://www.playframework.org/documentation/2.0.4/Home>.
- [Wen07] Makarius Wenzel. *Isabelle/Isar – a Generic Framework for Human-Readable Proof Documents*. 2007.
- [Wen09] Makarius Wenzel. *Parallel Proof Checking in Isabelle/Isar*. Techn. Ber. Technische Universität München, 2009.
- [Wen10] Makarius Wenzel. *Asynchronous Proof Processing with Isabelle/Scala and Isabelle/jEdit*. Techn. Ber. Université Paris Sud 11, LRI, Orsay France, 2010.
- [Wen11] Makarius Wenzel. »Isabelle as document-oriented proof assistant«. In: *Proceedings of the 18th Calculemus and 10th international conference on Intelligent computer*

- mathematics*. MKM'11. Bertinoro, Italy: Springer-Verlag, 2011. ISBN: 978-3-642-22672-4. URL: <http://dl.acm.org/citation.cfm?id=2032713.2032732>.
- [Wen12] Makarius Wenzel. *The Isabelle/Isar Reference Manual*. 2012.

A.4 Liste der Abkürzungen

ACE AJAX.org Cloud9 Editor, S. 35, 36

AJAX Asynchronous JavaScript and XML, S. 18, 20, 28

BSON Binary JSON, S. 29

CSS Cascading Style Sheets, S. 15, 16, 25, 26, 35

DOM Document Object Model, S. 15, 20, 55

HTML Hypertext Markup Language, S. 15, 16, 20, 25, 29, 31, 35

HTTP Hypertext Transfer Protocol, S. 18, 19, 24, 28, 32

Isar Intelligible semi-automated reasoning, S. 12

JS JavaScript, S. 15, 17, 18, 20, 21, 23, 25, 26, 30, 35, 58, 63

JSON JavaScript Object Notation, S. 18, 29

JVM Java Virtual Machine, S. 1, 5

LESS Less CSS, S. 26

REPL read-eval-print-loop, S. 1

sbt Simple Build Tool, S. 9, 11

SML Standard ML, S. 12

UI User Interface, S. 31, 32, 54

URL Uniform Resource Loactor, S. 26

XML Extensible Markup Language, S. 15

Anhang B

Installationsanweisungen

Für die lokale Ausführung des Servers kann die auf dem beiliegenden USB-Stick bereitgestellte Software verwendet werden. Dafür müssen zunächst folgende Voraussetzungen erfüllt sein:

1. Die Isabelle 2012 Plattform muss installiert werden. Diese ist zusammen mit detaillierten Installationsanweisungen zu finden unter:

<http://isabelle.in.tum.de/>

Wichtig ist hierbei, dass die Umgebungsvariable **ISABELLE_HOME** gesetzt ist, damit die Anwendung den Ort der Distribution feststellen kann.

2. Ein aktuelles *Java Development Kit* (JDK) der Version 7 oder höher muss installiert und als Standardplattform konfiguriert sein. (JDK < 7 führt zu unerwarteten Fehlermeldungen und Abstürzen, da Inkompatibilitäten bestehen.)

<http://www.oracle.com/technetwork/java/javase/downloads/index.html>

3. Das Play Framework in Version 2.1 (Nicht 2.0.1!!) ist zu installieren. Es muss sichergestellt sein, dass **play** im Pfad steht und damit von der Konsole aus aufrufbar ist. Zu diesem Zeitpunkt ist die aktuelle Version 2.1-RC1

<http://download.playframework.org/releases/>

Um die Anwendung zu starten, muss zunächst in das Stammverzeichnis gewechselt werden (entweder der USB-Stick oder eine Kopie davon auf der lokalen Platte).

Hier kann nun das Kommando

```
> play start
```

ausgeführt werden, wodurch der Server gestartet wird. Mit *CTRL-D* kann die Log-Ausgabe verlassen werden.

Die Anwendung ist nun unter <http://localhost:9000> vom Browser aus Verfügbar. Es wird empfohlen, Google Chrome in Version 24 oder höher für die Betrachtung zu verwenden.

Standardmäßig ist der Benutzer **martinring** mit dem Passwort **secret** eingerichtet. Weitere Nutzer können durch Anpassen der Datei `/data/.users` hinzugefügt werden.