UNIVERSITY OF TARTU
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
Institute of Computer Science

Kristjan Vedel

Eclipse-based IDE for the Agda Programming Language

Bachelor's Thesis (6 EAP)

Supervisor: Aivar Annamaa
Co-Supervisor: Sven Laur

Author: ..... "....." mai 2012

Supervisor: ..... "....." mai 2012

Co-Supervisor: ..... "....." mai 2012

Allowed to defend
Professor: ..... "....." mai 2012

TARTU 2012

Contents

Introduction

Agda is an young functional programming language with support for many innovative features like dependent types and interactive type-directed development. Currently the only choice for the interactive development of Agda programs is to use the Emacs-based development environment provided with Agda, which constrains the users interested in Agda to an editor many are not familiar with. Also Emacs, being an relatively old editor, does not provide many of the features supported by more recent integrated development environments.

The goal of this thesis is to design and implement an integrated development environment (IDE) for Agda on top of the Eclipse Platform. Given the unconventional nature of Agda and the complexity of developing a modern IDE, the resulting development environment is intended more as a starting point than a full-featured IDE in itself. Nevertheless it is a working software and demonstrates the areas where it can complement the existing IDE while also showing to be capable of supporting the main features from the existing tools like the interactive development of Agda programs.

Eclipse plug-ins developed in this work were written in Java and the components used for integration with Agda were written in Haskell. The source code for all implemented programs is made available for downloading under an open source license.

The work is divided into four chapters. The first chapter gives a brief introduction to dependent types and intuitionistic type theory, necessary for understanding the principles of Agda. Second chapter introduces Agda programming language and Agda Emacs mode, which is the existing IDE for Agda. Third chapter is about the Eclipse Platform, the creation of plug-ins for Eclipse and the core concepts of an Eclipse-based IDE. Fourth chapter is the main chapter of this thesis, describing the implementation of the IDE. First two sections of the fourth chapter examine the supportive tooling avail-

able for developing an Eclipse-based IDE and the DLTK framework chosen for current implementation. Third section describes the architecture of the solution, while the fourth section gives an overview of the supported features.

User guide and access to the source code in repositories and on a CD are provided as appendices.

Chapter 1

Background in Type Theory

1.1 Types and Type Systems

Most high-level programming languages use data types for classifying data and type systems that associate types with values to ensure some acceptable level of type safety. Type system can be either static, meaning type checking is done during compile time or dynamic, if typechecking happens at runtime. Type systems are usually either first-order (for common procedural languages) or second order (when extended with constructs as type parameters or type abstractions) and subtyping is also a frequently supported feature (almost universally for object-oriented languages) [1]. Still, in all those languages there is a clean separation between types and values: all types are either atomic or constructed from simpler types. Such type systems help to prevent a wide range of programming errors, but there are still cases that could be found with more powerful type systems.

1.2 Dependent Types

Let's consider a list of elements of some arbitrary type A and a function *head* that should return the first element from of a list of elements. This list would then have type $List\ A$ and function *head* would have type $List\ A \to A$. What should function *head* return when it is applied to an empty list? It's not possible to forbid this case by a type system where types can only talk about types. In case of Haskell, if the function *head* from module *Prelude* is applied to an empty list, an error is raised at runtime, which is rarely the

desired behavior. What we would want instead is to define function *head* in such a way that type system could enforce that it can be applied only to non-empty lists. This means that the type for *list* has to contain information about the length of the list or in other words it has to depend on a value that is the length of the list. Types that depend on value are called *dependent types* [2].

Having dependent types it is possible to define a type for a list of elements that depends on (is indexed by) the length of the list by $List\ A\ n$. Given the previous example of function *head*, we can now define it as only applicable for arguments of type $List\ A\ n$, where $n > 0$, so the type system can forbid the application of this function to an empty list. For a more extensive overview on the practical value of dependent types see *Why dependent types matter* by Altenkirch, McBride and McKinna [3].

1.3 Intuitionistic Type Theory

Dependent types play a major role in *intuitionistic type theory*, which is a logical framework for constructive mathematics developed by Martin-Löf [4]. As in constructive mathematics the law of excluded middle is omitted, it's necessary to actually construct the object to prove its existence. This bears some fundamental similarity with how computer programs are written. In programming, the type declaration `x:Int` states that there exists a value with the type `Int` and the value assigned to `x` serves as a proof for it. Similarly a declaration `f:Int` $\rightarrow$ `Int`, stating that there exists a function taking an integer and returning an integer, can be proved by writing any of such functions. Also no programs can be written for the declaration of empty type `x:`$\bot$. This exemplifies the central idea behind intuitionistic type theory, namely, a correspondence or isomorphism between proof systems and models of computation. By this relation, usually called the *Curry-Howard correspondence*, propositions correspond to types and proofs correspond to programs. From the perspective of proof theory this means that verifying proofs reduces to type checking and for program construction this gives the benefits of being able to express both specifications and programs with the same formalism, use the proof rules to derive correct program from specification or verify the properties of a program [5].

For an example of this correspondence, consider a program e_1 with type t_1 and another program e_2 with type t_2. We can think of e_1 to be the proof of

some logical formula t_1 and e_2 to be proof of t_2. For the proof of formula $t_1 \wedge t_2$ proofs for both t_1 and t_2 would be needed and we can write this as a pair of proofs (e_1, e_2). This, in turn, results in a program with type $t_1 \times t_2$, so the product type corresponds to conjunction. Similarly disjunction corresponds to disjoint union type, implication to function type etc.

Dependent types allow to encode properties of values as types whose elements are proofs that given property is true. For example $\forall a, b : \mathbb{N}.a + b = b + a$ is a type for a function that assigns for any pair of natural numbers a proof for commutativity of addition and by writing a program for this type we have a proof that addition is commutative for all natural numbers.

In order for the intuitionistic type theory to be consistent, all functions must be total. This also means that they are not allowed to allowed to crash or be nonterminating.

Chapter 2

Agda

2.1 Overview of Agda

Agda is a dependently typed functional programming language, based on the intensional variant of Martin-Löf's intuitionistic type theory [6]. Features of Agda include wide range of inductive data types, coinduction, dependent pattern matching, termination checking, mixfix operators, a module system. Agda also has a full support for unicode identifiers and keywords and an interactive type-directed development environment. The concrete syntax is strongly inspired by Haskell.

By the Curry-Howard isomorphism Agda is also a theorem prover: a proposition can be proved by writing a program for corresponding type. It has many similarities with other proof assistants with dependent types, like Coq, Epigram, Matita and NuPRL. However, unlike Coq and some other proof-assistants, Agda does not have a separate tactic language and proofs need to be written by hand, through interactive manipulation of the proof. Agda does include a proof-search tool called Agsy[7], which can construct some simple proofs automatically and recently there has been some work to integrate automated theorem proving into Agda[8][9].

Agda has a standard library, offering a range of facilities for algebra, category theory, data types, functions, IO and more. There are also other libraries and developments listed on the Agda wiki [10].

2.1.1 Code Examples

Every Agda file must contain a single top-level module declaration whose name has to match with the name of the file.

```
module List where
```

Rest of the program goes inside that module.

```
data Nat : Set where
  zero : Nat
  suc  : Nat → Nat
```

Here we defined a top level module `List` and an inductive datatype `Nat` for natural numbers with constructors `zero` and `suc`. The type of `Nat` is a predefined type `Set`, which itself is of type Set_1, which is of type Set_2 etc. Next we show an example of a list with a dependent type with constructors `[]` for empty list and `_::_` as an infix constructor for appending an element to a list.

```
data List (A : Set) : Nat → Set where
  []   : List A zero
  _::_ : {n : Nat} → A → List A n → List A (suc n)
```

Note that the type of a `List A` is `Nat → Set`. This means that this is a family of types indexed by natural numbers. `List A n` is a type for each natural number `n`. Underscores in constructor or function definitions mark a place for arguments, so `_::_` takes two arguments, an element and a list. Using such constructions allows us to define also *mixfix operators*, with arbitrary argument placements, like `if_then_else_` for the standard conditional expression. Curly braces in the type of `_::_` denote *implicit arguments*. Using implicit arguments we do not need to provide the type when applying the function or constructor, as long as type checker can infer it. Here the type checker can infer it from the type of the third argument.

Now we can define a function `head` for a non-empty list, using pattern matching.

```
head : {A : Set}{n : Nat} -> List A (suc n) -> A
-- head [] = ? -- Commented out, type checked would not allow this case
head (x :: xs) = x
```

We give the case for a non-empty list and not for empty list. This still satisfies the requirement that all functions must be total, because the case for constructing empty list would not be type correct as the argument list with type `List A (suc n)` cannot have a length of `zero`.

2.2 Emacs Mode

2.2.1 Overview

Agda programs are commonly edited using text editor Emacs customized with Agda Emacs mode. Agda Emacs mode is a major mode for Emacs, that allows programs to be developed interactively and in a type-directed way. [11] This means that it is possible to typecheck incomplete code, leaving a placeholders called holes or goals than can be filled later. It is also possible to do various things in the context of goal like get the context of the goal, infer the type, give or refine the type, evaluate a term and more. Communication between editor and Agda proof engine is currently done through an interactive GHCi session, which is initiated using a similar Emacs mode for Haskell. Work usually begins with loading the file that is being edited into the Agda proof engine. In response a temporary file is created with a list of tokens with both syntactic and some semantic information. This is used to provide semantic highlighting and a simple reference-based navigation.

Agda Emacs mode is being actively developed and currently it's the only IDE supporting interactive development of Agda programs. A screenshot of Agda Emacs mode with descriptions of various buffers used can be seen on Figure 2.1 .

2.2.2 Alternative to Agda Emacs Mode

Emacs is relatively popular as an IDE, but having only Emacs based development environment has also some drawbacks. Emacs UI and controls are historically evolved and differ from the guidelines for modern desktop environments, so new users often have to learn Emacs in addition Agda. Emacs is extended in Emacs Lisp dialect of Lisp programming language, which is not widely used, so developers are harder to find and the quantity of extensions tends to grow more slowly than, for example, for a Java-based Eclipse. While Emacs has a good support for editing code, it's not as capable

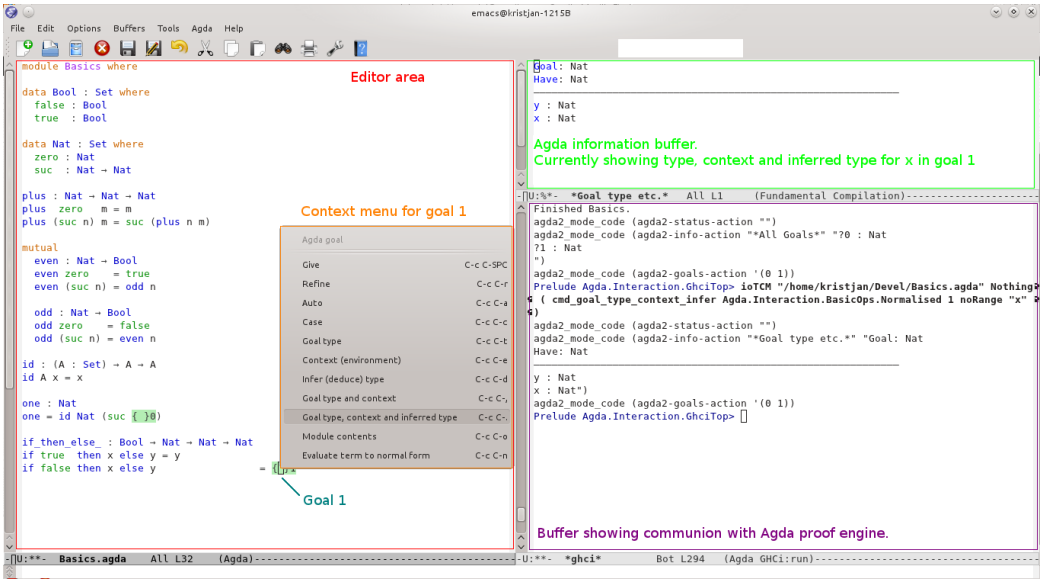

Figure 2.1: Agda Emacs mode

when browsing code or managing larger projects. Also with single front-end the components tend to get more tightly coupled so an alternative front-end would hopefully help to refine the API Agda provides for IDEs.

To complement the choices for development environments, the platform for this alternative IDE would need to address the listed shortcomings, while retaining the advantages. The most suitable option for this seems to be Eclipse, which is the dominant Java IDE and a popular platform for IDE development. There are also examples of creating an Eclipse based IDE for tools with existing Emacs interface, for both functional languages like Haskell [12] and interactive theorem proving tools like Proof General [13].

Chapter 3

Eclipse

3.1 Overview of Eclipse

The Eclipse Project is a Java-based open source project, developed by the Eclipse Community [14]. It is composed of multiple subprojects and built on a system of modules called plug-ins. While widely known as a powerful and popular IDE for programming language Java, it was designed as an "IDE for anything, and for nothing in particular" [15], that is to provide a common platform for diverse IDE-based software and aid with their integration. Main part of Eclipse Project is the Eclipse SDK which is a complete development environment for developing Eclipse based tools and also for developing Eclipse itself. Eclipse SDK is composed of:

- The Eclipse Platform, providing the core framework and services for all plug-ins as well as the runtime in which they are loaded, integrated and executed.
- Java Development Tools (JDT) , that provides the plug-ins composing the Java IDE.
- Plug-in Development Environment (PDE), wchich provides a number of plug-ins to facilitate building plug-ins for Eclipse.

The Eclipse Platform consists of three layers:

- Platform Runtime is core component of Eclipse, used for discovering, integrating and running plug-ins. It is based on an implementation of the OSGi core framework specification[16] and is the only part of Eclipse that is not itself implemented as a plug-in.

- Rich Client Platform (RCP) is a set of tools for building and deploying rich client applications.
- Workbench IDE UI is a set of plug-ins providing user interface functionality for integrated development environments. It is built on top of Eclipse RCP.

Figure 3.1 shows a more detailed architecture diagram of Eclipse.

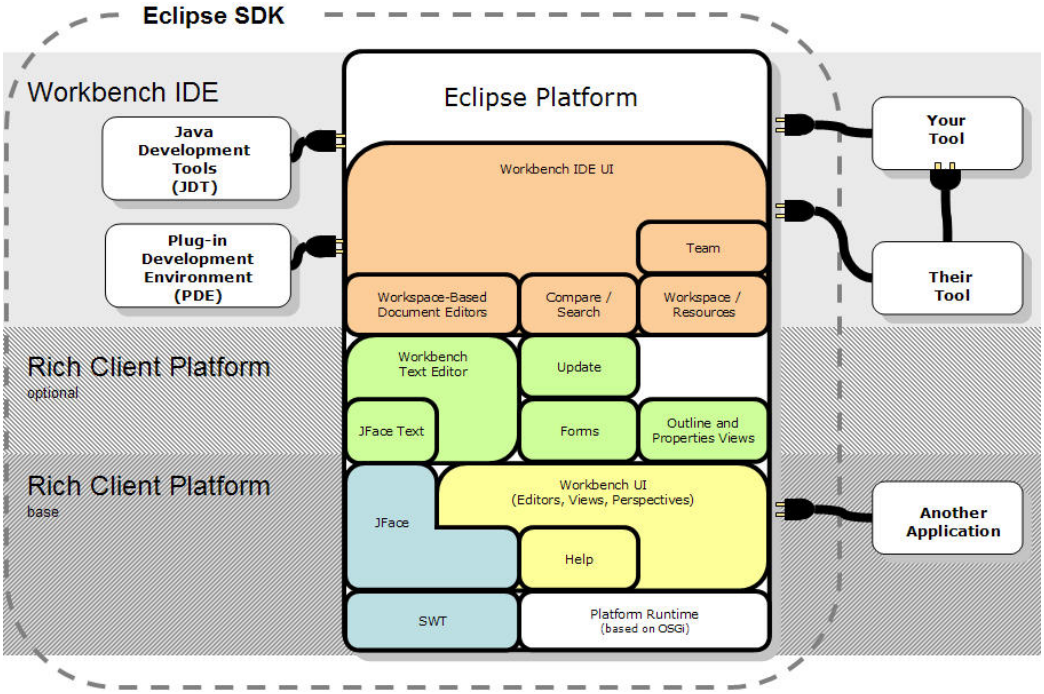

Figure 3.1: Overview of Eclipse Architecture [17]

3.2 Eclipse Plug-in Development

Eclipse plug-in is the smallest software component providing a service in the context of Eclipse Platform. At its core the plug-in is a module for OSGi framework, called *bundle* in the OSGi terminology. A simple Eclipse plug-in consists of a set of Java class files and a manifest file named *MANIFEST.MF* describing the bundle and its dependencies. In addition to the the OSGi framework, there is also an extension management system called Extension Registry, where *extensions points*, defining a contract for extension for other

plug-ins, are matched up with *extensions*, which implement these contracts. Extensions to an extension points are defined in the plug-ins via the file *plugin.xml* using XML. These tools provide great amount of flexibility in defining and managing services and allow development of loosely coupled, but well integrated software systems on top of Eclipse Platform.

Plug-in Development Environment includes a comprehensive tooling for developing plug-ins. It includes wizards for creating plug-in projects, specialized editors for plug-in configuration files and supports running a second instance with plug-ins being developed.

3.3 Eclipse-based IDE Development

Most Eclipse-based development environments have similar architecture where the IDE is divided into multiple major components and each of the components is packaged into a plug-in. A general rule in developing Eclipse plug-ins is to separate core functionality from user interface. The minimal set of plug-ins for an Eclipse-based IDE consists of at least *Core* and *UI* components. Support for debugging, if provided, is usually also separated into its own component.

Core plug-in provides the non-UI related infrastructure for the IDE. A typical Core plug-in for an Eclipse-based IDE consists of at least a project *nature* for associating projects with tools and plug-ins, an in-memory model of source code, usually derived from the corresponding *abstract syntax tree (AST)* and a source code *parser*, responsible for creating the AST. For more complex IDEs the Core component contains also many additional features based on traversing and manipulating the language model and AST, like indexing, searching, navigation, refactoring and code formatting. Compilers and parsers are usually integrated into Eclipse using *builders*. Builders are components responsible for incremental manipulation of project's resources. They work on *resource deltas*, which reflect all resource changes since the last invocation of the builder. Such incremental builder can become quite complex, so they are often packaged in a separate plug-in.

UI implements the user interface for IDE. The central component of an integrated development environment is usually the *editor*. The editor can be either structured or a text editor and is customized for working with the source code of IDE's target language. Two of the more important components of an Eclipse editor are the *source viewer* and the *document* compo-

nents. The document holds the actual content for editor while the source viewer responsible for displaying the document. *Source viewer configuration* is an extendable point of configuration for source viewer, allowing to plug in customizable UI behavior, like syntax highlighting or text hover. Typical features provided by UI also include *views* for outline of current module, type hierarchy and *wizards* for creating new projects and source files. Different *actions* and *commands* that allow user interaction with the IDE are also defined in the UI component. The visibility of components and actions and the overall layout of the user interface are managed by a feature called *perspective*. UI usually includes also the *preference pages* to configure the provided features.

Chapter 4

Implementation

4.1 Supporting Tools For Eclipse-based IDE Development

The Eclipse Platform offers just the APIs for implementing development environments but resulting IDEs usually share a lot of common features including syntax highlighting, parsing with error annotations and problem markers, abstract syntax tree based navigation and content assist [18]. Multiple projects have been created to facilitate IDE development with Eclipse, from which the following were evaluated:

- Xtext [19]
- The IDE Meta-Tooling Platform (IMP) [20]
- Dynamic Languages Toolkit (DLTK) [21]

Xtext is designed for implementing domain-specific languages and programming languages and offering the complete language infrastructure to support it. It's tightly integrated with the Eclipse Modeling Framework (EMF) and it's functionality is based on using a parser generator on a language grammar to create a parser and metamodel to work with. Is not designed with the purpose of integrating with existing tools and thus is not suitable for integration with existing tools for Agda.

IMP is a framework similar to Xtext. It includes a parser generator, but supports also other parser generators as well as hand-rolled parsers. It

takes more work to create an IDE with IMP than with Xtext, but it's more customizable and allows better integration with existing tools.

DLTK is set of extensible frameworks designed in the spirit of JDT to reduce the complexity of building full-featured development environments for dynamic languages. While DLTK has a lot of overlap with IMP, it does not include the meta-tooling support and a favors more direct approach instead. DLTK is designed for interacting with external language tools like external interpreters or debuggers.

Wider scope and meta-tooling features might make IMP suitable for integration with the Agda IDE in the future, but for this project DLTK was chosen in favor of IMP for its better support for language interoperability and great exemplary implementations.

4.2 Integration with DLTK

Eclipse plug-ins developed in current work are to a great extent based on the DLTK. DLTK provides exemplary development environments for Tcl, Ruby, Python and Javascript, which serve as documentation for this project in addition to a few available tutorials [22] [23]. There are also DLTK-based IDEs for many other languages, including a debugging-centric plug-in for functional programming language Scheme [24] and a development environment for a static and natively compiled language D [25].

DLTK provides a language model, similar to that of the JDT, to represent the workspace structure from the project level to the internals of source files. It also includes a generic but easily extensible implementation of an AST and a lot of extensible features, common to most IDEs that work on the language model or AST. Unfortunately the language model of DLTK is directed towards the object oriented languages and does not fit well with with unconvetional functional languages like Agda. The difficulties that arose in integrating Haskell with Eclipse [26] are also relevant in achieving a good integration between the semantic structure of Agda programs and Eclipse-based IDE tools. Considering the scope of this project, the choice was to use the model provided by DLTK and extend it where necessary, to match the more important programming constructs of Agda.

4.3 Overview of the Architecture

In the course of this work, an Eclipse-based IDE for Agda was developed, hereafter referred to as *AgdaEclipse*. It is composed of two major components: a set of Eclipse plug-ins comprising the IDE and a library written in Haskell to act as a proxy between Eclipse plug-ins and Agda proof engine. A TCP connection is established between the two components when the plug-ins are loaded in Eclipse. As plug-ins are initialized lazily, the connection is only established after Agda perspective has been opened. In spite of using a network protocol, both Eclipse and Agda installations currently have to be deployed on the same host, as some of the communication uses references to a local filesystem. Would this be replaced with streaming file contents over TCP, then the IDE and Agda backend could be installed on separate machines. One possible use case for this would be to let some more powerful computer handle the expensive type-level computations (e.g. proof verifications), while the development can be done on a laptop or other less powerful machine.

Figure 4.1 shows a high-level overview of the architecture of AgdaEclipse IDE.

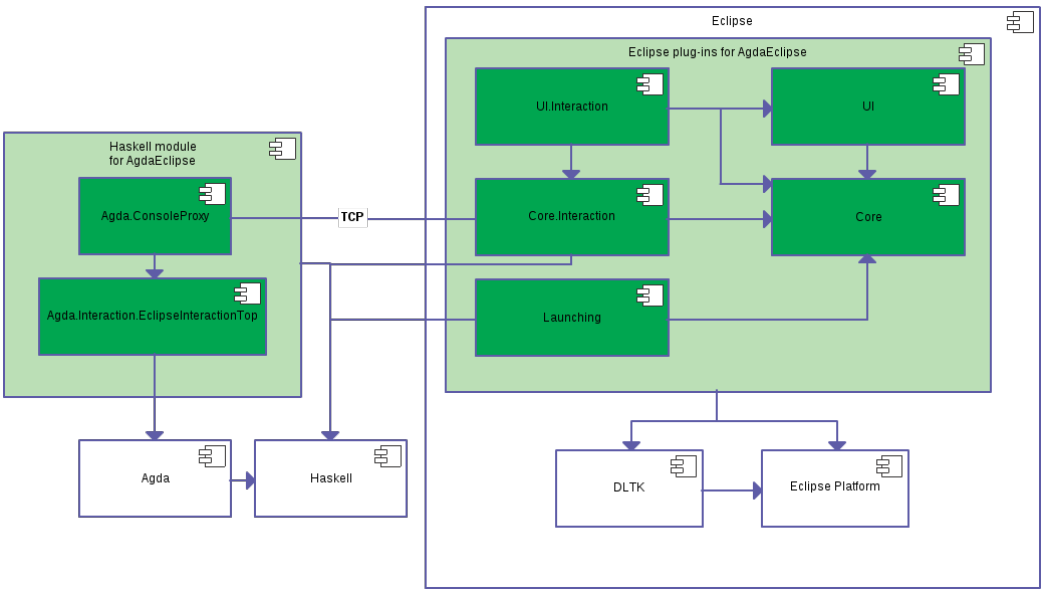

Figure 4.1: Overview of AgdaEclipse Architecture

4.3.1 Implemented Eclipse Plug-ins

The set of Eclipse plug-ins developed for AgdaEclipse with their corresponding identifiers are listed in Table 4.1.

Plug-in	Plug-in ID in Eclipse
Core	`org.bitbucket.agdaeclipse.core`
UI	`org.bitbucket.agdaeclipse.ui`
Core.Interaction	`org.bitbucket.agdaeclipse.core.interaction`
UI.Interaction	`org.bitbucket.agdaeclipse.ui.interaction`
Launching	`org.bitbucket.agdaeclipse.launching`

Table 4.1: AgdaEclipse Eclipse plug-ins

In general the Core and UI plug-ins contain the traditional IDE features mentioned in Section 3.3. Launching is used for executing Haskell programs and Core.Interaction and UI.Interaction are the core and user interface components for interacting with the Agda proof engine. The contents of implemented plug-ins can be summarized by following:

- **Core**
  The Core plug-in defines Agda nature and includes simple implementations of source parser, completion engine and selection engine by extending the base functionality provided by DLTK.

- **UI**
  UI plug-in is the standard user interface plug-in for an Eclipse based IDE. It is by far the largest plug-in developed in this project, but most of its features are supported by the underlying DLTK framework. Most important components included are the Agda perspective, providing the overall layout, and editor with the related source viewer configuration and token scanners providing the syntax highlighting and other customizations. Other components of UI plug-in are a completion processor, templates and template completion processor for code completion, a customized search page for Agda language elements, various preference pages for configuring the IDE and wizards for new projects and files.

- **Launching**
  Launching plug-in is more related to Haskell than Agda as it defines

the Haskell interpreters and the launching of Haskell programs. It is used for running the Haskell program that initiates the TCP client.

- **Core.Interaction**
  This is the core part of interacting with Agda proof engine. Some of the most important components in Core.Interaction plug-in are the interaction facade, which handles the requests to Agda proof engine and a parser that relies on the interaction facade for parsing. Core.Interaction also contains the Haskell script for launching the TCP client and an implementation of DLTK's script interpreter, which serves as a backend for TCP-based consoles. Rest of the components in Core.Interaction plug-in serve the interaction facade and parser.

- **UI.Interaction**
  UI.Interaction extends the Agda perspective with views for Agda console and goals. It also contains the actions for executing commands provided by the interaction facade.

Implemented Eclipse plug-ins are packaged into a single *feature* (with id `org.bitbucket.agdaeclipse.feature`) and the feature is contained in an *update site*.

4.3.2 Implemented Haskell Modules

In addition to the Eclipse plug-in a Haskell library was developed to provide a TCP client for the plug-ins and a proxy for Agda proof engine. It consists of the following two modules:

- **Agda.Interaction.EclipseInteractionTop** is a Haskell module responsible for managing the state for the Agda file that is active in the editor and also for communicating with the Agda proof engine.

- **Agda.ConsoleProxy** is a simple TCP client, that is launched with the description of the server it should connect to. When the connection is established it waits for the input from the server and forwards the requests to Agda.Interaction.EclipseInteractionTop module. After receiving the response it wraps it in XML and sends it back to the server. Synchronous request-response message exchange pattern is used here for simplicity reasons.

Haskell modules are packaged using the building and packaging system for Haskell libraries called Cabal[27]. They have external dependencies on the following Haskell modules: base, network, Agda, mtl, directory, split, xml, bytestring, utf8-string. All these dependencies should be resolved automatically when Cabal is used for installation.

4.3.3 Communication with Agda

The communication between the implemented Eclipse plug-ins and Agda is built on top of the DLTK's interpreter and console functionality. The console system of DLTK is a client-server solution designed for communicating with an external language interpreter over TCP. The console-based approach was preferred over interacting directly with the operating system process for it's design principles that promote better separation between the environments of interpreter and IDE and built-in support for handling an interactive console in Eclipse.

For AgdaEclipse, this process starts by opening of the Agda perspective, which opens the Agda console and by that triggers the connection initialization process. During the initialization a component called console server is started. Console server is a component from DLTK that communicates the information between the interpreter and console view. Then a Haskell program `consoleproxy.hs`, included with Eclipse plug-ins, is executed and it initiates the TCP client defined in Agda.ConsoleProxy. Started client then opens TCP connection to the script console server running in Eclipse and starts waiting for incoming requests. By that the connection to the Agda console is established.

All supported requests, except for closing the connection, are forwarded to Agda.Interaction.EclipseInteractionTop. The request itself is a simple comma-separated list of strings while the response is a set of commands in the form of XML, wrapped inside a response tag and with the length of the response as a prefix. Figure 4.2 shows an example of querying the type of the first goal for active file.

On the Eclipse side, components interested in communicating with Agda proof engine do not access the console directly, but instead use a service facade called `AgdaInteractionFacade`, which provides a simplified Java interface to all the supported services of the proof engine. A response from Agda to a single request can include a variety of commands and often they

Request

```
goalType,/home/kristjan/workspace/test.agda,False,0
```

Response

```
0000000403<response>
<command type="Resp_Status">(&quot;&quot;)</command>
<command type="Resp_DisplayInfo" subtype="Info_CurrentGoal">
  Set
</command>
<command type="Resp_InteractionPoints">
  <interactionids>
    <interactionid>0</interactionid>
  </interactionids>
</command>
</response>
```

Figure 4.2: Example of the communication protocol

are not directly related to the request itself, so returning the XML response to the caller to handle would not be the best solution. Instead `AgdaInteractionFacade` provides a way to register and unregister listeners for various responses. This allows actions like displaying responses in the console to be handled centrally and is closer to the asynchronous communication behavior introduced recently in the Agda Emacs mode.

4.4 Features

Most of the features provided by AgdaEclipse are in general very similar to those of any other Eclipse-based development environment. The layout, visible in Figure 4.3, and the overall perception of the IDE is probably familiar to those who have used JDT or other similar tools. In the core of the AgdaEclipse IDE is the Agda editor which is in essence just a standard Eclipse-based text editor, exhibiting behavior similar to that of modern graphical text editors. Functionality for specializing the editor for developing Agda programs is added through sub-components plugged into the editor.

The type system of Agda is powerful enough to represent very complex propositions and writing programs for such types and even the types themselves can be quite a challenge. This means that an important feature for

any Agda IDE would be to support the interaction with the type system. A subset of the features for the interactive development provided by the Agda Emacs mode were also implemented in AgdaEclipse to assist the development of more complex programs and proofs.

The customizations of the editor and also other more important features, most notable being the interaction with Agda proof engine, are described in the following subsections.

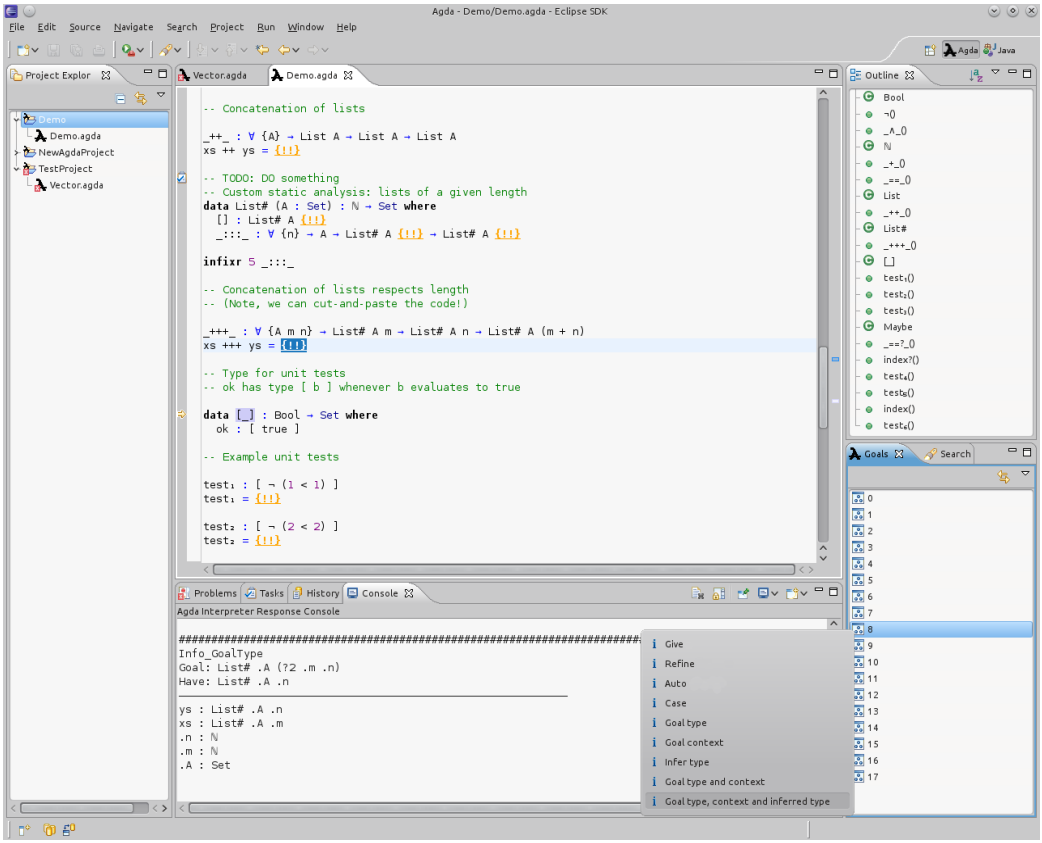

Figure 4.3: Working with AgdaEclipse

4.4.1 Agda Interaction

A significant part of developing more complex Agda programs consists of communication between the developer and the type system using placeholders for code to be filled later called holes or goals. Interactive development in AgdaEclipse currently relies on using a separate view to show the list of

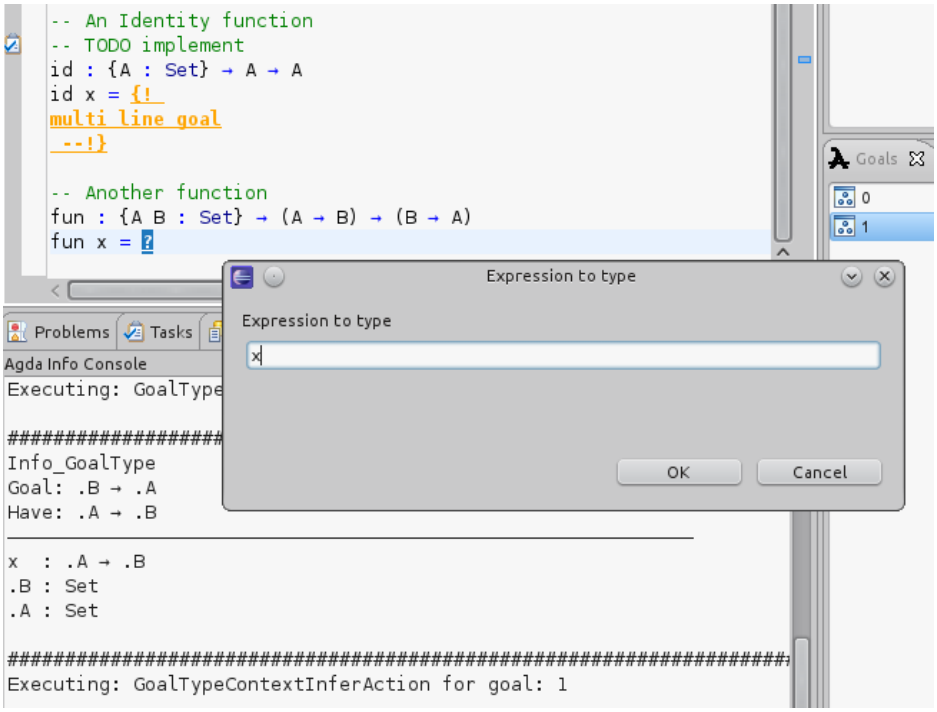

Figure 4.4: Interaction with Agda. Executing the command for getting the goal type, context and inferred type for x.

goals for the active file. Information about goals is cached on the Eclipse side behind the interaction facade and updated when Agda sends a response containing a new list of goals for given file.

Supported commands for Agda interaction include the informative actions in the context of the goal like getting the goal's type or context and inferring the type. Figure 4.4 shows an example of interacting with the Agda proof engine.

A command for reloading a file is also implemented and added to the menu bar of goals view. This is necessary as currently only one file at a time can be active on the Agda side, while Eclipse caches the information for all the workspace files. The support for keeping the whole workspace synchronized with the state in Agda would be a major improvement in the usability of the IDE.

4.4.2 Perspective

Agda Perspective includes following Views:

- Project Explorer – A standard Eclipse view for displaying projects and their contents in active workspace
- Outline – Displays the structure of the currently active Agda file
- Problems – Shows a list of problem markers denoting various unresolved problems for projects in active workspace, including problems reported by Agda
- Tasks – Displays a list of task markers found in the source code of projects in active workspace, includes tasks in Agda files.
- History – A standard view that shows combined history from version control system repositories with content in the local Eclipse history.
- Console – Displays a variety of consoles, including information console and proof engine communication console for Agda
- Goals – Lists the goals for currently active Agda file.

The standard layout of Agda perspective can be seen in Figure 4.3.

4.4.3 Parsing

Parsing is the most complex feature relying on the interaction with Agda. Core plug-in includes a simple parser class named `AgdaSourceParser` that extends the corresponding extension point from DLTK. When the connection to Agda proof engine is established, then `AgdaSourceParser` delegates the actual parsing to another parser located in Core.Interaction plug-in called `AgdaParser`. From `AgdaParser` a request is sent over the TCP connection to the Agda proof engine, which handles the actual parsing, creates a temporary file containing a list of resulting tokens and responds with the path to the temporary file. `AgdaParser` then parses the contents of this file into a simplified AST, containing declarations for functions, data types, records and fields and returns it to the `AgdaSourceParser` which in turn returns it to the caller. All information about references and other types of expressions is currently ignored, this is definitely one of the areas that should be improved as many other features like navigation, content assist and semantic highlighting would also benefit from this.

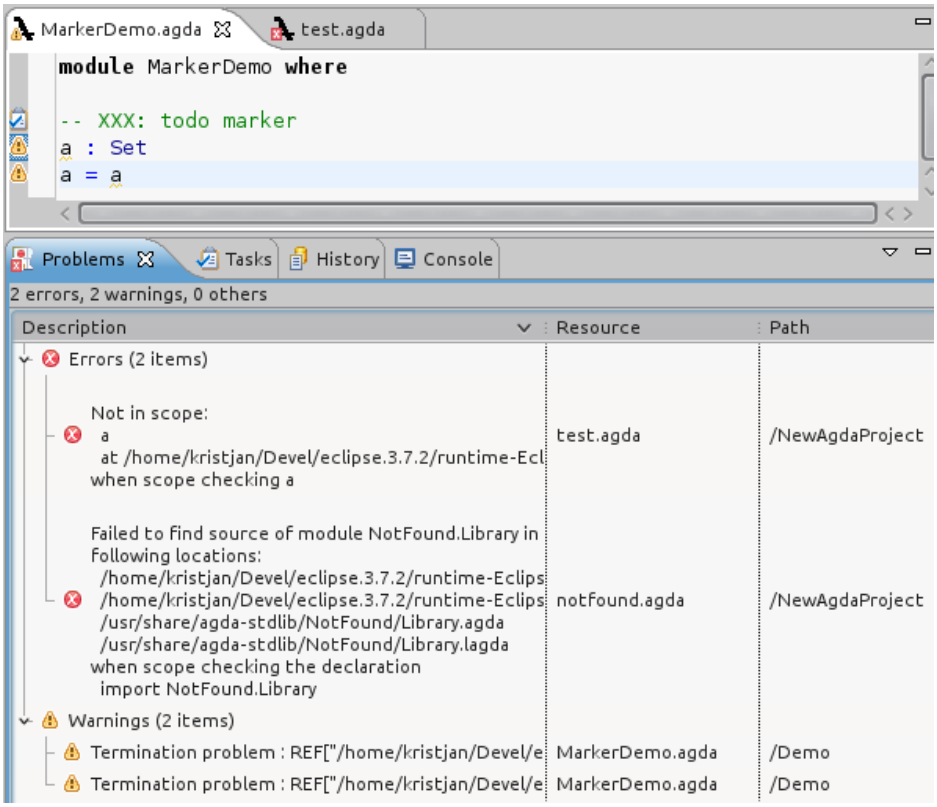

Figure 4.5: Problem markers.

4.4.4 Problem Marking

Parsing will not always succeed and it's important to mark the failures as close to the source as possible. Eclipse uses resource markers to add problems and other similar notes to the files. Markers supported in AgdaEclipse include error level problem markers for different building and parsing problems, while warning level problem markers are used to mark termination problems. Task markers are also supported and are used to insert tasks as comments in the source files. A screenshot with different markers can be seen on Figure 4.5.

4.4.5 Syntax Highlighting

Syntax highlighting is based on the standard rule-based reconciling provided by the Eclipse Platform. Two scanners, `AgdaPartitionScanner` and `AgdaCodeScanner`, responsible for tokenizing the document, are added to

the editor through `AgdaSourceViewerConfiguration`. Resulting tokens are highlighted according to the colors assigned to them in the preferences. Applying the tokenization rules is generally fast so highlighting can be updated immediately on text changes. Conversely, this means that the reconciling process is unaware of the semantic structure of the document, so it's not possible to distinguish between function and constructor references for example. Agda Emacs mode relies on interaction with Agda proof engine, and the syntax tree stored there, to provide semantic highlighting features. Same could be done for the AgdaEclipse by extending DLTK's API for semantic, AST-based highlighting, though this would need better support for the AST of Agda on the Eclipse side. Preferences for customizing the syntax highlighting can be seen on Figure 4.8

4.4.6 Content Assist

The Eclipse's standard "Ctrl+Space" combination is supported for content assist. Suggestions for auto-completion include the names of various user-defined declarations from the active file, templates defined for Agda and also Agda keywords. Some templates have been predefined and new templates can be added and existing ones changed in the Agda template preferences page. Figure 4.6 shows an example of using the content assist feature. Also a simple element comment header feature can be seen on the same screenshot.

4.4.7 Navigation

Outline view allows navigation to function, data type or record declarations in the context of an active file. Also items *Open Agda Type...* and *Open Agda Function...* were added to the navigation menu for searching type and function declarations over all the files in the workspace. Screenshot of using the *Open Agda Function...* feature can be seen on Figure 4.7. Support for navigating to the declaration by following the reference in the file is also included, but for this to be usable again the AST must be improved to properly include references.

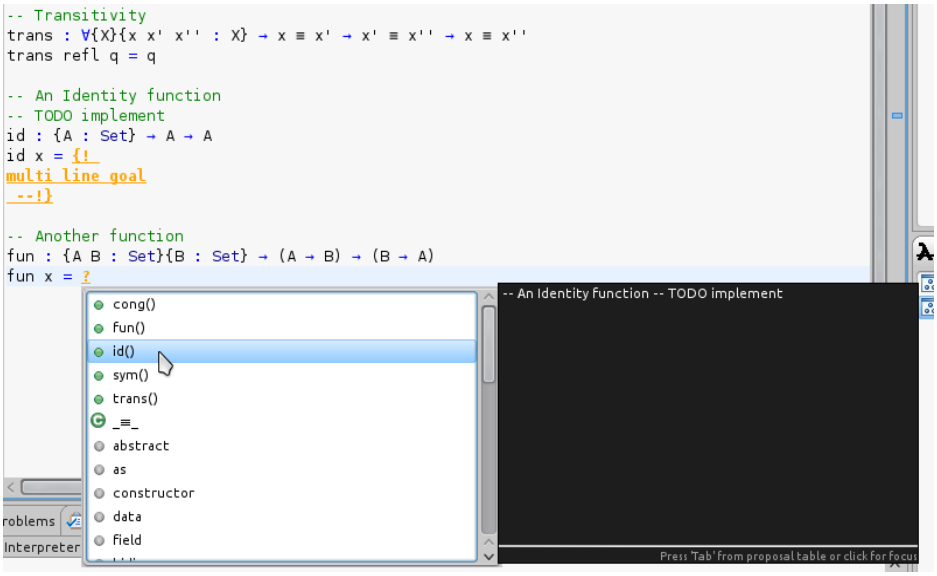

Figure 4.6: Content assist with a comment header

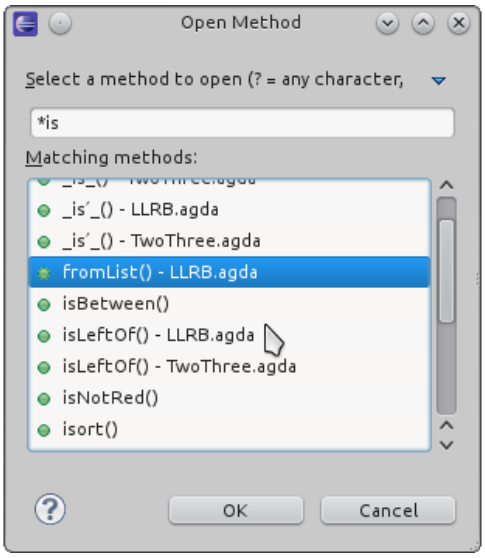

Figure 4.7: Open Agda function dialog

4.4.8 Configuration

A separate menu item, named *Agda* is created in the Eclipse Preferences dialog, allowing for the configuration of different features that are related to Agda. Most important of the preferences are the *Interpreters* and *Standard library* menu items, as these denote the platform-dependent preferences essential for using the IDE. Interpreters page shows a list of suitable interpreters for running Haskell programs and standard library should point at the Agda standard library location. The implementation of preference pages is mostly based on the corresponding preference page components provided by DLTK. Agda preferences and the preference page for syntax highlighting in particular can be seen on Figure 4.8.

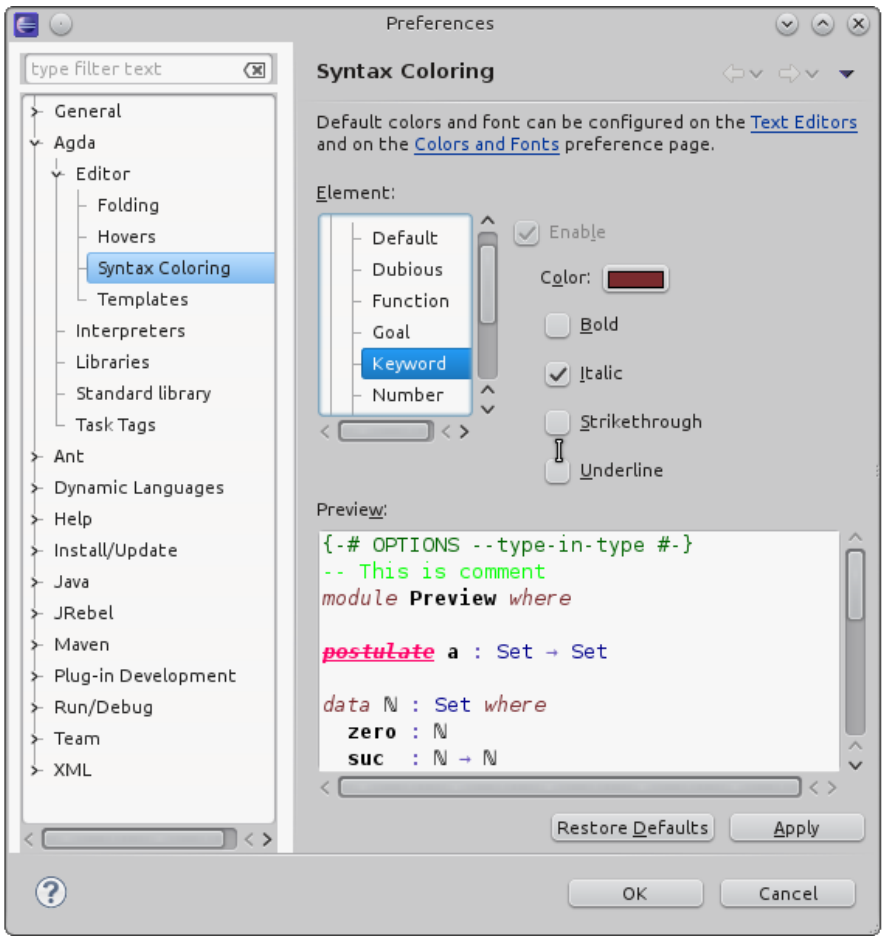

Figure 4.8: Agda Preferences. Customization of syntax coloring.

4.5 Current Status

The project is currently still in it's early stages and while it seems relatively stable and is usable for less complex tasks it is not mature enough for everyday work. Some of the features are included more as a proof-of-concept than as a fully developed implementation and in its current state the project might be more interesting for IDE developers or Agda developers rather than the actual Agda users.

There are also some known issues with the application. Only informative actions, that do not change the active file, are supported in interacting with Agda proof engine. This excludes `auto` and `refine` for example. This also means that changing the document requires full reload on the Agda side, which can be very slow. Another issue in the parsing phase is that parsing the temporary file for the semantic information can fail in some of the cases, as the file still uses the Emacs Lisp format from Agda Emacs mode interaction and the current solution for parsing a list of Emacs Lisp tokens in Java is rather poorly done.

A notable obstacle in improving the integration between Agda and Eclipse is that the Agda libraries are currently evolving very rapidly. In the presently unreleased development version of Agda, the changes to Agda Emacs mode include the removal of dependency to Haskell's Emacs mode and the replacement of highlighting based on temporary files with that of directly streaming highlighting information. While the changes themselves are also useful for integration with Eclipse, the API is currently not stable enough to implement some more complex solutions. On the other hand this makes it more likely to have support for features that would benefit an Eclipse-based IDE like having an implementation-independent API for IDEs, replacing all of the filesystem-based IO with streaming of file contents and support for working with many files in parallel.

Latest developments to this project are available from the repositories listed in Appendix B.

Conclusion

The goal of this thesis was to design and implement an Eclipse-based IDE for the dependently typed programming language Agda. Agda, Eclipse Platform and tools supporting the creation of Eclipse-based IDE were studied as a part of the thesis and based on the results an IDE was developed, comprising of a set of Eclipse plug-in supported by the Dynamic Languages Toolkit framework and a Haskell-based library to mediate the communication between Eclipse plug-ins and the Agda proof engine.

The resulting IDE supports creating of Agda project and files, has an Agda-specific perspective, syntax highlighting, support different queries in the context of the goal, simple navigation to declarations, searching, content assist features and problem marking.

Future work

Implemented solution, at it's current state, presents only a small subset of a full-featured development environment and in addition to stabilizing the existing implementation, some major improvements can be done both to the Haskell libraries and Eclipse plug-ins.

Developments that would benefit this IDE and could be implemented to the Haskell-based APIs for Agda include:

- a generic API for IDEs
- using some universal serialization format like xml or json,
- support for streaming the contents of the file instead of using the file system,
- support for working with multiple files in parallel.

Some of the areas of the implemented Eclipse plug-ins that should be improved include:

- improvements to the interaction plug-ins to cover the same set of actions as Emacs mode for Agda, including also actions that change the document, working inline in the editor instead of using the goals view and support of keyboard bindings,
- improvements to the AST and language model, possibly use some parser generator or methods for interacting with the existing Haskell-based AST
- support for unicode input.

Eclipse'i põhine integreeritud arenduskeskkond programmeerimiskeelele Agda

Bakalaureusetöö (6 EAP)

Kristjan Vedel

Resümee

Antud töö eesmärk oli kavandada ja implementeerida Eclipse'i põhine integreeritud arenduskeskkond (*IDE*) sõltuvate tüüpidega funktsionaalsele programmeerimiskeelele Agda. Töös vaadati lähemalt Agdat, Eclipse Platformi ja Eclipse'i põhiste arenduskeskkondade loomise raamistikke ning selle põhjal implementeeriti viis DLTK (*Dynamic Languages Toolkit*) raamistikule tuginevat Eclipse'i pistikprogrammi ning Haskelli teek, mis vahendab suhtlust Eclipse'i pistikprogrammide ja Agda vahel.

Implementeeritud IDE toetab Agda projektide ja failide loomist Eclipse'is, sisaldab Agda-spetsiifilist perspektiivi (*perspective*), süntaksi esiletõstmist, toetab erinevaid päringuid eesmärgi (*goal*) kontekstis, navigeerimist deklaratsioonide juurde, otsingut, sisu assisteerimist (*content assist*) ja probleemide markeerimist.

Bibliography

[1] L.Cardelli *Type Systems*, In *The Computer Science and Engineering Handbook*, A.B.Tucker, Jr., Ed. CRC Press, 1997.

[2] M.Hofmann *Syntax and semantics of dependent types*, In A.M.Pitts and P.Dybjer, editors, *Semantics and Logics of Computation*, Publications of the Newton Institute, pp. 79–130. Cambridge University Press, 1997.

[3] T.Altenkirch, C.McBride, J.McKinna *Why dependent types matter*, Manuscript, 2005.

[4] P.Martin-Löf *Intuitionistic Type Theory*, Bibliopolis, Napoli, 1984.

[5] B.Nordström, K.Petersson, J.Smith *Programming in Martin-Löf's type theory: an introduction*, Oxford University Press, 1990.

[6] U.Norell *Towards a practical programming language based on dependent type theory*, PhD thesis, Chalmers University of Technology, 2007.

[7] F.Lindblad, M.Benke, *A Tool for Automated Theorem Proving in Agda*, 2006, Lecture Notes in Computer Science 3839/2006, pp. 154–169.

[8] S.Foster, G.Struth *Integrating an Automated Theorem Prover into Agda* In: M.Bobaru, K.Havelund, G.Holzmann, R.Joshi, editors, *NASA Formal Methods*, 2011. LNCS, vol. 6617, pp. 116–130. Springer (2011)

[9] K.Kanso, A.Setzer *A Light-Weight Integration of Automated and Interactive Theorem Proving*, 2011, Under consideration for publication in Math. Struct. in Comp. Science

[10] *The Agda Wiki*, `wiki.portal.chalmers.se/agda/` (2012)

[11] C.Coquand, D.Synek, M.Takeyama *An Emacs interface for type directed support constructing proofs and programs*, ENTCS 2006.

[12] *EclipseFP*, http://eclipsefp.sourceforge.net/.

[13] D.Winterstein, D.Aspinall, C.Lüth, *Proof General/Eclipse: A generic interface for interactive proof.* In: International Workshop on User Interfaces for Theorem Provers, ENTCS, 2005.

[14] *Eclipse*, http://www.eclipse.org/ (2012)

[15] *Eclipse Platform: Technical Overview*, 2003, http://www.eclipse.org/whitepapers/eclipse-overview.pdf (2012)

[16] OSGi Alliance. OSGi Service Platform core specification, release 4, August 2005.

[17] http://www.jdg2e.com/ch08.architecture/doc/index.html (retrieved 23.04.2012)

[18] P.Charles, R.M.Fuhrer,S.M.Sutton Jr., E.Duesterwald, J.J.Vinju, *Accelerating the creation of customized, language-Specific IDEs in Eclipse*, In: S.Arora, G.T.Leavens, editors, *Proceedings OOPSLA 2009*, pp. 191–206. ACM, New York (2009)

[19] *Xtext*, http://www.eclipse.org/Xtext/ (2012)

[20] *IMP: The IDE Meta-Tooling Platform*, http://www.eclipse.org/imp/ (2012)

[21] *Dynamic Languages Toolkit (DLTK)*, http://www.eclipse.org/dltk/ (2012)

[22] *A guide to building a DLTK-based language IDE*, http://wiki.eclipse.org/A_guide_to_building_a_DLTK-based_language_IDE (2012)

[23] M.Scarpino, N.A.Good *Build an Eclipse development environment for Perl, Python, and PHP*, http://www.ibm.com/developerworks/opensource/tutorials/os-eclipse-octave/ (2012)

[24] *Schemeide*, http://schemeide.sourceforge.net/ (2012)

[25] B.D.O.Medeiros *Creation of an Eclipse-based IDE for the D programming language*, 2007 https://dspace.ist.utl.pt/bitstream/2295/149465/1/thesisdoc (2012)

[26] L.Frenzel *Experience report: building an eclipse-based IDE for Haskell*, ACM SIGPLAN Notices, Vol. 42, No. 9. (2007), 222

[27] *The Haskell Cabal*, `http://www.haskell.org/cabal/` (2012)

Appendix A

User Guide

Installation

Installation consists of two separate procedures as the cabal package and Eclipse plug-ins have to be installed separately.

Prerequisites

- The Haskell Platform (or at least Haskell and Cabal):
  `http://hackage.haskell.org/platform/`
- Agda:
  Currently a snapshot of a development version of Agda is required for the AgdaConsoleProxy cabal package. This snapshot is available from:
  `https://bitbucket.org/kvedel/agda2.3.1snapshot`
  After the release of Agda 2.3.2, AgdaConsoleProxy will depend on the last official release, available at:
  `http://wiki.portal.chalmers.se/agda/agda.php?n=Main.Download`
- Eclipse, with version $\geq$ 3.7: `http://www.eclipse.org/downloads/`

Installing AgdaConsoleProxy cabal package

- Check-out or download AgdaConsoleProxy:
  `https://bitbucket.org/kvedel/agdaconsoleproxy`
- Navigate into agdaconsoleproxy directory and run `cabal install`

Installing AgdaEclipse Eclipse plug-ins

- Open Eclipse
- In Eclipse choose **Help** → **Install new software** ... → **Add** ...
- Add the update site: `https://bitbucket.org/kvedel/org.bitbucket.agdaeclipse.updatesite/src/tip/`
- Select the update site and install **AgdaEclipse**.
- Open **Window** → **Preferences** → **Agda**
  - Select **Interpreters** and choose either **Add...** or **Search...** to add a Haskell interpreter like `runhaskell`.
  - Select **Standard library** and set the path to Agda standard library. Default path is `/usr/share/agda-stdlib`.

Getting Started

Next steps, after AgdaEclipse has been successfully installed, is to open Agda Perspective in Eclipse, create a new Agda project and an Agda file.

- Open Agda perspective:
  - Select **Window** → **Open Perspective** → **Other...** → **Agda**.
- Create a new Agda project:
  - Open dialog for creating a new Agda project by selecting **File** → **New** → **Project** → **Agda** → **Agda Project**
  - Enter a name for the project and select **Finish**.
- Create a new Agda file:
  - Open dialog for creating a new Agda file by selecting **File** → **New** → **Other...** → **Agda** → **Agda File**
  - Enter a name for the file and select **Finish**.

Module declaration is created automatically based on the name of the file and file is also loaded into Agda proof engine, so development of this new program can begin.

Appendix B

Source Code Repositories

The complete source code for the latest development versions of all components can be found from the following repositories:

- *Core*
  `https://bitbucket.org/kvedel/org.bitbucket.agdaeclipse.core/`
- *UI*
  `https://bitbucket.org/kvedel/org.bitbucket.agdaeclipse.ui/`
- *Core.Interaction*
  `https://bitbucket.org/kvedel/org.bitbucket.agdaeclipse.core.interaction/`
- *UI.Interaction*
  `https://bitbucket.org/kvedel/org.bitbucket.agdaeclipse.ui.interaction/`
- *Launching*
  `https://bitbucket.org/kvedel/org.bitbucket.agdaeclipse.launching/`
- *Feature*
  `https://bitbucket.org/kvedel/org.bitbucket.agdaeclipse.feature/`
- *Update Site*
  `https://bitbucket.org/kvedel/org.bitbucket.agdaeclipse.updatesite/`
- *Implemented Haskell modules*
  `https://bitbucket.org/kvedel/agdaconsoleproxy/`

The source code for all the components is available under the Eclipse Public License `http://www.eclipse.org/legal/epl-v10.html`.

Appendix C

Source Code on a CD

Attached to this thesis is the CD containing the source code for the implemented IDE and a digital copy of this thesis.