An Emacs-Interface for Type-Directed Support for Constructing Proofs and Programs.

Catarina Coquand [1]

Department of Computer Science
Chalmers University of Technology and University of Göteborg
Sweden

Dan Synek [2]

Institute for Computing and Information Sciences
Radboud University Nijmegen
The Netherlands

Makoto Takeyama [3,4]

Research Center for Verification and Semantics
National Institute of Advanced Industrial Science and Technology
Japan

Abstract

This paper presents an emacs interface for an interactive editor for proofs and programs. The interface allows for editing proofs in the same way as we write programs, but in addition offers commands that help term construction. It differs from most other proof editors for its support for direct construction of terms rather than tactics for building them.

Key words: User interface, interactive proof editor, type theory, emacs

1 Introduction.

Agda is an interactive system for incrementally constructing proofs and programs in Martin-Löf's type theory extended with records and modules. The

[1] Email:catarina@cs.chalmers.se
[2] Email:synek@cs.ru.nl
[3] Email:makoto.takeyama@aist.go.jp
[4] This author thanks CREST/JST (Japan Science and Technology Agency) for support.

system is built for direct manipulation of proof-objects and not based on tactics as most other proof-assistants. It builds on the long-standing development of the family of proof-assistants that includes ALF[10], which pioneered advanced graphical interactions for proof construction. Agda has two interfaces: the graphical, Alfa[9], supports structural editing driven by mouse and shortcut keys, guiding menus showing all possible choices, alternative display styles such as a natural deduction style, use of symbol fonts, systematic hiding of details, etc. The other is the text-based emacsagda this paper presents.

The text interface was built to meet the need for an interface that is close to the way we write programs. We wanted an interface where we can edit the code in the usual way but with a help in constructing terms just when it is asked for. One reason is that, for advanced users, a structural editing (graphical or not) can be something of a straight-jacket compared to a familiar advanced text-editors like emacs. When they know what they want, it is often quicker to type it in than to construct it stepwise. When they revise a large proof, they utilize global edit-commands cutting across structure boundaries. Advanced users seems to appreciate such raw but liberal editing even if they forgo nicer guidances and presentations. Examples of big developments done using the emacs-interface can be seen in [7,8,6,5,4]

The interface we present supports incremental constructions of proofs and programs via what we call communication points (also called meta-variables). These stands for holes in an *incomplete* proof or program that need to be filled to make it complete. Communication points are a legitimate part of the syntax extended for incomplete terms, which have correspondingly extended typing rules.

Users construct terms by filling those communication points. It can be done directly, or it can be incremental, in that users can fill a communication point with another incomplete term with new communication points. This is *refining* the goal of filling in the original to new subgoals. Such an proof-editing operation is immediately type-checked, and accepted only when it is well-typed.

Beside editing, users can ask for information helpful for construction at a communication point: its type, its context (identifiers in scope with types), the value of an expression in the context, etc.

Users can freely edit at any communication point regardless of when it is created or where in a proof it occurs, even if the proof is split across multiple files. This is unlike most command-line interfaces that force a default order to work on subgoals, requiring an explicit user commands to change the order.

The interface also provides the usual conveniences of a programmers' editor: keyword highlighting for increased readability, automatic indentation for layout-sensitive syntax, etc.

The interface we present is `agda-mode`, the emacs-mode for Agda. It is written in only about 850 lines of emacs lisp. The protocol between it and the proof engine is a simple string-based one. `agda-mode` sends command strings

to the engine, and the engine answers with strings telling `agda-mode` how communication points should be managed and displayed. There is basically nothing specific about the syntax or semantics of the Agda language in this. So it requires little to no changes to re-target this to another proof language, so long as its proof-engine has a command-line interface similar to that of Agda.

To understand how the system works we will present a simplification of Agda that we think captures much of the features of the system and how one interacts with it. We present a simply typed *lambda*-calculus with communication points and presents a possible implementation of a system for interactively constructing λ-terms. This language only serves to introduce the ideas of the system and has no theoretical status. The implementation that we propose has not been actually implemented, again it only serves as illustration of the full system.

Related to this work we have the Epigram system[11] developed by Conor McBride. Epigram is also inspired by Alf and supports direct manipulation of proof terms. It has an advanced syntax-directed emacs-interface with 2-dimensional layout. The main difference is that in Epigram the proof engine captures every key stroke from the user, instead of only exchanging information on buffers and communication points. This makes it possible for the proof engine to completely control the behavior of the system but stops the user from using standard features of emacs. Other related interfaces to type-theory proof-engines that support tactics-based interaction includes [1,2,12,3]

The emacs-interface was originally developed by Dan Synek and was later improved upon and rewritten by Makoto Takeyama. A modification of the interface to XEmacs has been done by Anton Setzer. The first version of Agda was developed by Thierry Coquand in Gofer. He has also been involved in all later developments. A second version of Agda, written in C, was developed by Dan Synek. The current version is written in Haskell and is implemented by Catarina Coquand and later improved upon by Makoto Takeyama.

2 Using the System.

For a user, Agda proof-assistant appears as emacs buffers using the major mode `agda-mode` for editing agda code. In part it is similar to other programming language modes, but what makes it a proof-assistant is that it supports, but not impose, the refinement-style code-development as explained below.

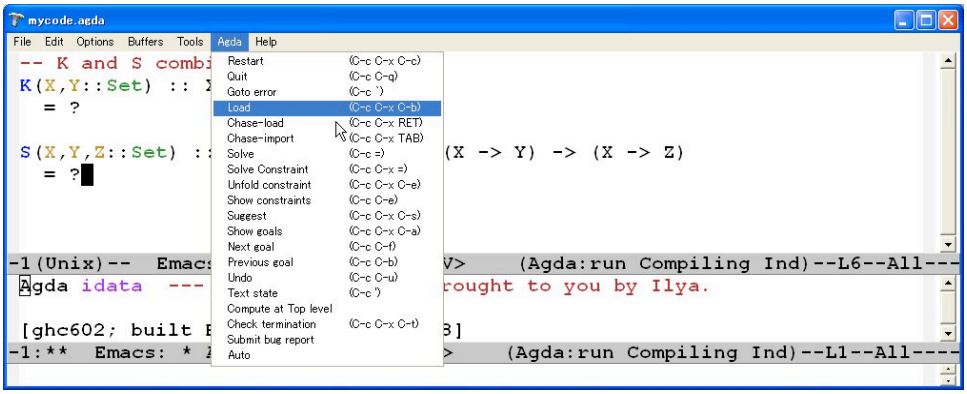
Buffers in `agda-mode` can be in either **Text-state** or **Interaction-state**. Initially the buffer is in Text-state. In Text-state the text can be freely edited and the interaction is just the usual emacs editing. In Interaction-state, communication points appear in the buffer and type-correct editing operations at the communication points become available. It is also possible to edit the text in Interaction-state, except that the communication points can not be removed. At first we had the restriction that no text editing were allowed in

Interaction-state other than the type-correct ones. However users found this to be inconvenient: for example, if they want to add a comment, it should not be necessary to re-typecheck the buffer. Of course, this means that the buffer, although it interacts with the proof-engine, need not be well-typed. If the user wants to avoid this situation s/he could toggle to Text-state before doing any changes. In any case typing error will be detected when the user re-typechecks the buffer.

2.1 General Commands

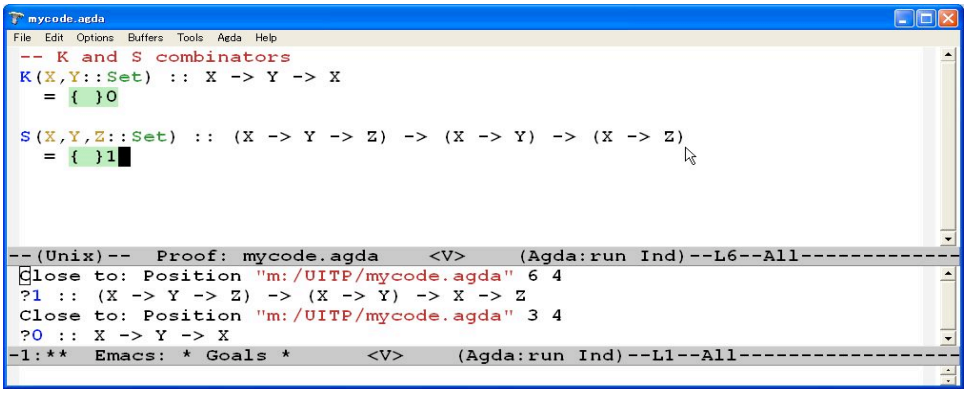
There is one major menu that is available in both states. This menu contains general commands that are not connected to any particular communication point: type checking the whole buffer, toggling between the two states, printing the type of all communication points, the undo-command, jumping around communication points, jumping to last reported error.

Example 2.1 Example of Text-state and the major menu. We can also see that the commands has short-cut-key bindings so that the user is not forced to use the mouse.



The text in the buffer is typically a list of definitions, $c = e :: \alpha; \ldots$. Both the expression e and the type α may contain communication points. These are written as `?` or `{!` *some text* `!}`. In Text-state, they have nothing to do with the proof-engine. To let the proof-engine know about them and get access to these communication points, the user `Load`s the buffer to the engine. This typechecks the buffer and if the buffer is well-typed, the buffer changes into Interaction-state. The communication points are now displayed highlighted and indexed. We can also see the type of all communication points in another buffer. It is possible to typecheck several buffers all of which might contain communication points.

Example 2.2 Here we can see how the text buffer above now is in Interaction-state and the communication points are highlighted.

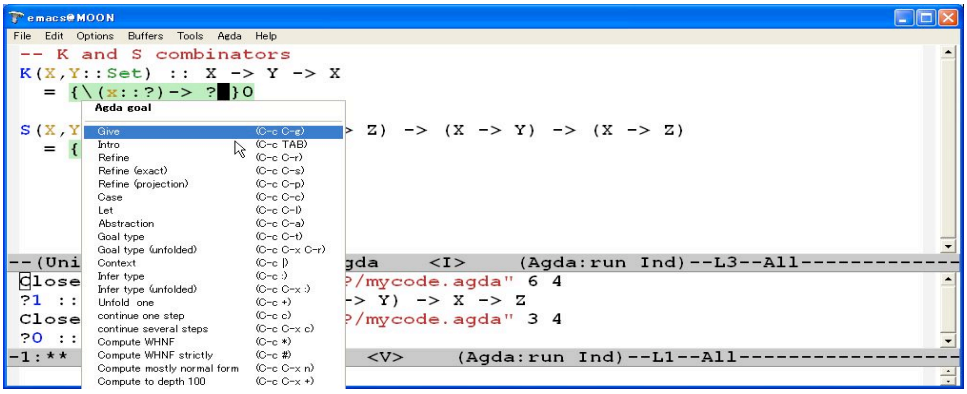


The undo-command will in Text-state behave just as the undo of emacs, i.e. it goes back in the editing history. If the last action in Interaction-state was a editing command, then the undo-command will undo that typing. If instead the last command did an update of a communication point, that update will be undone and the communication point is recreated.

2.2 Commands on Communication Points

Right clicking in a communication points pops up a new menu for commands acting on that particular communication point. We can now interact with the system to get help filling in the remaining parts of the proofs/terms and statements/types.

Example 2.3 This shows the menu for commands acting on communication points.



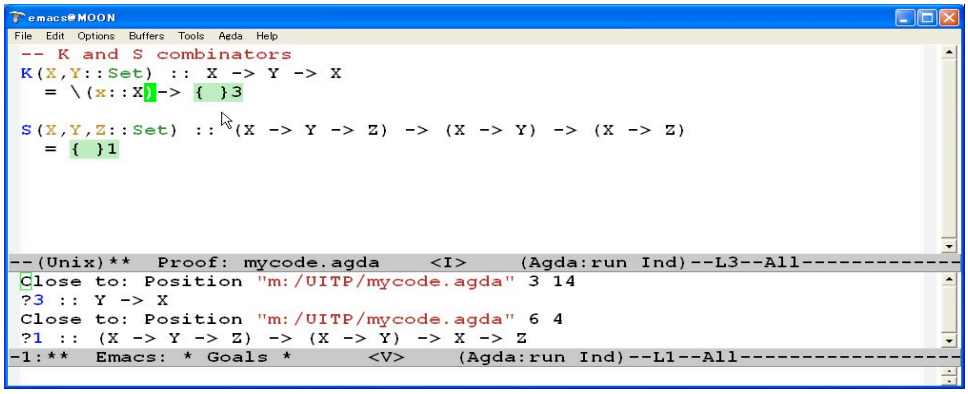
Examples of commands that can be performed at a communication point $?_i$ whose type is α are:

- **give.** A term e, which may contain communication points, is given by the user. The proof engine checks if $e : \alpha$ (see Section 3), and if so $?_i$ is updated to e. The communication points in e now becomes highlighted and indexed.
- **refine.** A term e is given by the user. The proof-engine repeatedly tries

`give` with terms e, e `?`, e `? ?`, $\cdots$ with increasingly many communication points until it succeeds or reaches a preset limit. Using the example 2.2, and a communication point with type $X \to Y$, then performing the command refine on `K` will update the communication point with the term `K ? ? ?`.

- **abstract** Given a list, `x1 x2 ... xn` the command does `give` with the term `\(x1::?) -> ... -> \(xn::?) -> ?`.
- **type, context.** The `type`-command prints the type of the communication point α and the `context`-command prints the type of bound variables and the defined identifiers that are in scope.
- **infer type.** This command infers and prints the type of an expression given by the user in a communication point. The communication point is not affected by this commands.

Example 2.4 Below we see the effect of the command `give` in example 2.3.



3 The Proof Engine

As the emacs-interface is basically only taking care of sending strings between the user and the proof-engine, we think it's important to understand how the proof-engine is working to understand how the whole system with the emacs-interface and the engine is interacting. The existing proof engine is an implementation in Haskell of a dependently typed functional language with data-types, records and modules. For the presentation of the engine we restrict ourselves to a simply typed λ-calculus with meta-variables (communication points). The main technical problems we avoid by this simplification is that we do not need to talk about evaluation of terms while type-checking and also constraint-solving is significantly simpler.

The simply typed λ-calculus with meta-variables has the following (incomplete) concrete syntax:

Types, denoted by `A, B`:	`o` | `A -> B` | `?`
Expressions, denoted by `t,tn`:	`(t)` | `x` | `\(x::A) -> t` | `f t1 ... tn` | `?` | `{!` *some text* `!}`
Definitions, denoted by `d` :	`f :: A = t`

Where `x`, `f` are identifiers. Lists of definitions are separated by semi-colon.

The proof engine consists of a state and operations on this state. What is in the state and the operations will be indicated below. The implementation has one module that defines the interface to the proof-engine. The graphical interface and the emacs-interface, both uses this interface.

The engine also keeps a list of old states, which is used to undo operations on the state. This is implemented as an ordinary list in Haskell, which might seem very space consuming, but we get sharing of common data structures thanks to Haskell being a functional language. We have not seen any problems with this solution in practice.

3.1 Internal Representation of Terms and Typechecking

We define a simply typed λ-calculus with meta-variables, which can be seen as communication points. Meta-variables are written as $?_k$. A meta-variable $?_k$ can only occur at one place in an expression. This restriction is not essential for this simply typed case, but makes the dependently typed calculus simpler. Types, expressions and definitions are defined as:

$$\begin{aligned} \alpha, \beta, \gamma &::= o \mid \alpha \to \beta \mid ?_k \\ e &::= \lambda(x : \alpha).e \mid x \mid e\ e \mid ?_k \\ d &::= f = e : \alpha \end{aligned}$$

where x and f are unique identifiers. The definitions are not recursive, they are just abbreviations.

The forms of judgements are

- $\Gamma \vdash f = e : \alpha$ correct definition
- IsType α correct type expression
- $\Gamma \vdash e : \alpha$ type checking
- $\Gamma \vdash e \Rightarrow \alpha$ type inference
- $\alpha = \beta$ type equality

where Γ is a context $[x_1 : \alpha_1, \ldots, x_n : \alpha_n]$, are:

$$\frac{\text{IsType } \alpha \quad \Gamma \vdash e : \alpha}{\Gamma \vdash f = e : \alpha} \qquad \frac{}{\vdash \text{IsType } o} \qquad \frac{\text{IsType } \alpha \quad \text{IsType } \beta}{\text{IsType } \alpha \to \beta}$$

$$\frac{\Gamma \vdash e \Rightarrow \beta \quad \alpha = \beta}{\Gamma \vdash e : \alpha}$$

$$\frac{x : \alpha \in \Gamma}{\Gamma \vdash x \Rightarrow \alpha} \qquad \frac{\text{IsType } \alpha \quad \Gamma, x : \alpha \vdash e \Rightarrow \beta}{\Gamma \vdash \lambda(x : \alpha).e \Rightarrow \alpha \to \beta}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow \alpha \to \beta \quad \Gamma \vdash e_2 : \alpha}{\Gamma \vdash e_1\ e_2 \Rightarrow \beta}$$

$$\frac{}{o = o} \qquad \frac{\alpha = \alpha' \quad \beta = \beta'}{\alpha \to \beta = \alpha' \to \beta'}$$

These syntax-directed rules defines a typechecking algorithm for expressions without meta-variables. Now we add the clauses for the new forms of the derivation leaves:

$$\frac{}{\Gamma \vdash ?_k : \alpha} \qquad \frac{}{\vdash \text{IsType } ?_k} \qquad \frac{}{?_k = \alpha} \qquad \frac{}{\alpha = ?_k}$$

These leaves are called constraints, since they put constraint on the possible solutions to the meta-variables. In these cases the constraints will be saved in the proof-state. Observe that we do not have type-inference of meta-variables. This would be possible by letting the type of the meta-variable being a fresh meta-variable, but this complicates the presentation.

A list of definitions $f_0 : \alpha_0 = e_0, \ldots, f_n, : \alpha_n = e_n$ is correct if $[] \vdash e_0 : \alpha_0, [f_0 : \alpha_0] \vdash e_1 : \alpha_1, \ldots, [f_0 : \alpha_0, \ldots, f_{n-1} : \alpha_{n-1}] \vdash e_n : \alpha_n$.

Example 3.1 We show how the state will be updated with constraints when the following list of definitions is typechecked:

$$f : \ ?_0 \to ?_1 = ?_2, g : o \to ?_5 \to o = f, \ h : o \to o = \lambda(x : ?_3).g\ ?_4\ x$$

Typechecking the definition of f adds the constraints:

$$[] \vdash ?_2 : ?_0 \to ?_1, \text{IsType } ?_0, \text{IsType } ?_1$$

to the state. Then typechecking g will add the constraints

$$?_0 = o, ?_1 = ?_5 \to o$$

and for the last definition we will add

$$\text{IsType } ?_3, [x : \ ?_3, g : o \to ?_5 \to o, f : \ ?_0 \to ?_1] \vdash ?_4 : o, ?_3 = o, ?_3 = ?_5$$

3.2 The Proof State and its Operations

The proof state consists of the following parts

- A list of definitions that has been typechecked.
- Constraints as described above 3.1
- A map from meta-variables to their local symbol table and precedences. In this simplified language the precedence simply tells if the meta-variable occurs in an application or in the left hand side of the arrow type.

There are two basic commands for updating the state, substituting a concrete expression for a meta-variable and adding a definition. For substituting an expression, `t` for a meta-variable, $?_k$, we perform the following steps:

(i) Lookup the symbol table of $?_k$ and translate the concrete term `t`, to an internal expression e. All identifiers in `t` are translated to their unique representation. The translation also adds fresh index to all meta-variables in `t` and save the symbol tables of the meta-variables in the expression.

(ii) Lookup the typing constraint for $?_k$ (there can only be one since meta-variables can only occur once.) If the constraint is $\Gamma \vdash ?_k : \alpha$, then we typecheck $\Gamma \vdash e : \alpha$. It is however not enough that the expression is type correct, we must also verify that the equality constraints are not violated. If we have a constraint $?_k = e'$, then we check that $e = e'$, which in turn can give rise to new equality constraints. If the typechecking and the equality checks succeeds, we perform the substitution. (In the actual implementation the substitutions aren't performed, they are simply recorded.) The case when the constraint is IsType $?_k$ is similar.

(iii) The result of this operation will be the indexes that were given to these new meta-variables in `t`, these indexes are used for the synchronization between the engine and the interface as will be described in 4

The second basic command is the addition of a definition `f :: A = t .` The definition will be translated into the internal representation, $f' = e : \alpha$ using the global symbol table of the state. The meta-variables in the types and the expression (if any) will get fresh indexes. We then check that that $\Gamma \vdash f = e : \alpha$ where Γ is the context we obtain from the list of global definitions in the state. The result is the indexing of the meta-variables in the order they occur in `f :: A = t .`

Given these commands we can derive commands like typechecking a buffer (list of definitions), `give`, `refine`, and `abstract`. We show how these commands work with two examples:

Example 3.2 Now we perform the command `give 0 o` in example 3.1, i.e. we want to substitute $?_3$ with the type o. We look up the typing constraint of $?_3$ which is IsType $?_3$, which holds since IsType o. What remains is to check the equalities, $?_3 = o$ which is true when substituting o for $?_3$ and then we check $?_3 = ?_5$ which will update the state with $o = ?_5$. Now we can substitute o for $?_3$ in the definitions. If we also perform `give 0 o` and `give 1 o -> o` we obtain the following list of definitions

$$f : \ o \to o \to o = ?_2, g : o \to ?_5 \to o = f, \ h : o \to o = \lambda(x : o).g \ ?_4 \ x$$

and the constraints

$$[] \vdash ?_2 : o \to o \to o, ?_5 = o, [x : o, g : o \to ?_5 \to o, f : \ o \to o \to o] \vdash ?_4 : o$$

Example 3.3 Given the previous example we now perform the command `refine 4 f`. We look up the type of $?_4$ in the constraints, which is o and then

the type of f which is $o \to o \to o$. We create two new meta-variables, $?_6$ and $?_7$, and we try to substitute $f\ ?_6\ ?_7$ for $?_4$. Typechecking

$$[x : o, g : o \to ?_5 \to o, f :\ o \to o \to o] \vdash f\ ?_6\ ?_7 : o$$

will give the new typing constraints

$$[x : o, g : o \to ?_5 \to o, f :\ o \to o \to o] \vdash ?_6 : o$$

and

$$[x : o, g : o \to ?_5 \to o, f :\ o \to o \to o] \vdash ?_7 : o$$

but no new equality constraints.

The result of this command should be the string that communication point 4 should be replaced with, in this case `(f ?  ?)`. The term is printed with parenthesis since we know by the precedence of $?_4$ that they are needed. Beside the string the command will also report the indexes of the meta-variables in the term, 6 and 7 (see 4 for a discussion on these indexes.)

Beside these commands regarding construction of terms we also have command `undo` command that given a index, i returns to the state i which is the state in position i in the list of former states.

It would also be possible to use constraint solving to automatically find out how to fill in certain communication points. We will not give the algorithm for that in this presentation, we just point out that this is done in the implemented system.

4 Emacs Interface Protocol

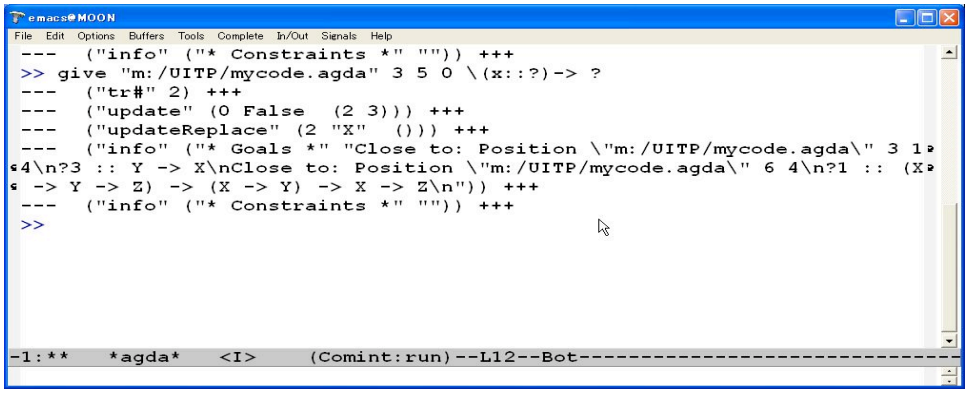
The protocol between the interface and the proof engine is a simple command language based on message passing of strings. The interface and the proof engine are synchronized in two ways, by the indexing of the communication points and the states.

We have chosen to synchronize the states explicitly, the states are indexed (the indexes are used for the undo) and every time a command change the state of the proof-engine it will report back the index of the new state to the interface. In this way the agda-mode does not need to know which commands changes the state of the proof engine.

The other synchronization is done on the indexing of the communication points. After a buffer is typechecked the proof engine will report back the indexes of the communication points of that buffer in the order they occur in the buffer. Observe that the users can not number the meta-variables themselves, the indexing is done by the proof engine. Also when a communication point is updated with a new term possibly containing meta-variables, the proof engine reports back the indexes of these meta-variables.

We will simply describe the protocol with a hopefully illustrating example.

Example 4.1 This example shows how the agda-mode and the proof engine interact on the example 2.3 above.



First we see the command `give` that is sent to the proof engine. The first three arguments represents the position of the communication point, then comes the index of the communication point, 0, and the string that the user wants to replace it with. The lines enclosed in `---` and `+++` following the command is output from the proof engine to be interpreted by the agda-mode. The first line gives the index of the new state. The second line says that the communication point 0 should be updated with the string given by the user, that it does not need parenthesis (the boolean `False`) and the indexing of the two new communication points, $?_2$ and $?_3$. The third line says that communication point 2 should be replaced with the string "X". The last two lines will make the agda-mode create two buffers, * Goals * and * Constraints *, containing the texts in the given string.

5 The agda-mode

The agda-mode is a single emacs library file `agda-mode.el` containing 850 lines of GNU emacs lisp. It uses a customized version of Lapalme's Haskell-mode for automatic indentation. The total time that has been spent on implementing and modifying the implementation we estimate to about two months of work. This being relatively small, we hope that it should be fairly easy to adapt it to other proof systems based on similar ideas, though we have not experimented yet with other systems.

The implementation consists of two parts. One part translates user actions in agda-code buffers (emacs buffers using `agda-mode` where agda-code is edited) to textual command inputs to the proof-engine. The other part processes outputs from the proof-engine and updates agda-code buffers accordingly.

When `agda-mode` is enabled, it starts up the proof-engine as a sub-process running in parallel, if it is not running already. The process is started in a separate buffer `*agda*` that uses `comint-mode` (command-interpreter-in-a-buffer) of Emacs, which is similar to the shell-mode. In a typical Agda session, several agda-code buffers share a single `*agda*` process.

Most user commands in agda-mode construct and send textual commands to the proof-engine, inserting them to the buffer `*agda*`. Doing so requires contextual information, e.g. in which communication point a command is invoked. For this, `agda-mode` in Interaction-state maintains annotations for communication points with 'text-properties' and 'overlay'. These are the emacs way of associating information with regions of text. A user command can look up the text property at the current cursor position and obtain the communication-point index-number in it. The annotation is also responsible for popping-up the communication-point menu on a mouse-click, displaying communication points in the highlight color and with index numbers, moving the cursor to next / previous communication points, and protecting them from deletion by non-agda-mode commands.

The annotation is created and maintained by the part of `agda-mode` that processes outputs from the proof-engine. The process buffer `*agda*` is set up with a hook function to be called whenever the proof-engine outputs something. The function examines the output and dispatches appropriate processing functions.

A typical output would tell that the proof-engine successfully typechecked a piece of code given in a communication point with index i, that the communication point should be replaced by a new piece, and that indices j and k are assigned to new communication points in the new piece. `agda-mode` then deletes the text range of the communication point i, insert the new piece, searches for the communication points j and k, and annotate them. The search simply looks for the syntax of communication points (`?` or `{!···!}`); this is the only syntax of the Agda language that `agda-mode` needs to know.

The undo operation in an agda-code buffer is somewhat complicated: the proof-engine must be kept synchronized, several changes made by output processing must be undone in one step if they resulted from a single user command, and these may span across several agda-code buffers other than the current buffer. Normally, emacs automatically records any undoable changes in an "undo list" for each buffer. Using this, `agda-mode` maintains its own extended undo list for each buffer, in which it can tell whether a change is made by output processing (or by the user), and if so, the state index of the proof-engine at the time the change was made. The undo command looks at the extended list of the current buffer. If the topmost change is for an output processing at state n, it repeats undoing in every agda-code buffer until all changes at state n are undone, and then sends to the proof-engine the command to go back to the state before n.

6 Conclusions and Future Work

We have presented a rather simple, but still powerful interface for incrementally building λ-terms. Using emacs as the base for the interaction, makes it simple for users to get started with the system, since emacs is a familiar tool for many. We think that this tool can be rather easily modified to fit other systems using type-based editing of programs or proofs, since it basically knows nothing about the underlying language.

There are however also some disadvantages with the system. One disadvantage is that it only offers help for constructing terms and not definitions. This is not a problem for the toy language presented in the paper, but is impractical in the full language. If we ,for example, want to write definitions in a pattern matching style the interface offers no help for constructing the patterns.

For future work we think that being able to interact with the proof engine via the the buffers it creates and not only via the communication points would be useful. One example is that today we can ask for suggestions for filling in a communication point. This will result in a buffer where possible completions are written, but we can not choose one of them. Instead we will have to cut-and-paste the solution into the communication point.

Another possible extension is dynamic menus which will depend on the type of the communication points or some other information. Having dynamic menus also helps when connecting the system with plug-ins, which might add new commands to the built-in ones.

References

[1] David Aspinall. Proof general - a generic tool for proof development. In *Proceedings of Tacas 2000*, LNCS 1785, pages 38–. Springer-Verlag New York, Inc., 2000.

[2] David Aspinall and Christoph Lüth. Proof general meets isawin: Combining text-based and graphical user interfaces. *Electronic Notes in Theoretical Computer Science*, 2004.

[3] Yves Bertot, Gilles Kahn, and Laurent Théry. Proof by pointing. In Masami Hagiya and John C. Mitchell, editors, *TACS*, volume 789 of *Lecture Notes in Computer Science*, pages 141–160. Springer, 1994.

[4] Ana Bove and Thierry Coquand. Formalising bitonic sort in type theory. *Submitted for publication*, 2005.

[5] Jan Cederquist. An implementation of the heine-borel covering theorem in type theory. In Eduardo Giménez and Christine Paulin-Mohring, editors, *TYPES*, volume 1512 of *Lecture Notes in Computer Science*, pages 46–65. Springer, 1996.

[6] Jan Cederquist and Sara Negri. A constructive proof of the heine-borel covering theorem for formal reals. In Stefano Berardi and Mario Coppo, editors, *TYPES*, volume 1158 of *Lecture Notes in Computer Science*, pages 62–75. Springer, 1995.

[7] Thierry Coquand and Henrik Persson. A proof-theoretical investigation of zantema's problem. In Mogens Nielsen and Wolfgang Thomas, editors, *CSL*, volume 1414 of *Lecture Notes in Computer Science*, pages 177–188. Springer, 1997.

[8] Thierry Coquand and Henrik Persson. Gröbner bases in type theory. In Thorsten Altenkirch, Wolfgang Naraschewski, and Bernhard Reus, editors, *TYPES*, volume 1657 of *Lecture Notes in Computer Science*, pages 33–46. Springer, 1998.

[9] Thomas Hallgren. Alfa, 2000. http://www.cs.chalmers.se/~hallgren/Alfa/.

[10] Lena Magnusson and Bengt Nordström. The alf proof editor and its proof engine. In *TYPES '93: Proceedings of the international workshop on Types for proofs and programs*, pages 213–237. Springer-Verlag New York, Inc., 1994.

[11] Conor McBride. Epigram, 2004. http://www.dur.ac.uk/CARG/epigram.

[12] Markus Wenzel. Isar - a generic interpretative approach to readable formal proof documents. In *Proceedings of TPHOL-99*, LNCS 1690, pages 167–. Springer-Verlag New York, Inc., 1999.