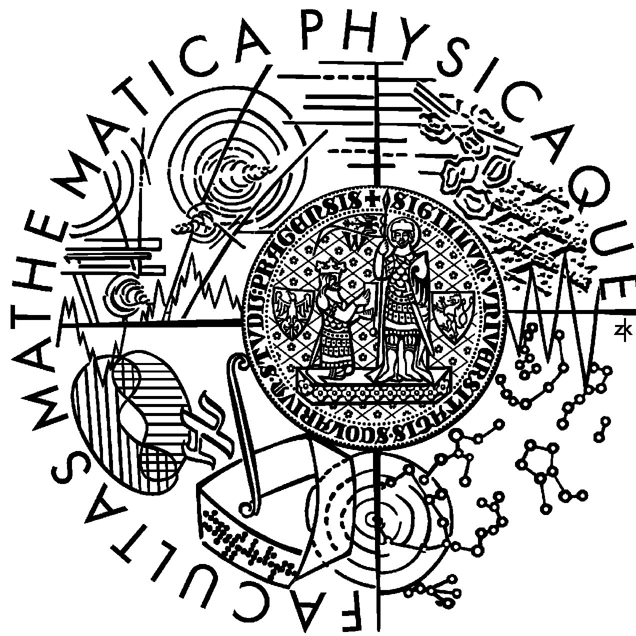
Charles University in Prague

Faculty of Mathematics and Physics

MASTER THESIS



Adam Abonyi

IntelliSense Integration for Coq theorem prover

Department of Software Engineering

Supervisor of the master thesis: Mgr. Tomáš Petříček

Study programme: Computer Science
Specialization: Software Systems

2012

I would like to thank Tomáš Petříček for masterfully guiding me through this thesis even though it was no easy feat, also for all his helpful advice and feedback, but mainly for having utter patience with me.

I would like to also thank my parents for supporting me throughout my studies. Even encouraging me to prioritize my studies over work to get me over the line, and mostly they have just been awesome parents for as long as I can remember.

I also shouldn't forget to mention Honza Ambrož and Daniel Balaš for being great friends and colleagues and for taking over my work duties during the finalization process of this thesis.

Finally and most importantly, I would like to express my utmost gratitude to my loving girlfriend Veronika for always taking care of me in times of need and for supporting me all the way. If it wasn't for her, this thesis would probably never have been completed.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In Prague date 12.4.2012 signature

Název práce: Integrace IntelliSense pro Coq - nástroje na dokazování vět

Autor: Adam Abonyi

Katedra: KSI

Vedoucí diplomové práce: Mgr. Tomáš Petříček

Abstrakt:

Nástroje pro automatické (interaktivní) počítačové dokazování vět se používají na vytváření a ověřování formálních důkazů. Během posledního desetiletí se tyto nástroje postupně vyvíjely a začaly být čím dál častěji používány. Vývojová prostředí používaná pro tvorbu těchto důkazů však bohužel ve vývoji zaostávají a proto neposkytují takové možnosti, jako jiná modernější prostředí, se kterými se v dnešní době můžeme běžně setkat.

Cílem této práce je zlepšit vývojové nástroje pro dokazování vět a to konkrétně nástroj Coq. Ten integrujeme do moderního vývojového prostředí a tím získáme výhody novodobých nástrojů i při tvorbě důkazů. Takovéto vylepšení by mohlo zvýšit povědomí o počítačově asistovaném dokazování a tudíž vést k usnadnění formální verifikace a ke zvýšení kvality jak hardwaru, tak softwaru.

Klíčová slova: Coq, Teorém, Důkaz, Integrace, IntelliSense

Title: IntelliSense Integration for Coq theorem prover

Author: Adam Abonyi

Department: KSI

Supervisor of the master thesis: Mgr. Tomáš Petříček

Abstract:

Computer-assisted (interactive) theorem provers are software tools that help with the development of formal proofs. While theorem provers are constantly evolving and have started to be used more often over the last decade, the development tools used for creating computer-assisted proofs have not improved as much and they do not offer features that are common to other modern development tools.

This thesis aims to improve the state of the art of development tools for interactive theorem provers, namely Coq. We integrate the Coq interactive theorem prover into an existing modern development environment and bring the benefits of modern tools to interactive proving. Improving the development tools could help popularize computer-assisted theorem proving and thus improve the quality of software by simplifying formal verification and by allowing development of verified hardware.

Keywords: Coq, Theorem, Proof, Integration, IntelliSense

Table of Contents

TABLE OF CONTENTS 1

1. INTRODUCTION 3

1.1. INTERACTIVE THEOREM PROVING 3
1.2. TOOL SUPPORT AND COQ 4
1.3. WORK STRUCTURE 5

2. BACKGROUND 6

2.1. COQ AND THE GALLINA LANGUAGE 6
2.2. THE LANGUAGE OF COQ 6
2.2.1. PROVING IN COQ 7
2.3. IDE INTEGRATION 9
2.3.1. SOURCE CODE EDITOR 10
2.3.2. COMPILER AND INTERPRETER 10
2.3.3. BUILD AUTOMATION 10
2.3.4. DEBUGGER 11
2.3.5. IDE IN CONTEXT OF THEOREM PROVING 11
2.4. MONODEVELOP 12
2.5. IMPLEMENTATION TECHNOLOGIES 13
2.5.1. C# LANGUAGE 14
2.5.2. F# LANGUAGE 14

3. RELATED WORK AND GOALS 17

3.1. COQ IDE AND PROOF GENERAL 17
3.1.1. COQ IDE 17
3.1.2. PROOF GENERAL 18
3.1.3. COMPARISON 19
3.2. MICROSOFT VISUAL STUDIO 19
3.2.1. VISUAL C# 19
3.2.2. VISUAL F# 20
3.2.3. SUMMARY 21
3.3. JEDIT IDE FOR ISABELLE 21
3.4. GOALS 22

4. ANALYSIS 24

4.1. LANGUAGE SPECIFICATION 24
4.1.1. ASSUMPTIONS 25

4.1.2. ASSERTIONS AND PROOFS 25
4.2. ABSTRACT SYNTAX TREE 25
4.2.1. OBJECT ORIENTED REPRESENTATION 25
4.2.2. F# REPRESENTATION OF THE AST 27
4.2.3. SUMMARY 27
4.3. PARSING OF GALLINA LANGUAGE 27
4.3.1. FSLEX AND FSYACC 28
4.3.2. LEXICAL ANALYSIS 28
4.3.3. SYNTACTIC ANALYSIS 28
4.3.4. OTHER ALTERNATIVES 29
4.4. BACKGROUND PROOF CHECKING 30
4.4.1. STEPWISE CHECKING 31
4.5. IDE SELECTION 31
4.5.1. PROJECT AND FILE TEMPLATES 32
4.5.2. SYNTAX HIGHLIGHTING 32

5. IMPLEMENTATION 34

5.1. LEXER AND PARSER 34
5.1.1. LEXER 34
5.1.2. PARSER 36
5.2. BACKGROUND AGENTS AND PROOF CHECKING 39
5.2.1. AGENT IMPLEMENTATION 40
5.2.2. SIMPLE PARSER 42
5.3. IDE INTEGRATION 43
5.3.1. CODE HIGHLIGHTING 43
5.3.2. PARSER 45
5.3.3. SCOPES 45
5.3.4. IDENTIFIER RESOLUTION 46
5.3.5. INTELLISENSE 47
5.3.6. TOOLTIPS 48
5.3.7. PADS/PANELS 48

6. SUMMARY 51

6.1. SAMPLE 51
6.2. COMPARISON 54
6.3. EVALUATION AND FUTURE WORK 55
6.3.1. FUTURE WORK 55
6.4. CONCLUSIONS 56

7. REFERENCES 57

1. INTRODUCTION

According to [1], a proof is the process or an instance of establishing the validity of a statement especially by derivation from other statements in accordance with principles of reasoning. In mathematics, a proof is absolute and its correctness can be in principle determined by anyone by following primitive mathematical steps.

To avoid human mistakes proof assistants[1] are being used more and more in computer aided verification of software and hardware. Design of critical pieces of code and/or critical pieces of hardware can highly benefit from this verification, because writing handmade proofs of C-compilers or Pentium processors is close to impossible. This approach can not only save time but a lot of money too. As mentioned in [2], Intel's estimated loss of the FDIV bug (a bug in the floating point unit in the Pentium processor) was around $475 million. Another example is the explosion of the space rocket Ariane 5 where an overflow error occurred in the conversion from 64-bit floating point number to a 16-bit integer [2].

The use of tool support for the design, simulation and verification of systems is paramount. When proof assistants get easier to use and contain more basic knowledge, their usage will get more widespread.

Just like modern software development tools simplify writing of programs, we want to simplify construction of proofs by offering some of the benefits of classical integrated development environments.

1.1. Interactive Theorem Proving

In the year 2012 the third conference on Interactive Theorem Proving and related issues will be held. This and many other indicators show an increased interest in this field.

The Four Color Theorem[2] is famous for being the first long-standing mathematical problem to be resolved using a computer program. The theorem was first postulated by Francis Guthrie. Many famous mathematicians worked on proving this theorem and many incorrect proofs were stated. In 1976 it was proven by Appel and Haken [3]. They broke down the problem into many smaller cases that the computer (IBM 370) solved individually. In that time this kind of proof was met with skepticism as computer programming is known to be error prone.

According to [4] in 1995, Robertson, Sanders, Seymour, and Thomas helped clear up most of the doubts of the Appel and Haken approach. They used a program written in C to analyze all individual cases.

---

[1] Proof assistant is a software tool that aids with the development of formal proofs.

[2] The Four Color Theorem: The regions of any simple planar map can be colored with only four colors, in such a way that any two adjacent regions have different colors.

In [4] the so called “Ultimate step” has been made in clarification of these efforts. Formal proof script that covers both the mathematical and computational parts of the proof was written in Coq proof checking system [5], which mechanically verified its correctness in all respects.

Another example of the recent developments in interactive theorem proving is the project called CompCert [6]. Its goal is to investigate the development and correctness proof of verified compilers and to improve the tools needed for such proofs. The result of this work is a formally verified compiler for a large subset of the C language, called Clight. The Coq proof assistant was used to conduct the correctness proof. It was also used to program most of the compiler.

In [7] the author is arguing that the cost of formal program verification outweigh the benefits and that the technology of program verification is mature enough today so that it makes sense to us it in a support role in computer science.

If this field is becoming more mainstream and will continue to increase in popularity it is crucial that the development tools are adequately advanced. Efficient tools can save a lot of time and can help make this field available to more people.

1.2. **Tool Support and Coq**

Development of tool support was always hand in hand with the development of any programming language. It simplifies and increases programming efficiency as it shifts many requirements from the programmer to the development tools.

In this thesis, we focus at developing tool support for Coq. Coq is a Proof Assistant for a logical framework known as the Calculus of Inductive Constructions. It allows the interactive constructions of formal proofs, and also the manipulation of functional programs consistently with their specifications. It runs as a computer program on various architectures [5].

The standard distribution of Coq includes Coq Integrated Development Environment (Coq Ide) a graphical tool included with the installation of Coq. As described by the authors it is a user-friendly replacement to command-line user input. Its main purpose is to allow the user to navigate forward and backwards into a Coq vernacular[1] file, executing corresponding commands or undoing them respectively [5]. Another popular tool used for various proof assistants[2] is *Proof General* [8]. It is based on a generic Emacs[3] interface for interactive proof assistants, developed at the LFCS in the University of Edinburgh.

---

[1] Vernacular is the language of commands of Gallina i.e. definitions, lemmas, theorems, etc.

[2] A proof assistant is a computerized helper for developing mathematical proofs. For short, we sometimes call it a prover, although we always have in mind an interactive system rather than a fully automated theorem prover.

[3] Emacs is an extensible, customizable, self-documenting, real-time display editor.

When compared with modern IDEs used by software engineers such as Microsoft Visual Studio [9] or Eclipse [10], the Coq Ide and Proof General are relatively simple.

Modern software development IDEs consist of several components all of which can be in some way useful. Coq Ide and Proof General include some of these but usually in a very basic manner. Some features that are not present are uninterrupted code editing, code autocompletion (also called IntelliSense by Microsoft), background code verification, inline tooltips, etc. Bringing these features to the Proof Assistant IDEs can improve development time and comfort.

1.3. Work Structure

So far we have discussed the main areas that will be targeted in this thesis. The rest of the thesis is organized as follows.

In **Chapter 2** we will introduce Coq and the Gallina language. We will also learn about the main ideas of Integrated Development Environments (autocompletion, refactoring, etc.). The chapter will be concluded with specific information on the frameworks and programming languages used in this work.

In **Chapter 3** we will present works related to the main areas in this thesis. First off we will talk about Coq Ide and Proof General. This will be followed by the IDE details of higher-level languages C# and F#. Subsequently we will talk about Isabelle, a proof language alternative to Coq. This chapter will conclude by defining the goals of this thesis.

The analysis of the problem will be discussed in **Chapter 4**. We will consider the specifics of all the problems that we are going to encounter: The representation of the Abstract Syntax Tree (Section 4.2) and the Background Proof Checking (Section 4.4) mechanisms and Parsing (Section 4.3). Eventually we will talk about the choice of IDE that will be extending (Section 4.5).

Implementation details of the work can be found in **Chapter 5**. This mainly includes the Parser and the Lexer for the language (Section 5.1), representation of scopes (Section 5.3.3), background agents (Section 5.2) and IDE integration (Section 5.3).

Examples of the final result will be shown in **Chapter 6**. The solution provided will be compared with Coq Ide and Proof General. We will assess the goals we have set in **Chapter 3** and future work on the project will be discussed. The benefits of the work can be found in the Conclusion that will summarize this thesis.

2. BACKGROUND

To completely understand the problem that this thesis attempts to solve we must start with understanding the language we want to support. Coq Proof Assistant uses the Gallina language for proof definitions. In this chapter we are going to provide a brief introduction to the Gallina language since we will be referencing to it often in the later chapters. Furthermore we will talk about IDEs as they are used in software engineering. Languages and frameworks used in the implementation will be also mentioned.

2.1. Coq and the Gallina Language

Coq implements a program specification and mathematical higher-level language called Gallina that is based on an expressive formal language called the Calculus of Inductive Constructions [11].

With its commands Coq allows:

- Definition of functions or predicates
- Stating mathematical theorems and software specifications
- Interactive development of formal proofs
- Machine-checking of proofs

As a proof development system, Coq provides interactive proof methods, decision and semi-decision algorithms, and a tactic language for letting the user define its own proof methods.

As a platform for the formalization of mathematics or the development of programs, Coq provides support for high-level notations, implicit contents and various other useful kinds of macros.

We start by introducing the *Gallina language,* which is the specification language of Coq. It is used to prove mathematical theories and program specifications. The proofs are built using formal notions of mathematical constructs, such as axioms, hypotheses, parameters, lemmas, theorems and definitions of constants, functions, predicates and sets.

2.2. The language of Coq

Coq objects are divided into two categories: the Prop sort and the Type sort. Prop is the sort for propositions, i.e. well-formed propositions are of sort Prop. The following listing shows two examples of propositions:

```
∀ A B : Prop, A /\ B -> B \/ B
∀ x y : Z, x * y = 0 -> x = 0 \/ y = 0
```

New predicates can be defined inductively:

```
Inductive even : N -> Prop :=
  | even_0 : even 0
  | even_S n : odd n -> even (n + 1)
  with odd : N -> Prop :=
  | odd_S n : even n -> odd (n + 1).
```

In this example the constructors *even_0*, *even_S* are the defining clause of the *even* predicate. Similarly for the odd predicate we have the constructor *odd_S*. These constructors can be used later on in the proofs.

Predicates can be also defined by abstracting over other existing propositions:

```
Definition divide (x y:N) := ∃ z, x * z = y.
Definition prime x := ∀ y, divide y x -> y = 1 \/ y = x.
```

Type is the sort for data types and mathematical structures. Well-formed types or structures are of sort Type. Here is a basic example of a type representing a function that takes an integer and returns a tuple of two integers:

```
Z -> Z * Z
```

Corresponding to the induction principle in maths, inductive structures can be used for proof by induction:

```
Inductive nat : Set :=
  | 0 : nat
  | S : nat -> nat.

Inductive list (A:Type) : Type :=
  | nil : list A
  | cons : A -> list A -> list A.
```

2.2.1. PROVING IN COQ

Proof development in Coq is done through a language of tactics that allows a user-guided proof process. In this section, we demonstrate the use of the command line interactive mode.

The following section is inspired by the Coq tutorial [12], but we focus on features that are interesting from the perspective of IDE. First we are going to consider atomic propositions A, B and C. This can be achieved by introducing them as variables of the type Prop (propositions).

```
Variables A B C : Prop.
```

Following is the output of Coq. These outputs will be displayed in with a darker background than the input code samples.

```
A is assumed
B is assumed
C is assumed
```

Now we can use these variables to create a lemma. We will take one of the axioms from the propositional logic.

```
Lemma distr_impl : (A -> B -> C) -> (A -> B) -> A -> C.
```

```
1 subgoal

  A : Prop
  B : Prop
  C : Prop
  ============================
   (A -> B -> C) -> (A -> B) -> A -> C
```

The current goal is displayed below the double line. To achieve this we combine local hypotheses (they will be introduced in the next step) with the goal.

Now that the lemma has been defined Coq is expecting a formal proof of it (we have entered the proof mode. This is done using commands and tactics. We can omit the names of local hypotheses created by the introduction tactic. We can define their names one by one or let them be generated automatically.

```
intros.
```

```
1 subgoal

  A : Prop
  B : Prop
  C : Prop
  H : A -> B -> C
  H0 : A -> B
  H1 : A
  ============================
   C
```

In the output we can notice that hypotheses H, H0 and H1 have been introduced and the goal has changed. The goal C can be obtained from hypothesis H.

```
apply H.

2 subgoals

  A : Prop
  B : Prop
  C : Prop
  H : A -> B -> C
  H0 : A -> B
  H1 : A
  ============================
   A

subgoal 2 is:
 B
```

A new goal has been introduced. Notice that this goal is not fully listed with its list of local hypotheses and variables. To solve the current goal we just have to use the available hypothesis H1. Now we can continue with the second subgoal where only B must be proven. This can be achieved by applying hypothesis H0 and concluded with H1 again.

```
exact H1.      (* Finalizes subgoal 1 *)
apply H0.      (* Proof of subgoal 2 *)
exact H1.      (* Finalizes subgoal 2 *)

Proof completed.
```

Now the proof is complete.

Error Messages

In this demonstration we assume the same lemma used in the previous example. This time we will apply a wrong hypothesis and observe the error message.

```
apply H0.

Toplevel input, characters 6-8:
> apply H0.
>       ^^
Error: Impossible to unify "B" with "C".
```

The error message contains all the information necessary to identify the mistake. The exact position and the message with details explaining the problem.

2.3. IDE Integration

The Integrated Development Environment is a software application that provides comprehensive facilities to computer programmers for software development [13].

In an IDE these features are usually present:

- a source code editor

- a compiler and/or an interpreter
- build automation tools
- a debugger

IDEs increase productivity of the programmer by providing components with similar interfaces that can easily take advantage of each other. This reduces the time needed to learn new software conventions and also eliminates the necessity to use various development programs.

IDEs can be divided into two categories. Single purpose IDEs, which are not that common nowadays, like TurboPascal, Delphi and Borland C++ Builder were used by one programming language and had features that were common for that respective language. On the other hand Multi-purpose IDEs like Eclipse, Netbeans, MonoDevelop, Visual Studio try to be feature rich so that it is possible to simply add a new language trough a system of extensions and add-ins.

2.3.1. SOURCE CODE EDITOR

Source code editors have features specifically designed to simplify and speed up input of source code, such as *syntax highlighting*, *autocomplete* and *bracket matching* functionality. So, while many text editors can be used to edit source code, if they don't enhance, automate or ease the editing of code, they are not source code editors, but simply text editors that can also be used to edit source code.

Some of the modern source code editors check syntax while the user is typing, immediately warning of syntax errors. Some also have the possibility to work with the source code. Providing features like automatic alignment, removal of extra whitespaces.

2.3.2. COMPILER AND INTERPRETER

Some IDEs contain the compiler or an interpreter for a certain language directly with them. Others use compilers already installed in the system. The compiler and interpreter are used to run the program but they can be used for the background checking.

The source code from the IDE is used as the input for the compiler. The compiler then uses this input to create output data. In the case of Coq each command can be processed by the compiler and has its own output, which can be displayed in output windows or directly in the source code editor (underline errors). It can also create a representation of the source code that can be used to extend functionality of the IDE.

2.3.3. BUILD AUTOMATION

Build automation is the act of scripting or automating a wide variety of tasks that software developers do every day:

1. compiling computer source code into binary code

2. packaging binary code
3. running tests
4. deployment to production systems
5. creating documentation and/or release notes
6. source control

As mentioned in the previous chapter, compiler serves for translating source code from the programming language to a language that can be executed or interpreted. Usually this is implemented by assigning a key command that gives the order to the IDE to send the code to the compiler, however sometimes it is a good idea to do this in the background because informing about errors right away can be quite useful.

Similarly all other features work with the source code in some way and build this support directly into the menus of the IDE, so that all the options are easily accessible and do not rely on running other programs or on the programmers extra effort.

2.3.4. DEBUGGER

A debugger is a computer program that is used to test and debug other programs. This technique adds great power in the ability to halt when specific conditions are encountered. A program running like this will typically be slower than executing the code directly.

The debugger is integrated into the IDE in a variety of ways mainly because finding and removing errors can be a very lengthy process. The IDE must provide all the information that it can at a specific time. To achieve this IDEs support breakpoints to halt the execution of the application and then the IDE usually displays many panes, where values of all the variables are displayed, callstack for the breakpoint is listed and other helpful features are provided.

2.3.5. IDE IN CONTEXT OF THEOREM PROVING

The previous section summarized IDE features available in standard IDEs used by software engineers. Now, we consider which of these features might be usable in the interactive theorem proving context. In related work (Section 3.1), we discuss the features supported by existing IDE tools for theorem provers.

From what has been said about IDEs it can be seen that there is a large variety of features that can be used. Most of the mentioned source code editor features (Section 2.3.1) are useful in the theorem proving context. Most importantly, these include syntax highlighting and autocompletion. Understanding of the code can help us suggest names of defined

Types, Lemmas or other constructs. In the same manner refactoring[1] can be used too to simply alter code.

Close integration with the interpreter (Section 2.3.2) is the key for streamlining the process of building proofs. Providing some form of hints and output messages instantly and seamlessly without the need to do anything extra can help the developer focus more on the proof then on the interaction with the IDE.

Debugger and build automation (running tests, deployment to production) are mainly useful in the programming language context and do not have direct equivalents in the theorem proving context.

2.4. MonoDevelop

MonoDevelop is the IDE that we eventually chose to extend. It is an essential component for this work, because it can demonstrate the benefits that the integration of a theorem prover into a modern IDE can bring. The reason why MonoDevelop was chosen over Microsoft Visual Studio is discussed later in Section 4.5.

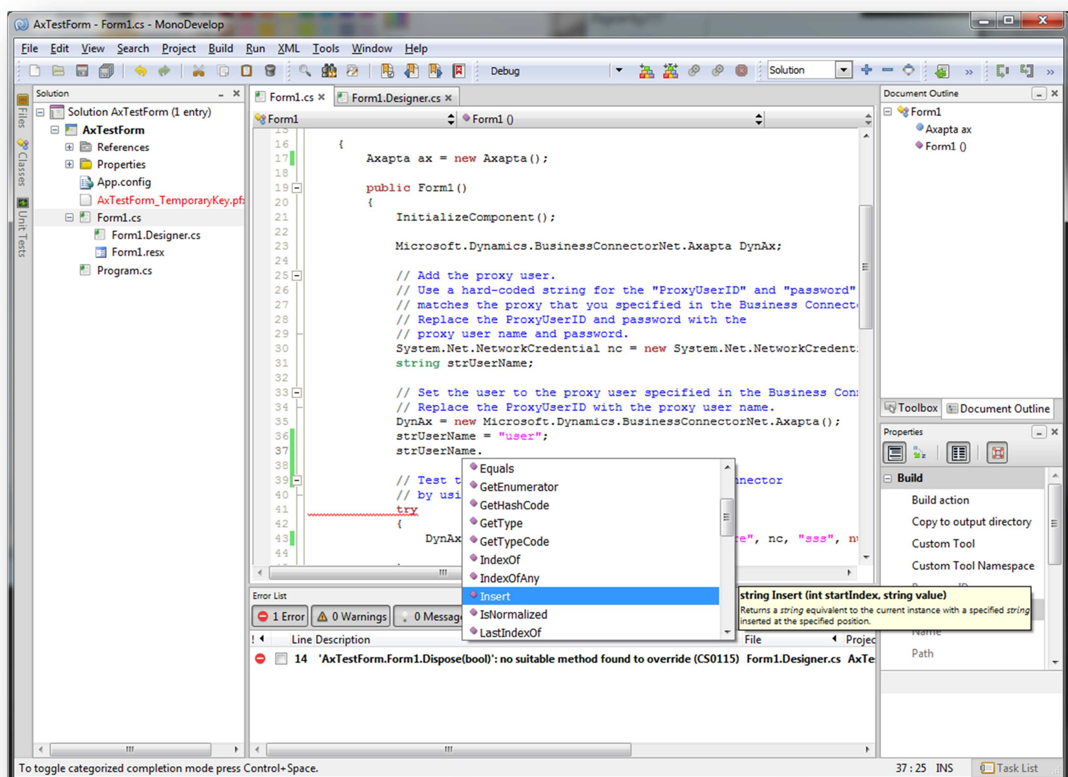


Figure 1 - MonoDevelop IDE

[1] Code refactoring is a technique for restructuring an existing body of code, altering its internal structure without changing its external behavior.

MonoDevelop is an IDE primarily designed for C# and other .NET languages. It enables developers to quickly write desktop and ASP.NET Web applications on Linux, Windows and Mac OSX. MonoDevelop makes it easy for developers to port .NET applications created with Visual Studio to Linux and to maintain a single code base for all platforms.

In Figure 1 we can see some features of MonoDevelop. A C# project is loaded, but many different languages are supported as well. There also are customizable panes; on the left there is the project pane, in the bottom the error pane and on the left the document outlines and properties pane. All of them can be moved to different locations, closed or reenabled. There are of course many other panes that are not shown in this example. We can also notice the line underlined with a red wave, where an error has occurred. Its description can be found in the bottom pane or in a tooltip displayed when the mouse hovers over the line. The code is appropriately highlighted and while typing the environment provides hints and autocompletion.

MonoDevelop contains features that are not evident from the picture like an integrated debugger, visual designer for GTK#, ASP.Net support, source control, unit testing and many others. These features are not relevant to this thesis and won't be described here but more information can be found on the MonoDevelop page [14].

Extending MonoDevelop

The functionality of MonoDevelop can be extended and it can be done by add-ins. They consist of two basic parts: the xml definition and the code running the add-in functionality. The xml file contains some basic meta-information such as the author, copyright. It also tells MonoDevelop what parts you want to extend.

The add-in architecture of MonoDevelop is designed to extend any part of the IDE. To achieve this, the concept of an extension tree is used. Each extension (a set of extensions is an add-in) can be plugged into an extension point defined in other add-ins. MonoDeveop is built this way, so many extensions are available. More information about extensions point can be found in the extension tree reference manual [15]. The details of the implementation of our plugin will be discussed further in Section 5.3.

2.5. Implementation Technologies

The *.NET Framework* is a software framework that runs on Microsoft Windows. It includes a large library and provides language interoperability[1] across several programming languages. Programs written for the .NET Framework execute in a software environment, known as the Common Language Runtime (CLR), an application virtual machine. The class library and the CLR together constitute the .NET Framework.

[1] Each .NET language can use code written in other .NET languages.

The .NET Framework's Base Class Library provides user interface, database connectivity, web application development, network communications and others. Programmers produce software by combining their own source code with the .NET Framework and other libraries.

Alternatively Mono is a software platform designed to allow developers to easily create cross platform applications. It is an open source implementation of Microsoft's .Net Framework. The purpose of Mono is not only to be able to run Microsoft .NET applications cross-platform, but also to bring better development tools to Linux developers. F# is among many other languages supported by Mono.

.NET Languages

C# is the main programming languages designed for the Common Language Infrastructure. There are of course other languages like F#, C++/CLI, IronPython, IronRuby. While being completely different, they have some common attributes. They all have access to the Base Class Library and they all get compiled into the CIL object code, which is then Just In Time compiled (JITed) into machine code.

2.5.1. C# LANGUAGE

C# is a multi-paradigm programming language encompassing strong typing, imperative, declarative, functional, generic, object-oriented (class-based), and component-oriented programming disciplines. It was developed by Microsoft within its .NET initiative and later approved as a standard by Ecma (ECMA-334) and ISO (ISO/IEC 23270:2006).

MonoDevelop can be extended by any .NET language. For the purpose of this thesis C# will be used for implementing add-ins, not only because MonoDevelop is programmed in C# but also because all other add-in implementations use C# so it will be easier to incorporate them.

2.5.2. F# LANGUAGE

F# is a multi-paradigm programming language, targeting the .NET Framework, that encompasses functional programming as well as imperative and object-oriented programming disciplines. It is a variant of ML and is largely compatible with the OCaml implementation. F# was developed by Don Syme at Microsoft Research, but is now fully supported part of Visual Studio and there is also an open-source release for Mono.

Benefits of using F# are similar to benefits of other functional languages over imperative languages. Formulating problems is much closer to their definition and is less error-prone (immutability, more powerful type system, intuitive recursive algorithms). Additional advantages of F# are easier asynchronous programming, easy integration of compiler compilers and domain-specific languages, units of measure, flexible syntax. Some notable features of F# that will be used in this work are discussed next.

Discriminated Unions

With F# we have the ability to use constructs that are not common in imperative languages, called d*iscriminated unions*. The benefits of (recursive) discriminated unions is that they can be very useful for representing tree data structures with heterogeneous nodes, such as the AST or other tree-like structures.

```
type BinaryTree =
    | Tip
    | Node of int * Tree * Tree
```

This discriminated union represents a binary tree with an integer in each node that is not the tip. Match can be used to add operations that can traverse trough this tree.

```
let rec sumTree tree =
    match tree with
    | Tip -> 0
    | Node(value, leftTree, rightTree) ->
        value + sumTree(leftTree) + sumTree(rightTree)
```

The pattern matching determines the type of the current node and returns the corresponding data for the node.

Abstract Syntax Tree

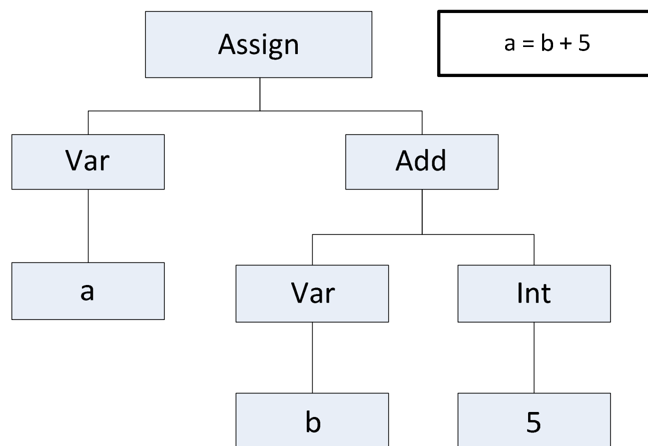


Figure 2 - A simple AST representation of the expression a = b + 5

In computer science, an abstract syntax tree (AST), or just syntax tree, is a tree representation of the abstract syntactic structure of source code written in a programming language [16].

Each node of the tree denotes a construct occurring in the source code. The syntax is abstract in the sense that it does not represent every detail that appears in the real syntax. For instance, grouping parentheses are implicit in the tree structure, and a syntactic construct such as an if-then-else-condition may be denoted by a single node with two branches.

Lexing and Parsing in F#

Fslex and Fsyacc are F# implementations of Lex and Yacc known from OCaml. They are a part of the F# PowerPack release for Visual Studio 2010. Together they take an input string and create a parse tree of typed objects. This parsed tree can be used for further processing and understanding the code.

Fslex converts the input string into a series of tokens. A token is a string label for the parser. Simply put the parser decides based on a series of rules (regular expressions), what does each part of the input string represent. For example 123 is a Number and text in-between apostrophes like “Hello” is a String.

The parser imposes syntactical rules onto the tokens generated by the lexer. The parser uses grammar rules that define the syntax to construct an AST.

F# Agents

Another advantage of F# is that it supports many features for asynchronous programming. This is not as common in other (especially imperative) languages. Asynchronous workflows [17] support performing computations asynchronously without blocking execution. F# agents are based on asynchronous workflows but they also offer:

- An easy way of implementing asynchronous code.
- They can be used for background tasks as they do not block the main thread.
- Many agents can be created without the need to create a thread for each one.

More information about the implementation of F# Agents can be found in Section 5.2 and in [18]. For our purposes it is not important to create many background agents. We incorporate the use of the background task and the fact, that these agents can be called very easily from our code, without the need for any locks.

3. RELATED WORK AND GOALS

This chapter introduces the current most popular tools available for Coq. Next we will talk about the features present in Visual Studio for imperative and functional languages. Also a similar project for a different proof language will be discussed. Finally we will state our goals.

3.1. Coq Ide and Proof General

Coq Ide and Proof General are the two most commonly used editors for Coq. First we discuss their behavior and features separately and then compare them together.

3.1.1. COQ IDE

The Coq Integrated Development Environment is a development tool with better, visual feedback compared to the command-line tool. Its purpose is to improve development experience. It allows the user to navigate forward and backward through a Coq proof script[1]. This means executing corresponding commands. The commands that were accepted cannot be changed unless the user returns to a point prior to them.

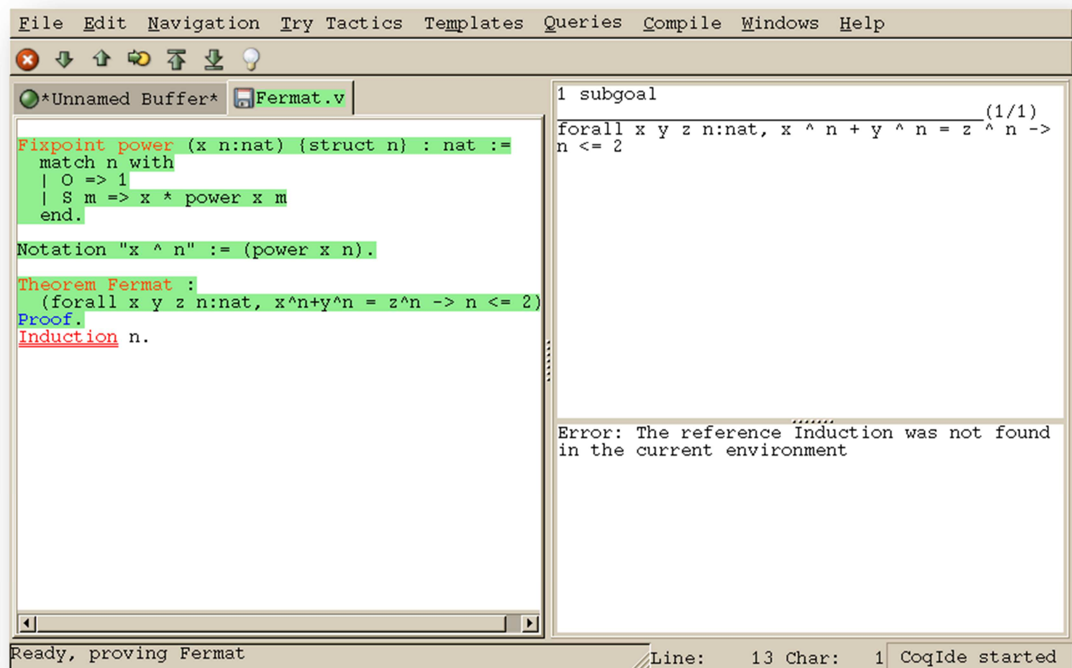


Figure 3 - Coq Ide

[1] A proof script is a sequence of commands which constructs a proof, usually stored in a file.

The Graphical User Interface (GUI) of Coq Ide show in Figure 3 consists of these main parts:

1. Script window – you may open numerous files here and send inputs to the Coq interpreter. Lines with a green highlighted color were already successfully processed. Red underline means the current text produces an error.
2. Proof window – all subgoals that have to be proven for the current proof are displayed here.
3. Output window – the output of the coq interpreter for the last input line is displayed here

3.1.2. PROOF GENERAL

Proof General [8] is a generic Emacs interface for interactive proof assistants, developed at the LFCS in the University of Edinburgh.

The interface is designed to be easy to use especially to people familiar with Emacs. The proof script is developed similarly to Coq Ide. Proof General also keeps track of which proof steps have been processed by the prover, to prevent accidental changes. In Figure 4 the processed lines are highlighted with the blue color

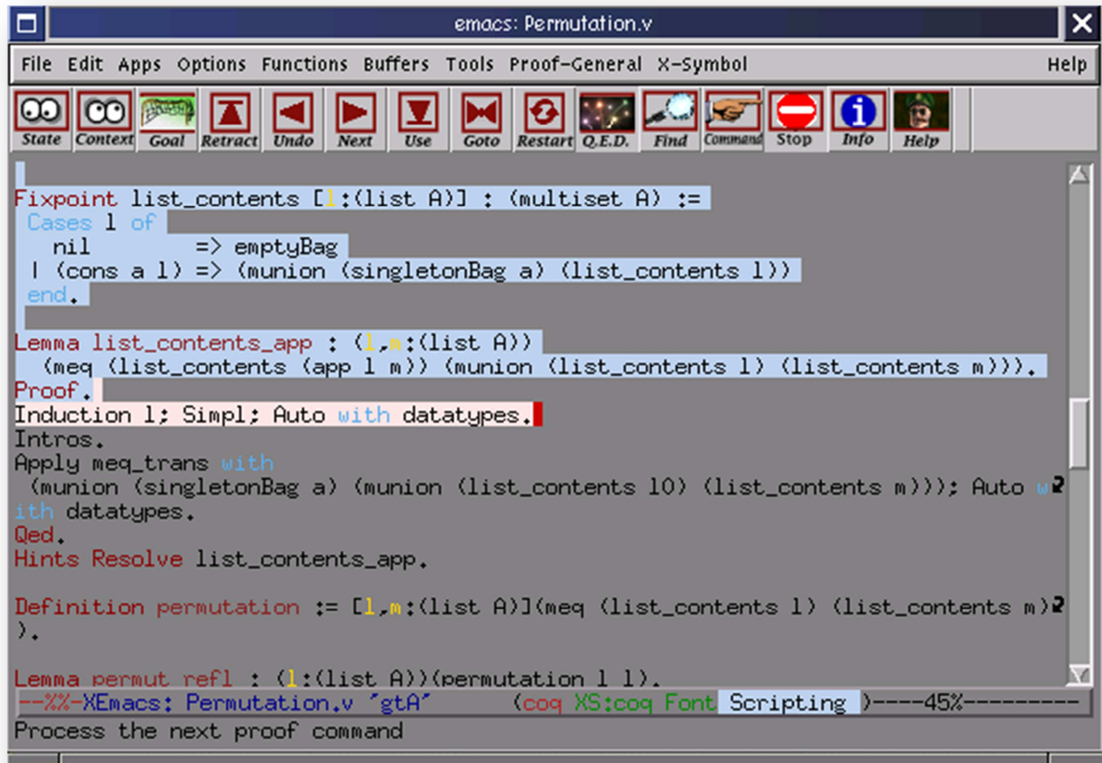


Figure 4 - Coq in Proof general module in Emacs

The aim of Proof General is to provide a powerful and configurable interface for numerous interactive proof assistants.

3.1.3. COMPARISON

Coq Ide and Proof General share similar features. Both use simple syntax highlighting. They also offer a simple set of commands and tactics that can be picked from the menu but do not offer autocomplete feature. Feedback from the interpreter is displayed in a separate window, but it always refers only to the last processed command.

The biggest difference between these IDEs is that Coq Ide is a single purpose IDE with a lot of Coq specific functions, while Emacs is a text (source code) editor that can be simply customized and extended with support for any language. The familiarity with Emacs can be a big factor in considering the development environment for many developers.

3.2. Microsoft Visual Studio

Although Visual Studio does not provide support for any theorem provers, it is worth considering its features as it is one of the most advanced IDEs used by professional developers and it may provide useful hints for developing IDE for interactive theorem proving. This can help people to feel more comfortable with theorem proving.

3.2.1. VISUAL C#

The C# support in Visual Studio C# is a typical example of how to implement support for an imperative language. It contains most of the features described above.

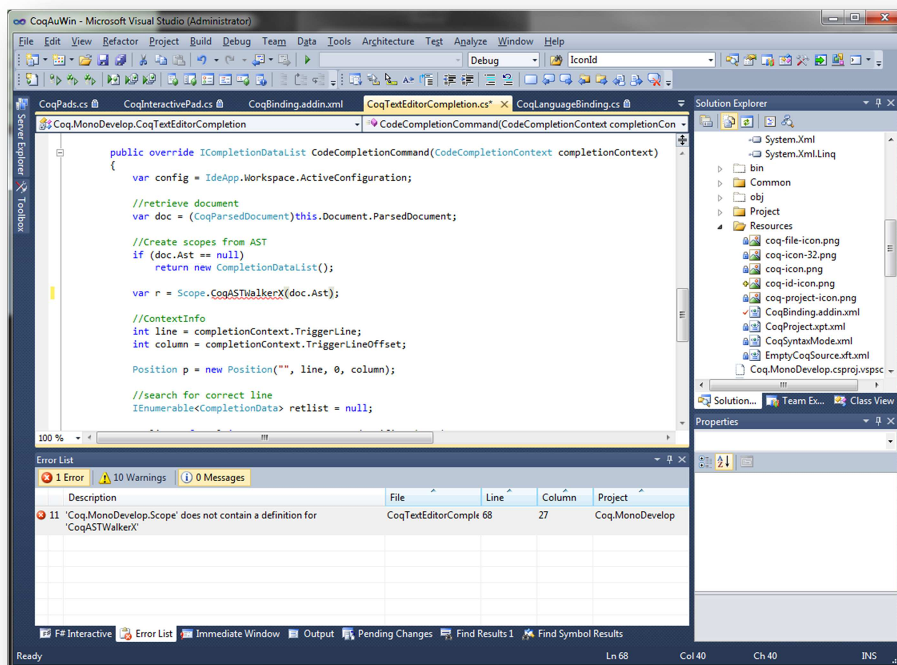


Figure 5 - Visual Studio 2010 with an opened C# project

In Figure 5 we can see an opened solution. To the right we can see the Solution Explorer window, where one can easily switch between projects and files and doesn't need to open other source code files elsewhere. The small icons on the left of the file names indicate the state of the file against the source control. The lock icon means that the file is the unchanged from the last synchronization with the server; the check icon indicates that a change was made in the IDE.

The syntax highlighted source code is written in C#. The IDE checks classes and their members on the background so if some method name is typed incorrectly or incompatible types are used the IDE immediately informs the user by underlining the corresponding parts of the source code. An error message with more information is displayed in the bottom pane or in a tooltip over the highlighted error.

3.2.2. VISUAL F#

Since F# is a functional language some IDE features differ from the ones that are present in the C# editor. This is mainly because F# has type inference[1] and supports interactive scripting (F# interactive panel in Figure 6).

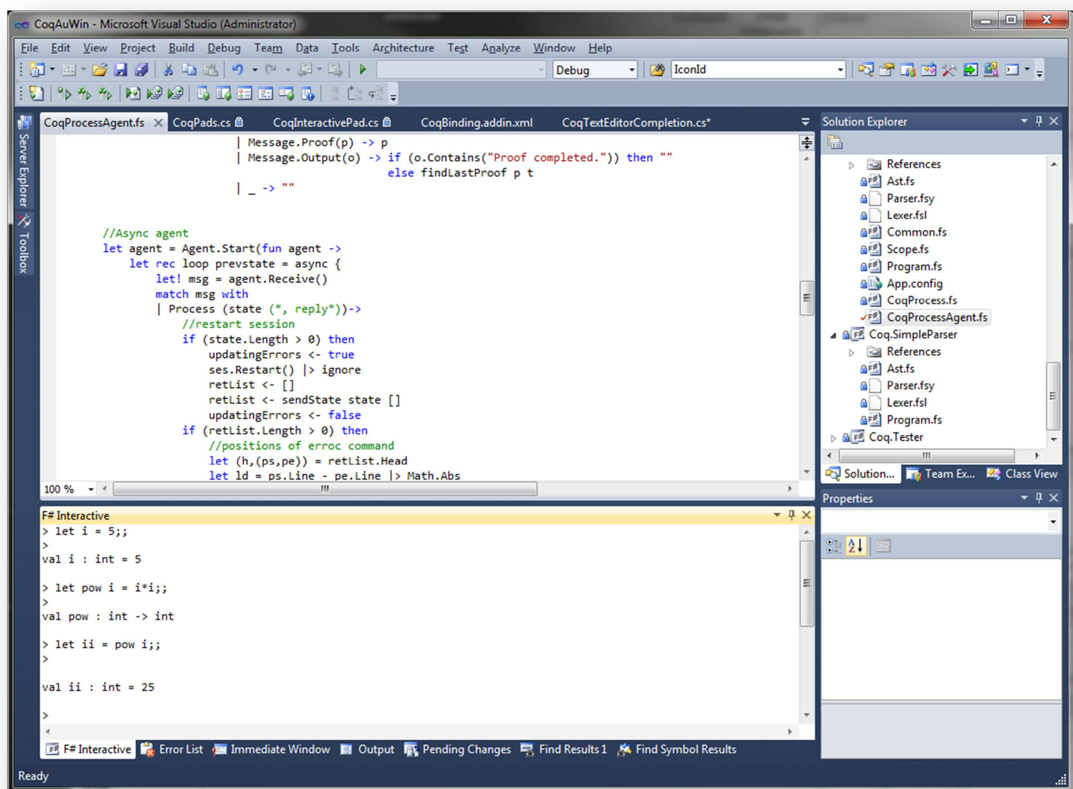


Figure 6 - F# file opened in Visual Studio 2010

---

[1] Type inference refers to the automatic deduction of the type of an expression.

In comparison to the C# IDE features, F# is more of a scripting language. For this purpose the F# Interactive window was introduced. One can execute selected parts of code by pressing the key commands Alt + Enter. The code is then executed in the Interactive pane and it is possible to immediately see the results of the processed code, which can be called directly from the same pane.

Another convenient feature is F# type checking. It is done seamlessly on the background so that the user experience is not interrupted. This process of determining the types in F# is quite complicated and is slightly comparable to simple proofs.

3.2.3. SUMMARY

The features provided by Visual Studio are mutual to most mainstream IDEs and languages but they are not as common in the world of proof assistants. Having freedom to move around in the source code and getting instant feedback is not standard and is a feature, we would like to introduce. This will hopefully make developing proofs more comfortable for people used to these standard features.

3.3. jEdit IDE for Isabelle

Isabelle is a generic proof assistant. Similarly to Coq Isabelle provides tools for proving various formulas. This goal of a project described in [19] is to introduce a new Isabelle/Scala layer that will take advantages of the asynchronous support of Isabelle and can be called using the Scala language from plugins for JVM-based IDEs. The Isabelle/jEdit implementation introduces an enhanced graphical editor of proof scripts.

The authors point out some of the shortcomings of editors like Proof General or other standard proof assistant IDEs. The main issue being that the provers process commands one after another while the results of these commands are displayed in a separate window. Once processed these commands can't be changed which separates the code into two parts, the *locked* and the *editable* region. This limits the development to a single line, where the output is displayed and the system is awaiting the new command.

The removal of this *single focus* model is the main goal that this work wants to eradicate.

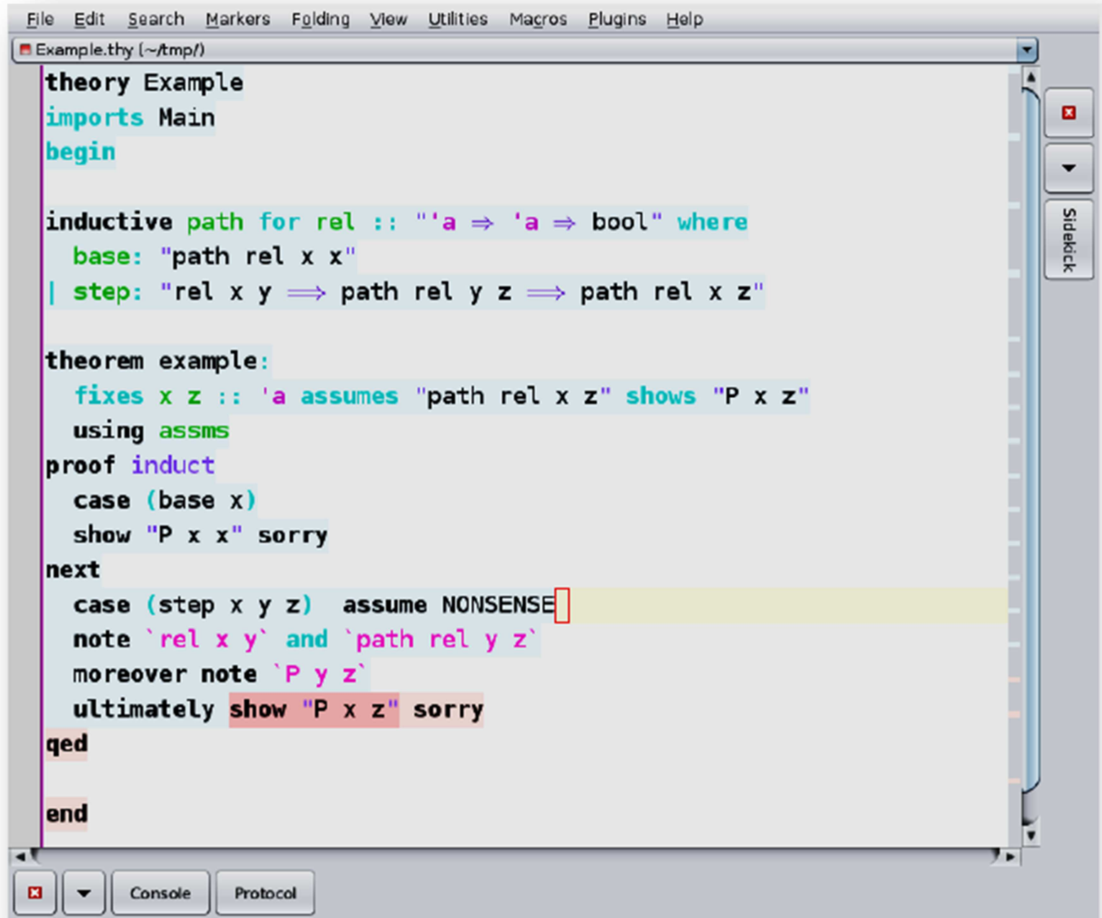


Figure 7 - Main editor window of Isabelle/jEdit, with semantic highlighting

As shown in Figure 7 the text can be typed, edited, changed as the user wants. There are no locked parts of the proof script like in Coq Ide or Proof General. The editor provides useful feedback thanks to the semantic information from the partially processed document. This feedback is displayed trough tooltips, coloring, etc.

3.4. Goals

In chapter 2 and 3 we have talked about IDEs used with imperative languages, functional languages and used for proof assistants. We discussed their features that they possess and that might be useful in our implementation. Some of them will be integrated. For some, we will lay the grounds so their implementation will not be as difficult as it would be with current proof assistant IDEs.

We want to implement an add-In for MonoDevelop (Sections 4.5, 5.3) so that the user can create edit and run Coq Proofs with the benefit of autocompletion, IntelliSense and an intelligent representation of the vernacular file (Coq source code). This requires the

implementation of a processor of the Gallina language that can create an abstract representation of the code and make this representation available for further use.

The goal of this thesis is to create the basics of a modern Coq Ide and simplify the process of creating smart editing tools for the Gallina language in the future. Practically with Coq and MonoDevelop we are hoping to achieve the same as the jEdit IDE with Isabelle we talked about in the previous chapter.

The main goals of this thesis are:

- **Advanced editing.** Existing modern IDEs give us the possibility to use advanced editing features. Similarly like the Emacs implementation gave the users the ability to use the commands they were used to. This was a significant improvement to the basic options of Coq Ide.
- **Background error checking.** Modern IDEs like Visual Studio integration for F# provide background error-checking that is non-intrusive and runs on a background thread while the user is typing code. To some extent, this goal is achieved by jEdit for Isabelle. We aim to provide similar functionality for the Coq proof assistant. The user should be able to edit any part of a file and see how changes affect the correctness of the proof.
- **IntelliSense.** The highlighting of the code allows the programmer to quickly orient in the code, while autocompletion provides the relevant identifiers without the need to remember every identifier correctly. This helps the productivity and simplifies the development process.
- **Simple installation.** By integrating with a mainstream IDE, we allow the developer to quickly start proving in an environment that they are used to, which can help to bring more people to this area of expertise.
- **Extensibility.** The AST of the code provides basis for advanced IDE extensions like refactoring. This is easily accessible to anyone who wishes to implement such an extension in the future.

4. ANALYSIS

In this chapter we will talk about the specifics of the Gallina language. The representation of this language using AST will be discussed and the analysis of what is needed to implement a parser of this language. Finally, we discuss the choice of base IDE for our plugin.

4.1. Language Specification

The Lexical conventions of the language are quite similar to other functional languages. Blanks (also called Whitespace) are typically spaces, newlines, tabs. Comments are enclosed in (* and *) and are treated as blanks too. The most important for the IDE implementation are probably the *Identifiers*, which will be the important for Intellisense. The I*dentifier* is a sequence of letters, digits, underscores or apostrophes that does not start with a digit or apostrophe.

```
x                    (* Identifier *)
a.b.c                (* Qualified identifier *)
```

The language contains keywords and reserved keywords, that can't be used in any other manner.

The basic part of the Gallina language is the Term. It serves as the basic logical expression. Terms (also called expressions) are used to describe mathematical objects (numbers, etc.), logical values and other objects. There are over 19 rules in the language specification that define Term.

The next basic building block of the Gallina language is a sentence. Sentence is a command that instructs Coq. It typically consists of a keyword and terms that specify arguments of the command. A command can be either related to the proof or to the Coq compiler. Commands like *Check* or Search inform about the specified term. The example shows us the result of a command *Check 0*.

```
Check 0.
0
        : nat
```

The proof commands can by of three types:

- Assumptions – extends the environment with axioms, parameters, hypotheses or variables
- Definitions – extends the environment with associations of names to terms
- Inductive Definitions - inductive types, simple annotated inductive types, simple parametric inductive types, mutually inductive types

- Fixpoints – definitions of recursive functions
- Assertions and Proofs – states a proposition with a proof

The Listing 1 shows and example of a few sentences written in the Gallina language:

```
Definition one := (S O).            (* Definition *)
Variables A B C : Prop.             (* Assumption *)
Lemma distr_impl : (A -> B -> C) -> (A -> B) -> A -> C.
                                    (* Assertion *)
```

Listing 1 - Sentences in the Gallina language

We will not talk about the differences in the various assumptions and assertions. Information about them can be found in the specification of the Gallina language.

4.1.1. Assumptions

Assumptions bind and identifier to a type. An assumption can be an axiom, parameter, hypotheses or a variable.

4.1.2. Assertions and Proofs

Assertions state a proposition of which the proof is interactively built using tactics. The most standard assertion is a theorem followed by lemma, remark, fact, corollary and proposition.

A proof starts with the keyword *Proof*. Then Coq enters the proof editing mode until the proof is completed. In this mode tactics (chapter 8 of [20]) are used to complete the proof. The proof is ended by the keyword Qed, which returns the user back to the environment.

4.2. Abstract Syntax Tree

For programmatic processing, the text-based source code must be converted into a structure that simultaneously represents the original code and its syntactical structure. Such a structure is called the Abstract Syntax Tree and is previously described in chapter 2.5.2. In this section, we consider the required structure of the AST for Coq that will be used by our source code editor. We also discuss various ways of representing the AST in different types of programming languages and choose the most appropriate approach for our work.

4.2.1. Object oriented representation

In object oriented languages like C# or JAVA, the AST would be created by a tree of various objects called Nodes. A simple AST is shown in Figure 8.

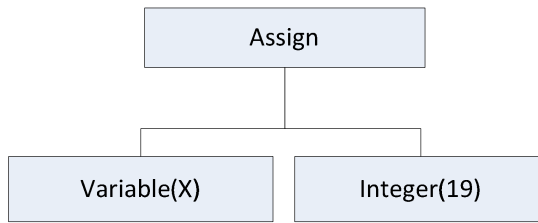


Figure 8 - Example of a simple AST

In larger codebases these structures can get fairly complicated so the resulting structure have a lot of various tree nodes. Creating an extensible representation in which the tree is traversed can be done using the Visitor design pattern[1].

The visitor pattern [21] provides a unified way to add new functionality to a tree based data structure without modifying the structure or itself. The Visitor itself encapsulates the logic of traversal through the structure and allows the derived structures to only implement the functionality for each node. This separates the data from the algorithms on the same data structure.

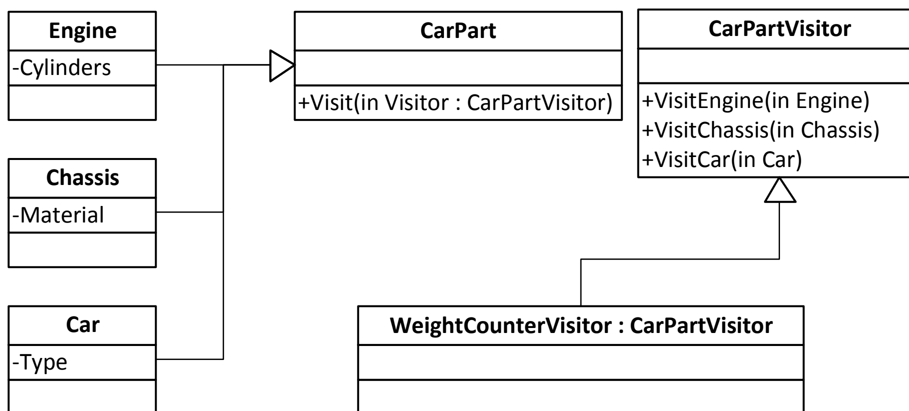


Figure 9 - UML diagram of the Visitor pattern

On the left side of Figure 9 we can see the tree representation of car parts. The visitor then implements a visit function for each node of the hierarchy. This is the place, where the desired functionality for each is coded. This allows extensibility because the way the tree structure is traversed is already defined and does not need reimplementation. The only thing that remains to be written is the action for each node. In the example we have a *WeightCounterVisitor* which can easily calculate the total weight of the CarPart.

When implementing the Visitor design pattern, each node in the AST structure has to have its implementation for each operation. This largely increases the size of the code compared to code that implements functionality in virtual methods. However, the Visitor pattern is the best object-oriented approach for representing data structures that rarely change, but allow adding new functionality.

[1] A design pattern is defined as a general reusable solution to a common problem. It serves as a template how such a problem should be solved.

4.2.2. F# REPRESENTATION OF THE AST

Adding new operations to an existing data structure is the primary way of implementing any code that works with data in a functional language [22].

The AST will correspond with the parser grammar of the Gallina language and will be created using the F# specific discriminated unions described in chapter 2.5.2. In Listing 2 an example of the structure described in Figure 9 is captured using discriminated unions.

```
type CarPart =
    | Engine of Cylinders
    | Chassis of Material
    | Car of Type
```

Listing 2 - Example of a discriminated union

This approach is basically the same as was described in the Visitor pattern but it is more common in the domain of functional languages where it has direct support.

4.2.3. SUMMARY

In the implementation of CoqAuWin, we choose the functional representation, because it provides a more succinct way to express the structure of AST and it makes it possible to add other functions that manipulate with the AST.

Another benefit of using discriminated unions and F# is also that most of the people that will be using the work are going to be used to Coq which is based on ML and because F# is also a derivate of ML then altering and adding functionality will be easy for these people.

4.3. Parsing of Gallina Language

The specification of the Gallina language can be found on the web page of the authors of Coq [20]. We use this documentation as the basis for our parser, because the reference manual describes the basic structure of the language and the grammar rules. Unfortunately, not every detail is explained, but it provides enough detail for the purpose of this thesis. The original Gallina language parser is written in OCaml, this imposes a slight similarity in certain respects.

For transformation of text-based source code into a structured representation (AST) F# provides the Fslex and Fsyacc tools. These come with the F# PowerPack [23] and we consider them in the next section .

4.3.1. FSLEX AND FSYACC

Fslex and Fsyacc come with the F# PowerPack in Visual Studio. They are based of the OCaml parser, but have some extended functionality. Also the F# compiler is written in Fslex and Fsyacc.

The main benefit of using this parser is the simplicity of the whole process from creating the rules for the lexer and parser all the way to the creation of our AST of discriminated unions. Also using an ML based language to parse an ML like language is also a logical choice.

As described in [24] Fsyacc generates an LALR parser which is the standard with most of the commonly known parsers. LALR stands for Look-Ahead Left to right parser that produces the Rightmost derivation. LR parsing can handle a larger range of languages compared to the LL parsers

4.3.2. LEXICAL ANALYSIS

Syntactically, Gallina is not any different from other languages. The lexical elements could be split into a few main categories:

- Blanks (Whitespaces) – Spaces, newlines, horizontal tabs and they are ignored by the lexer.
- Comments – Text enclosed between (* and *) and are treated as blanks.
- Identifiers and access identifiers – Described in chapter 2.2.
- Natural numbers and integers – Sequences of digits. Integers can be preceded by a minus sign.
- Strings – Strings are delimited by double quotes and enclose a sequence of any characters.
- Keywords – Reserved keywords. (list can be found in [20])
- Special tokens – Operator signs. (list can be found same place as keywords)

The Fslex tool can be used to transform source code into a sequence of tokens described in the previous list.

4.3.3. SYNTACTIC ANALYSIS

As discussed in Section 4.1, the Gallina language is a sequence of sentences. These sentences can be either proof related or they can be aimed at the Coq interpreter for informative purposes. One more feature worth pointing out is a special section called proof, where only tactics or Coq commands can be used. This behavior must be maintained, even if really won't need. It is crucial that the code can be parsed and worked with even when we can simulate these informative commands in some way in the IDE.

Notations make parsing of the Coq language somewhat difficult. They allow the programmer to define a text pattern with parameters that represents a small piece of Coq code. It can be viewed as more flexible version of Macros in the C++ language.

```
Notation "A /\ B" := (and A B).
```

In this example from this point forward if such a text pattern is located in the Coq proof code it is translated into the expression (and A B).

For our purposes and the purposes of this thesis we simplified the Notations to only have the form of Identifier Operator+ Identifier [Operator+ Identifier]*. Expressions such as “True /\ False” do not present a problem, because even if the notation was not previously defined we can (if we need to) check this information in the AST. Either way once an undefined notation is sent to the Coq compiler an error message is reported and it can then be displayed in the IDE.

This approach handles the most common uses of notations. Adding support for the definition of notations in the reference manual would require a significant modification to the way the parsing and lexing process works. Notations are effectively smart macros, so we would have to either go through the code 2 times, to replace all the notations correctly and then parse the code again, or we would have to create a system, that would be able to dynamically add tokens and grammar rules once a new notation would be detected.

4.3.4. OTHER ALTERNATIVES

In C# there are several parsers that can be used. These compilers can be used to create an AST that would have each node represented by an object. This would mean that we would have to create many classes and implement the Visitor for each possible tree node.

The list of other possible parsers:

- **Other parser generators.** There is a number of other parser generators including GOLD Parser [25], Irony.NET [26], Antlr [27] and Coco/R for C# [28]. However, these do not provide easy integration with F# and do not provide significant benefits over Fslex/Fsyacc.
- **Parser combinators.** Another approach used in functional languages is to develop parser using parser combinators [29] [30]. In F#, parser combinators are available using the FParsec library. However, parser combinators are generally less efficient and are more appropriate for small-scale solutions.
- **Reusing Coq parser.** We could create our own hand-written parser derived from the Coq source code that is a part of the Coq installation. However this option would be significantly harder, because the ML code used for this parser is not fully compatible with F# and it significantly exceeds the needs that we have of our parser.

We choose to develop parser for Coq using Fslex and Fsyacc. These are the standard tools for parsing in F# and are well integrated with F#, which is used to represent the AST. Another reason worth mentioning is the similarity with OCaml in which Coq itself is written. This may be useful in the development process as it allows comparing some portions of our implementation with the reference parser in Coq.

4.4. Background Proof Checking

Another aspect of our IDE that requires consideration is the checking of proofs in opened Coq source files. Proof checking is achieved using the standard command line Coq tool (by sending the code to the Coq process). It can be achieved in several ways. The most notable are:

1. Line by line feeding like in Coq Ide - The programmer manually sends the code to the Coq process.
2. Using a keyboard shortcut on a selection of the code. This behavior can be seen in the F# interactive window in Visual studio.
3. Automatically while typing. The way F# or C# code is highlighted immediately after it is written.

The first option is typical to the classic Proof Assistants and limits the programmer by separating the code into a locked and editable part. We want to avoid using this system of proof checking. On the other hand implementing the second option is straightforward. Because of the necessity of the manual selection it is more likely to be used on smaller proofs. For this reason, we intend to support this approach as an alternative for simple proofs.

Implementing the background checking of the proof script immediately while it is being written requires a more complicated approach. It is not possible to send the current code to the compiler and wait for the outcome because each keystroke would result in a few seconds of idle time, when the IDE would be nonresponsive.

To circumvent this effect it is possible to do several things. One of which is process the code in some way on a background thread and update the IDE user interface once the results are known Also sending the whole code after each keystroke could be resource consuming so just allowing the user to type freely and send the code again after the last UI update has occurred and only if the code has changed in the meantime. An important feature of the Coq interpreter is that is always in some state. This means that if a line is changed all that is needed to do is to undo a few commands until the compiler returns before to the edited line and then the new code can be processed again. This means that whole proof doesn’t necessarily need to be recalculated from the beginning after each edited line.

4.4.1. STEPWISE CHECKING

To avoid sending the entire input to Coq interpreter, we have to be able to split the source code into sentences so that is will be possible to send these sentences to the compiler one by one. Outputs and other information are stored for each sentence successfully sent to the compiler up until the first sentence that produces an error or until the whole proof script is complete.

Simple Parser (discussed later in Section 5.2.2) is used to split up the input text into segments that can be sent one by one to be processed. Each segment should have an output from the Coq compiler.

4.5. IDE selection

When considering what IDE should be used, the main goal was to find one that is well known and easily extensible. The main focus of this thesis is not the integration with a specific IDE. The IDE provides the means to build tooling that matches the goals highlighted in Section 3.4. Integrating into a different IDE would not change the structure of the project and should not be a major problem.

Originally the IDE should have been Microsoft Visual Studio, but for several reasons we decided to reconsider this decision. A major problem is that Visual Studio runs only on Microsoft Windows, while different IDEs have the possibility to run on different platforms and because Coq is also multiplatform the decision has been made to eventually prefer a multiplatform IDE. Another benefit of integrating into MonoDevelop instead of Microsoft Visual Studio is that implementing language support will be easier because of the technical reasons discussed in the following section.

There are many possibilities when considering an IDE to extend with language support. The most notable IDEs that can be extended with language support are:

- Visual Studio
- MonoDevelop
- SharpDevelop
- NetBeans
- Eclipse

From this list only Visual Studio is bound to one platform. JVM[1] languages are used to program add-ins for NetBeans and Eclipse. Plugins into the other IDEs can be written in any .NET Language but mostly they are written using C#.

[1] Java Virtual Machine (JVM) is a virtual machine capable of executing Java bytecode.

Visual Studio Integration

Integration into Visual Studio is done using the Managed Extensibility Framework. This is a library for creating extensible applications. The actual integration is done by creating a language package class which inherits the classes ProjectPackage, IVsInstalledProduct and IOleComponent, where many of the entry points for the various extensions are define using attributes.

Developing integration for Visual Studio 2010 is largely complicated by the fact that only a part of the extensibility process is done using MEF components. The rest is provided through legacy COM API. These two different methods for integrating into VS 2010 make the process more complicated.

MonoDevelop Integration

MonoDevelop uses a system based on a tree of extension points. Each point is a placeholder where an add-in can add a new item which provides extra functionality.

A MonoDevelop add-in consists of a manifest, which is an xml file describing the corresponding add-ins, their dependencies and declares the location of the actual implementation. More information can be found on the MonoDevelop homepage [14].

4.5.1. PROJECT AND FILE TEMPLATES

Both MonoDevelop and VS 2010 share a similar approach to the addition of new projects and file templates. Template files have to be created and they are accompanied by an XML file describing them. There are several records that have to be in this description file:

- The name and the description of the item
- Icon that will be displayed
- Project Type under which the item will be categorized

MonoDevelop can contain the content of the file directly in the XML description file. This can be useful for creating projects with one file that is empty or contains very little code.

4.5.2. SYNTAX HIGHLIGHTING

There are a few ways how Syntax Highlighting can be implemented. The most basic option is to have a set of defined rules and keywords and rules for recognizing common syntactic elements such as comments and strings. Texts in-between double quotes (strings), keywords, numbers and other parts of the code can all have different colors.

A more sophisticated way to create a syntax highlighter is to use data directly from the lexer and/or parser and separate tokens into categories. Then each category can have a separate style in which they are displayed. This approach has a major advantage – thanks to the lexer and parser, more information about the code is available (which identifiers were

defined, etc.). This provides the option to create dynamically changing categories and the highlighting of a certain word can be changed while the code is being written.

To implement highlighting into MonoDevelop a Syntax Mode XML file must be created. In this file the MIME Type[1] must be described and it is followed by the definition of keywords and their corresponding categories (in this case the color identifier that will be used from MonoDevelop settings). Comments, strings are defined by rules. Creating a highlighter using information from the lexer and parser can be done, but it is not documented and there are not many examples to be found.

Visual Studio on the other hand offers classes to implement the highlighting in code. The *IClassificationProvider* interface contains the method *GetClassifier* that has the whole text in the editor as its parameter. The classifier can then create lexical or syntactical analysis (or just go through a list of keywords) and return the highlighting information of the whole code.

[1] MIME Types or more correctly Media types are used for identifying media content more precisely then by the file extension (if there is any). Applications usually use MIME types for interpretation of embedded data.

5. IMPLEMENTATION

In this chapter we describe the implementation details of the parser required for parsing the Gallina language. This will be followed by the specifics of implementing the background compiler using F# agents. Finally we show how to implement all that we have created into the IDE MonoDevelop.

5.1. Lexer and Parser

Implementing a language parser requires the creation of 3 crucial parts: The lexer, parser and the AST into which the input code is eventually transformed. In this chapter, we discuss the specifics of the implementation - irregularities in the F# parser generator tools (Fslex and Fsyacc) and problems specific to Coq.

5.1.1. LEXER

The lexical analysis is defined in the file called Lexer.fsl and it does not look different from similar files of other parsers. It can be separated into 2 logical parts.

The first part is placed in-between brackets and code that can be used later on is placed. Usually this is the place where the lexer buffer (the tokenizer) is declared and it contains helper functions that are used for text processing and calculation of positions.

The second part consists of rules and definitions that describe the way the tokens should be recognized. Most of these rules are simple regular expressions. More complex constructs like strings or nested comments cannot be expressed by them so it is possible to write a set of separate rules that parse these parts separately.

Lexer Position

To correctly record the position of the lexer buffer first the initial positions must be declared. This is done by the function *setInitialPos* that has to be correctly called before the lexer starts to convert the input code into tokens:

```
let setInitialPos (lexbuf:LexBuffer<char>) filename =
   lexbuf.EndPos <-{ pos_bol = 0;
   pos_fname=filename;
   pos_cnum=0;
   pos_lnum=1 }
```

The end position of the lexer buffer is set to count from the column number (pos_bol) 0, position number (pos_bol) 0 and line number (pos_lnum) 1. Once the lexer encounters a newline the column and line number of the lexer buffer must be changed. This is done using a function called newline.

```
let newline (lexbuf:LexBuffer<char>) =
        lexbuf.EndPos <- lexbuf.EndPos.NextLine
```

Using these functions helps us identify the position of each token. The StartPos and the EndPos attributes have the location of the token in the input code. The input buffer is an array of characters. The position number indicates the index where the token starts or ends. Similarly the column and line provide us with the information in a more UI friendly way thanks to the functions we described above.

Identifier and Qualid identifiers

The rules of our Coq lexer first define the letters that can be used in the definition of identifiers. An Identifier can begin with any letter and an underscore. The following characters can be alphanumeric, underscore and an apostrophe.

```
let firstletter = ['a'-'z' 'A'-'Z' '_' ]
let subsequentletter = [ 'a'-'z' 'A'-'Z' '0'-'9' '_' '\'' ]
```

Following these definitions, the identifier and access identifier can be defined using regular expressions.

```
let ident = firstletter subsequentletter*
let accessident = '.' firstletter subsequentletter*
```

These can now be used to create a token for the identifier and the qualified identifier. A qualified identifier is a list of identifiers divided by a dot and without a whitespace between them for this we need to use a combination the ident and accessident expression.

```
| ident accessident+   { QUALID (lexeme lexbuf) }
| ident                { IDENTIFIER (lexeme lexbuf) }
```

Strings and Comments

Constructs such as strings or nested comments can be identified by creating a small rule set. The most notable difficulty with these rules is that the handling of the position has to be done separately and, if needed, one must pass the information about the position of the tokenizer and use the helper functions correctly.

Strings require correct handling of special characters like tabs or newlines or (back)slashes, also if a newline or the end of file comes before the ending apostrophes it is good to identify such a malformed string instead of not being able to create any token.

On the other hand, comments in Coq have to be aware of how deeply nested they are so that the token can be correctly assigned. Listing 3 shows an example of the comment rule set from our implementation.

```
and comment n = parse
| eof          { EOF(None) }
| newline      { newline lexbuf; comment n lexbuf }
| "(*"         { comment (n+1) lexbuf }
| _            { comment (n) lexbuf }
| "*)"         { if (n > 1) then comment (n-1) lexbuf
                 else tokenize lexbuf }
```

Listing 3- Comment lexer rules

5.1.2. PARSER

The parser definition is located in the file Parser.fsy. Like the lexer it is similar to parser files of other parsers. As before, the file can be split up into several parts. In the first part the used namespaces are enclosed by the characters %{ and }%. This is followed by the definition of the used tokens. A token can be either a plain token, or an informational token, that can contain some additional data. The type of this data must be defined with the token as can be seen in Listing 4.

```
%token <System.Int32> INTEGER
%token <System.String> STRING
%token <System.String> IDENTIFIER

%token UNDERSCORE
%token EQ
```

Listing 4 - Token definition

This is followed by the definition of the associativity of the token. This helps the parser decide which way it should interpret certain expressions that are otherwise ambiguous. For example if we set the associativity of the operator + to left the expression A + B + C will be interpreted as (A + B) + C. When the right associativity is set to the symbol ^ the expression A^B^C will be interpreted as (A^(B^C)). This sections is eventually ended by the %% symbol.

In the last section the parsing rules are described. First of the starting rule is defined and its corresponding AST node. In our implementation the language is defined as a list of sentences.

```
SentenceList:
  | Sentence SentenceList          { $1::$2 }
  | Sentence                       { [ $1 ] }
```

Each sentence can be an assumption, definition, inductive definition, fixpoint, assertion, records, commands and other as can be seen in Listing 5.

```
Sentence:
 //Gallina
 | AssumptionKeyword Assums DOT
                            {Sentence.Assumption($1, $2,
                                         parseState.ResultRange)}
 | Definition               {Sentence.Definition($1,
                                         parseState.ResultRange)}
 | Inductive                {Sentence.Inductive($1)}
 | Fixpoint                 {Sentence.Fixpoint($1)}
 | Assertion Opt_Proof      {Sentence.AssertionBlock($1, $2,
                                  parseState.ResultRange)}
 | Record                   {Sentence.Record($1)}
 //Commands - should be elsewhere maybe
 | Command                  { Sentence.Command($1) }
```

Listing 5 - Sentence parser rules

The code that is next to each rule represents the instruction for the parser and contains information on which node should be created in the AST for the corresponding parsing rule.

Abstract Syntax Tree

The abstract syntax tree is represented by discriminated unions described in Section 2.5.2 and Section 4.2. Using the tokens from the lexer with the parsing rules already gives us some understanding of the source code structure. The source code must be stored in a discriminated union that corresponds to the parsing rules. An example of corresponding AST using discriminated unions can be found in the following Listing 6.

```
type CoqAST =
   | CoqAST of list<Sentence> * Location

and Sentence =
    | Assumption of AssumptionKeyword * list<Assume> * Location
    | Definition of Definition * Location
    | AssertionBlock of Assertion * option<Proof> * Location
    | Fixpoint of Fixpoint
    | Inductive of Inductive
    | Record of Record
    //command
    | Command of Command
```

Listing 6 - Discriminated union for the CoqAST and Sentence

The root element of the tree is a list of sentences and information about the location – the start and the end of the code represented in the tree. This is similar for each sentence type.

Notations

A notation in Gallina is a symbolic abbreviation denoting some term or term pattern. A notation is always surrounded by double quotes. The notation is composed of tokens separated by spaces.

Notation implementation used in our thesis is very simple. It allows parsing certain expressions that will be rejected by the Coq interpreter, but it covers most of the common occurrences. More information about this matter can be found in Section 4.3.3.

To implement a notation definition we first need to add a notation type to the sentence discriminated union type in the AST:

```
| Notation of string * Term
```

Then a corresponding parser rule to the Sentence rule:

```
| NOTATION STRING COLONEQUALS Term DOT
      { Sentence.Notation($2, $4) }
```

To identify the use of a notation we have to add the notation functionality to the term parsing rules and add an AST representation of such a notation term:

```
//Notations AST
and Notation =
    | Notation of list<Term>
```

And the addition of to the parsing rules:

```
NotationExpression:
| Term Operator Operator Term
                          { Notation.Notation( [ $1 ] @ [ $4 ] ) }
```

This notation denotes two terms separated by two operator symbols. Adding other possibilities exceeds the needs of this thesis. But to correctly handle this problem it would be necessary to use a different approach.

Proof Body

The tactical language is used to provide the proof for the declared assumptions. The use of these tactics differs from the rest of the language. Usually they are just simple commands with none or a few terms as arguments. We categorize these commands by the number of arguments they have. The parser rules are shown in Listing 7.

```
TacticInvocationList:
  | TacticInvocation TacticInvocationList { $1::$2 }
  | TacticInvocation                      { [ $1 ] }

TacticInvocation:
 |INTEGER COLON Tactic DOT
              { TacticInvocation(Some($1), $3,
                parseState.ResultRange) }
 | Tactic DOT {TacticInvocation(None, $1, parseState.ResultRange)
}

Tactic:
  | IDWithoutKeyWords             { TacticNoArg($1) }
  | IDWithoutKeyWords Term        { TacticOneArg($1, $2) }
  | IDWithoutKeyWords Term Term   { TacticTwoArgs($1, $2, $3) }
  ........
  | RENAME ID INTO ID       { Rename($2, $4) }
```

Listing 7 - Tactics parser rules

TacticInvocationList processes the tactics one by one. Each TacticInvocation is an applied tactic that is used for the proof (i.e. intros, exact, apply) and it can be either directly a tactic or it can start with a number followed by a colon and a tactic. This defines the subgoal that is being targeted by the tactic.

Tactics can be divided into groups depending on how many arguments they take. We don't need to process each tactic because they usually don't mostly do not have any immediate products that can be used in our IntelliSense.

5.2. Background Agents and Proof Checking

An F# agent running in the background is used to process the editor code asynchronously so that the user experience of the IDE is not interrupted.

To achieve this we first have to create a Coq process that can accept commands and returns output from the interactive complier. This is done by the class CoqSession, where the Coq compiler is started and methods for communication are defined. The return type of the SendCommand method is defined here too and represents the possible outputs of the compiler (Error, Output or Proof).

```
type Message =
    | Error of String * Option<(int*int)>
    | Output of String
    | Proof of String
```

The Error case can optionally have additional information about where the error occured. Combined with other information we use Message values to highlight the error directly in the user interface. Output and Proof types each server as information for the corresponding panes (Section 5.3.7).

The CoqSession class gives us the ability to check code. Now we have to do this asynchronously on the background. For this we define a background agent (implemented by the MailboxProcessor type in F#) and specify the communication method.

It is only possible to send messages to the Agent. That is why the reply method must be also a part of this message. For this purpose the generic AsyncReplyChannel type is used. The messages used can be found in Listing 8.

```
type internal ProcessMessage =
    | Process of list<(string*Location)> *
        AsyncReplyChannel<option<string * option<int*int> * int>>
    | Retrieve of Lexing.Position *
        AsyncReplyChannel<option<Message>>
    | Proof of Lexing.Position * AsyncReplyChannel<string>
    | Check of string * AsyncReplyChannel<string>
```

Listing 8 - Agent Messages

Process message is used to send text the whole text to the compiler and returns the error string and its position if it occurs. The retrieve and proof messages are used to get the corresponding output for the position in the parameter. for line that is currently selected in the source code editor. The check message is used for tooltips.

5.2.1. AGENT IMPLEMENTATION

The Agent [18] consists of several methods that are essential for the correct function. The agent receives the code as a list of sentences and for each sentence there is an output up to the first encountered error. The sendState function creates such a list of outputs for each sentence. Another function that is used is the findLastProof which finds takes the output list and finds the last proof output before the given location.

```
let agent = Agent.Start(fun agent ->
    let rec loop prevstate = async {
        let! msg = agent.Receive()
        match msg with
        | Process (state)->
            //Send the lines to the coq parser
            //results are saved in retList
            //This may take some time
            if (retList.Length > 0) then
                //Calculate some positional info
                match h with
                | Message.Error(s,i) -> //event.Trigger(….)
                | None               -> //event.Trigger(….)
            return! loop state
        | Proof (p, reply) ->
            //Returns proof information for the given position
            reply.Reply(findLastProof p retList)
            return! loop prevstate
        | Check (c, reply) ->
            let m = ses.SendCommand("\nCheck " + c)
            //Returns tooltip info for the given position
            match m with
            | Message.Output(r) -> reply.Reply(r)
            | _ -> reply.Reply("Error")
        | Retrieve (p,reply) ->
            //Returns the output for the given position
            let mR = List.tryFind (fun (_,(b,e)) ->
                                  Common.IsBefore b p) retList
            match mR with
            | Some (s,l) -> reply.Reply(Some(s))
            | None -> reply.Reply(None)
            return! loop prevstate
    }
loop [] )
```

Listing 9 - Implementation of the Agent for the corresponding messages

The core part of the agent implementation is shown in Listing 9. According to the message received we can create the appropriate response. This is where the functions defined before can be used.

The *Process* message receives the current state of the code (list of sentences), restarts the Coq compiler and creates a new output list eventually triggering an event that informs of the completion of the process. If any errors are encountered all the information is passed through the event arguments. We can then subscribe this event from the IDE and update the highlighted lines correspondingly as shown in Listing 10.

```
CoqProcessAgent.Service.NotifyParsingFinished +=
        new FSharpHandler<EventArgs>
               (Service_NotifyParsingFinished);

void Service_NotifyParsingFinished(object sender, EventArgs arg)
{
 //checks
 document.Errors.Clear();

 var args = arg as ErrorResultEventArgs;
 if (args.ContainsError)
   document.Errors.Add(
    new Error(ErrorType.Error,
              /*Position in code*/,
              args.Message);
}
```

Listing 10 - Simplified version of the subscribed event that higlights errors

The *Proof* and *Retrieve* messages get the location as a parameter along with the reply channel through which the data is passed back to the caller. In response, the agent immediately returns output or proof information for the desired position.

Finally we create methods that can be called from any .NET languages. These are shown in Listing 11.

```
member x.ProcessMessage(ssl) =
        agent.Post(Process(ssl))

member x.RetrieveMessage(p) =
        agent.PostAndReply(fun replyChan ->
            Retrieve(p, replyChan))

member x.ProofMessage(p) =
        agent.PostAndReply(fun replyChan ->
            Proof(p, replyChan))
```

Listing 11 - Methods of the agents

5.2.2. Simple Parser

As mentioned before in the analysis of simple parser the implementation is very minimalistic and only a few lexer rules have to be created. The parser then has only a single rule to collect a list of sentences and create an appropriate list of sentences. Compared to the normal parser the sentence here is only a string (that can be passed to the Coq interpreter).

The problem lies in correct separation of sentences in the code that is being changed by the user and can possibly be incorrect at a specific time. This can be achieved by finding complete sentences that are ended by a dot symbol. All blanks (described in Section 4.3.3)

are ignored and the only problem that arises is correctly identifying access identifiers and dots used in strings as can be seen in the example.

```
“AAA.BBB”                    (*String*)
Space.space.space.ident      (*Qualified Identifier*)

a.b.c.                       (*Sentence with one Qualid*)
a. b.                        (*Two sentences *)
“aaaa”.”bbb”. cc.            (*Two sentences*)
```

The first two lines show the possible elements that can create a problem for the simple parser. The next two sentences show us how the sentences should be separated.

The tokenizer differentiates only two types of tokens. It tries to separate sentences, strings (in quotes) and comments. The purpose of the sentence rule is to identify the end of the sentence and mark it with the string token. The string rule is a simplified version of the one used in the Coq language lexer. Its purpose is to only correctly identify the end of the string and return it. The comment rule is equivalent to the one in the language parser and only handles the correct end of the comment.

Once the code is processed the parser just takes the strings and creates a list of sentences. These sentences are then sent to the compiler like mentioned in the previous chapter.

5.3. IDE Integration

The main topic in this section is the integration of Coq into the MonoDevelop IDE. The process of converting information from the AST into data structures was described in the previous section. Now we use the information from these structures and convert it into something that can be useful while writing code. The background agent interconnectivity with IDE will be described. The additional panels (interactive, output, proof) added to the MonoDevelop IDE will be also presented.

5.3.1. Code Highlighting

MonoDevelop uses an XML file to define the way syntax is highlighted. The definition of the syntax is stored in the file CoqSyntaxMode.xml and the root element is called SyntaxMode.

First the language specific properties are set. This tells the interpreter the meaning of some special characters. The properties that can be defined with a description:

- LineComment – The tag that starts a single line.
- BlockCommentStart – The tag that starts a block comment.
- BlockCommentEnd – The tag that ends a block comment.
- StringQuote – The quotation mark.

Each property can be set multiple times, so it is possible to set multiple types of string quotes (i.e. double apostrophe or just a single apostrophe). Unfortunately Coq only knows the one type of string quotes. The definition for the Gallina language is shown in Listing 12.

```
<Property name="BlockCommentStart">(*</Property>
<Property name="BlockCommentEnd">*)</Property>
<Property name="StringQuote">"</Property>
```

Listing 12 - Coq syntax properties

The span tag (shown in Listing 13) is a part of the buffer that is highlighted using a begin and an end string. The attributes of the span tags are:

- rule – The name of the rule that is valid inside the span.
- color – The color of the span content
- tagcolor – The color of the begin and end string
- escape – The escape string inside the span
- stopateol – Multiple line span

The order of the span definition may have a role in the way they are colored, especially if the begin string of one spam is a prefix of another span.

```
<Span color="comment.block" rule="Comment"
      tagColor="comment.tag.block">
  <Begin>(*</Begin>
  <End>*)</End>
</Span>

<Span color="string.double" rule="String" stopateol="true"
        escape='\'>
  <Begin>"</Begin>
  <End>"</End>
</Span>
```

Listing 13 - Comment and string span definition

The rule tag (shown in Listing 14) defines an own set of delimiters, keywords, spans, matches that can be used for highlighting. When a span is entered it activates the corresponding rule tag.. Keywords can be defined in the tag Word is located inside a Keywords tag. The color of the keywords category is set by the color parameter.

```
<Rule name="Comment">
  <Keywords color="comment.keyword.todo" ignorecase="True">
    <Word>TODO</Word>
    <Word>NOTICE</Word>
  </Keywords>
</Rule>
```

Listing 14 - Rule example

5.3.2. Parser

Adding a parser requires an addition to the add-in manifest file, where the entry point of the parser is specified.

```
<Extension path = "/MonoDevelop/ProjectModel/DomParser">
  <Class class = "Coq.MonoDevelop.CoqParser" id = "CoqParser" />
</Extension>
```

In this case it is in the CoqParser class in the Coq.MonoDevelop namespace. The CoqParser class inherits the MonoDevelops AbtractParser class. Here methods like *CanParse* and *Parse* are defined.

The *Parse* method is invoked when the source code is changed. It returns a class called ParsedDocument. This is where the code from the IDE is parsed by our parser and stored as the AST. Basically our implementation of the ParsedDocument has only one read-only attribute, the AST, which is set when the class is created.

The background agent is also called here to determine the outputs from the Coq compiler. If an error occurred it can be displayed in the IDE by setting the ParsedDocument Error property. An example of an error displayed in the IDE can be found later on in Section 6.1.

The SimpleParse method calls the background agent and returns an Error if it has occurred.

```
var e = SimpleParse(content);
if (e != null)
{
 doc.Errors.Clear();
 doc.Errors.Add(e);
}
```

5.3.3. Scopes

Scopes are the first structure created from the traversal of the AST. The purpose of scopes is to easily remember the context of used identifiers and allow easy access to this information. It is then used by the IntelliSense.

The scope is represented by the type ScopeState that is defines as a list of identifiers, Location and a list of ScopeStates.

```
// ScopeState is Location, list of
// ids defined in the scope and a list of subscopes
type ScopeState =
  | ScopeState of list<Identifier> * Location * list<ScopeState>
```

In the proof a new ScopeState is created after a definition of assertion denotes a new identifier (or identifiers). The location defines where the identifier is valid as can be seen in Listing 15.

```
Variables A B C : Prop.
Lemma distr_impl : (A -> B -> C) -> (A -> B) -> A -> C.
 intro H.
 intro H0.
 intro H1.
 apply H.
 exact H1.
 apply H0.
 assumption.
```

Listing 15 - Identifier validity in a proof script

In the example above each line represent a new scope being created. First a ScopeState type is created that contains identifires *A*,*B* and *C* and the location from the end of the sentence to the end of the file. This type is added to o global ScopeState. Next the lemma is added, where the identifier *distr_impl* is added to the list and the location is similar to the variables. In this case new subscopes are created. The identifiers *H, H0, H1* are gradually added to this list with the corresponding location (that ends where the proof of the lemma ends).

5.3.4. IDENTIFIER RESOLUTION

To correctly retrieve the identifiers relevant to a certain point first Scopes need to be created so that the relevant identifiers for a certain point can be then retrieved from them.

To traverse the AST in F#, a recursive function that matches the discriminated union against individual cases is needed. This means we need a match expression with a clause for each AST node that we are interested in visiting and then we can set the corresponding action we want to perform on that node. The nodes that are not matched can be ignored in the last (catch-all) clause so it is not necessary to implement every type of the discriminated union.

```
let rec CoqASTWalker ast =
    match ast with
      | CoqAST(sl,l) ->
              let (s,e) = l
              ScopeState([], l, List.map SentenceWalker e) sl)
```

In this example the root AST node is matched against the CoqAST type and the global scope is created. The end of the global scope is then passed to the SentenceWalker function along with the list of sentences:

```
and SentenceWalker endLoc s =
    match s with
    | Sentence.Assumption(ak, la, (sp, ep)) -> ...
    | Sentence.Definition (def, (sp, ep)) -> ...
    | ...
    | _ -> []
```

Listing 16 - The SentenceWalker function

In Listing 16 the Assumptions and Definition (and others not shown in this example) sentences are matched and the corresponding action is executed. In this case the return value of each walker function is a ScopeState filled identifiers that were defined within that scope.

Resolving the identifiers

Returning a list of identifiers for a certain position from this structure is a simple recursive function. If the position is in the location the identifiers are added to the list and this is done recursively for all sub-scopes.

```
let rec GetScopeStateIdentifiers pos (ss:ScopeState) =
    let (ScopeState(liid, loc, lss)) = ss
    if (Common.IsPartOf pos loc) then
        liid @ Common.Merge (GetScopeStateIdentifiers pos) lss
    else
        []
```

Listing 17 - A recursive function that returns a list of identifiers for a given position in a ScopeState

As can be in Listing 17 the ScopeState is first separated into the list of ids, location and list of ScopeStates. Then if the position falls into the ScopeState the list of identifiers is merged with the result of this function for each of the subscopes.

5.3.5. INTELLISENSE

To implement support for Intellisense into MonoDevelop we first have to add this information to the add-in integration manifest XML file.

```
<Extension path = "/MonoDevelop/Ide/TextEditorExtensions">
 <Class fileExtensions=".v"
        class="Coq.MonoDevelop.CoqTextEditorCompletion" />
</Extension>
```

This informs MonoDevelop that the Intellisense will be used with the files that have the .v extension and that its implementation can be found in the class *CoqTextEditorCompletion* in the namespace Coq.MonoDevelop.

This class inherits the *CompletionTextEditorExtension* class. The most important part to implement is the *CodeCompletionCommand* method, which is called every time the need

for IntelliSense is required. In this method the list of identifiers is constructed. This is done using the F# function described in the previous chapter and using LINQ[1].

```
//ContextInfo
int line = completionContext.TriggerLine;
int column = completionContext.TriggerLineOffset;
Position p = new Position("", line, 0, column);

//search for correct line
IEnumerable<CompletionData> retlist = null;

retlist = from l in Scope.GetScopeStateIdentifiers(p, r)
          select new CompletionData(l.Item1);
```

The information about the position is extracted from the completionContext passed as the parameter and the results are returned as a CompletionDataList.

5.3.6. TOOLTIPS

Tooltips are added by extending the TextEditorResolver node.

```
<Extension path="/MonoDevelop/Ide/TextEditorResolver">
    <Resolver class="Coq.MonoDevelop.CoqResolverProvider"
              mimeType="text/x-coq" />
  </Extension>
```

A class implementing the interface *ITextEditorResolverProvider* must be created. This is done by implementing two methods. First is the *CreateTooltip* method where the string displayed in the tooltip is returned. Next are the GetLanguageItem methods (one overload has an extra parameter), where the instance of a class *ResolveResult* is returned. Here it is possible to decide whether a tooltip for the item on the current location should be displayed. The information for the tooltip can then be recovered from the background agent.

5.3.7. PADS/PANELS

There are several pads in the Coq integration that need to be added to the IDE. This can be done by adding the pad definition to the add-in manifest file.

[1]Language Integrated Query (LINQ) adds native data querying capabilities to C#.

```xml
<Extension path="/MonoDevelop/Ide/Pads">
    <Pad id="Coq.MonoDevelop.CoqInteractivePad" defaultLayout="*"
         defaultPlacement="Bottom" _label="Coq Interactive"
         icon="md-coq-project"
         class="Coq.MonoDevelop.CoqInteractivePad" />
    <Pad id="Coq.MonoDevelop.CoqOutputPad" defaultLayout="*"
         defaultPlacement="Right" _label="Coq Output"
         icon="md-coq-project"
         class="Coq.MonoDevelop.CoqOutputPad" />
    <Pad id="Coq.MonoDevelop.CoqProofPad" defaultLayout="*"
         defaultPlacement="Right"
         _label="Coq Proof" icon="md-coq-project"
         class="Coq.MonoDevelop.CoqProofPad" />
</Extension>
```

For each pad the default layout, placement, label and icon can be set. The implementation of the pads can be found in the Coq.MonoDevelop namespace. For each pad a separate class that inherits from the *IPadContent* class is set.

In the *Initialize* method the content of the panel can be set. Usually this means emptying the content and setting the way the text is spread and if the content is editable. Also a method for displaying text is added.

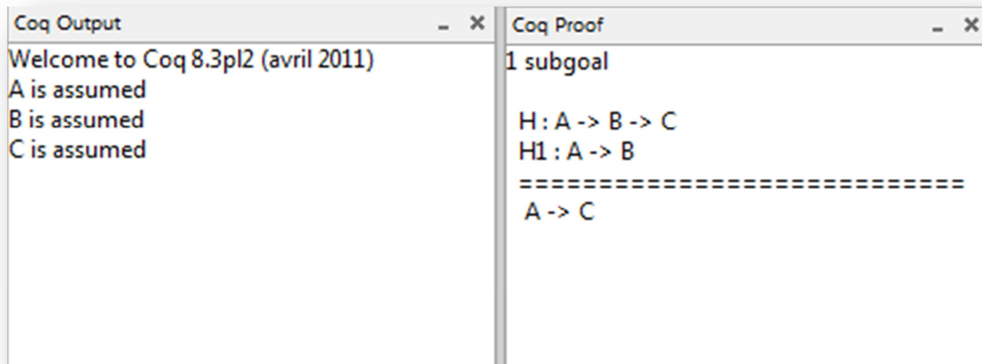


Figure 10 - Output and Proof panels

The proof panel is the simples displayed panel. It reacts to the change of the active document (the document that is being currently edited in the IDE) and to the line the cursor is on in the editor. According to these two options the corresponding proof is displayed by contacting the background agent.

```
Lemma distr_impl : (A -> B -> C) -> (A -> B) -> A -> C. Proof.
 intro H.
 intro H1.
 auto.
Qed.
```

Listing 18 - A simple proof of a lemma with a selected line

The proof displayed in Figure 10 is displayed when the cursor is somewhere on the highlighted line.

The output panel is basically the same as the proof panel. It displays the output for the corresponding sentences where the cursor is located in.

```
Variable A B C : Prop.
```

The output for this line is also displayed in Figure 10.

Coq Interactive Pad

The Coq interactive pad is a window, where a separate Coq process is running and one can type the commands directly into it and they are executed separately to the Coq process used in the background compiler. This is useful for checking the Coq syntax or behavior in simple cases. It can be also used to find the documentation and types of built-in Coq functions and predicates."

The interactive pad also has the ability to send the lines of code directly from the code editor using keyboard shortcuts. To implement such a feature commands have to be added. First to the manifest:

```
<Extension path="/MonoDevelop/Ide/Commands">
    <Category _name = "Coq Integration" id="Coq Integration">
      <Command id="Coq.MonoDevelop.CoqCommands.SendSelection"
        _label = "Send selection to Coq Interactive"
        _description="Send the selected text to Coq Interactive"
        shortcut="Ctrl|Return"
        macShortcut="Alt|Return"
        defaultHandler="Coq.MonoDevelop.SendSelection" />

      <Command id="Coq.MonoDevelop.CoqCommands.SendLine"
        _label = "Send line to Coq Interactive"
        _description="Send the current line to Coq Interactive"
        shortcut="Ctrl|Alt|L"
        macShortcut="Meta|Control|L"
        defaultHandler="Coq.MonoDevelop.SendLine" />
    </Category>
</Extension>
```

First the category is set in which the commands reside. The command has an id, label, description, shortcut attribute. In defaultHandler the class that handles the command is set.

And then implement the Command classes that inherit from the class *CommandHandler*, where the Run method has to be overridden which defines the action that will be executed when the command is invoked.

6. SUMMARY

First we will present the results of our work in a step by step walkthrough, using screenshots to demonstrate the IDE. Next we will compare our solution to the current IDEs used for theorem proving. This will be followed by the evaluation and an elaboration on future work. Finally the conclusion of this thesis will be stated.

6.1. Sample

When MonoDevelop starts a new project can be created by clicking in the top menu on File-New-Solution. A new option for a Coq solution is available in the dialog window that appears. A new Coq project can be now created. This dialog can be seen in Figure 11.

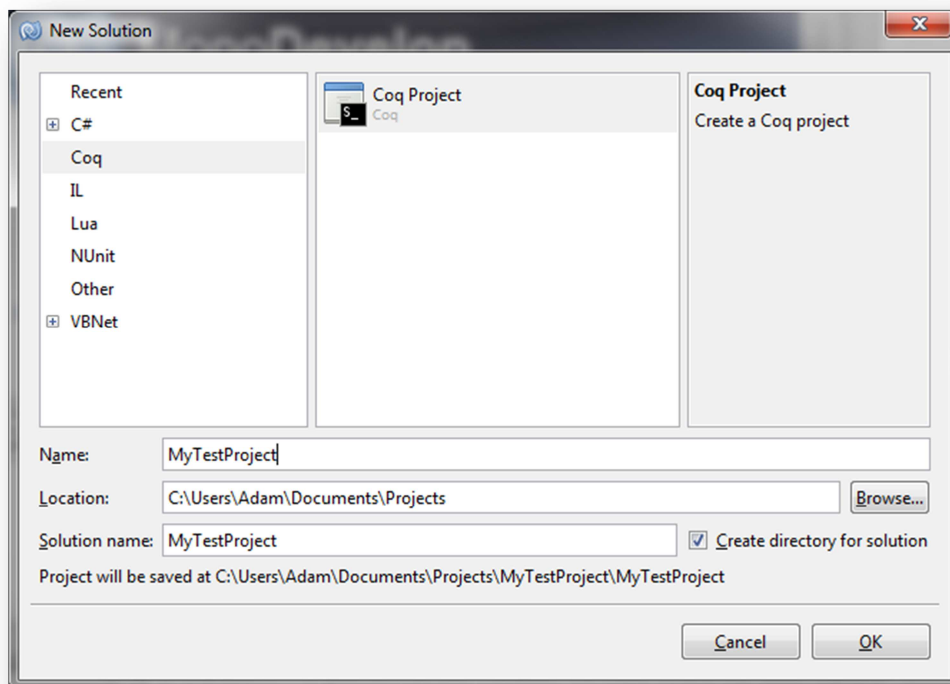


Figure 11 - New solution dialog

When a new project is created the IDE switches the view to the source code editor with an opened default script file. This script file is called main.v the extension is common to Coq script files. In Figure 12 we can see how the IDE looks in script editing mode. In our example the proof and the output panes are located on the right. The Coq immediate panel is attached at the bottom. This can be easily changed by dragging the mouse over the title of the pads and dragging them to another location. If the panes are hidden for any reason, they can be found in the View-Pads option in the MonoDevelop top menu.

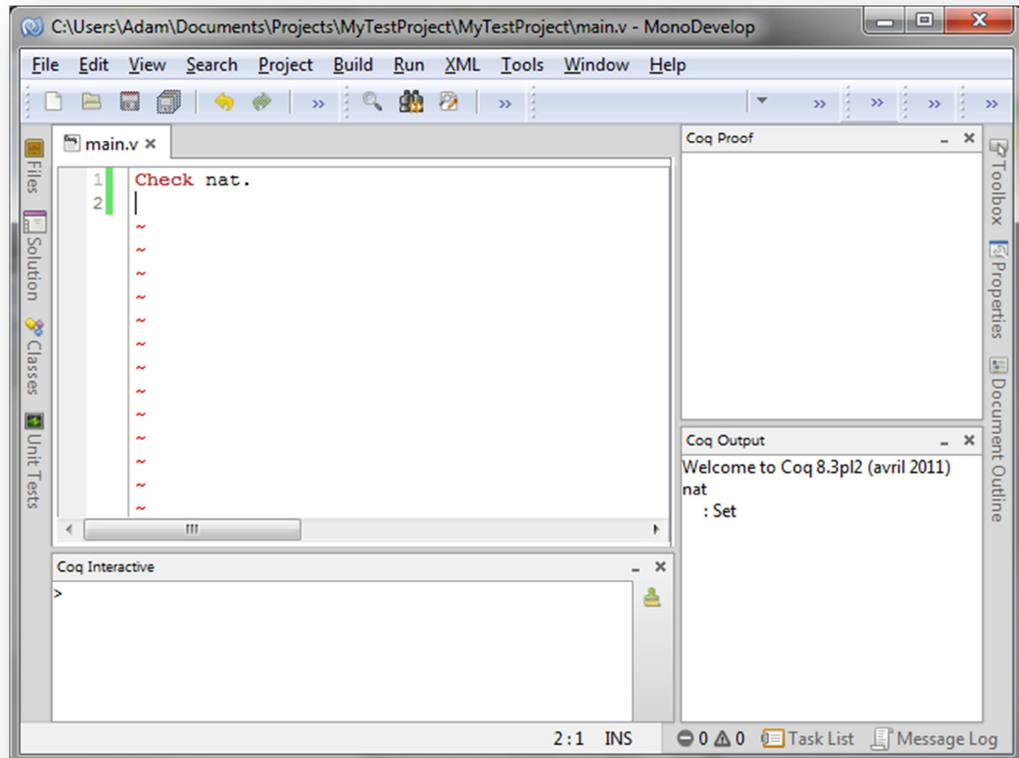


Figure 12 - MonoDevelop IDE when developing in Coq

While writing a script file, we can access the identifiers that are relevant to the current context using the common CTRL+SPACE keyboard shortcut. A list of identifiers then appears close to the cursor. This behavior can be seen in the Figure 13.

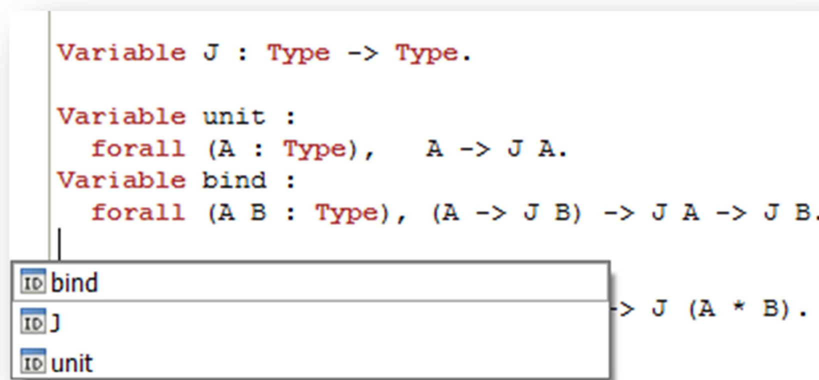


Figure 13 - IntelliSense support

To view the normal or proof output of lines, we have previously entered one must move the cursor into the sentence/command where the information is required. In Figure 14 we have selected the whole line to demonstrate the cursor position. The output is then in the Coq proof pad.

```
26  Notation "a >>= f" := (bind _ _ f a) (at level 60, right as
27
28  Lemma mduplicate_eq :
29    forall A f g,
30    f = g -> mduplicate A f = mduplicate _ g.
31  Proof. intros until 0; intros ->; trivial. Qed.
32
33  Lemma mzip_eq :
34    forall A B a1 a2 b1 b2,
```

Coq Output

Coq Proof

1 subgoal

```
============================
forall (A : Type) (f g : J A), f = g -> mduplicate A f =
mduplicate A g
```

Figure 14 - Output example

Tooltip information about an identifier or type can be displayed by hovering the mouse cursor over the desired part of code. The tooltip appears nearby as displayed in Figure 15.

```
Variable J : Type -> Type.

Variable unit :
  forall (A : Type),   A -> J A.
Variable bind :
  forall (A B : Type), (A -> J B) -> J A -> J B.

Variable mzip :
  forall (A B : Type), J A -> J B -> J (A * B).

Variable morelse :
  forall (A : Type),   J A -> J A -> J A.
Variable mzero :
  forall (A : Type),   J A.

Variable mduplicate :
                                A -> J (J A).
```

mduplicate
: forall A : Type, J A -> J (J A)

Figure 15 - Tooltip example

In Figure 16 the way error messages are presented is shown. A tooltip is available when hovering over the problematic line. In the output panel the same error message is offered but only if the cursor is in or after the position of the error in the code.

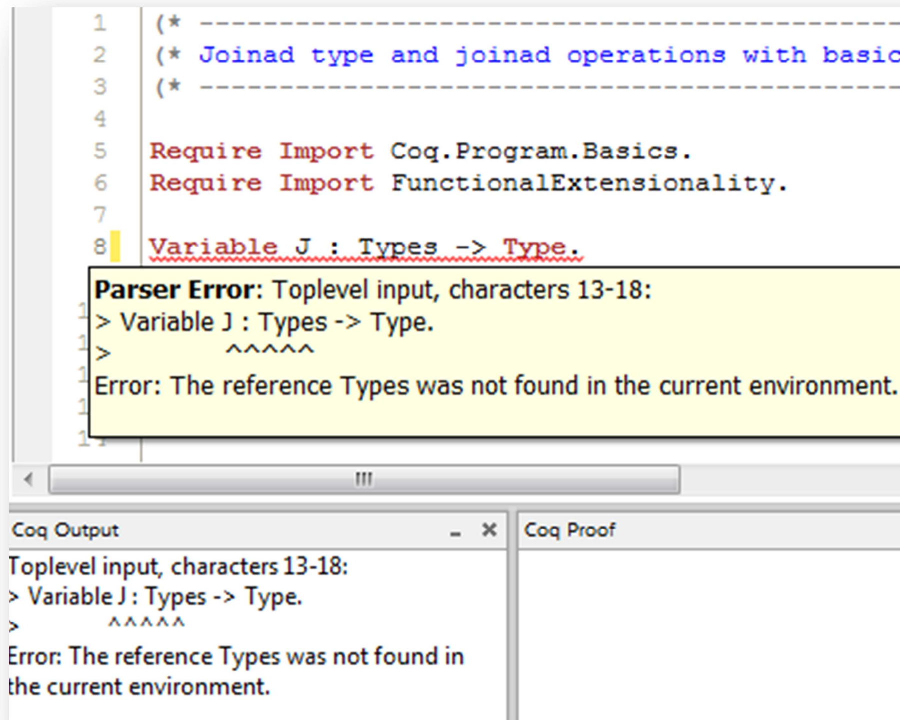


Figure 16 - Error tooltip and output example

6.2. Comparison

To compare our Coq integration for MonoDevelop with other theorem prover IDEs we have to consider many aspects.

We provide a modern, modifiable and extensible IDE which gives the possibility to move and hide pads and benefits from MonoDevelops advanced source code editor. Easily modifying the IDE and the editor is not possible in Coq Ide and it is not as comfortable with Proof General.

On the other hand Coq Ide and Proof General still provide the user with more options when developing proofs. They contain a list of tactics and templates with often used sentences. This could be easily added to MonoDevelop by extending corresponding menu nodes. Another useful feature of Coq Ide that we did not implement (yet) is that one can directly use tactics from the proof panel.

Neither Coq Ide nor Proof General offer tooltips with information about identifiers (such as types) or error messages. Getting this information means the addition of extra check commands.

The process of proof development differs significantly, because we offer seamless background checking instead of the line by line processing in Coq Ide and Proof General. This is probably the largest difference between our implementation and the other IDEs.

6.3. **Evaluation and Future work**

To evaluate our work we review the goals (discussed in Section 3.4) and compare them with the results presented in Section 6.1. This will be followed by presenting areas, where this thesis could be improved in the future.

Advanced editing. We have provided the basic integration of Coq into MonoDevelop. The functionality of modern editing tools is at the disposal of the user. We have added syntax highlighting and autocompletion (IntelliSense). We have also added pads and windows for instant feedback and/or help.

Background error checking. We have implemented seamless background checking without the need to send the code to the Coq compiler manually. The result of this background checking is the ability to display outputs and errors immediately in the panes without having to move the processed lines.

Simple installation. We have to use an installer or a simple tool to copy our implementation into MonoDevelop. Once copied/installed, MonoDevelop should contain all new supported features. We developed this integration under Microsoft Windows, but adding Linux support should not be a problem.

Extensibility. We provide anyone who is willing to extend support for Coq with access to our AST representation of the code. Using a similar approach like we used to implement the identifier resolution they can gather information from it and use it any way they please.

6.3.1. **FUTURE WORK**

There are many ways to improve the work in this thesis. It can be done in breadth or in depth. Each feature we have introduced is only a basic implementation and can be improved and extended.

Parsing. Our implementation of the parser was developed by the reference manual description of the Gallina Language. We have not implemented support for various libraries that add functionality to Coq. Implementing the language extensions of each library would be a good way to further extend the capabilities of the parser. Also an important feature that needs to be implemented in some point is to create a parser that can recover from an parsing error and that will continue parsing.

IDE. The IDE can be extended in many ways. Extending the menus to have similar options as Coq Ide, i.e. Templates or listed Tactics. The proof output window can be also extended so it is possible to directly click on the proof and apply tactics using the context menu. Finally adding more features that take advantage of the AST would provide probably the best improvements. A feature that comes to mind is refactoring.

Installation. In the future, it is possible to added Coq integration into the MonoDevelop repository. Coq support will then be directly available from the menus of the IDE.

6.4. **Conclusions**

In this thesis we focused on implementing Coq support into a modern IDE, more specifically MonoDevelop. We have identified the main features of other IDEs used for Coq –Coq Ide, Proof General and also we have gone through the features present in modern development tools for imperative and for functional languages. From this we created a subset of features that we eventually implemented into MonoDevelop.

We have proven it is possible to use modern IDE features for interactive theorem provers- some of which were not available before. We have created our own parser of the Gallina language to create an AST which we used for providing the autocomplete feature. This can be used to gather any information about the source code for more advance editing features.

We have successfully achieved the goals we have set but there is still a lot of work that can done in many areas that were discussed. Probably the biggest shortcoming is in the parser support that does not contain error recovery and does not implement some advanced aspects of the Gallina language. Nevertheless we have laid the ground for further work in this area and hopefully created a useful tool for theorem proof developers.

7. REFERENCES

[1] (2012) Proof - Webster's Online Dictionary. [Online]. http://www.merriam-webster.com/dictionary/proof

[2] Keijo Heljanko. (2006, May) Model Checking based Software Verification. [Online]. iplu.vtt.fi/digitalo/modelchecking.pdf

[3] K. Appel and W. Haken, "Every map is four colourable," *Bulletin of the American Mathematical Society*, vol. 82, pp. 711-712, 1976.

[4] G. Gonthier. (2005) A computer-checked proof of the Four Colour Theorem.

[5] The Coq Development Team. The Coq Proof Asistant. [Online]. http://coq.inria.fr/

[6] INRIA. (2012) CompCert. [Online]. http://compcert.inria.fr/research.html

[7] Adam Chlipala. (2008) Certified programming with dependent types. [Online]. http://adam.chlipala.net/cpdt

[8] David Aspinall. (2012) Proof General. [Online]. http://proofgeneral.inf.ed.ac.uk/

[9] Microsoft™. (2012) Visual Studio. [Online]. http://www.microsoft.com/visualstudio/en-us

[10] The Eclipse Foundation. (2012) Eclipse. [Online]. http://www.eclipse.org/

[11] The Coq Development Team. The Coq Proof Assistant, What is Coq. [Online]. http://coq.inria.fr/what-is-coq

[12] Gérard Huet, Gilles Kahn, and Christine Paulin Mohring. (2009, July) The Coq Proof Assistant - A Tutorial. [Online]. coq.inria.fr/V8.2pl1/files/Tutorial.pdf

[13] Wikipedia. (2012) Integrated development environment. [Online]. http://en.wikipedia.org/wiki/Integrated_development_environment

[14] Xamarin and The Mono Comunity. (2012) MonoDevelop. [Online]. http://monodevelop.com/

[15] Xamarin and The Mono Comunity. (2012) MonoDevelop - Extension Tree Reference. [Online]. http://monodevelop.com/Developers/Articles/Extension_Tree_Reference

[16] Wikipedia. (2012) Abstract syntax tree. [Online]. http://en.wikipedia.org/wiki/Abstract_syntax_tree

[17] Don Syme, Tomáš Petříček, and Dmitry Lomov, "The F# Asynchronous Programming Model," in *In PADL '11: Proc. 13th International Symposium on Practical Aspects of*

Declarative Languages, 2011.

[18] Tomáš Petříček and Jon Skeet. (2012) MSDN - Server-Side Functional Programming. [Online]. http://msdn.microsoft.com/en-us/library/hh273081.aspx

[19] Makarius Wenzel, "Asynchronous Proof Processing with Isabelle/Scala and Isabelle/jEdit," *Electronic Notes in Theoretical Computer Science*, 2010.

[20] The Coq Development Team. (2012) Reference Manual. [Online]. http://coq.inria.fr/doc/

[21] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides,.: Addison-Wesley Professional, 1994, ch. 5.

[22] Tomáš Petříček and Jon Skeet, *Real-World Functional Programming*.: Manning, 2010.

[23] Microsoft. (2012) The F# PowerPack, with F# Compiler Source Drops. [Online]. http://fsharppowerpack.codeplex.com/

[24] J. Jeuring and D. Swierstra. (2001) Grammars and Parsing. [Online]. http://www.cs.uu.nl/docs/vakken/gont/diktaat.pdf

[25] Devin Cook. (2012) GOLD Parsing System. [Online]. http://goldparser.org/

[26] Roman Ivantsov. (2012) Irony -.NET Language Implementation Kit. [Online]. http://irony.codeplex.com/

[27] Terence Parr. (2012) ANTLR Parser Generator. [Online]. http://www.antlr.org/

[28] Hanspeter Mössenböck. (2012) The Compiler Generator Coco/R. [Online]. http://ssw.jku.at/Coco/

[29] Graham Hutton and Erik Meijer, "Monadic parser combinators," University of Nottingham, Technical Report NOTTCS-TR-96-4 1996.

[30] Daan Leijen and Erik Meijer, "Parsec: Direct style monadic parser combinators for the real world," Utrecht University, Technical Report UU-CS-2001-35 2001.