# ATOM EDITOR SUPPORT FOR THE COQ PROOF ASSISTANT

Ken Hosogoe

Bachelor of Engineering
Software Engineering

Department of Engineering
Macquarie University

5 September, 2016

Supervisor: Anthony Sloane

## ACKNOWLEDGMENTS

I would like to acknowledge my supervisor Associate Professor Anthony Sloane for his help and guidance throughout this thesis project.

## STATEMENT OF CANDIDATE

I, Ken Hosogoe, declare that this report, submitted as part of the requirement for the award of Bachelor of Engineering in the Department of Engineering, Macquarie University, is entirely my own work unless otherwise referenced or acknowledged. This document has not been submitted for qualification or assessment an any academic institution.

Student's Name: Ken Hosogoe

Student's Signature: Ken Hosogoe (electronic)

Date: 5/9/16

## ABSTRACT

Interactive theorem proves are important tools which aid in the creation of formal proofs, specification, and software verification. The current state of proof assistant IDEs are dated, and missing features that could be offered by modern IDEs. This project will design and implement an Atom package to provide support for the Coq proof assistant. The project evaluated IDEs and proof assistants for various features to implement. Identified features were syntax highlighting, auto completion, IDE to Coq interaction (PIDE), and linting. These features were designed and implemented. This project was able to create a package that implement basic support for the Coq proof Assistant for the Atom source code editor.

# Contents

**Acknowledgments** **iii**

**Abstract** **vii**

**Table of Contents** **ix**

**List of Figures** **xiii**

**List of Tables** **xv**

**1 Introduction** **1**

1.1 Motivation . . . . . . . . . . 1
1.2 Scope . . . . . . . . . . 2

**2 Background and Related Work** **3**

2.1 Coq . . . . . . . . . . 3
2.1.1 Examples . . . . . . . . . . 4
2.2 IDE . . . . . . . . . . 5
2.3 Atom . . . . . . . . . . 5
2.4 Coq IDEs . . . . . . . . . . 5
2.4.1 Proof General . . . . . . . . . . 5
2.4.2 CoqIDE . . . . . . . . . . 5
2.4.3 Coq/jEdit . . . . . . . . . . 7
2.4.4 Coqoon . . . . . . . . . . 7
2.5 Features of IDEs . . . . . . . . . . 7
2.5.1 Text Editor . . . . . . . . . . 7
2.5.2 Syntax Highlighting . . . . . . . . . . 7
2.5.3 Code Completion . . . . . . . . . . 8
2.5.4 Compiler Interaction . . . . . . . . . . 8
2.6 Waterfall (REPL) . . . . . . . . . . 9
2.7 PIDE . . . . . . . . . . 9
2.7.1 Protocol . . . . . . . . . . 10

**3 Design** **17**
3.1 Syntax Highlighting . . . 17
3.2 Prettification of Coq code . . . 17
3.3 Auto complete . . . 18
3.4 Basic Atom/Coq Interaction . . . 18
3.4.1 Coqtop Communication . . . 18
3.4.2 Proof Model . . . 18
3.4.3 Highlighting . . . 18
3.4.4 UI Windows . . . 19
3.5 Advanced Atom/Coq Interaction . . . 19
3.5.1 PIDE Commands . . . 19
3.5.2 Document Model . . . 20
3.5.3 Parsing . . . 21
3.5.4 Change Parsing . . . 21
3.5.5 Socket Communication . . . 21
3.5.6 Communication Messaging . . . 21
3.5.7 Execution List . . . 22
3.5.8 Coq Output . . . 22
3.6 Linter . . . 22

**4 Implementation** **23**
4.1 Syntax Highlighting . . . 23
4.2 Prettification of Coq code . . . 24
4.3 Auto complete . . . 25
4.4 Basic Atom/Coq Interaction . . . 26
4.4.1 Coqtop Communication . . . 26
4.4.2 Proof Model . . . 26
4.4.3 Highlighting . . . 27
4.4.4 UI Window . . . 27
4.5 Advanced Atom/Coq Interaction . . . 27
4.5.1 PIDE Commands . . . 27
4.5.2 Document Model . . . 28
4.5.3 Parsing . . . 28
4.5.4 Change Parsing . . . 29
4.5.5 Socket Communication . . . 31
4.5.6 Communication Messages . . . 31
4.5.7 Execution List . . . 31
4.5.8 Coq Output . . . 32
4.6 Linter . . . 32

**5 Discussion** **35**
5.1 Sample Proof . . . 35
5.2 Comparison . . . 37

**6 Conclusion** **41**
6.1 Future Work . . . . . . . . . . 41

**7 Abbreviations** **43**

**A Code** **45**
A.1 coq.cson . . . . . . . . . . 45
A.2 provider.js . . . . . . . . . . 50
A.3 syntax.json . . . . . . . . . . 51
A.4 PIDECommand.js . . . . . . . . . . 57
A.5 updateNodeHelper.js . . . . . . . . . . 59
A.6 yxml.pegjs . . . . . . . . . . 60
A.7 yxmlNodes.js . . . . . . . . . . 62
A.8 DocumentModel.js . . . . . . . . . . 65
A.9 errorMessages.js . . . . . . . . . . 71
A.10 goals-view.js . . . . . . . . . . 72
A.11 messages.js . . . . . . . . . . 73
A.12 pideTop.js . . . . . . . . . . 74
A.13 coqCommunicate.js . . . . . . . . . . 76
A.14 ken-coq.js . . . . . . . . . . 78
A.15 ken-coq.atom-text-editor.less . . . . . . . . . . 79

**B Consultation Meetings Attendance Form** **83**

**Bibliography** **83**

# List of Figures

2.1 Proof General . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
2.2 CoqIDE . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
2.3 Coq/jEdit . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
2.4 Coqoon . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8

4.1 Prettified Right Arrow . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25

5.1 Open Coq File . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
5.2 Opened Proof . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36
5.3 Syntax Highlighting . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36
5.4 Autocomplete . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37
5.5 PIDE . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37
5.6 Different command output . . . . . . . . . . . . . . . . . . . . . . . . . . 38
5.7 Linter . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38

B.1 Consultation Meetings Attendance Form . . . . . . . . . . . . . . . . . . . . 84

# List of Tables

2.1 YXML transfer syntax . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15

# Chapter 1

# Introduction

As computer systems are becoming more ubiquitous they have a large impact on society. Many computer systems are used in safety-critical applications like fly-by-wire aircraft, antilock braking, and radiation therapy machines [1]. The correctness of computer systems are of importance as bugs can lead loss of money, time, and life.

Examples are the Intel floating-point division bug which was estimated to be a $475 million loss [2], and the Ariane 5 rocket bug causing it to explode [3]. Other bugs which many people experience regularly are system glitches and crashes. With the growing dependence on computers there is a need for more robust, and reliable computer systems.

## 1.1 Motivation

These examples illustrate the impact software bugs can cause on our lives. With our lives becoming ever more dependant on software we require more reliable software. Interactive proof assistants help in the creation of correct computer systems through the use of formal proofs, formal verification, and formal specifications.

The Coq proof assistant has two main IDE Proof General, and CoqIDE. Both of these IDEs have a dated UI, and rely on a traditional sequential form of proving. There have been efforts in creating IDEs (Coqoon, and Jedit) that use a newer asynchronous approach as used by Isabelle but has not had a large uptake.

This project will implement Coq support on the Atom platform. Atom is a relatively new text editor that is modern, and has improvements in technology. Atom has advantages over other editors as it is free software, and written in JavaScript on the electron framework. Targeting Atom one of the popular web technology based editors this project will increase interest in proof assistants, and on the asynchronous interaction of Coq proofs. There has also been a rise in web technology based applications, as is within text editors. This project could also be adapted to work on other electron, or web based editors.

## 1.2 Scope

This project aims to design and implement support for the Coq proof assistant for the Atom editor platform.

This project will analyse typical IDE and proof assistants for features to be implemented on the modern Atom text editor.

The identified features will be designed, and implemented for the Atom text editor as a package.

The goal is to create a proof assistant which looks modern, is easy to use, and uses asynchronous interaction to increase their use with the expanding computer system environment.

# Chapter 2

# Background and Related Work

The project aimed to design and implement support for the development of proofs with the Coq proof assistant using the Atom editor. Existing editor and IDE support for Coq was surveyed and information gathered was used to develop requirements for the Atom package.

Proof assistants are important as they are used to develop mathematical proofs, and write formal specifications, programs, and to verify that programs are correct.

Current Coq IDE offerings are CoqIDE, Emacs with Proof General, Coq PIDE Jedit, and Coqoon. This project's aim was to improve upon these using the Atom text editor.

## 2.1 Coq

Coq is the formal proof assistant that provided the backend logic of proof checking for the package. Coq is a formal proof assistant that implements a program specification and mathematical higher-level language called Gallina [4].

A proof assistant is a software tool used to assist in the definition and proving of formal proofs through human-machine interaction. Proof assistants require user input to guide the software tool to a formal proof. Proof assistants are used to formalise mathematical theories, prove theorems, write formal specifications of system and software, and software verification.

A formal proof is a "finite sequence of sentences each of which is an axiom, an assumption, or follows from the preceding sentences. The last sentence of the sequence is a theorem" [5]. Formal proofs can be checked by computers easily, but finding proofs is difficult.

Coq is made up of three languages:

1. Gallina (Specification Language) For developing mathematical theories and to prove specifications of programs

2. Vernacular (Command Language) For controlling the Coq environment

3. Ltac (Tactic Language) For defining new tactics

### 2.1.1 Examples

The following is an example from *Software Foundations*. [6]. The days of the week can be defined using the Inductive command:

```
Inductive day : Type :=
  | monday : day
  | tuesday : day
  | wednesday : day
  | thursday : day
  | friday : day
  | saturday : day
  | sunday : day.
```

This example has created a type called day which defines the days of the week.

A definition for the next day of the week can be defined using Definition:

```
Definition next_weekday (d:day) : day :=
  match d with
  | monday => tuesday
  | tuesday => wednesday
  | wednesday => thursday
  | thursday => friday
  | friday => monday
  | saturday => monday
  | sunday => monday
  end.
```

This example is a function which will match the given day and return the next day.

Proving in Coq is performed through user-machine interaction using tactics. To prove the following:

```
Example test_next_weekday:
  (next_weekday (next_weekday saturday)) = tuesday.
```

Coq will give you this goal to prove:

```
1 subgoal
______________________________________(1/1)
next_weekday (next_weekday saturday) = tuesday
```

Which can be proved with:

```
Proof.
  simpl.
```

The simpl tactic performs simplification.

Giving:

```
1 subgoal
```

```
______________________________________(1/1)
tuesday = tuesday
```

The proof can be finished by:

```
  reflexivity.
Qed.
```

The reflexivity tactic checks that that both sides of the equal evaluate to the same thing.

## 2.2 IDE

Integrated Development Environments (IDE) and source code editors are software tools which provide useful features for the creation of programs. Using an IDE improves productivity as it centralises all the tools required into a single application, making it easier to use and navigate between different tools.

## 2.3 Atom

Atom is a free and open source cross platform source code text editor developed by GitHub in 2014. Atom is built upon the Electron framework which utilises chromium and Node.js to create desktop applications [7].

Atom is highly customisable and extendible through packages, which can modify the look and feel, and add features. It is a modern, approachable, and hackable source code text editor.

Atom packages are written in JavaScript or any language that can be transpiled into JavaScript. The two most popular languages packages are written in are CoffeeScript and ECMAScript 6 using Babel. Packages can be styled using CSS and LESS.

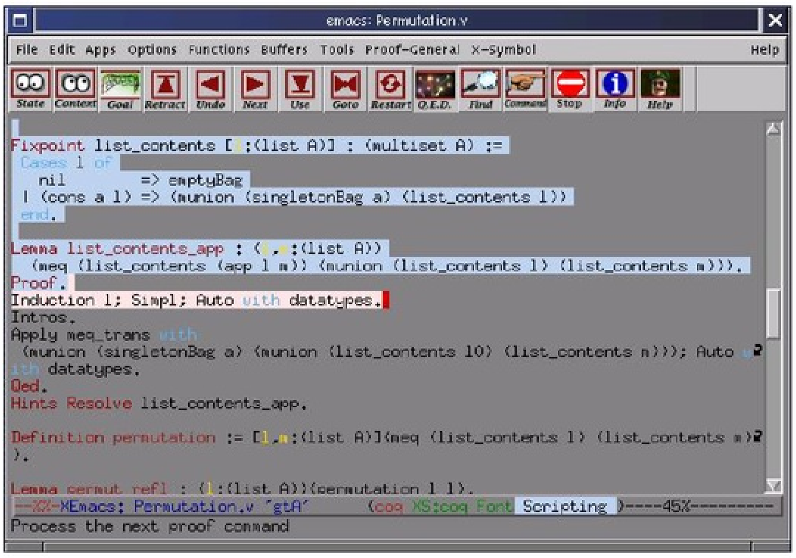
## 2.4 Coq IDEs

### 2.4.1 Proof General

Proof General is a generic interface for interactive proof assistants for Emacs, developed at the LCFS in the University of Edinburgh. Proof General uses the waterfall method of proof interaction, this is shown in Fig.2.1 where the blue region is the processed/locked region.

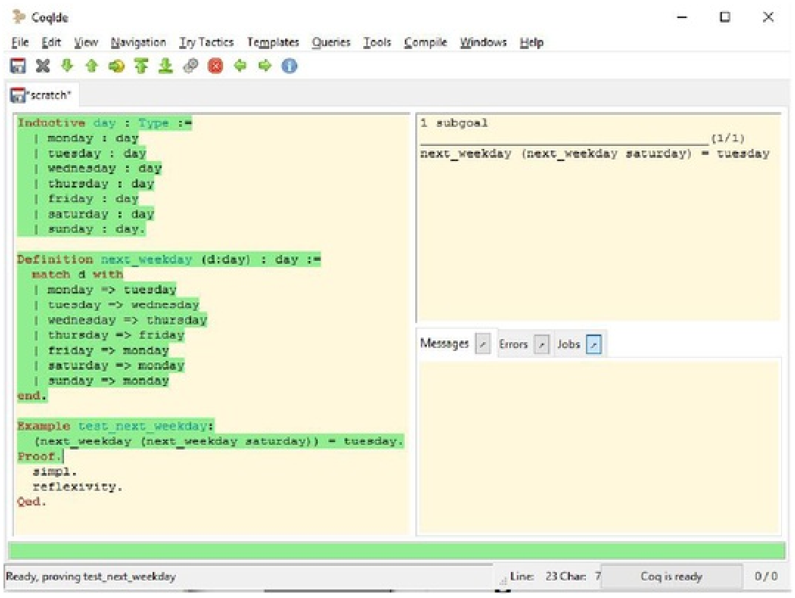
### 2.4.2 CoqIDE

The Coq Integrated Development Environment is a tool that comes with the coq package. It is based on GTK and written in Ocaml. Similar to Proof General it uses the waterfall method of proof interaction, shown in Fig.2.2 where the processed region is in green.


emacs: Permutation.v
File Edit Apps Options Functions Buffers Tools Proof-General X-Symbol Help
State Context Goal Retract Undo Next Use Goto Restart Q.E.D. Find Command Stop Info Help

```
Fixpoint list_contents [l:(list A)] : (multiset A) :=
 Cases l of
   nil         => emptyBag
 | (cons a l) => (munion (singletonBag a) (list_contents l))
 end.

Lemma list_contents_app : (l,m:(list A))
  (meq (list_contents (app l m)) (munion (list_contents l) (list_contents m))).
Proof.
Induction l; Simpl; Auto with datatypes.
Intros.
Apply meq_trans with
 (munion (singletonBag a) (munion (list_contents l0) (list_contents m))); Auto w
ith datatypes.
Qed.
Hints Resolve list_contents_app.

Definition permutation := [l,m:(list A)](meq (list_contents l) (list_contents m)
).

Lemma permut_refl : (l:(list A))(permutation l l).
--%%-XEmacs: Permutation.v  'gtA'       (coq XS:coq Font Scripting )----45%---------
Process the next proof command
```

Figure 2.1: Proof General


CoqIde
File Edit View Navigation Try Tactics Templates Queries Tools Compile Windows Help
*scratch*
Messages
Errors
Jobs

```
Inductive day : Type :=
  | monday : day
  | tuesday : day
  | wednesday : day
  | thursday : day
  | friday : day
  | saturday : day
  | sunday : day.

Definition next_weekday (d:day) : day :=
  match d with
  | monday => tuesday
  | tuesday => wednesday
  | wednesday => thursday
  | thursday => friday
  | friday => monday
  | saturday => monday
  | sunday => monday
end.

Example test_next_weekday:
  (next_weekday (next_weekday saturday)) = tuesday.
Proof.
  simpl.
  reflexivity.
Qed.
```

```
1 subgoal
______________________________________(1/1)
next_weekday (next_weekday saturday) = tuesday
```

```
Ready, proving test_next_weekday    Line:  23 Char:  7    Coq is ready    0 / 0
```

Figure 2.2: CoqIDE

### 2.4.3 Coq/jEdit

Coq/jEdit is an IDE that uses the jEdit text editor. It was made as a proof of concept of connecting Coq to the Isabelle Pide infrastructure. Unlike the previous two IDE's that used the waterfall method of interaction, this IDE uses PIDE which is an asynchronous form of interaction. In Fig.2.3 there are no highlighted locked regions because of the use of PIDE.

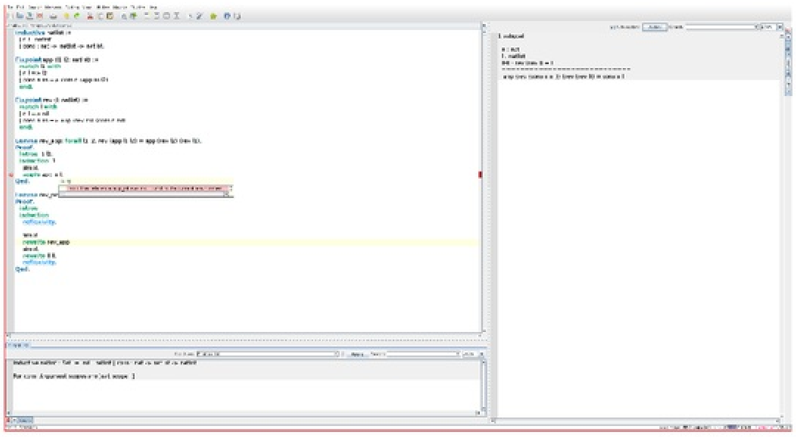


**Figure 2.3:** Coq/jEdit

### 2.4.4 Coqoon

Coqoon is a plugin built for the Eclipse IDE. It has an automatic project builder, java-style project organisation, and uses the PIDE protocol. Fig.2.4 shows a screenshot of Coqoon.

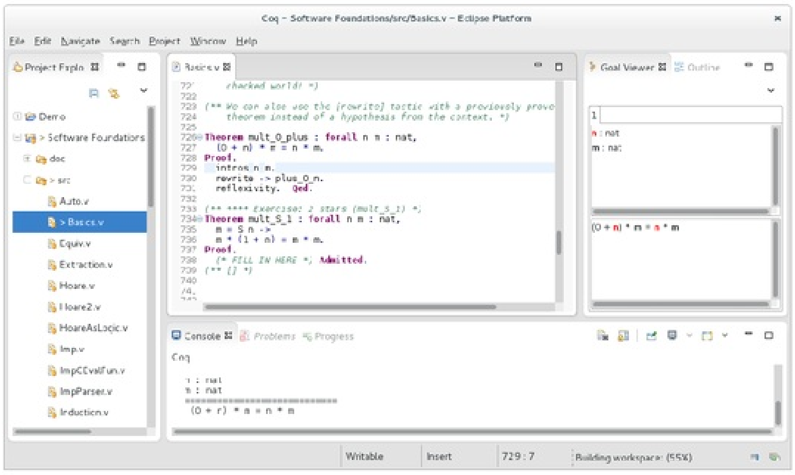
## 2.5 Features of IDEs

### 2.5.1 Text Editor

Text editors are used to display and edit text files and provide the features undo/redo, find/replace, and cut/copy/paste.

### 2.5.2 Syntax Highlighting

Syntax highlighting is a feature which colours lexical tokens in different colours according to the category of the token. Highlighting improves readability by making structures visually distinct, allowing users to quickly differentiate between different structures such



```
    checked world! *)

(** We can also use the [rewrite] tactic with a previously prove
    theorem instead of a hypothesis from the context. *)

Theorem mult_0_plus : forall n m : nat,
  (0 + n) * m = n * m.
Proof.
  intros n m.
  rewrite -> plus_O_n.
  reflexivity.  Qed.

(** **** Exercise: 2 stars (mult_S_1) *)
Theorem mult_S_1 : forall n m : nat,
  m = S n ->
  m * (1 + n) = m * m.
Proof.
  (* FILL IN HERE *) Admitted.
(** [] *)
```

```
1
n : nat
m : nat

(0 + n) * m = n * m
```

```
Coq

 n : nat
 m : nat
 ============================
  (0 + n) * m = n * m
```

Figure 2.4: Coqoon

as different keywords. Syntax highlighting is for improving readability of code and does not change the semantics of the code [8].

Adam Abonyis paper “IntelliSense Integration for Coq theorem prover” [9] states the basic method for implementing syntax highlighting is to have a set of defined keywords and rules for recognising different syntactic structures. The paper also states another method to create a syntax highlighter using data from a lexer and/or parser and separate tokens into categories. This method has an advantage as it provides more information about the code.

### 2.5.3 Code Completion

Code completion displays code suggestions based on keystroke input to the editor [10]. The user can continue writing or select a command from a list of commands that is then inserted into the editor. Code completion makes coding easier and improves developer productivity.

### 2.5.4 Compiler Interaction

IDEs typically offer tools to the user to compile their code within the IDE environment.

Coq has two modes of operation a compiled mode where commands are processed from a file and an interactive mode where user commands are interpreted [11].

- The compiled mode takes a file containing a whole development and compiles it to check correctness. The compiler creates an output file containing a compact representation of its input. This mode does not provide real-time feedback to users as they write proofs so it will not be used as the main process of generating proofs.

- The interactive mode is a read-eval-print loop (REPL). The user will develop their proofs step by step, and backtrack if needed. This mode does provide the real-time interactivity wanted but is limited by the nature of REPLs.

## 2.6 Waterfall (REPL)

The waterfall method builds upon the REPL method and is main method of interaction used with the Coq proof assistant. This style is used by Proof General, and Coq IDE. The REPL method is augmented through the IDE by using:

- Script buffer holds input, the commands to construct a proof
- Goals buffer displays the current sub-goals to be solved
- Response buffer displays other output from the proof assistant

The user writes proofs in the script buffer and steps through the script which sends the commands to the prover. Processed commands are placed in a locked region to prevent unwanted editing. To edit the locked region commands must be undone to unlock the region of code that needs modifying. The prover will then return an output; users will only see the output from the latest proof step.

## 2.7 PIDE

Prover IDE (PIDE) developed by Makarius Wenzel is as an approach to communicate between Isabelle and jEdit which are proof assistants and text editors respectively. PIDE is a communication protocol to send text edits to the prover and back [12]. This is used by Coqoon, and Jedit.

PIDE solves the issue of the waterfall approach by using an asynchronous document model, the prover receives edits continuously and updates its internal state [13]. This method of interaction is different from the previous as the user will write proof commands whilst the backend automatically process these commands asynchronously without the user needing to step through commands manually.

Coq-PIDE is a project by Makarius Wenzel that re-implements the core PIDE protocol for Isaabelle/ML to Coq [13]. In Makarius Wenzel's paper "PIDE as front-end technology for Coq" [13] he outlined how the PIDE protocol interacts with Coq. The paper also states that the PIDE protocol is written in OCaml for Coq and Scala for the editor side.

Carst Tankink's paper "PIDE for Asynchronous Interaction with Coq" [12] outlined the basic design of PIDE from the JVM side (Scala) with the ML side being symmetrical:

- Functions for capturing a document. A document represents all versions of a proof and can be updated by these functions. Additional functions are hooks which allow the tool to receive changes in the document made by the prover

- The functions, data, and results are represented as XML that can be serialized and de-serialized
- The XML is translated in YXML (custom transfer syntax for XML)
- YXML packets are sent on input/output channels (FIFO buffers, TCP/IP)

### 2.7.1 Protocol

This section will outline the messages used in the PIDE protocol.

**Editor Messages**

**Define Command**

This message provides the prover with the contents of a command span, and maps these to an identifier. All future messages that refer to a defined command uses its identifier ID instead of the command span text.

The prover stores a look up table of defined commands in the document model, which allows it to search for commands when an identifier is given. The Defined command message is triggered when there is an edit to proof document (file load, keyboard entry, copy-paste). The editor identifies changed regions in the proof document and sends these to the prover.

The following is an XML representation of what the editor sends to the prover.

```
<prover_command name="Document.define_command">
  <prover_arg>-19</prover_arg>
  <prover_arg>0</prover_arg>
  <prover_arg>
    Example test_next_weekday:
      (next_weekday (next_weekday saturday)) = tuesday.
  </prover_arg>
</prover_command>
```

The editor sends a *prover_command* with name attribute 'Document.define_command' to indicate it is a Define Command. The message contains three arguments *prover_arg*. The first argument is the *command identifier*. The second argument identifies the *type of command* the message contains. 0 indicates it is an executable command, and 1 means it is not such as white space. The third argument contains all the *proof text* from the editors buffer from a single command span, including all white space and newlines.

**Update**

The update message is the core of the protocol, it updates the prover's document model of changes by supplying the insertion and deletions history as a list of command identifiers.

The prover uses the message to update its representation of the proof document by executing the insertions and deletions of the edit. Then each command is mapped in a new document to a future computation of a proof state, the first computations of the commands which are unchanged between document states are kept. Shared command computations after the first difference in commands are not kept as they could have been changed by the differences. Next, an *assign update* message is sent to the editor. Finally, once all the commands have been executed the proof states are computed.

The following is an XML representation of what the editor sends to the prover.

```
<prover_command name="Document.update">
  <prover_arg>-37</prover_arg>
  <prover_arg>-47</prover_arg>
  <prover_arg>
    <:>
      <:>C:\Users\Ken\Coq\day.v</:>
      <:><2 0="0" 1="-19" 2="-20" 3="-21" 4="-22" 5="-23" 6="-24"
          7="-44" 8="-45" 9="-46" 10="-28"/></:>
    </:>
    <:>
      <:>C:\Users\Ken\Coq\day.v</:>
      <:><0>
          <:>
            <:><:>-24</:></:>
            <:/>
          </:>
          <:>
            <:><:>-24</:></:>
            <:><:>-44</:></:>
          </:>
          <:>
            <:><:>-44</:></:>
            <:><:>-45</:></:>
          </:>
          <:>
            <:><:>-45</:></:>
            <:><:>-46</:></:>
          </:>
      </0></:>
    </:>
  </prover_arg>
</prover_command>
```

The editor sends a *prover_command* with name attribute *Document.update* to indicate it is a Define Command. The message contains three arguments *prover_arg*. The first argument contains the identifier of the version of the document to update from. It is 0 when there is no previous version. The second argument is the identifier of the version to update to. The example is updating from document version -37 to -47. The third argument is the bulk of the data required for updating document versions. The ¡:¿ element is used as a separator.It has the perspective data, and second set has the update data.

The perspective data contains the directory and file name of the current proof document shown to the user, and the command identifiers which can seen on the screen. The visible identifiers are shown by an element with the name '2' and has a list of incrementing attributes starting from '0' indicating what commands are visible. The example indicates the command identifiers -19, ...-24, -44, ...-46, -28 are visible.

The update section contains the directory and file name of the proof document the operation is on, and a list of operations to apply to the document model. PIDE supports several types of operations but only one is implemented for Coq that is *Edit* operation (type '0'). The edit operation takes list of pairs, there are three types of pairs. The first indicates the beginning of the document, (None, -1) means -1 is the first command identifier of the proof. The second is an insertion which indicates a command that follows another command, (-1, -2) means the -2 command follows the -1 command. The last type is the deletion to show when a command is removed, (-1, None) means remove the command which followed the -1 command.

The example has four edit operations, one deletion and three insertions. It first removed the command following identifier -24. Then inserted command -44 after command -24. Next it inserted command -45 after command -44. Finally it inserted command -46 after command -45.

**Prover Messages**

**Update Assign**

This message provides a mapping for the editor between a command identifier and execution identifiers.

The editor uses the mapping to assign messages from the prover which contain execution identifiers to their corresponding command.

The following is an XML representation of an *assign_update* message from the prover.

```
<protocol function=assign_update>
  <:>-22</:>
  <:>
    <:>
      <:>-17</:>
      <:></:>
    </:>

    <:>
      <:>-19</:>
      <:>
        <:>26</:>
        <:>33</:>
      </:>
    </:>

    <:>
      <:>-21</:>
      <:>
```

```
        <:>28</:>
      </:>
    </:>

  </:>
</protocol>
```

The prover sends a *protocol* element with 'function=assign_update' as an attribute.

The message has two parts. The first has the identifier of the corresponding version document. The second part contains a list of command identifiers with a list of execution identifiers. Each command identifiers can have zero or more execution identifiers assigned.

The example is an *assign_update* for the document version -22. It has three command-execution mappings. The first -17 command identifier has no execution identifiers mapped. The following -19 has two execution identifiers assigned 26, and 33. The final -21 has a single execution identifier assigned.

**Writeln**

This message contains the output produced by Coq when the user in a proving section of a document requiring goal, or when a command is defined. Users of Coq will be familiar with these messages, as they are the standard messages used to relay proof progress. The editor stores these messages and displays them to the user as requested by the user. This primary form of communication for the proof progress.

The following is an XML representation of a *writeln* message.

```
<writeln serial="9" id="21" source="goal">
1 subgoal

  ============================
   tuesday = tuesday
 </writeln>
```

The prover sends a *writeln* element with three attributes. The first *serial* is an identifier for the message. Second *id* is the execution identifier the message is associated to. Lastly *source* describes what type of output the message contains.

The example has a serial of 9 and is about the command associated with execution identifier 21. It is of goal type and contains the output text from Coq of a goal for a proof.

**Report**

This message contains markup data about command spans. The data provides the name, kind, and location of the code identifier.

The markup data can be used by the editor to produce hyperlinks, which allow users to click on the name of an identifier and be taken to where it was defined.

The following is an XML representation of a *report* message.

```
<report offset="31" end_offset="43" id="6">
```

```
  <entity id="6" offset="31" end_offset="43" def_id="4"
     def_offset="12" def_end_offset="24" name="next_weekday" kind
     ="def"/>
</report>
```

The prover sends a *report* element with three attributes. The first two are text offsets which indicate the location the report is referring to in a command span. The third *id* is the execution identifier for which this report is about. The inner section contains an *entity* element with attributes containing the report data. *offset* and *off_set* is the text location the report refers to.

Attributes prefixed with 'def' refer to the command which defined the construct the reported command uses. *id* is the execution identifier of the defining command, and offset is the same as before but for the defining command text. Finally the *name* and *kind* of the defining command is included.

**Error**

This message contains the errors produced by the prover. Error messages that are associated with command spans will have their associated execution id and an offset which indicates which part of the command is wrong if applicable. Error messages are stored, and displayed to the user so they can take action in correcting these errors.

The following is an XML representation of an *error* message.

```
<error serial="25" offset="1" end_offset="17" id="105">
  Error: Illegal tactic application.
</error>
```

The prover sends an *error* element has attributes similar to messages previously seen. The example refers to execution id 105, and characters 1-17 of the corresponding command span. Which refers to 'reflexivity Qed.' because it is missing a '.' after 'reflexivity'. The inner section contains the Coq ouput text for the error.

**Message Representation**

PIDE uses Extensible Markup Language (XML) as its underlying data structure for commands. XML is a standard for markup language that defines rules for encoding documents for human and machine reading. Under all the extensive XML standard documents there is a simple tree structure of text and markup elements.

The essence of XML can be expressed in 3 lines.

```
Tree datatype =
  Elem(markup: String, List[(String, String)], body: List[Tree])
  | Text(content: String)
```

The downside of XML was its text representation was not orthogonal to UTF-8 which the next sections YXML will solve.

**YXML**

PIDE uses YXML as its text representation of data between the editor and prover. It has the advantage of being orthogonal to UTF-8, meaning text does not have to be modified between reading and writing.

YXML achieves this by leaving behind angle brackets '<>' to indicate tag starts, and endings and using ASCII characters which are never used by the XML standard. YXML uses chr 5 (X), and chr 6 (Y) as control characters. [14]

**Table 2.1:** YXML transfer syntax

| | XML | YXML |
|---|---|---|
| Open Tag | <*name attribute*=*value* ...> | **XY***name***Y***attribute*=*value*...**X** |
| Close Tag | </*name*> | **XYX** |

As the YXML syntax no longer relies on angle brackets YXML can be directly in-lined into string representation of messages. YXML is free to contain any markup as YXML does not use ><\&"' so escaping these characters is not required thus there is no altering of the message semantics unlike XML where escaping is required to properly represent the information.

**Low Level Representation**

This section will detail how PIDE sends and receives messages over its socket connection. The PIDE system does not use a straight YXML syntax representation for its transmission of data between the editor and prover, additionally the sending and receiving has differences in formatting.

Examples used in this section are from Wireshark which was used to listen on the loopback (127.0.01) socket connection between the Jedit and Coq-PIDE. YXML representations of X and Y are shown as '.' as Wireshark has no textual representation of chr 5 and 6.

**Editor Sending / Prover Receiving**

Pide does not use the YXML syntax for the complete representation of data for prover receiving. It uses a count of characters separated by commas to indicate boundaries of data. YXML can be used in the boundaries to represent data. Byte counts and data are separated by the Line feed chr 10.

```
32 33 2c 33 2c 31 2c 36 31 0a 44 6f 63 75 6d 65   23,3,1,61.Docume
6e 74 2e 64 65 66 69 6e 65 5f 63 6f 6d 6d 61 6e   nt.define_comman
64 2d 34 31 30 54 68 65 6f 72 65 6d 20 62 61 73   d-410Theorem bas
69 63 5f 63 6f 6e 6a 20 3a 20 66 6f 72 61 6c 6c   ic_conj : forall
20 28 41 20 42 20 3a 20 50 72 6f 70 29 2c 0a 20    (A B : Prop),.
20 41 20 2d 3e 20 42 20 2d 3e 20 41 20 2f 5c 20    A -> B -> A /\
42 2e                                             B.
```

This example shows a define command. It starts with a series of numbers '23,3,1,61' which correspond to the character lengths of each data boundary. The following text in the message corresponds to a define command being command, command identifier, command type, and command text.

**Editor Receive / Prover Sending**

Unlike prover receiving PIDE uses YXML for sending messages to the editor, but a slightly modified version for the command. Instead of sending the header and body in a single message it sends them as two separate messages. Messages start with a byte count of header YXML length then followed by YXML message and repeated for the body. Byte counts and data are separated by the Line feed chr 10.

```
34 31 0a 05 06 77 72 69 74 65 6c 6e 06 73 65 72   41...writeln.ser
69 61 6c 3d 31 30 06 69 64 3d 31 39 06 73 6f 75   ial=10.id=19.sou
72 63 65 3d 67 6f 61 6c 05 05 06 05 31 37 0a 4e   rce=goal....17.N
6f 20 6d 6f 72 65 20 73 75 62 67 6f 61 6c 73 2e   o more subgoals.
```

This example shows the two separate messages of the header and body. It starts with 41 indicating the byte length and is followed by the YXML message, this is repeated again for the body with a length of 17.

# Chapter 3

# Design

A package for the Atom source code editor was created to implement Coq support. This project is a single large package that is composed of smaller subcomponents which each implemented a feature for Coq Support. The package was written in ECMAScript 6 using babel.

## 3.1 Syntax Highlighting

Syntax highlighting for Coq the language was built upon the existing framework Atom uses for syntax highlighting. Atom calls its syntax's as grammars, which are defined in a method similar to Textmate (source code editor on the OSX platform). The syntax highlighting provided a similar level of highlighting as the CoqDoc tool used to produce documentation of Coq code.

The Coq language is large and was difficult to fully implement so a smaller subsection of Coq was implemented for syntax highlighting.

Atom supports many different themes to colour the editor differently. To enhance compatibility with various Atom themes, specific colours for keywords were not be used. Keywords of the Coq language were placed into categories which dictated the colour the Atom theme applied to the text. Categories for each keyword was picked based upon the syntax, and the CoqDoc colouring of keywords.

## 3.2 Prettification of Coq code

Prettification of Coq code substituted Coq text tokens for their unicode equivalent. This made reading code simpler and faster. Atom does not have an built-in support for token substitution. Solutions that were explored are modifying the edit buffer, and using CSS.

## 3.3 Auto complete

Auto completion was delivered by creating a provider for the Autocomplete+ Atom package. Similar to syntax highlighting a smaller subset of the Coq language was targeted for implementation.

To determine which keywords to suggest, the provider needs the following data:

- Scope information This information indicates which code block the cursor is currently in. This determines which keywords are valid to display.
- Prefix This is the current word being typed. The list of keywords will be determined through the use of this keyword and a match algorithm.

## 3.4 Basic Atom/Coq Interaction

This feature coordinated messaging between Atom and Coq for proof creation using the waterfall method of interaction.

### 3.4.1 Coqtop Communication

This feature initialised Atom for communication with the coqtop process. It set-up the hooks for displaying coq output to the UI windows and provided a method for sending text to coq.

### 3.4.2 Proof Model

The proof model keeps a state history of all the commands which have been sent to the prover. This history is used to undo command as the history will give an identifier to tell the prover to go back to.

The proof model has commands that are invoked by hotkey to progress through the proof text.

It has the commands:

- step forward
  will send a command span to coq
- step backward
  undoes a command in coq

### 3.4.3 Highlighting

The highlighter was responsible for highlighting regions of code which have been sent to Coq. Highlighting was determined by a list of ranges. A range is the start and end coordinates of text in the buffer.

This has the features:

- add highlighting adds a range of text to the list
- remove highlighting removes a range of text from the list

### 3.4.4 UI Windows

There was a UI window for goal and coq messages. The UI window opened when a proof text file was opened. It would split the panel placing the proof text at the top and message window underneath.

## 3.5 Advanced Atom/Coq Interaction

The advanced interaction between Atom and Coq module used the PIDE protocol. This module replaced the Basic interaction module. The PIDE protocol is a different method of interaction than in the basic module, thus it required a separate module to be created.

Users will interact with this module in a different way than the basic module. Users will interact with the editor and have their actions automatically be sent to the prover, and receive output asynchronously. Unlike the basic method where the user has to manually send commands to the prover to receive output.

It was initially planned to use the existing PIDE Pure.jar a Scala implementation of the editor side of the PIDE architecture. Unfortunately using a JVM implementation with Javascript was not feasible. Two methods were explored for utilising Pure, ScalaJS and the 'java' node package.

The first method was to use ScalaJS a compiler that compiles Scala source code into JavaScript code, to convert the Pure Scala implementation into a JavaScript version. This ran into compile errors as the underlying Scala code was not implemented in ScalaJS.

The second method was to use the 'java' a node package which allows the use of Java within Javascript code. This turned out be quite cumbersome as some data structures of Scala had no orthogonal versions to JavaScript, meaning many data stores had to converted manually which added complexity. Another complexity was that the PIDE implementation had limited documentation so navigating between Scala, Java, JavaScript, and Pure it self was not feasible.

Ultimately the PIDE protocol had to be reimplemented for the Atom package in JavaScript. The papers gave an overview of how the PIDE protocol worked, but were outdated in the formation of some commands and did not illustrate commands were constructed for transmission.

### 3.5.1 PIDE Commands

The editor stored the data for PIDE in a data structure that could easily be created and converted to its transport syntax. The commands which are used:

- Define Command

- Discontinue Execution
- Update

The first two commands have simple structure compared to update which is complex. It required a helper function with inputs: file, perspective, and document. File is the filepath and name of the current proof document. Perspective is what the commands the users is current looking at. Document is the links representing the relationship between commands.

### 3.5.2 Document Model

The document model is a representation of the proof document on the editor side. It is a collection of data structures that contain proof text and meta-data, proof state, and identifier count. The model also contains information for the prover on how to modify its own data model to reflect the editor.

The proof state and identifier count are negative integers, that need to easily convertible to their string representations. A simple data type that supports negative integers and type conversion is what is required which JavaScript can natively supply.

The data structure that contains the proof text requires additional data about the proof text. This data structure stored the raw proof text (including all white space, and comments), location of text, order of commands, and their command identifiers. This data was constantly searched, accessed, inserted and deleted from.

**Actions on Document Model**

**Insertion**

An insertion occurs when new proof text in inserted into the text buffer. For the document model and text buffer to reach a synchronous state the model must be updated to reflect the text insertions.

On an insertion the proof text will be augmented with a new command identifier, and location information (location within the buffer) will be added to the data model.

**Deletion**

A deletion occurs when a proof text is removed from the text buffer. For the document model and text buffer to reach a synchronous state the model must be updated to reflect the text deletions.

On a deletion the deleted command will be removed from the model. Its preceding command link is updated to point at the command following the deleted link. The deleted command will also have its command identifier marked for garbage collection.

**Search & Access**

Searching occurs when the editor needs the to find which command required based on location information so insertions, and deletions are performed on the correct data. Data is also accessible by their command identifiers so prover messages can get the meta-data associated with the command.

### 3.5.3 Parsing

The proof document was parsed to convert lines of text into individual proof commands. Proof commands were inserted into the document model. Parsing would occur when new proof text needed to be added to the document model e.g. document first opened, user modifies text.

The parser grouped proof text between two types as required by the PIDE protocol. It required the raw proof text to be separated between executable and non-executable code. Executable type text are spans of text which end with a '.' or text which is not classified as white space (partially written commands). Non-executable code is the white space between commands (spaces, newlines) and comments.

### 3.5.4 Change Parsing

To have changes to the text buffer affect the document model change parsing occurred. This was a difficult task as not only does the changed text have to parsed, but also a whole region of command spans may have to be parsed. Meaning any changed text and the surrounding command text must be re-parsed to ensure the document model reflects the text buffer. This is because a change may affect more than a single command span.

In addition, the editor stores the new command text along with their update links to send to the prover to update its own internal document model. Update links for the update model are formed by the (to, from) syntax.

### 3.5.5 Socket Communication

The editor and PIDE's coqtop communicated via a socket connection. The editor will initiated the creation of the socket server and started the PIDE coqtop process with the server address. The socket would have an input method for Coq commands to be sent to the prover, and an output for Coq message responses.

### 3.5.6 Communication Messaging

The PIDE communication syntax was used. The editor would translate proof text and update links data structures into the communication syntax. When the editor receives Coq messages it translates the YXML into a data structure which can then be processed by their relevant modules.

Proccessed Coq Messages:

- Assign Update
- Writeln
- Error

### 3.5.7 Execution List

This module processed Assign update commands. A mapping of command, and execution identifiers are created using the information from the assign update commands. The mapping is used by the message and error list modules to associate Coq output messages with their associated Coq command.

### 3.5.8 Coq Output

This module processed the Writeln commands. It stored all the Coq output contained within the command with its execution identifier. The module also facilitated the retrieval of these messages when a users cursor was on a command.

## 3.6 Linter

The linter gave the user feedback to the user on any Coq errors which may have occurred such as invalid syntax and invalid commands. Feedback was given in underlining the offending text much like how word processors underline spelling mistakes.

Error information was be supplied by processing the Error coq messages produced by the prover. Offending command are identified by using an execution identifier to find the relevant command identifier in the execution list. Offset information by the prover was be used to mark text to be underlined.

# Chapter 4

# Implementation

Boilerplate code for Atom packages was generated by using the Package Generator package included with Atom. The skeleton package it created was set up for ECMAscript 6 and provided a starting point for the project. Documentation for creating an Atom package and its API are located on its site [15]. Documentation was sparse at times, so another useful reference was reading other Atom packages [16].

## 4.1 Syntax Highlighting

The Coq grammar was specified in the Textmate style. The grammar was written in .cson (coffeescript object notation) file. The top level contained four key/value pairs:

1. scopeName
   The unique name for the grammar, 'source.coq' was used for this grammar.
2. fileTypes
   This indicates what file types the grammar will be applied to. To target Coq files the value 'v' was used.
3. patterns
   A list of textmate rules defining the grammar.
4. repository
   A dictionary of named rules.

Coq syntax is highlighted by creating a rule which matches the text which needs to be highlighted. The colour of the highlighted text can be assigned by naming the rule of of the pre defined scope names. Scope names are generic constructs that Atom themes use to target many different languages without requiring specific colours to be chosen. An example of a keyword:

```
'let':
    'name': 'keyword.other.coq'
    'match': '\\b(let|in)\\b'
```

The default Atom theme coloured keywords pink, so the text 'let and 'in' will be coloured pink.

There are three types of textmate rules:

1. include
2. match
3. begin/end

An include is a rule that specifies a named rule from the repository to use. This was used in the patterns list as it allowed a simpler reading/navigation of the rules of the Coq grammar during implementation, and implicit documentation of rules had names in the repository.

A match is a rule that matches regular expression to text and assigns it to the rules name. This rule type is the main rule type used to define the grammar. An example of an ident rule:

```
'ident':
    'name': 'enity.name.function.coq'
    'match': '([a-zA-Z_][a-zA-Z0-9_]*)'
```

An ident is an identifier which can start with a letter or underscore followed by any letter, number, or underscores.

A begin/end is a rule that matches two regular expressions. It is used to match text which may span multiple lines as the match rule can only match on a single line. As its name implies the rule has a regex indicating the start and a regex for the end. This rule is used for comments, and quote marks for strings. An example of a comment rule:

```
'comment':
      'name': 'comment.block.coq'
      'begin': '(\\(\\*)'
      'end': '(\\*\\))'
      'patterns': [
        {
          'include': '#comment'
        }
      ]
```

This rule first matches '(*' which indicates the start of a comment block and ends with '*)' for the end of a comment block. This rule can also contain a comment between the begin and end regex, through the use of the include rule.

## 4.2 Prettification of Coq code

Coq code was prettified by extending the previous grammar. The grammar was extended by creating a new rule for each operator and scope name.

To style the new operator rules, Atom packages can define cascading style sheets (CSS). Each new scope name had a CSS rule which replaced the text with its Unicode equivalent.

Operator text was hidden using CSS visibility property. Unicode was displayed using CSS pseudo elements which added styling to specific parts of an element. A pseudo element was added after each rule a content property had the unicode text.

To ensure the positioning of elements was correct 'relative' was used as the position for the main rule, and 'absolute' for the pseudo element. The pseudo elements have a right offset to position the Unicode text in the middle of the text which is needed as the text operators are often longer than their Unicode versions.

An example of a prettified right arrow:

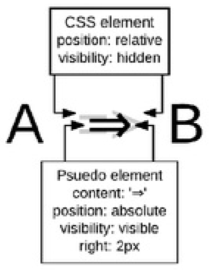


**Figure 4.1:** Prettified Right Arrow

## 4.3 Auto complete

Two algorithms for matching prefix's with keywords were designed left to right matching, and fuzzy string matching.

#### Left to Right Matching

This algorithm checked if a prefix exactly matches with a keywords from left to right. Keywords which matched were displayed as possible suggestions. Matching keywords were sorted according to the size of the word in ascending order.

**Example**
Two examples of prefixes with possible keyword completions.
Input: d
Output: delta, Debug, Defined ...

Input: del
Output: delta, Delimit

**Fuzzy String Matching**

This algorithm used approximate string matching rather than an exact matching which was used in the algorithm before. Approximate matching allows users to type in keywords partially without requiring it to exactly match a keyword from left to right. This is useful when the user does not know the exact keyword they require but an idea of what they require, allowing them to find the required keyword. The fuzzaldrin library for fuzzy filtering and string scoring was used.

The prefix and keywords were given to the fuzzaldrin library. It would give possible completions based upon its own string matching algorithm.

**Example**
Two examples of prefixes with possible keyword completions.
Input: ab
Output: About, Abort, Variable ... Generalizable
This example shows that keywords that contain 'ab' are shown.

Input: module
Output: Module, Module Type, Declare Module
This example shows if a user is interested in 'module' but not sure of the beginning of the command they will still get a useful list of completions.

## 4.4 Basic Atom/Coq Interaction

### 4.4.1 Coqtop Communication

The coqtop process is started using Atom's bufferedProcess. Coqtop was used in emacs mode as it provided greater information about the state of the proof than the normal mode. Emacs mode is toggled by giving coqtop '-emacs' as an argument at start up.

Input to coq is provided by a method that writes directly to standard in of the coqtop process. Standard out had its output connected to the UI window, and standard error prints directly to the console.

### 4.4.2 Proof Model

The state of the proof is kept by storing a history of state IDs in a list, and maintaining a range of proven commands.

Sending of proof text was triggered by pressing the hotkey 'ctrl-down' which called the stepForward method. This method uses Atom's 'scanInBufferRange' method to get the first text which ended with a '.' using regex '(.—
n—
r)*?.' on the section of text which was not proven. Matching text was added to the proven range, and text sent to Coq. The state is then incremented and pushed onto state history list.

Undoing proof text is triggered by the the hot key 'ctrl-up'. This method uses 'backwardsScanInBufferRange' on the proven range. It finds the last command text and removes it from the proven range. To undo the prover state the proof ID is popped from the state history list and sent to Coq using 'BackTo stateID' which puts the prover in the state of stateID.

### 4.4.3 Highlighting

The highlighting of text is performed using Atom's built in methods: 'markBufferRange' and decorate marker. A mark is a data structure which marks a location of text. This marker maintains its logical location marking as the buffer is changed. A decorate marker allows decorating a marker with css, allowing it to be coloured.

Highlighting of the proof document was achieved by storing a range variable which maintained the highlight state of the proof document. A decorate marker was used to colour highlighted markers green. Highlighted regions are added by passing in a region to be highlighted and making the end of its region the new highlighted end range, then update the decorate marker with the new range. Removal was is a similar process but using the passed in region's starting point as the new highlighted end range.

### 4.4.4 UI Window

The output window is an extended ScrollView from the 'atom-space-pen-views' package. It is a simple HTML page containing two div elements Goals, and container. Output is displayed on the window by calling 'view.container.html' which edits the html inside the container div.

## 4.5 Advanced Atom/Coq Interaction

### 4.5.1 PIDE Commands

Pide commands are represented by JavaScript objects. Each command has a kind tag indicating the type of command.

Define command has a command identifier, command type, command text for PIDE and range for use by the editor.

Discontinue execution only required a kind type as it held no data.

Update has a old and new state identifiers, and update YXML. Update YXML is where the bulk of the data is stored, and helper function was required to generate this complex structure.

This complex structure when broken down, has two main parts perspective, and document. The perspective contains a file node which is a text node with file path and name, and a list of commands the viewer can see. The document node although complex can be broken down into file node, and a list of lists structure. Firstly list of command links from the parsing process get converted into the list of list of list structure. Next, this list is placed in a zero node, then in a colon node. These last two nodes are YXML element nodes.

### 4.5.2 Document Model

The command counter and state identifiers were stored in a number data type. Proof text was stored in a list of define command objects. Using a list allowed all the commands to be ordered. Define command objects were used as internally they contained the exact data. A separate data structure would have just duplicated the same structure but without the ability to convert to the transmission syntax.

### 4.5.3 Parsing

Parsing command text from the text editor used Atom's text editor scan method. The scan method takes regex and a callback. The method parsed the text according to the provided regex and when a match was found the callback is called with match information.

The regex used to parse was white space was:

```
(\(\*[\w\W]*?\*\)|\s+)
```

This regex matches the start/ends of comments anything between, or one or more whitespace characters.

Regex for commands was:

```
([^.]*\.|[\s\S]+)
```

This regex matches command text that ends with a '.' or partial commands which is text that is not matched by the white space regex.

Both of these regex commands are ORed into a single global regex command so the scan method finds all matches in the proof document.

The callback created a define command object and pushed it onto a list. Define command objects were created using the matched text, display marker, type information by the callback data.

Type information is gained by checking which part of the regex matched, if the second part is undefined it is of type '0' and vice versa.

Once the whole document is parsed update links for each command must be generated. To create update links for an empty document model loop through the new commands. Push a tuple containing the previous command identifier and current identifier to a list. For the first command there is no previous command so it is assigned as Null.

Finally the command list can be assigned as the new document model as the model is empty. The command list, and update links list can be sent to the prover to update its document model with the editors.

### 4.5.4 Change Parsing

This type of parsing modifies an existing document model with changes performed by the user. Two different methods of change parsing where implemented. The first method A was a slower approach and the second method B is a faster method.

**Method A**

This Method re-parses the whole document when any changes occur and updating command identifiers with their old identifiers if they match and assigning new command with identifiers. This method is slower as it requires re-parsing of a whole document when any change occurs, especially when a few commands may only need to be re-parsed.

Atom provides an 'onDidStopChanging' method which calls a callback after the editor has stopped changing for 300ms. The callback is provided with location data of the change (old text range and new text range). It is used for both methods to start the change parsing process.

For method A Parsing occurs similarly as in the previous section but command identifiers are not assigned. After parsing spans into a list command identifiers are assigned for the old, and new identifiers. Assignment of identifiers takes place in two loops. The outer loop loops through parsed commands and inner loop loops through the document model.

There are five cases which occur between the parsed, and model commands whilst looping through the lists:

1. Equal Commands
   When both commands have the same text and location they are equal. Text and location must both match as just text could match with any white-space which could have the incorrect document links. Equal commands are assigned same identifier assigned as they have not been changed.

2. Inserted Command
   When the parsed command's location is beyond the current document command it is a new command. The parsed command is assigned a new identifier and added to the pending edits list. An update command is also added to pending updates to update links between the previous document command and parsed command.

3. Removed Command
   If the commands do not satisfy the first two commands it means a command has been removed. An update command which removes this command link is added to the pending updates list.

4. End of Document Model
   When we reach the end of the document model and still have parsed commands its means they are all new commands. New commands have an identifier and link assigned along with pending commands added to their lists.

5. End of Parsed Commands
   If we there are no more parsed commands but document commands it means they

were removed. Removed commands have their links updated and added to the pending updates list.

Pending Command & Update: These lists are the messages that will be sent to Pide-Coq to update its own internal state.

After, the assignment process the parsed commands can now be set as the new document model, and pending commands & updates can be sent to PIDE-Coq.

**Method B** This method, instead of parsing everything it only parses text which would be affected by the change. There are four main stages in the algorithm: identify changed region, parse changed region, generate update links for the new document, and update document model.

The changed regions of text was found by finding a start and end point of commands of text that were affected. The starting point was found by searching the document model for the first command containing the starting change point, if no command was found the start point is [0, 0] (start of text). The last point is similar but using the end change point, and if no point is found the end point is [Infinity, Infinity] (end of text).

The next step was to parse the text using these points. Using Atom's scanInBufferRange a specific range was parsed. The range was from the start point found in the previous stage and the end point was [Infinity, Infinity] (end of text). Each parsed command is assigned a new identifier and added to a pending edit list. Finally the command was checked if it contained the end point found from the previous step, if it did the parsing process was stopped. The end point was not specified at the start of parsing because the changed region could extend past this point, the following example illustrates this.

```
Proof.
  simpl.
  reflexivity.
Qed.
```

If the '.' after simpl is removed the new command would be:

```
simpl
  reflexivity.
```

The first stage would say the end point is just before 'reflexivity' as it is a white space command that was edited, but the change has also modified the next command. This is why it is required to parse the text until a command that contains the end point is found.

The third stage is the creation of update commands for PIDE. Removed commands have their links to them removed, by looping backwards through the the document model commands which are contained in the parsed command range and creating update links to Null. Next, an update link is created from the command preceding the changed region, to the first parsed command. Finally links are created for commands in the parsed commands.

In the final stage the document model is updated with the parsed commands, and parsed commands and update links are sent to PIDE.

### 4.5.5 Socket Communication

Atom and PIDE/Coq used 'net' a Node.js asynchronous network wrapper. A server is created using net's createServer method. A callback is registered on the 'connection' event which starts the document model and connects the output of the socket to the message processing module. The socket is also saved so data can be sent to Coq.

The server is started by the listen method which after starting created the Coq PIDE process. The PIDE process was initiated by Atom's BufferedProcess with arguments "-async-proofs", "on", "-toploop", "pidetop", "-main-channel", and socket's IP:port address to coqtop.exe.

The editor sends messages to the prover by creating a list of Coq messages which are written to the socket.

### 4.5.6 Communication Messages

YXML messages from PIDE are parsed using the parser generator PEG.js. The grammar used to parse YXML:

```
element
  = XY text attribute* element* XYX
  | text
attribute
  = Y text '=' text
attrText
  = [_a-zA-Z0-9]+
text
  = [- =_.:;()~<>/\\a-zA-Z0-9\n,'"\[\]]+
X
  = '\u0005'
Y
  = '\u0006'
```

Attributes required a separate definition attrText as text contained a '=' which caused it to consume the '=' between the name and value of an attribute. This grammar is used to parse the text into trees using the nodes element, attribute, and text.

Once messages are parsed they are placed into a list. The list gets looped through and each message will be processed based upon its '.name.text'. Message and Module:

- writeln processed by Coq output module
- protocol processed by Execution list module
- error processed by Linter module

### 4.5.7 Execution List

Execution data (commandID, executionID*) from PIDE is stored in a Map.

To update the execution map the module would loop through the body of an assign update command. Each command would then be put into execution map with its commandID used as the key, and the list of executionIDs as the value.

If a command had an empty executionID list it would remove the messages of the associated command identifier (Coq output), and remove the command identifier from the execution map.

### 4.5.8 Coq Output

This module used a map for its data structure. It has four functions insertion, removal, retrieval, and display.

**Insertion**

An insertion takes a 'writeln' command. A single execution ID may have multiple lines of output associated, but they are sent as separate output commands. To group multiple outputs, it is first checked if there is an existing message, if so it is prepended to the new output. Messages are stored with execution IDs as the key, and output as as the value.

**Removal**

Messages are removed using their command ID. A message is removed by getting the list of execution IDs assigned to a command ID from the execution map. Then the list is looped through removing associated IDs from the message map.

**Retrieval**

Messages were retrieved by a command ID. A list of execution IDs associated with the command ID is retrieved from the execution map. Next, an output string is built by appending the output from the message map with the execution ID list.

**Display**

Coq output is displayed to user by identifying what command they want information for, retrieving the message, and displaying it.

Using Atom's 'onDidCursorChangePosition' the editor can get the coordinates of where the users clicks. The coordinate information is used in a linear search to find the command the user is interested in. Next, the command ID is used to retrieve the output from the message map. The output is then sent to the Coq view module.

## 4.6 Linter

This module uses Steelbrain's Linter package for Atom. It provides the user interface of the linter; underlining text, error dialogue box, popup dialogues, and status bar. All that is required by the editor is to format messages for the linter.

Messages for the linter require an object with type, filePath, range, and text. All the data besides the range is simple to obtain from the error command. Range required converting the offset data into a text buffer range.

To convert a command start/end offset to a textbuffer range, loop through the command text keeping track of row and column positions. These are the steps as follows: Set a row and column variable to the starting point of the error command. Loop through the command text. Set the start point as the row/column if index = startOffset. Next, if the character is a new line increment row and set column to 0, else increment column. Finally the start point and row/column variables is the range which is sent to the linter.

The linter's message storer is emptied when the editor sends a command to PIDE to ensure only valid error messages are shown.

# Chapter 5

# Discussion

## 5.1 Sample Proof

A Coq proof can be started by creating or opening a file with the Coq '.v' extension. Atom will require the package created in this project, and its dependencies packages (linter). The file opening dialogue can be seen on fig 5.1.

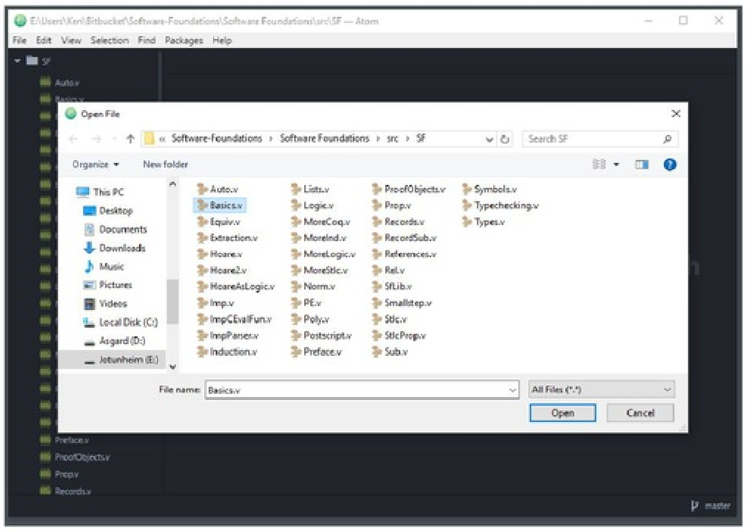
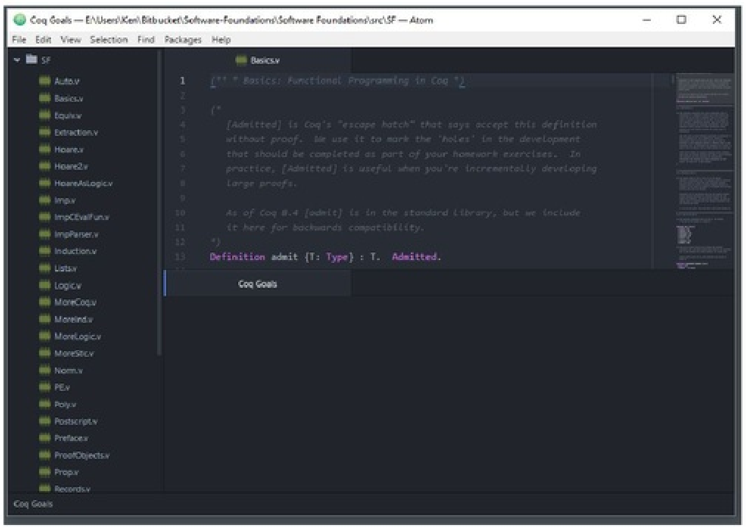


**Figure 5.1:** Open Coq File

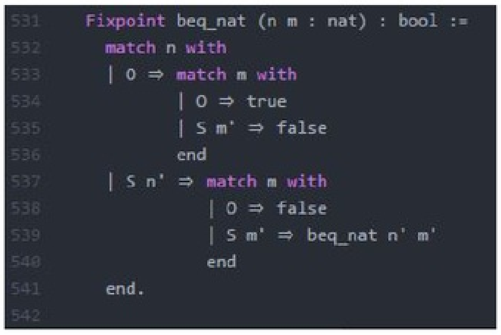
When a Coq file is opened two panes will be displayed. One Texteditor pane which contains the proof text at the top, and an output pane which contains Coq output underneath. As shown in fig 5.2



```
(** * Basics: Functional Programming in Coq *)

(*
   [Admitted] is Coq's "escape hatch" that says accept this definition
   without proof.  We use it to mark the 'holes' in the development
   that should be completed as part of your homework exercises.  In
   practice, [Admitted] is useful when you're incrementally developing
   large proofs.

   As of Coq 8.4 [admit] is in the standard library, but we include
   it here for backwards compatibility.
*)
Definition admit {T: Type} : T.  Admitted.
```

**Figure 5.2:** Opened Proof

The proof text colours Coq keywords differently, making code structures easier to read. Mathematical operators and symbols are replaced with their unicode equivalents increasing readability. As shown in fig5.3

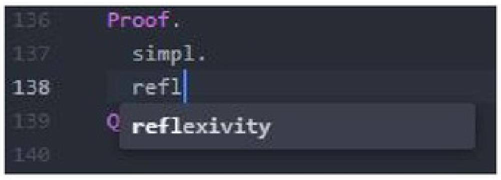


```
Fixpoint beq_nat (n m : nat) : bool :=
  match n with
  | O ⇒ match m with
         | O ⇒ true
         | S m' ⇒ false
         end
  | S n' ⇒ match m with
            | O ⇒ false
            | S m' ⇒ beq_nat n' m'
            end
  end.
```

**Figure 5.3:** Syntax Highlighting

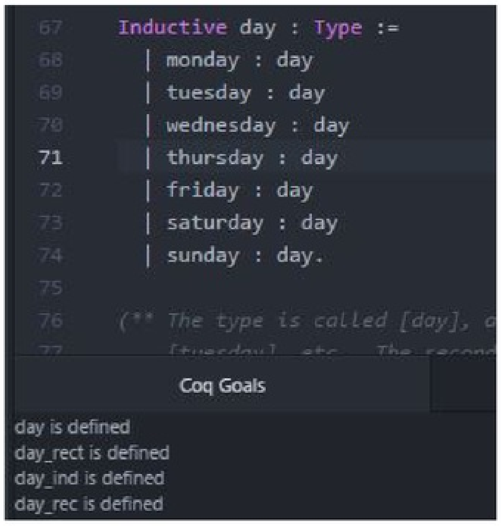
As text is written onto the text pane a pop up dialogue appears suggesting possible completions based upon what has already been typed. As shown in fig5.4

The advanced method automatically sends proof text to Coq using the PIDE protocol, so command text does not have to be manually sent. The output of a command the cursor is on will be displayed as the prover processes through the proof text. Output with multiple lines have all their lines shown as a new line. As shown in fig 5.5



```
Proof.
  simpl.
  refl
Q reflexivity
```

**Figure 5.4:** Autocomplete



```
Inductive day : Type :=
  | monday : day
  | tuesday : day
  | wednesday : day
  | thursday : day
  | friday : day
  | saturday : day
  | sunday : day.

(** The type is called [day], a
```

```
day is defined
day_rect is defined
day_ind is defined
day_rec is defined
```

**Figure 5.5:** PIDE

The user is can also move their cursor onto different commands to display the associated output . As shown in fig 5.6

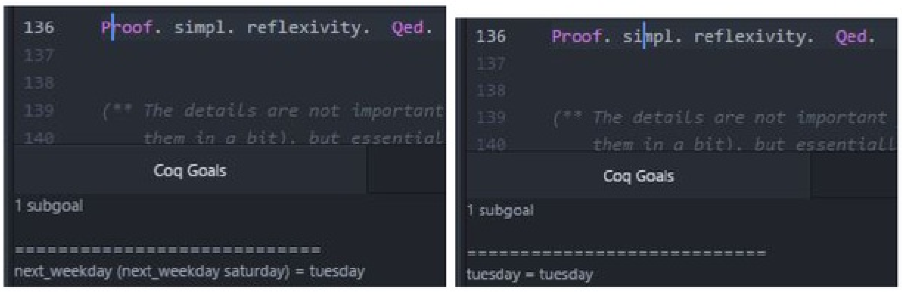
Errors are shown to the user through the use of the linter package. The example 5.7 shows an error by underlining it, and displaying an error dialogue.

## 5.2 Comparison

This package will be compared to other IDEs: Proof General, CoqIDE, Coqoon, and Jedit.

This package with Atom provides a modern user interface that is customisable and extendible. The other IDE have more dated interfaced. CoqIDE is not very customisable nor extendible. Proof General although customisable on the emacs platform it is written in Lisp which unlike Atom or, Coqoon and Jedit using JavaScript and Java respectively which are more popular and simpler to learn languages.

All IDEs provide Coq syntax highlighting although Jedit on my computer did not



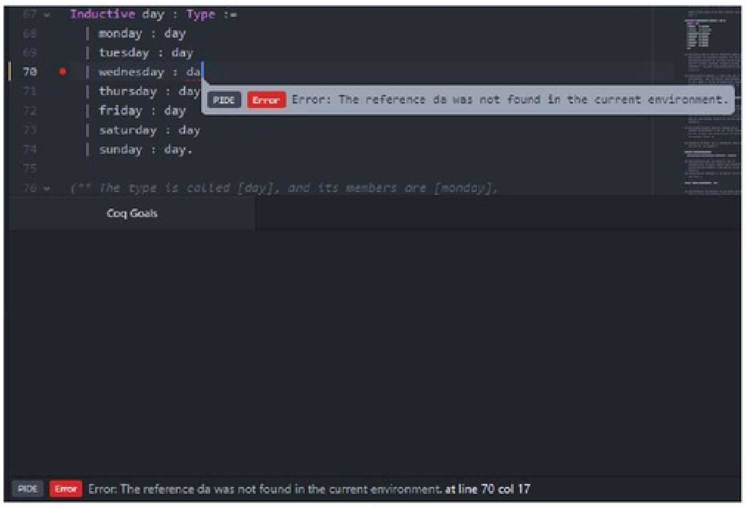
```
Proof. simpl. reflexivity.  Qed.


(** The details are not important
    them in a bit), but essential
```

```
1 subgoal
============================
next_weekday (next_weekday saturday) = tuesday
```

```
Proof. simpl. reflexivity.  Qed.


(** The details are not important
    them in a bit), but essentiall
```

```
1 subgoal
============================
tuesday = tuesday
```

**Figure 5.6:** Different command output



```
Inductive day : Type :=
  | monday : day
  | tuesday : day
  | wednesday : da
  | thursday : day
  | friday : day
  | saturday : day
  | sunday : day.

(** The type is called [day], and its members are [monday],
```

```
Error: The reference da was not found in the current environment.
```

```
Error: The reference da was not found in the current environment. at line 70 col 17
```

**Figure 5.7:** Linter

properly highlight. Proof General with an additional package also provides prettification of operators, whilst the other IDEs do not.

Auto completion is also offered by CoqIDE, and Proof General with an additional package. An advantage of this project implementation is that it uses fuzzy string matching allowing keywords with partial string matches to be suggested.

A key advantage this project has over Proof General, and CoqIDE was its use of the PIDE protocol for interaction with Coq. This protocol is a significant change of interaction, as the proof text is automatically sent to the prover for proving instead of manually sending lines.

The error reporting is a step above CoqIDE, and Proof General which only display when an alert when errors occur. This project along with Jedit display the part of the text that caused an error by underlining it, making it easier to locate, and fix errors.

**Evaluation** The syntax highlighting provides a basic highlighting of a subset of Coq syntax, this could be further extended by highlighting a greater portion of Coq. Information from PIDE could also be used to augment syntax highlighting.

Operator and symbols were prettified by utilising CSS. Although this worked well some may argue this use was a 'hack'. An alternative method which could be investigated would be to use folds, which used to hide code blocks. Instead operators could be folded and the Unicode replacement displayed.

Auto completion does a good job a suggesting possible Coq keywords through semi matching, and partial matches. But it does not suggest user created names, this could be implemented by utilising the reports generated by the PIDE protocol.

The basic interaction method between Atom and Coq was partially implemented. This could be further implemented to support the newer XML protocol, although it still uses the slower waterfall method and with CoqIDE and Proof General (in development) supporting it and already and being established IDEs it may not be as useful.

The document model currently stores command spans in a list. This has performance drawbacks on access. To access a certain element in a list a linear search must be performed which is O(n). These searches happen numerous times during processing and may add-up especially for a real-time proof editing and longer documents. An alternative method would be to store all commands in a map, and command ordering could stored as extra link data, or a list. This method would have a O(1) to access commands using their command identifier.

There is a limitation on parsing comments in Coq proofs. Comments that contain comments cannot be parsed correctly through regular expressions. Regex will either be too greedy or lazy.

```
1 (* Comment 1
2   (* Comment 2 *)
3 *)
4 Definition admit {T: Type} : T.  Admitted.
5 (* Comment 3 *)
```

A greedy regex will match lines 1 - 5. This is a problem as the command on line 4 will no longer be processed as its sent to the prover as a comment. A lazy regex would match lines 1 - 2 and 5. This does not work as the closing '*)' on line 3 will be part of the command on line 4 making it invalid.

A solution would be to use a regular expression library that supports recursive expressions, or creating a parser.

# Chapter 6

# Conclusion

This project has implemented a package adding Coq proof assistant support for the Atom text editor. The project identified features offered by IDEs, and proof editors to be implemented for the package. Each feature was designed and implemented. Some implementations were improved upon as better designs were created.

A literature review was conducted to research the area and background of this project. The Coq language and Atom platform was researched. Current offerings of IDEs and proof assistants were surveyed for features. PIDE required deep research of the protocol and reverse engineering how it worked by observing messages between Coq and the editor.

Syntax highlighting, auto complete, prettification, basic and advanced interaction, and linting were identified during the background research for design and implementation.

During the design/implementation of the advanced coq interaction (PIDE) it was found the existing editor library written in Scala. Two methods were explored to interoperate Scala and JavaScript, by using ScalaJS, or the npm library java. Unfortunately these methods were unsuccessful as underlying data structures were not implemented. Thus, the editor side of PIDE had to be redesigned. A basic implementation of the PIDE protocol was able to be created in JavaScript for the Atom package.

## 6.1 Future Work

This project has designed and implemented basic Coq support for Atom. It provides a base for extension, and expansion. In Evaluation short comings of implemented features were outlined and possible improvements were suggested.

There are also many more features which are offered by IDE, and proof assistants which could be implement. Suggested features:

1. An Identifier list that maps names with locations allowing users to jump to declarations

2. Displaying documentation of Coq keywords within the editor

3. Visual progress of the Proof state

4. Code snippets

The PIDE protocol is an interesting area for the Coq proof assistant. It has been embraced by Isabelle, changing how its proofs are performed. A greater portion of PIDE could be implemented for Coq, providing more information for both editor and prover. The documentation of PIDE could also be improved, as this would increase knowledge and understanding of the protocol. Thus, increasing developer interest, and support. This project has implemented the PIDE protocol strongly to the Atom and Coq platform but general implementation could be developed. Allowing for more proof assistants, to be implemented on JavaScript based editors.

# Chapter 7

# Abbreviations

| | |
|---|---|
| IDE | Integrated Development Environment |
| PIDE | Prover Integrated Development Environment |
| REPL | read-eval-print loop |

# Appendix A

# Code

## A.1 coq.cson

```
'scopeName': 'source.coq'
'name': 'Coq'
'fileTypes': ['v', 'atom://coq']
'patterns': [
  {
    'include': '#comment'
  }
  {
    'include': '#string'
  }
  {
    'include': '#sort'
  }
  {
    'include': '#binder'
  }
  {
    'include': '#let'
  }
  {
    'include': '#match'
  }
  {
    'include': '#if'
  }
  {
    'include': '#axiom'
  }
  {
    'include': '#variable'
  }
  {
    'include': '#definitions'
  }
  {
    'include': '#inductive-definitions'
  }
  {
    'include': '#recursive-functions'
  }
  {
    'include': '#assertions-proofs'
  }
  {
    'include': '#special-token'
  }
]
'repository':
  'comment':
      'name': 'comment.block.coq'
      'begin': '(\\(\\*)'
      'end': '(\\*\\))'
      'patterns': [
        {
          'include': '#comment'
        }
      ]

  'ident':
    'name': 'enity.name.function.coq'
    'match': '([a-zA-Z_][a-zA-Z0-9_]*)'

  'access_ident':
    'name': 'enity.name.function.coq'
    'match': '(\\.[a-zA-Z_][a-zA-Z0-9_]*)'

  'num':
    'name': 'constant.numeric.coq'
    'match': '([0-9]+)'

  'integer':
    'name': 'constant.numeric.coq'
    'match': '(-?[0-9]+)'

  'string':
    'name': 'string.quoted.double.coq'
    'begin': '"'
    'end': '"'

  'keyword':
    'name': 'keyword.other.coq'
    'match': '\\b(_|as|at|cofix|else|end|
              exists|exists2|fix|for|forall|fun|
              if|IF|in|let|match|mod|
              Prop|return|Set|then|Type|using|
              where|with)\\b'

  'special-token':
    'patterns': [
      {
        'name': 'keyword.operator.times.coq'
        'match': '(\\*)'
      }
      {
        'name': 'keyword.operator.leftrightarrow.coq'
        'match': '(<->)'
      }
      {
        'name': 'keyword.operator.rightarrow.coq'
        'match': '(->)'
      }
      {
        'name': 'keyword.operator.leftarrow.coq'
        'match': '(<-)'
      }
      {
        'name': 'keyword.operator.Rightarrow.coq'
        'match': '(=>)'
      }
      {
        'name': 'keyword.operator.lessequal.coq'
        'match': '(<=)'
      }
      {
        'name': 'keyword.operator.greaterequal.coq'
        'match': '(>=)'
      }
      {
        'name': 'keyword.operator.notequal.coq'
        'match': '(<>)'
      }
      {
        'name': 'keyword.operator.not.coq'
        'match': '(~)'
      }
      {
        'name': 'keyword.operator.and.coq'
        'match': '(/\\\\)'
      }
      {
        'name': 'keyword.operator.or.coq'
        'match': '(\\\\/)'
      }
      {
        'name': 'keyword.operator.turnstile.coq'
        'match': '(\\|-)'
      }
      {
        'name': 'keyword.operator.ken.coq'
        'match': '(!|%|&|&&|\\(|\\(\\)|\\)|
                    \\*|\\+|\\+\\+|,|-|->|\\.|
                    \\.\\(|\\.\\.|\\/|\\\\|:|::|:<|
                    :=|:>|;|<|<-|<->|<:|
                    <=|<>|=|=>|=_D|>|>->|
                    >=|\\?|\\?=|@|\\[|\\\/|\\]|
                    \\^|{|\\||\\|-|\\|\\||}|~)'
      }
    ]

  'sort':
    'name': 'keyword.other.coq'
    'match': '(Prop|Set|Type)'

  'binder':
    'name': 'keyword.other.coq'
    'match': '\\b(fun|forall|fix|cofix)\\b'

  'let':
    'name': 'keyword.other.coq'
    'match': '\\b(let|in)\\b'

  'match':
    'name': 'keyword.other.coq'
    'match': '\\b(match|with)\\b'

  'if':
    'name': 'keyword.other.coq'
    'match': '\\b(if|then|else)\\b'

  'axiom':
    'name': 'keyword.other.coq'
    'match': '\\b(Axiom|Parameter|Local Axiom|Conjecture)\\b'

  'variable':
    'name': 'keyword.other.coq'
    'match': '\\b(Variable|Variables|Hypothesis|Hypotheses)\\b'

  'definitions':
    'patterns': [
      {
        'name': 'keyword.other.coq'
        'match': '\\b(Definition|Local|Hypothesis|Hypotheses)\\b'
      }
      {
        'name': 'keyword.other.coq'
        'match': '\\b(Let|Fixpoint|CoFixpoint|with)\\b'
      }
    ]

  'inductive-definitions':
    'patterns': [
      {
        'name': 'keyword.other.coq'
        'match': '\\b(Inductive|with)\\b'
      }
      {
        'name': 'keyword.other.coq'
        'match': '\\b(CoInductive|with)\\b'
      }
    ]

  'recursive-functions':
    'patterns': [
      {
        'name': 'keyword.other.coq'
        'match': '\\b(Fixpoint|with)\\b'
      }
      {
        'name': 'keyword.other.coq'
        'match': '\\b(CoFixpoint|with)\\b'
      }
    ]

  'assertions-proofs':
    'patterns': [
      {
        'name': 'keyword.other.coq'
        'match': '\\b(Theorem|Lemma|Remark|Fact|Corollary|Proposition)\\b'
      }
      {
        'name': 'keyword.other.coq'
        'match': '\\b(Proof|Qed|Defined|Admitted)\\b'
      }
    ]
```

## A.2 provider.js

```
'use babel';

import fuzzaldrin from "fuzzaldrin";

let syntax = require('./syntax');

export default class Provider {
    selector = '.source.coq';
    inclusionPriority = 1;
    excludeLowerPriority = true;

    constructor() {
    }

    getSuggestions({editor, bufferPosition, scopeDescriptor, prefix, activatedManually}) {
        return new Promise(resolve => {
            var wordList = this.getWordList(prefix);
            var suggestionList = this.getSuggestionList(wordList);
            resolve(suggestionList);
        });
    }

    /**
     * Creates a list of possible keywords according to prefix.
     * Comparison of keyword and prefix are not case sensitive.
     */
    getWordList(prefix) {
        // return syntax.keyword.filter(word => word.toLowerCase
           ().startsWith(prefix.toLowerCase()));
        return fuzzaldrin.filter(syntax.keyword.concat(syntax.
           tactic), prefix);
    }

    /**
     * Creates a suggestion object list.
     * List is sorted in ascending order of text length.
     */
    getSuggestionList(wordList) {
        // return wordList.sort((a, b) => a.length - b.length).
           map(this.getSuggestion);
        return wordList.map(this.getSuggestion);
    }

    getSuggestion(word) {
        return {
            text: word
        }
    }

}
```

## A.3 syntax.json

```
{
  "keyword": [
    "About",
    "AddPath",
    "Axiom",
    "Abort",
    "Chapter",
    "Check",
    "Coercion",
    "Compute",
    "CoFixpoint",
    "CoInductive",
    "Corollary",
    "Defined",
    "Definition",
    "End",
    "Eval",
    "Example",
    "Export",
    "Fact",
    "Fix",
    "Fixpoint",
    "Function",
    "Generalizable",
    "Global",
    "Grammar",
    "Guarded",
    "Goal",
    "Hint",
    "Debug",
    "On",
    "Hypothesis",
    "Hypotheses",
    "Resolve",
    "Unfold",
    "Immediate",
    "Extern",
    "Constructors",
    "Rewrite",
    "Implicit",
    "Import",
    "Inductive",
    "Infix",
    "Lemma",
    "Let",
    "Load",
    "Local",
    "Ltac",
    "Module",
    "Module Type",
    "Declare Module",
    "Include",
    "Mutual",
    "Parameter",
    "Parameters",
    "Print",
    "Printing",
    "All",
    "Proof",
    "Proof with",
    "Qed",
    "Record",
    "Recursive",
    "Remark",
    "Require",
    "Save",
    "Scheme",
    "Assumptions",
    "Axioms",
    "Universes",
    "Induction",
    "for",
    "Sort",
    "Section",
    "Show",
    "Structure",
    "Syntactic",
    "Syntax",
    "Tactic",
    "Theorem",
    "Search",
    "SearchAbout",
    "SearchHead",
    "SearchPattern",
    "SearchRewrite",
    "Set",
    "Types",
    "Undo",
    "Unset",
    "Variable",
    "Variables",
    "Context",
    "Notation",
    "Reserved Notation",
    "Tactic Notation",
    "Delimit",
    "Bind",
    "Open",
    "Scope",
    "Inline",
    "Implicit Arguments",
    "Add",
    "Strict",
    "Typeclasses",
    "Instance",
    "Global Instance",
    "Class",
    "Instantiation",
    "subgoal",
    "subgoals",
    "vm_compute",
    "Opaque",
    "Transparent",
    "Time",
    "Extraction",
    "Extract",
    "Variant",
    "Program Definition",
    "Program Example",
    "Program Fixpoint",
    "Program Lemma",
    "Obligation",
    "Obligations",
    "Solve",
    "using",
    "Next Obligation",
    "Next",
    "Program Instance",
    "Equations",
    "Equations_nocomp",
    "forall",
    "match",
    "as",
    "in",
    "return",
    "with",
    "end",
    "let",
    "fun",
    "if",
    "then",
    "else",
    "Prop",
    "Set",
    "Type",
    ":=",
    "where",
    "struct",
    "wf",
    "measure",
    "fix",
    "cofix",
    "before",
    "after",
    "constr",
    "ltac",
    "goal",
    "context",
    "beta",
    "delta",
    "iota",
    "zeta",
    "lazymatch",
    "level",
    "associativity",
    "no"
  ],
  "tactic": [
    "intro",
    "intros",
    "apply",
    "rewrite",
    "refine",
    "case",
    "clear",
    "injection",
    "elimtype",
    "progress",
    "setoid_rewrite",
    "left",
    "right",
    "constructor",
    "econstructor",
    "decide equality",
    "abstract",
    "exists",
    "cbv",
    "simple destruct",
    "info",
    "fourier",
    "field",
    "specialize",
    "evar",
    "solve",
    "instanciate",
    "quote",
    "eexact",
    "autorewrite",
    "destruct",
    "destruction",
    "destruct_call",
    "dependent",
    "elim",
    "extensionality",
    "f_equal",
    "generalize",
    "generalize_eqs",
    "generalize_eqs_vars",
    "induction",
    "rename",
    "move",
    "omega",
    "set",
    "assert",
    "do",
    "repeat",
    "cut",
    "assumption",
    "exact",
    "split",
    "subst",
    "try",
    "discriminate",
    "simpl",
    "unfold",
    "red",
    "compute",
    "at",
    "in",
    "by",
    "reflexivity",
    "symmetry",
    "transitivity",
    "replace",
    "setoid_replace",
    "inversion",
    "inversion_clear",
    "pattern",
    "intuition",
    "congruence",
    "fail",
    "fresh",
    "trivial",
    "tauto",
    "firstorder",
    "ring",
    "clapply",
    "program_simpl",
    "program_simplify",
    "eapply",
    "auto",
    "eauto",
    "change",
    "fold",
    "hnf",
    "lazy",
    "simple",
    "eexists",
    "debug",
    "idtac",
    "first",
    "type of",
    "pose",
    "eval",
    "instantiate",
    "until"
  ]
}
```

## A.4 PIDECommand.js

```
'use babel'

import yxmlNodes from '../yxml/yxmlNodes.js';

export default class PIDECommands {

    static printCommandsYXML(node) {
        switch (node.kind) {
            case 'Document.define_command':
                return node.kind.length + ',' +
                    node.commandID.length + ',' +
                    node.commandType.length + ',' +
                    node.command.length + '\n' +
                    node.kind +
                    node.commandID +
                    node.commandType +
                    node.command;
            case 'Document.discontinue_execution':
                return node.kind.length + '\n' + node.kind;
            case 'Document.update':
                let yxml = yxmlNodes.printNodesYXML(node.
                   updateYXML);
                return node.kind.length + ',' +
                    node.oldState.length + ',' +
                    node.newState.length + ',' +
                    yxml.length + '\n' +
                    node.kind +
                    node.oldState +
                    node.newState +
                    yxml;
            default:
                console.log('Command print Error');
                console.log(node);
                return 'ERROR';
        }
    }

    static defineCommand(commandID, commandType, command, range)
       {
        return {
            kind: 'Document.define_command',
            commandID,
            commandType,
            command,
            range
        }
    }

    static discontinueExecution() {
        return {
            kind: 'Document.discontinue_execution'
        }
    }

    static update(oldState, newState, updateYXML) {
        return {
            kind :'Document.update',
            oldState,
            newState,
            updateYXML
        }
    }

}
```

## A.5 updateNodeHelper.js

```
'use babel';

import YxmlNodes from "../yxml/yxmlNodes";

export default class YxmlTree {

    static updateNode(file, perspective, document) {
        let fileNode = this.fileNode(file);
        let perspectiveElement = this.perspectiveNode(fileNode,
           perspective);
        let documentElement = this.documentNode(fileNode,
           document);
        return YxmlNodes.update(perspectiveElement,
           documentElement);
    }

    static perspectiveNode(file, perspective) {
        let body = this.perspectiveBody(perspective);
        return this.colonNode([file, body]);
    }

    static perspectiveBody(perspective) {
        let attributes = this.listToAttributes(perspective);
        let node = YxmlNodes.element(this.textNode(2), attributes
          , []);
        return this.colonNode([node]);
    }

    static listToAttributes(perspective) {
        let attributes = [YxmlNodes.attribute(this.textNode(0),
           this.textNode(0))];
        perspective.forEach((value, i) => {
            attributes.push(YxmlNodes.attribute(this.textNode(i +
                1), this.textNode(value)));
        });
        return attributes;
    }

    static documentNode(file, document) {
        let pairList = document.map(this.documentPair, this);
        let zeroBody = this.zeroNode(pairList);
        let documentBody = this.colonNode([zeroBody]);
        return this.colonNode([file, documentBody]);
    }

    static zeroNode(body) {
        return YxmlNodes.element(this.textNode(0), [], body);
    }

    static documentPair(pair) {
        let from;
        let to;
        if (pair[0] == null) {
            from = this.colonNode([]);
        } else {
            from = this.colonNode([this.colonNode([this.textNode(
               pair[0])])]);
        }
        if (pair[1] == null) {
            to = this.colonNode([]);
        } else {
            to = this.colonNode([this.colonNode([this.textNode(
               pair[1])])]);
        }
        return this.colonNode([from, to])
    }

    static fileNode(file) {
        return this.colonNode([this.textNode(file)]);
    }

    static colonNode(body) {
        return YxmlNodes.element(this.textNode(':'), [], body);
    }

    static textNode(text) {
        return YxmlNodes.text(text.toString());
    }

}
```

## A.6 yxml.pegjs

```
{
    function elementNode(name, attributes, body) {
        return {
            kind: 'element',
            name,
            attributes,
            body
        }
    }

    function attributeNode(name, value) {
        return {
            kind: 'attribute',
            name,
            value
        }
    }

    function textNode(text) {
        return {
            kind: 'text',
            text
        }
    }
}

element
        = X Y name:text attributes:(attribute)* X body:(element)*
            X Y X { return elementNode(name, attributes, body); }
    / text

attribute
        = Y name:attrText '=' value:text { return attributeNode(
           name, value); }

attrText
    = text:[_a-zA-Z0-9]+ { return textNode(text.join('')); }

text
        = text:[- =_.:;()~<>/\\a-zA-Z0-9\n,'"\[\]+*|]+ { return
           textNode(text.join('')); }

X
        = '\u0005'

Y
        = '\u0006'
```

## A.7 yxmlNodes.js

```
'use babel'

let parser = require('./yxml.js');

export default class YxmlNodes {

    /**
     * Parses YXML header/body data into a tree.
     */
    static parseData(data) {
        let dataMessages = [];
        let i = 0;
        while (i < data.length) {
            let outer = this.getMessageSlice(data, i);
            let inner = this.getMessageSlice(data, outer.end);
            i = inner.end;
            let message = outer.data.slice(0, -3).toString() +
                inner.data.toString() + outer.data.slice(-3).
                   toString();
            let parsedMessage = parser.parse(message);
            dataMessages.push(parsedMessage);
        }
        return dataMessages;
    }

    static getMessageSlice(data, start) {
        let i = start;
        let n = 0;
        while (i < data.length) {
            if (data[i] == 10) {
                break;
            } else {
                n = n * 10 + parseInt(String.fromCharCode(data[i
                   ]));
                i++;
            }
        }
        return {
            data: data.slice(i + 1, i + n + 1),
            end: i + 1 + n
        }
    }

    static printNodesYXML(node) {
        switch (node.kind) {
            case 'element':
                let attributeString = '';
                let bodyString = '';
                node.attributes.forEach((attr) => attributeString += this.printNodesYXML(attr));
                node.body.forEach((body) => bodyString += this.printNodesYXML(body));
                return '\u0005\u0006' + this.printNodesYXML(node.name) + attributeString + '\u0005' + bodyString + '\u0005\u0006\u0005';
            case 'attribute':
                return '\u0006' + this.printNodesYXML(node.name) + '=' + this.printNodesYXML(node.value);
            case 'text':
                return node.text;
            case 'update':
                return this.printNodesYXML(node.perspectiveElement) + this.printNodesYXML(node.documentElement);
            default:
                console.log('YXML print error');
                console.log(node);
                return 'ERROR';
        }
    }

    static printNodesXML(node) {
        switch (node.kind) {
            case 'element':
                let name = this.printNodesXML(node.name);
                let attributeString = '';
                let bodyString = '';
                node.attributes.forEach((attr) => {
                    attributeString += ` ${this.printNodesXML(attr)}`
                });
                node.body.forEach((body) => {
                    bodyString += `\n${this.printNodesXML(body)}`
                });
                return `<${name}${attributeString}>${bodyString}\n</${name}>`;
            case 'attribute':
                return `${this.printNodesXML(node.name)}=${this.printNodesXML(node.value)}`;
            case 'text':
                return node.text;
            case 'update':
                return this.printNodesXML(node.perspectiveElement
                   ) + this.printNodesXML(node.documentElement);
            default:
                console.log('XML print error');
                console.log(node);
                return 'ERROR';
        }
    }

    static element(name, attributes, body) {
        return {
            kind: 'element',
            name,
            attributes,
            body
        }
    }

    static attribute(name, value) {
        return {
            kind: 'attribute',
            name,
            value
        }
    }

    static text(text) {
        return {
            kind: 'text',
            text,
        }
    }

    static update(perspectiveElement, documentElement) {
        return {
            kind: 'update',
            perspectiveElement,
            documentElement
        }
    }

}
```

## A.8 DocumentModel.js

```
'use babel'

import PIDECommands from "./protocol/PIDECommands";
import YxmlTree from "./protocol/updateNodeHelper";
import YxmlNodes from "./yxml/yxmlNodes";
import Messages from './messages';
import errorMessages from './errorMessages';

export default class DocumentModel {
    textEditor;
    coq;
    commandCounter;
    model;
    state;
    view;
    linter;

    constructor(textEditor, view, linter) {
        this.textEditor = textEditor;
        this.commandCounter = 0;
        this.state = '0';
        this.model = [];
        this.message = new Messages();
        this.view = view;
        this.linter = linter;
        atom.workspace.open('atom://coq', {split: 'down'});
    }

    start(coq) {
        this.coq = coq;
        this.onLoad();
        this.textEditor.onDidStopChanging(this.onChange.bind(this
           ));
        this.textEditor.onDidChangeCursorPosition(this.
           onCursorChange.bind(this));
        this.textEditor.onDidDestroy(this.coq.close.bind(this.coq
           ));
    }

    processOutput(data) {
        YxmlNodes.parseData(data).forEach((message) => {
            switch (message.name.text) {
                case 'writeln':
                    this.message.insertMessage(message);
                    break;
                case 'protocol':
                    this.message.updateExecutionMap(message);
                    break;
                case 'error':
                    if (message.attributes.length === 4) {
                        let commandID = this.message.getCommandID(message.attributes[3].value.text);
                        let command = this.getCommand(commandID);
                        let range = errorMessages.getRange(command, message);
                        let error = this.textEditor.getTextInRange(range);
                        this.linter.setMessages([{
                            type: 'Error',
                            filePath: this.textEditor.getPath(),
                            range: range,
                            text: message.body[0].text,
                        }])
                    }
                    break;
                case 'report':
                    break;
                case 'status':
                    break;
                default:
                    console.log(YxmlNodes.printNodesXML(message));
            }
        });
        this.displayMessage(this.textEditor.getCursorBufferPosition());
    }

    onLoad() {
        let edits = this.parseSpans();
        let pendingEdits = [];
        let updateLinks = [];
        for (let i = 0; i < edits.length; i++) {
            pendingEdits.push(edits[i]);
            this.pushUpdateLink(i, edits, updateLinks);
        }
        this.model = edits;
        this.sendEdits(pendingEdits, updateLinks);
    }

    parseSpans() {
        let edits = [];
        this.textEditor.scan(/(\(\*[\w\W]*?\*\)|\s+)|([^.]*\.|[\s
           \S]+)/g,
            (span) => {
                let command = span.matchText;
                let type = span.match[1] == undefined ? '0' :
                   '1';
                let defineCommand = PIDECommands.defineCommand(
                    (--this.commandCounter).toString(), type,
                    command, this.textEditor.markBufferRange(span
                       .range)
                );
                edits.push(defineCommand);
            }
        );
        return edits;
    }

    sendEdits(pendingEdits, updateLinks) {
        this.linter.setMessages([]);
        let update = this.getUpdate(updateLinks);
        pendingEdits.push(update);
        this.coq.input(pendingEdits);
    }

    onCursorChange(event) {
        if (!event.textChanged) {
                let messageDisplayed = this.displayMessage(event.
                   newBufferPosition);
            if (!messageDisplayed) {
                let update = this.getUpdate([]);
                this.coq.input([update]);
            }
        }
    }

    displayMessage(point) {
        let span = this.findCommandIn(point);
        if (span) {
            let commandID = span.command.commandID;
            let output = this.message.getMessage(commandID);
            this.view.container.html(output.replace(/\n/g, '<br
               >'));
            return output !== '';
        }
    }

    findCommandIn(point) {
        for (let i = 0; i < this.model.length; i++) {
            let command = this.model[i];
            if (command.range.getBufferRange().containsPoint(
               point)) {
                return {
                    command,
                    index: i
                };
            }
        }
    }

    findCommandInRev(point) {
        for (let i = this.model.length - 1; i >= 0; i--) {
            let command = this.model[i];
            if (command.range.getBufferRange().containsPoint(
               point)) {
                return {
                    command,
                    index: i
                }
            }
        }
    }

    onChange(object) {
        for (let change of object.changes) {
            this.onTextChange(change);
        }
    }

    onTextChange(change) {
        let updateLinks = [];
        let startSpan = this.findCommandIn(change.start);
        let changeEnd = this.sumPoints(change.start, change.
           newExtent);
        let endSpan = this.findCommandInRev(changeEnd);
        let startPoint = startSpan ? startSpan.command.range.
           getBufferRange().start : [0, 0];
        let endPoint = endSpan ? endSpan.command.range.
           getBufferRange().end : [Infinity, Infinity];
        let edits = this.parseSpanChanges(
            startPoint,
            endPoint
        );

        if (endSpan) {
            let dirtyIndex = this.getLastDirtyIndex(endSpan.index
                , edits);
            for (let i = dirtyIndex; i >= startSpan.index; i--) {
                this.removeLink(i, this.model, updateLinks);
            }

            this.newLink(startSpan.index, edits, updateLinks);
            for (let i = 1; i < edits.length; i++) {
                this.pushUpdateLink(i, edits, updateLinks);
            }

            let dirtyCount = 1 + dirtyIndex - startSpan.index;
            this.model.splice(startSpan.index, dirtyCount, ...
                edits);
        } else {
            for (let i = 0; i < edits.length; i++) {
                this.pushUpdateLink(i, edits, updateLinks);
            }
            this.model.push(...edits);
        }
        this.sendEdits(edits, updateLinks);
    }


    parseSpanChanges(startPoint, endPoint) {
        let edits = [];
        this.textEditor.scanInBufferRange(
            /(\(\*.*\*\)|\s+)|([^.]*\.|[\s\S]+)/g,
            [startPoint, [Infinity, Infinity]],
            (span) => {
                let command = span.matchText;
                let type = span.match[1] == undefined ? '0' :
                    '1';
                let defineCommand = PIDECommands.defineCommand(
                    (--this.commandCounter).toString(), type,
                    command, this.textEditor.markBufferRange(span
                        .range)
                );
                edits.push(defineCommand);
                if (span.range.containsPoint(endPoint)) {
                    span.stop();
                }
            }
        );
        return edits;
    }

    getLastDirtyIndex(endIndex, edits) {
        let dirtyIndex = endIndex;
        let lastEdit = edits[edits.length - 1];
        if (lastEdit) {
            for (let i = endIndex; i < this.model.length; i++) {
                if (lastEdit.range.getBufferRange().end.
                   isGreaterThan(
                         this.model[i].range.getBufferRange().
                            start)) {
                    dirtyIndex = i;
                } else {
                    break;
                }
            }
        }
        return dirtyIndex;
    }

    pushUpdateLink(index, edits, updateLinks) {
        let from = null;
        if (index > 0) {
            from = edits[index - 1].commandID;
        }
        updateLinks.push([from, edits[index].commandID]);
    }

    newLink(index, edits, updateLinks) {
        let from = null;
        if (index > 0) {
            from = this.model[index - 1].commandID;
        }
        updateLinks.push([from, edits[0].commandID]);
    }

    removeLink(index, edits, updateLinks) {
        let from = null;
        if (index > 0) {
            from = edits[index - 1].commandID;
        }
        updateLinks.push([from, null]);
    }

    sumPoints(a, b) {
        return [a.row + b.row, b.row === 0 ? a.column + b.column
           : b.column];
    }

    getUpdate(updateLinks) {
        let perspective = this.getPerspective();
        let updateYXML = YxmlTree.updateNode(this.textEditor.
           getPath(), perspective, updateLinks);
        let state = this.state;
        this.state = (--this.commandCounter).toString();
        return PIDECommands.update(state, this.commandCounter.
           toString(), updateYXML);
    }

    getPerspective() {
        let perspective = [];
        let cursorPoint = this.textEditor.getCursorBufferPosition
           ();
        this.model.forEach((command) => {
            if (command.range.getBufferRange().containsPoint(
               cursorPoint)) {
                perspective.push(command.commandID.toString());
            }
        });
        return perspective;
    }

    getCommand(ID) {
        for (let i = 0; i < this.model.length; i++) {
            let command = this.model[i];
            if (ID === command.commandID) {
                return command;
            }
        }
    }

}
```

## A.9 errorMessages.js

```
'use babel'
```

```

export default class errorMessages {

    static getRange(command, message) {
        return this.findRange(command.command, command.range.
           getBufferRange(),
            message.attributes[1].value.text, message.attributes
               [2].value.text);
    }

    static findRange(text, offsetRange, startOffset, endOffset) {
        let row = offsetRange.start.row;
        let column = offsetRange.start.column;
        let startRange = [0, 0];
        for (let i = 0; i < endOffset - 1; i++) {
            if (i == startOffset - 1) {
                startRange = [row, column];
            }
            if (text.charAt(i) === '\n') {
                row++;
                column = 0;
            } else {
                column++;
            }
        }
        let endRange = [row, column];
        return [startRange, endRange];
    }
}
```

## A.10 goals-view.js

```
'use babel';

import {ScrollView} from "atom-space-pen-views";

export default class GoalsView extends ScrollView {
    static content() {
        this.div({class: 'Goals'}, () => {
            this.div({outlet: 'container'});
        });
    }

    constructor() {
        super();
    }

    getTitle() {
        return 'Coq Goals';
    }
}
```

## A.11 messages.js

```
'use babel'

export default class Messages {
    commandExecutionMap;
    messageMap;

    constructor() {
        this.commandExecutionMap = new Map();
        this.messageMap = new Map();
    }

    updateExecutionMap(updateMessage) {
        updateMessage.body[1].body.forEach((data) => this.
           executionData(data));
    }

    executionData(data) {
        if (data.body[1].body.length === 0) {
            this.removeMessage(data.body[0].body[0].text);
            this.commandExecutionMap.delete(data.body[0].body[0].
               text);
        } else {
            this.commandExecutionMap.set(data.body[0].body[0].
               text,
                data.body[1].body.map((execution) => execution.
                   body[0].text));
        }
    }

    getCommandID(executionID) {
        for (let [key, value] of this.commandExecutionMap.entries
           ()) {
            if (value.includes(executionID.toString())) {
                return key;
            }
        }
        return 1;
    }

    insertMessage(message) {
        let output = this.messageMap.get(message.attributes[1].
           value.text) || '';
        output += message.body[0].text + '\n';
        this.messageMap.set(message.attributes[1].value.text,
           output);
    }

    removeMessage(commandID) {
        this.commandExecutionMap.get(commandID).forEach((key) =>
           {
            this.messageMap.delete(key);
        });
    }

    getMessage(commandID) {
        let output = '';
        if (this.commandExecutionMap.has(commandID)) {
            this.commandExecutionMap.get(commandID).forEach((key)
                => {
                output += this.messageMap.get(key) || '';
            });
        }
        return output;
    }

}
```

## A.12 pideTop.js

```
'use babel';

let net = require('net');
import {BufferedProcess} from "atom";
import PIDECommands from "./protocol/PIDECommands";

export default class PideTop {
    bufferedProcess;
    socket;
    server;
    documentModel;

    constructor(documentModel) {
        this.createServer();
        this.createCoqProcess();
        this.documentModel = documentModel;
    }

    /**
     * Creates a TCP server, for the editor and Coq to
        communicate on.
     */
    createServer() {
        this.server = net.createServer();
        this.server.on('connection', (socket) => {
            this.socket = socket;
            this.documentModel.start(this);
            this.socket.on('data', (data) => {
                this.documentModel.processOutput(data);
            });
        });
        this.server.on('error', (err) => {
            throw err;
        });
    }

    /**
     * Creates a Coq process when it receives a port from the
        server.
     */
    createCoqProcess() {
        let that = this;
        this.server.listen(() => {
            coqProcess(this.server.address());
        });

        function coqProcess(address) {
            let ipPort = '127.0.0.1:' + address.port;
            let command = 'coqTop';
            let args = [
                "-async-proofs", "on",
                "-toploop", "pidetop",
                "-main-channel", ipPort
            ];
            let stdout = (output) => {
                console.log('stdout: ' + output);
            };
            let stderr = (output) => {
                console.log('stderr: ' + output);
            };
            let exit = (code) => {
                console.log("exit: " + code);
            };
            that.bufferedProcess = new BufferedProcess({command,
               args, stdout, stderr, exit});
        }
    }

    /**
     * Writes data to the Coq TCP socket.
     * @param data PIDE formatted command
     */
    input(data) {
        data.unshift(PIDECommands.discontinueExecution());
        data.forEach((input) => {
            this.socket.write(PIDECommands.printCommandsYXML(
               input))
        });
    }

    /**
     * Kills the Coq process, and closes the server.
     */
    close() {
        this.server.close();
        this.bufferedProcess.kill();
    }
}
```

## A.13 coqCommunicate.js

```
'use babel'

import PideTop from "./pide/pideTop";
import DocumentModel from "./pide/DocumentModel";
import GoalsView from "./pide/goals-view";

export default class CoqCommunicate {
    subscriptions;
    coqTop;
    textEditor;
    linter;
    GoalsView;

    constructor(subscriptions, linter) {
        this.subscriptions = subscriptions;
        this.linter = linter;


        atom.workspace.addOpener((uri) => {
            if (uri.slice(-1) == 'v') {
                this.disposeText = atom.workspace.
                   observeTextEditors ((editor) => {
                    this.textEditor = editor;
                    this.startCoq(this.getGoalsView());
                    this.disposeText.dispose();
                });
            }

            if (uri == 'atom://coq') {
                return this.getGoalsView();
            }
        });
    }

    startCoq(view) {
        this.documentModel = new DocumentModel(this.textEditor,
           view, this.linter);
        this.coqTop = new PideTop(this.documentModel);
    }

    close() {
        this.coqTop.close();
    }

    getGoalsView(){
        if (this.GoalsView) {
            return this.GoalsView;
        } else {
            this.GoalsView = new GoalsView();
            return this.GoalsView;
        }
    }
}
```

## A.14 ken-coq.js

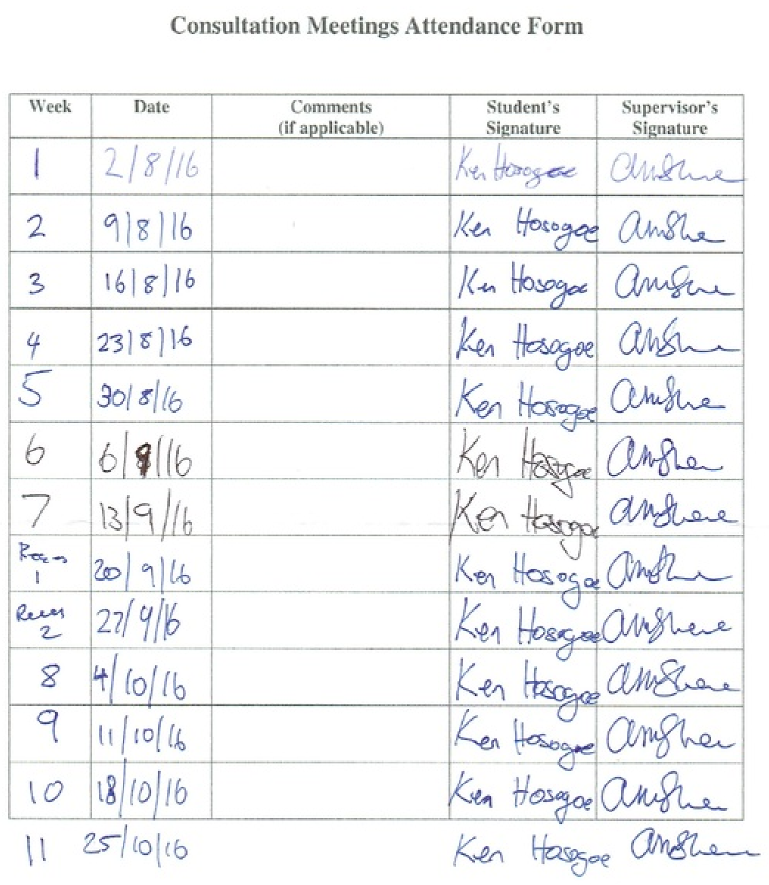
```
'use babel';

import KenCoqView from "./ken-coq-view";
import {CompositeDisposable} from "atom";
import Provider from "./autocomplete/provider";
import CoqCommunicate from "./communication/coqCommunicate";

export default {
    kenCoqView: null,
    modalPanel: null,
    subscriptions: null,
    linter: {},

    activate(state) {
        this.provider = new Provider();
        this.kenCoqView = new KenCoqView(state.kenCoqViewState);
        this.modalPanel = atom.workspace.addModalPanel({
            item: this.kenCoqView.getElement(),
            visible: false
        });

        // Events subscribed to in atom's system can be easily
           cleaned up with a CompositeDisposable
        this.subscriptions = new CompositeDisposable();

        // Register command that toggles this view
        // this.subscriptions.add(atom.commands.add('atom-
           workspace', {
        //     'ken-coq:toggle': () => this.toggle(),
        // }));
    },

    deactivate() {
      this.coqCommunicate.close();
      this.modalPanel.destroy();
      this.subscriptions.dispose();
      this.kenCoqView.destroy();
    },

    serialize() {
      return {
          kenCoqViewState: this.kenCoqView.serialize()
      };
    },

    provide() {
        return this.provider;
    },

    consumeLinter (indieRegistry) {
        this.linter = indieRegistry.register({name: 'PIDE'});
        this.subscriptions.add(this.linter);
        this.toggle()
    },

    toggle() {
        console.log('KenCoq was toggled!');
        this.coqCommunicate = new CoqCommunicate(this.
           subscriptions, this.linter);
    }
}
```

## A.15 ken-coq.atom-text-editor.less

```
// The ui-variables file is provided by base themes provided by
   Atom.
//
// See https://github.com/atom/atom-dark-ui/blob/master/styles/ui
   -variables.less
// for a full listing of what's available.
@import "ui-variables";

.ken-coq {
}

.highlight-green .region {
  background-color: #00ff00;
}

.times.coq {
  position: relative;
  visibility: hidden;

  &:after {
    content: '  ';
    position: absolute;
    visibility: visible;
    right: 0px;
  }
}

.rightarrow.coq {
  position: relative;
  visibility: hidden;

  &:after {
    content: '   ';
    position: absolute;
    visibility: visible;
    right: 4px;
  }
}

.leftarrow.coq {
  position: relative;
  visibility: hidden;

  &:after {
    content: '   ';
    position: absolute;
    visibility: visible;
    right: 4px;
  }
}

.leftrightarrow.coq {
  position: relative;
  visibility: hidden;

  &:after {
    content: '   ';
    position: absolute;
    visibility: visible;
    right: 3px;
  }
}


.Rightarrow.coq {
  position: relative;
  visibility: hidden;

  &:after {
    content: '   ';
    position: absolute;
    visibility: visible;
    right: 2px;
  }
}

.lessequal.coq {
  position: relative;
  visibility: hidden;

  &:after {
    content: '   ';
    position: absolute;
    visibility: visible;
    right: 4px;
  }
}

.greaterequal.coq {
  position: relative;
  visibility: hidden;

  &:after {
    content: '   ';
    position: absolute;
    visibility: visible;
    right: 4px;
  }
}

.notequal.coq {
  position: relative;
  visibility: hidden;

  &:after {
    content: '   ';
    position: absolute;
    visibility: visible;
    right: 4px;
  }
}

.not.coq {
  position: relative;
  visibility: hidden;

  &:after {
    content: '  ';
    position: absolute;
    visibility: visible;
    right: 0px;
  }
}

.and.coq {
  position: relative;
  visibility: hidden;

  &:after {
    content: '   ';
    position: absolute;
    visibility: visible;
    right:3px;
  }
}

.or.coq {
  position: relative;
  visibility: hidden;

  &:after {
    content: '   ';
    position: absolute;
    visibility: visible;
    right: 3px;
  }
}

.turnstile.coq {
  position: relative;
  visibility: hidden;

  &:after {
    content: '   ';
    position: absolute;
    visibility: visible;
    right: 3px;
  }
}
```

# Appendix B

# Consultation Meetings Attendance Form



Consultation Meetings Attendance Form

| Week | Date | Comments (if applicable) | Student's Signature | Supervisor's Signature |
|---|---|---|---|---|
| 1 | 2/8/16 | | Ken Hosogoe | [illegible] |
| 2 | 9/8/16 | | Ken Hosogoe | [illegible] |
| 3 | 16/8/16 | | Ken Hosogoe | [illegible] |
| 4 | 23/8/16 | | Ken Hosogoe | [illegible] |
| 5 | 30/8/16 | | Ken Hosogoe | [illegible] |
| 6 | 6/9/16 | | Ken Hosogoe | [illegible] |
| 7 | 13/9/16 | | Ken Hosogoe | [illegible] |
| Recess 1 | 20/9/16 | | Ken Hosogoe | [illegible] |
| Recess 2 | 27/9/16 | | Ken Hosogoe | [illegible] |
| 8 | 4/10/16 | | Ken Hosogoe | [illegible] |
| 9 | 11/10/16 | | Ken Hosogoe | [illegible] |
| 10 | 18/10/16 | | Ken Hosogoe | [illegible] |
| 11 | 25/10/16 | | Ken Hosogoe | [illegible] |

**Figure B.1:** Consultation Meetings Attendance Form

# Bibliography

[1] J. Harrison, "Formal prooftheory and practice," *Notices of the AMS*, vol. 55, no. 11, pp. 1395–1406, 2008.

[2] V. Pratt, "Anatomy of the pentium bug," in *Colloquium on Trees in Algebra and Programming.* Springer, 1995, pp. 97–107.

[3] J.-L. Lions *et al.*, "Ariane 5 flight 501 failure," 1996.

[4] C. Development Team, "What is coq?" https://coq.inria.fr/about-coq, 2016.

[5] Wikipedia, "Formal proof," https://en.wikipedia.org/wiki/Formal_proof, 2016.

[6] A. Development Team, "Atom," https://www.cis.upenn.edu/~bcpierce/sf/current/index.html, 2016.

[7] ——, "Atom," https://atom.io/, 2016.

[8] A. Sarkar, "The impact of syntax colouring on program comprehension," in *Proceedings of the 26th annual workshop of the psychology of programming interest group (ppig 2015)*, 2015.

[9] A. Abonyi, "Intellisense integration for coq theorem prover," *WORK*, vol. 5, pp. 1–3.

[10] A. Scandurra, "autocomplete+," https://github.com/atom/autocomplete-plus, 2016.

[11] C. Development Team, "The coq proof assistant reference manual: Version 8.5pl1," 2016.

[12] C. Tankink, "Pide for asynchronous interaction with coq," *arXiv preprint arXiv:1410.8221*, 2014.

[13] M. Wenzel, "Pide as front-end technology for coq," *arXiv preprint arXiv:1304.6626*, 2013.

[14] ——, "Isabelle/jedit–a prover ide within the pide framework," in *International Conference on Intelligent Computer Mathematics.* Springer, 2012, pp. 468–471.

[15] A. Development Team, “Atom documentation,” https://atom.io/docs, 2016.

[16] ——, “Packages make atom do amazing things.” https://atom.io/packages, 2016.