Proving with Refinements
$The LEGO_r - system$

Merijn van Erp

August 29, 1996

Contents

Acknowledgements

This master's thesis was written as part of my education at the University of Nijmegen, Faculty of Mathematics and Computing Science, Department of Foundations of Computing Science.

I want to thank Erik Barendsen, Henk Barendregt and Mark Ruys for their supervision and support.

I want to thank my family for their moral support during the last hectic weeks. And also thanks to my fellow students for the computer-games played, lifting the pressure a bit.

Merijn van Erp

Chapter 1

Introduction

This master's thesis will be about the process of proving. Aristotle already knew that though proving a statement might be very difficult (if not impossible), validating the proof of a statement is relatively simple. Imagine one has an informal proof of A in the mathematical context Γ, then one con formalize this into logic as $\{\Gamma\} \vdash \{A\}$, where $\{\}$ is the formalization. If one manages to formalize one's proof into lambda-calculus, this proof-checking can even be automated. One can formalize a proof $\Gamma \vdash A$ given in logic with a proof p into type-theory. One get's $[\Gamma] \vdash [p] : [A]$, where [ ] is the formalization. This means the statement A has been transformed into a type and the proof p now is the lambda-term inhabiting that type. But to formalize a proof, especially the long and complex ones, is quite a task itself.

This is where systems like LEGO [1] [2] come in. LEGO is a computer-program written to develop such formalized proofs. If one can formalize the statement and its context, LEGO will help constructing and verifying a proof for that statement. One can even start proving without an informal proof to be formalized. LEGO is a proof-development-system.

What LEGO is really all about, is finding lambda-terms (also called inhabitants in this context) for lambda-types. One formalizes one's statement as a lambda-type and one tries to find its inhabitant. If a inhabitant is found, that inhabitant is the formalized proof of the statement. The reason LEGO works so well is the relative simpleness of inhabitant-checking.

This all sounds quite usefull, but not everything is as nice as it could be. The problem with LEGO (and similar systems) is the bottom-up style needed when giving a proof. One has to specify all lemmas used prior to starting the proof. This is to impose some structure in the proofs (see also below). If one has no clear idea how the proof is going to be (which will be quite often), this is cumbersome. Every time one wants to isolate a lemma during the proving, one has to interrupt the proof-session and insert the lemma.

LEGO does support a kind of preliminary lemmas. One states the lemma as an axiom and later (after proving the main statement) one proves the lemma. A certain LEGO-command (Cut) will now replace all occurrences of the axiomized lemma by its proof. This does still need a lot of hacking when one wants to add a lemma in the middle of a proving-session.

Another problem of LEGO is the structuring of the proof. LEGO only support what we will call linear dependency. This means that in the proof of any statement (either the main statement or any lemmas) all statements can be used which were declared earlier in the text. Although it can be seen as a restraint, posing the above mentioned problem with stating lemmas, it is mandatory to make the proof readable. (Imagine a book with all proofs scattered all over, in any particular order.) However, it does still leave some references which can point way back to a distant, seemingly unimportant lemma.

We will try and solve these problems. We will use an interface called $LEGO_r$, which uses the text-editor Emacs, to deal with them.

The problem of the late lemma insertion will be countered by refinements. Refinements are lemmas which can be stated but need not be proven right away. The $LEGO_r$-system will regulate these refinements. They can be seen as automatized versions of the Cut-command.

The structure of LEGO will be extended with a new dependency, giving the structure a more two-dimensional look. The $LEGO_r$-system will impose a so-called vertical dependency. This is done with views (see below). This structuring makes it more clear what statements are more important than others. Referring back can only be done to statements which have enough importance. This might seem belittling, but it produces a clearly structured and readable proof. This helps enormously when one looks at a proof did not construct oneself.

In this document we will describe the various contributions we made to the LEGO-system, to make the $LEGO_r$-system. These are:

- The development of a system, based on LEGO, that worked with refinements and adds the view-structure.
- The proof that this system is as powerfull as the LEGO-system.
- The implementation of this system as $LEGO_r$.

Chapter 2

Concept of proving with refinements

One of the problems one might have with LEGO and similar systems is the use of lemmas. In LEGO a lemma has to be proven before one starts proving a theorem which uses that lemma. Most mathematical proofs however are conceived the other way around. First one starts with the idea for a theorem and then one attempts to prove this. While proving, auxiliary proofs are isolated into separate lemmas. Sometimes one immediately proves those lemmas, sometimes one finishes proving the theorem first. LEGO doesn't really support either way of working. It does have the Cut-command, but the proof-session has to be stopped to accept the axiomized lemma. So one has to rely on word-processor facilities, like paste, to work this way.

We wanted to overcome this problem and so we will introduce the concept of refinements. Refinements are lemmas which are formulated and used before they are proven. One could start proving a theorem and then one could come up with a lemma which would be really useful. With refinements one could claim the lemma and save its proof for later. At that time the claimed lemma will simply be substituted by a definition of the proof. One could prove the lemma right away too.

2.1 Proving with refinements

A refinement is a statement which can be used, before a proof is given. Until the proof of the statement is given any use of the refinement will be handled as a temporary axiom. If, at any time, the proof of the statement is produced, the axiom status of the refinement will be substituted by a conventional lemma status. We will further explain it by a typical example.

Example

Imagine one wants to prove the following: $\Gamma \vdash A$. While proving one finds a lemma $\Gamma \vdash B$ and a lemma $\Gamma \vdash C$. To prove those lemmas one might need sub-lemmas, let's say $\Gamma \vdash D$ for lemma B and both $\Gamma \vdash E$ and $\Gamma \vdash F$ for lemma C. Normally one would have to prove this in the following order:

1. $\Gamma \vdash u : D$
2. $\Gamma \vdash q : B$
3. $\Gamma \vdash v : E$
4. $\Gamma \vdash w : F$
5. $\Gamma \vdash r : C$
6. and finally $\Gamma \vdash p : A$.

This cannot be done this way unless one has a very good idea of how the proof is going to be. Most of the time one will be proving A and then one will find B and the others. This means that one has to interrupt the proving of A and prove the others first or one has to claim the lemmas and prove them before the entire proof of A is given (the last option is a LEGO option). This is either quite cumbersome or tends to become incomprehensible quite rapidly.

Our interface will use some of both methods while reducing their drawbacks. It will allow the proof of the theorem and lemmas as given above in any particular order. Moreover it will give a clear hierarchy in the dependency of the lemmas.

Our interface would do the above mentioned proof as follows:

1. One starts with proving A in context Γ.
2. One formulates lemma B and one refines it.
   The interface will now stop the proving, insert the lemma as an axiom and then it will allow the user to continue proving A but now with the possible use of the lemma. It will record the fact that the lemma is in an unproven state.
3. One formulates lemma C and one refines it.
   The system will treat this refinement of a lemma the same as the previous one. It will be inserted after the previous refinement.
4. One finishes proving A. One has constructed a p so that $\Gamma \vdash p : A$.
   $\Gamma \vdash p : A$ *has been proven correct relative to the lemmas B and C. Now all that's left to do is prove it locally. This means proving the refinements. The system will administrate this for the user.*

5. One can start proving B or C. The order does not matter. Let's assume one wants to prove B first.
The system will now allow the user to start proving the refinement. The context in which this is to happen, will already have been inserted. The system has the same functionality as in the proof of ordinary theorems, as explained before.

6. One formulates sub-lemma D and one refines it.
The system will treat this sub-lemma refinement exactly the same as a regular lemma refinement. It does not matter how often one keeps refining, the system treats it all the same (as it should do).

7. One finishes proving B relative to sub-lemma D.
This too happens exactly the same as proving the theorem relatively. The system will show only a relative proof has been given and will show the refinement(s) still to prove.

8. At this moment one might be interested in the status of the proof so far.
It is possible to ask the system to show which parts (theorem and refinements) of the proof have been proven (relatively or locally) and which parts are still to prove.

9. Once again the user can choose different paths to an local proof. The proof of C is still open, but the proof of D can also be started. Let's say the latter option is chosen.
A deeper level of refinement is started and again it's treated as before.

10. One proves D without any refinements.
The system notes this. The proof will be administrated as locally correct. Moreover as the last (and only) refinement of B has been proven, so the system will grant the proof of B with a sign of local correctness.

11. The only thing left is the proof of C.
This proof is treated analogously to the previous parts. The refinements of C will get the same treatment. A important detail which should be mentioned is the view during this proving. As said before, the system will create a hierarchy in the lemmas and theorem. This will cause a limited view as will be explained later. In this example it means that the result of $\Gamma \vdash q : B$ *can and the result of* $\Gamma \vdash u : D$ *can not be used in proving* $\Gamma \vdash r : C$ *and its refinements.*

12. After finishing the proof of $\Gamma \vdash r : C$ and its refinements the theorem has been proven globally.
The system will administrate this local correctness. If the files are used in LEGO now, the proof would be accepted but the context would still

contain the axioms used as refinements. This is done to prevent a breach in the view of the proofs, as will be explained later. One final command needs to be given to straighten this out. When the command is given, the system checks if the theorem has been proven locally and if so it will produce files that will be accepted by LEGO without the axioms in the context.

2.2 The View of a Refinement

In this paragraph we will explain the idea of views for refinements. In mathematics it is considered good style to limit the use of a sub-lemma. It can be very confusing if, for example, in proving theorem 5 a reference is made to lemma 2.3.16. A sub-lemma apparently needed only to prove one thing should not appear again elsewhere. This will be called the scope of the (sub-)lemma, the places where the (sub-)lemma are meant to be used. Outside the scope the lemma should not be used.

The view of a refinement consists of those other refinements that have this refinement in their scope. In other words, the view of a refinement are those other refinements whose statements may be used to prove it. To better explain the view of refinements in our system we will first explain the notion of the refinement-tree.

Let a numbering exist on the refinements and the theorem as follows:
We will number with sequences of natural numbers. The theorem will be numbered with the empty sequence. every refinement will be numbered as: If the refinement was defined as n^th refinement in a proof, it will be given the sequence of the statement of the proof with n-1 appended to it.

Example
If a refinement was 4^{th} in the proof of refinement 3.2.7, then it would be numbered 3.2.7.3

These sequences put the refinements in some kind of structure, which we will call the refinemnt-tree. The refinement-tree is a structure of refinements and relations between those refinements. The so-called top of the tree consists of the theorem and its axioms. All other nodes in the tree are the refined statements. A connection of to nodes means one of nodes contains a refinement given in the proof of the other. The refinemnts are put in the tree as follows:
On top is the theorem with sequence [ ]. There is a direct contact between two refinements of sequence s and s' if and only if s is shorter than s' and s is substring of s' placed at the start of s'. Now s will be placed higher than s'. The s is now called the parent and s' the child.

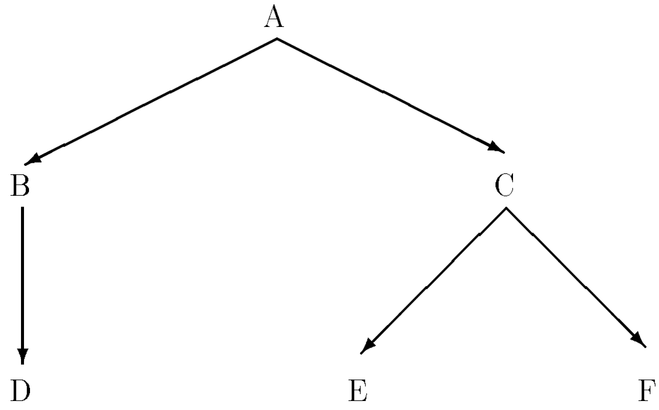

Figure 2.1: An example of a refinement-tree

By this convention the lower refinement is given in the proof of the higher one. Another important notion is the left-right order, which is not at random. If refinements are on the same level, the one on the left will be the one formulated earlier. As working with refinements gives one freedom in the order of proving and formulating, "formulated earlier" is meant in the conventional way of proving, without refinements. In sequences:

Let L(x), $x = [x_0, x_1, .., x_{n-1}]$, be n, the lenght of x. L([ ]) = 0. Let $x = [x_0, x_1, .., x_{n-1}]$ and $y = [y_0, y_2, .., y_{m-1}]$ be sequences and k = min(n,m), then $x =_k y \Leftrightarrow \forall_{i<k} x_i = y_i$. Now $x <_{seq} y \Leftrightarrow x =_{L(x)} y$. This means that sequence x is a starting-segment of sequence y. Define a new ordening on sequences called $<_{lego}$.
Let two sequences $x = [x_0, x_1, .., x_{n-1}$ and $y = [y_0, y_1, .., y_{m-1}]$ then $x <_{lego} y$ $\Leftrightarrow (\exists_{i<L(x)} x =_i y \wedge x_i < y_i]) \vee x <_{seq} y$
Now for all refinements with the same parent the left-right ordening is that of $<_{lego}$, if $x <_{lego} y$ then the refinemnt of x is to the left of the refinement of y. This ordening is the ordening LEGO uses. In LEGO a refinement with sequence r can use all other refinements with sequences s iff s $<_{lego}$ r.

Some other important relations will be explained below in an example.

Example

See figure 2.1. In this figure the refinement-tree of the proof in the previous example is given. The top node contains A, the theorem and its (possible) axioms. As both B and C are directly defined in the proof of A they are directly connected to it and below it. This relation is called parent-child with the higher node (A) being the parent of its lower children nodes (B and C). Ancestors are the set of parents and their parents. B

is parent of D. Both E and F are children of C. A is every refinements ancestor. Siblings are those refinements that have the same refinement as parent. Thus B and C are siblings and E and F are siblings too. An elder sibling is an earlier defined one (the left one). Thus B is older than C.

To define the view of a refinement we have to add three new definitions of relations on sequences:
Define $<_{child}$ for a $x = [x_0, x_1, .., x_{n-1}]$ and $y = [y_0, y_1, .., y_{m-1}]$ as:
$x <_{child} y \Leftrightarrow (n = m + 1) \wedge (x =_m y)$. This is the relationship parent-child as described before. the refinement of y is the parent and the refinement of x the child.

Define $<_{sib}$ for $x = [x_0, x_1, .., x_{n-1}]$ and $y = [y_0, y_1, .., y_{m-1}]$ as:
$x <_{sib} y \Leftrightarrow (n = m) \wedge x =_{m-1} y \wedge (x_m < y_m)$. This is the elder-sibling relationship as described before. The refinement of x is the elder sibling of the refinement y.
Let sequence $x = [x_0, x_1, .., x_{n-1}]$ then elder-siblings(x)= $\{y | y <_{sib} x\} \cup$ elder-siblings($[x_0, x_1, .., x_{n-2}]$). This is a recursive definition giving all the elder siblings of the refinement of x and its ancestors.

The last relation is $<_{LEGO_r}$. Let $x = [x_0, x_1, .., x_{n-1}]$ and $y = [y_0, y_1, .., y_{m-1}]$ be sequences, then $x <_{LEGO_r} y \Leftrightarrow x <_{child} y \vee x \in$ elder-siblings(y). The view of a refinement consists of the following other refinements:

- All elder siblings.
- All elder siblings of ancestors.
- Its children

Or in terms of sequences:
Let $x = [x_0, x_1, .., x_{n-1}]$ then View(x)= $\{y | y <_{LEGO_r} x\}$.

In a proof of a statement in $LEGO_r$ one can use the axioms of the top-node and all statements given in the view of the statement. In normal LEGO the view of x would have been all y with $y <_{lego} x$. The view of the theorem (top-node) are its children, the first refinements.

Example

See figure 2.1. The view of refinement E is $\{\Gamma, B\}$. F is not included, because it is younger than E. C is the parent and therefore not included. A is the theorem which is never included and D does not share parents with E and neither is it one of its ancestor's elder siblings.

This structuring is new in LEGO, though it does not harm the power of the proving ability of the program. It merely enforces the good use of scope. It does however give a clear picture of how refinements are related, thereby simplifying good use of scope and increasing the understandability of the proof.

Chapter 3

Provability of Refinements

In this section we will show that the power of a system proving with refinements is not greater (or less) than the power of a system without. To do this we need some definitions.

Definition: Relatively correct

Imagine a proof p of statement A, contained in a refinement with sequence r, in context Γ. The proof p is relatively correct, iff Γ can be extended with Γ_r, all elements of Γ_r must be elements of View(r), such that $\Gamma, \Gamma_r \vdash p : A$.

Definition: Locally correct

Imagine a proof p of statement A, contained in a refinement with sequence r, in context Γ. The proof p is locally correct, iff p is relatively correct and all (if any) refinements with sequence y $<_{child}$ r are locally correct.

In the refinement-tree this means that all leafs (nodes without children) are automatically local when they are proven relatively. A refinement is local if it is proven relativly correct and all its children are proven locally correct.

In $LEGO_r$ the user has to prove the relative correctness of a statement and if no refinements are used the system will conclude local correctness. If refinements are used, the local correctness of these has to be proven (by use of the system) before the system will grant a certificate of local correctness to the statement. (For those who do not trust the system, $LEGO_r$ will be able to produce a file without refinements of any locally correct proof. This file can then be fully checked by normal LEGO.) More formally, if one wants to prove A in context Γ with use of a lemma B :

- The user proves $\Gamma, l : B \vdash p : A$
- The user proves $\Gamma \vdash q : B$
- $LEGO_r$ will conclude $\Gamma \vdash p[l := q] : A[l := q]$

The first two points can happen in any order (although if switched no refinement needs to be used). The conclusion of the system is done automatically.

The following statements are made with just a sketch of a proof.

If the entire (finite) refinement-tree is relatively correct, it is also entirely locally correct.
Proof: See the definition of local corectness.

If the top of the refinement-tree is locally correct, the entire tree is absolutely correct or LEGO-correct. In that case LEGO will accept the proof.
Proof: If the top of the tree is locally correct, the entire refinement-tree is locally correct, because the top has no elder siblings. If everything is locally correct, nothing depends on unproven statements anymore. Thus it is an absolutely correct proof.

A theorem A is considered provable in $LEGO_r$, if it is at the top of the refinement-tree and is locally provable in $LEGO_r$. Due to many difficulties involved in checking for elder-siblings, this has not been implemented in $LEGO_r$. Thus it is only checked if the top is locally correct and that is why a theorem has to be on top of the tree to be provable.

Theorem: The system with refinements is no more or less powerful than the system without refinements.

Proof: This is a sketch of what the proof will be like. The system with refinements is at least as powerful as the system without, because all the refinements offer is something extra. One does not have to use refinements. Therefore any proof in the normal system will be accepted in the system with refinements.

It is not more powerful. For this proof we need the following substitution-lemma [3]:

Substitution lemma:

Suppose $\Gamma, x : B, \Delta \vdash M : A$ *and* $\Gamma \vdash N : B$. *Then* $\Gamma, \Delta[x := N] \vdash M[x := N] : A[x := N]$.

As the system makes refinements into axioms, the first and second supposition are the relative proof of a theorem with a refinement and the local proof of that refinement respectively. Therefore the system makes a legal substitution by replacing the axiom with the actual proof. Also becomes clear that nothing new has happened. So the system with refinements is indeed no more powerful than the system without. This means that both systems are equally powerful.

Chapter 4

The $LEGO_r$-system

In this chapter the $LEGO_r$-system will be fully explained. We will start with a few decisions about the general idea of the system. Then the workings of the system will be explained. The last part will contain the actual programming-description.

4.1 Why the $LEGO_r$-system?

The fact that LEGO was chosen above all other prove-checkers was based upon the use of LEGO on the University of Nijmegen. Although the Coq-system [4] was used as teaching-system, only LEGO was used in the academical research.

The first real decision to be made was in what kind of system the concept of refinements in LEGO could best be represented. After some searching four possibilities remained.

The first was the Emacs text-editor [5] which already supported a LEGO-interface written by Thomas Schreiber [6] in elisp [7]. This system allows the user to write a LEGO-file and test it at the same time.

The second option was LaTeXLEGO by Peter Aczel, Anthony Bailey and David Carlisle [8], a system to write LEGO-files and explaining text together in one file. The output will be one nicely readable file.

The third possibility was ASF+SDF developed at the University of Amsterdam [9], a system for generating language-specific environments from algebraic specifications of (programming) languages.

Fourth and last there was HTML [10], the well known hyper-text language.

As mentioned before, we chose Emacs and the LEGO-interface. In our research we had invested much time in this system so we knew it quite well. Also the system already worked with LEGO and promised to be easily extended. It is used at the University of Nijmegen.

LaTeXLEGO was still too experimental and it did not really support use of refinements. It is to text-based and it did not offer anything to support an interface.

Using ASF+SDF would expect LEGO to be fitted for the system. As Emacs already had an interface this would be redoing much work. ASF+SDF was very new to us and it would have taken much time to fully understand its use. It also had no connection with LEGO yet.

Roughly the same arguments can be used for HTML as can be used for ASF+SDF. The link with LEGO was not directly apparent and it would take too long to get into the language.

Now the base was there, the system had to be worked out. Should a new system be build based upon the general ideas given by Schreiber's interface or should that interface be extended? Although extension often takes more time than was expected, the building of a new system would probably take too long anyway. And the interface that was already there, seemed programmed clear enough to allow for an easy extension.

4.2 How to use the $LEGO_r$-system

In this section the workings of the $LEGO_r$-system will be explained. For the use of Emacs and the original LEGO functions we refer to [5] and and [6],[2] respectively. The starting of system will be explained first along with the functions of the main (working) window. Next the administration window will be explained and finally the refinement-tree option will be elucidated.

4.2.1 Starting the system

To start the system, choose a directory to work in. It is recommended that for every proof a new directory will be chosen. This to prevent interference with previous proofs and to keep a clear distinction in the files.

Start Emacs in this directory. Once Emacs is started one types "M-x load-file" to load the program. Now one will be asked for the program-name and one gives the proper name and directory of the $LEGO_r$-program. To start one types "M-x LEGO-mode". Further explanation will be given in the next paragraphs.

4.2.2 The main window

Once the system is started, two new menus will have appeared. The one marked LEGO is the original LEGO-menu, see [6] for more information.

The other menu is the refinement-menu. It will have seven menu-items of which two may be used at this time. Those items are:

- New proof
  This item allows the user to start proving a new theorem. The system will ask the name of the theorem and next the statement. If the theorem-name is already known the system will say so and nothing further will happen. Otherwise, the statement should be given.It should not be ended by a ";" at the end. Next the administration-window will be opened and a LEGO-shell will be started.

- Open proof
  This item will open an old proof saved previously. The system will ask the user for the name of the proof. This name should be given without any file-extensions, just the name is enough. If it does not exist the system will say so and nothing will happen. Otherwise the administration-window will be started and filled with the right information. The LEGO-shell will be started and the $LEGO_r$-file asked for will be shown in the main window. The file will also be send to the LEGO-shell, so proving can continue where it was left off.

These items are necessary to start proving something. No other commands will work before one of these commands is chosen. After one has been chosen both commands will not function anymore. Now the user can start giving axioms and lemmas that are not meant to be refined. If one wants to start the actual proof, two other commands are now available:

- Begin proof
  This item will tell the system the proof of the actual statement will now begin. If all axioms have been given and all (non-refinement) lemmas have been proven, this is the command one wants to use. It will automatically insert the Goal-rule for the statement and send it to the LEGO-shell. The two last menu-items will now become available. Once again, the system needs this command so starting the proof oneself will certainly result in errors.

- Close proof
  This item represents two commands. It both saves the current proof and it closes it so a different proof can be started. If this command is chosen the system will ask the user if the proof should be saved. Note that this is the only save way to store the file, "Save Buffer" and the likes will NOT work properly as they will not update the administration. No matter whether the file should be saved or not, the system now asks the user if the file should be closed. This will remove the current proof-file, the administration-window and the LEGO-shell. The system will now behave as if just started. So this command can save a file, start anew or do both.

After this command is given the last menu-items will be available, the "Begin proof" will be turned of and cannot be chosen again. The last three items are:

- Give refinement
  This is one of the most important commands of the system. When actually proving the user can give a refinement with this command. Note that the proof must have been started already. (This is not really so restraining as it may seem. To start a proof one only has to use the function "Begin proof". To postpone stating a refinement till that time does not hamper the proving in any way.) This command will make sure LEGO interrupts the proof-mode to accept the refinement. The user will be asked to give the refinement name (a new name) and the statement of the refinement (the original lemma). The system will now insert this information in the proof-file preceded by the header "(* Refinement *)". The system will expect this header, so one should not remove it. The refinement itself will be written as a axiom and the LEGO-system will treat it this way. The user however should only use it as a normal (saved) lemma would be used. The administration window will be updated to show the refinement. The LEGO-shell will now be restored to proof-mode exactly where it was and the refinement can now be used in the proof. This item may be used more than once.

- Insert refinement
  This is the command used to promote (or degrade) refinements to a new level in the refinement tree. It is used in combination with the "Delete refinement"-command in the administration window. If a refinement A is to be upgraded to node of proof B for example, do the following. Delete A in the node it is declared as a refinement. How this is done will be explained later. Go to the node of proof B and do save the proof where A was removed. It may not work anymore, but that will be corrected later. In node B use the "Insert refinement"-command and give the name of A when asked for. Next the place of A should be determined. Give the name of the refinement A should be placed before. Choose so that A is valid where it should be valid! Now go back to the proof A was removed from (save the file of proof B) and simply save this file. If A is placed correctly this proof should be returned to its previous state.

- Build LEGO-file
  This item is the last command one needs to make an local $LEGO_r$-proof into an absolute LEGO-proof. Although LEGO accepts all $LEGO_r$-files, the proofs in those files will have their refinements still in their contexts. To get one large file which LEGO accepts without any refinements in any context, this command should be chosen. It

will check the absoluteness of the proof and if it is absolute it will produce the LEGO-files. The LEGO-files will be almost the same as the $LEGO - r$-files except for the appearance of the refinements. They will be represented with an Include of their proofs and not with an axiom. This only be done now because of the strict rules on the views of refinements. The new files will be written in a new directory called "finished". As the system can be fooled into believing a proof is relative when it is not (though only by abusing it on purpose), a final check is made on LEGO-file once more. This is done by running trough LEGO.

4.2.3 The administration window

The administration-window serves two purposes. Firstly it shows the statement currently being proven, its refinements and their status and secondly it allows quick access in the refinement-tree.

As said this window shows the current statement under proof, its lemmas and their status. How is this done? On the top of the window the current proof is shown by its name and its statement. It will be followed by its status which can be "ABS" for an local proof, "REL" for a relative proof and "TODO" for a proof that is not proven in any way yet. Below the proof's indication is a line of dashes to separate it from the refinements. The refinements will be shown the same way the current statement being proven is shown. Their statuses have the same meanings too.

The administration menu has four menu-items. These are:

- Delete refinement
  With this item the user can delete a refinement. It will be taken of the administration window and from the main window. Note however that only the refinements definition (the axiom) will be deleted from the working window. All later occurrences of the name will not be deleted. The user should only remove a refinement that is not used or that is to be moved elsewhere (see "Insert refinement"). The files related to this refinement (and any sub-refinements) are not deleted. This again to make moving refinements possible. To delete those files one has to delete them oneself. To delete a refinement the refinement should be highlighted by the cursor. Once that is done the deletion command can be given. This function is only available once the proof has begun.

- Go to refinement
  This allows the user to go one step deeper into the refinement-tree. It will assume the chosen refinement as the current statement to be proven and will adjust the system properly. To activate this command

the refinement should be highlighted by the cursor. Then the command can be given. The system will ask if the user would like to save the current proof. If the user does not want to save any changes to that proof will be lost. It will then set the system to the new proof-session as if one of the starting options of the main window has been chosen. It will create a new proof-session if the refinement has never been worked on and it will load an old session otherwise. Both the administration window and the LEGO-shell will be updated. This option is available as soon as the proof has been started.

- Step back
  This is the reverse of "Go to refinement". It will go one step higher into the refinement tree to the parent of the current statement.This command does not need highlighting as there is only one option. The command can be given as soon as the proof is started. It will be treated exactly the same as a "Go to refinement"-command. If this command is given in the top of the refinement-tree nothing will happen.

- Show refinement-tree
  This item will go to a new screen to show the refinement-tree. The user will be asked which tree should be shown, the entire tree from the top or the subtree from the actual node? If "top" is answered the entire tree is shown, the current subtree is shown otherwise. Further explanation of this command can be found below.

It should be noted that the administration window is read-only. It is not to be edited upon. Saving this file by "Save buffer" and similar commands might very well damage the proof.

4.2.4 The refinement-tree screen

This screen is a sub-screen pasted over the administration window. It serves to show the refinement-tree at the moment and allows jumps in this tree. If the screen is opened it will show the tree as follows. On top of the screen will be the top of the chosen tree (either the top of the entire refinement-tree or the current node). It will be shown as in the administration window, with status. Next the first refinement will be shown with a slight indentation. Beneath every refinement are its own refinements (with their own refinements and so on) each time more indented then before. See figure 4.1.

Example
As one can see in figure 4.1, the theorem (top-node) is shown first. All its children (lemma1, lemma2 and lemma3) are shown with a slight indentation. Their sub-lemmas (lemma3.1 for example) are indented even more, and so on. Below each lemma its own sub-lemmas are shown before its next

theorem:statement REL
  lemma1:statement ABS
    lemma1.1:statement ABS
    lemma1.2:statement ABS
  lemma2:statement TODO
  lemma3:statement REL
    lemma3.1:statement REL
      lemma3.1.1:statement TODO

Figure 4.1: A possible picture of the reference-tree.

sibling. For example, lemma1's sub-lemmas (lemma1.1 and lemma1.2) are shown before lemma2 is shown.

Two possible commands can be given now:

- Close refinement-tree
  This simply closes the refinement-tree screen and returns to the administration window. Proving can now continue.

- Jump to refinement
  This is somewhat similar to the "Go to refinement" in the administration window. However this option will take more time and therefore small "jumps" can be done faster with the administration window. To go to entirely different branches of the refinement-tree step by step is too slow and for those occasions this jump option is perfect. Simply highlight the chosen refinement with the cursor and give the command. The top of the tree can also be chosen (although it is no refinement). The system will ask if the current proof shall be saved. After that it will set the system to the proof of the chosen refinement. The refinement-tree screen will be replaced by the administration screen.

Like the administration-window, this screen is read-only. It should not be edited. Saving it in any way will not harm the system.

4.3 How $LEGO_r$ was implemented

Here a light will be thrown on the implementation of $LEGO_r$. Some important implementation decisions will be handled and then some important algorithms will be shown.

4.3.1 Design decisions

As said, the system was implemented in elisp, a lisp variant specially designed for the Emacs text-editor. A LEGO-interface has already been written for it and the $LEGO_r$-system is an extension to this system. The first important implementation decision was how to regulate working with refinements. The first options were all based on the idea of hypertext. This is text which has "links", places sensitive for mouse-clicks. A mouse-click on a link can result in all sorts of functions. This proved to be very hard to implement and not necessarily better than other options. To also implement keyboard variants of commands with hypertext would prove to be another difficulty.

The system of the administration window is both elegant and simple. The mouse can still be used for activating menus. The keyboard commands are implemented easily in conjunction with the mouse-menus. The fact that the user has to concentrate on three windows instead of two (working window and LEGO-shell) is a drawback, but the increased surveyability makes up for that. It also solved another problem of hypertext. The files produced by $LEGO_r$ should be readable to normal LEGO. As hypertext needs a lot of notations for links in the actual files, this might prove to be difficult for LEGO to accept. This could be solved by writing the notations as LEGO-comments, but this would need much programmed parsing of the files. By using another window for the extra text-properties, the information for a $LEGO_r$-proof could be split into different files. The working window now only contains a "normal" LEGO-file and saves it that way.

The strict view rules brought with them a new problem. How does one keep the user from using lemmas outside their view? Having decided that the system would enforce good use of views, this enforcing should be done properly and preferably without any loopholes around it. This meant some programming has been done that may not seem very logical. One of the most important decisions resulting from this was the division of the LEGO-proof in different files for every refinement-statement This way the views were quite easy to formulate.

It had its own problems however.For example the "Build LEGO-file" command. Why not include the already proven files directly. Aside from the problems arising from changing the proofs of those already proven statements (because of wrong assumptions for example), it would allow for easy abuse of available lemmas outside their scope. The files would be included and therefore all sub-lemmas would become available. There are several ways to prevent this and the "Build LEGO-file" command was by far the easiest one.

The skipping from the proof of one statement to another had some problems too. The contexts for different statements had to be separated. Otherwise circular dependencies and other grave errors might result. This meant problems for the LEGO-shell. Just switching contexts is not possible, so the only option was to rebuild contexts as far as necessary. Two implementations were possible for this rebuilding. Either a new LEGO-session was started every time a new refinement was chosen or one LEGO-session handled all rebuilding. Both options had their pros and cons.

The multi-shell option needed a long time to start up every time. It was not implemented in the original LEGO-interface. On the other side it would allow swift access to proofs already available. The main drawback, which was not apparent at first is the large amount of memory used by a LEGO-shell. To have more than one shell running at a given time severely hampers the performance of many systems.

The one-LEGO-shell option had other problems. It required a lot of LEGO-commands by the system in the LEGO-shell as handles for the program. It does not interfere with the LEGO-shell, but the user might get confused. But by placing those handles carefully a lot of reloading can be skipped when taking small steps in the refinement-tree. Especially when the proofs get really large this time-saving makes up for any possible confusion.

Lastly a comment about the refinement-tree window. This window can be seen as superfluous as all its functions can be performed by the administration window. The real purpose of the refinement tree is to give the user a good look at what is to be done. It then simply jumps to such an area under construction and then the administration window can be used to move around in smaller steps. It is unusual to jump around while proving and this option is not build for that purpose.

4.3.2 The system

The greater part of the extension to the LEGO-interface revolves around handling the abstract elisp data structures known as administrations. An administration holds the information about one node in the refinement tree. An administration of a node n is a structure containing three elements:

1. The requires part
   This part contains the name of the (possible) parent node of n. If the described node is the top node, it contains "(**)". This is a bogus name, needed because of problems with reading the empty string "" from file.

2. The proof part
   This is a small abstract structure containing the status of the described node (coded by color: "green" for local, "yellow" for relative and "red"

otherwise) and the name and statement (in a string). The proof part is never empty.

3. The refinements part
This is a list of the proof parts of the refinements of the described node. The refinements part can be the empty list, if the current node has no refinements.

Example
The following example is meant to show the idea of the administration, it's not a relevant proof. Suppose a node exists containing the statement $\{A : Prop\}A \to A$ named *true*. It is a refinement of the node *truth* and it has one refinement *false* stating $\{A : Prop\}A$. The refinement *false* has not been proven yet, but *true* has been. Therefore *true* is proven relative to *false*. The administration would be:

$$("truth" \; ("yellow" \; "true : \{A : Prop\}A \to A") \\ \; (("red" \; "false : \{A : Prop\}A")))$$

There are several functions for getting and setting parts of an administration. All these are implemented in the file administration.el.

A special administration exists called the current-proof-administration. This contains the administration of the node currently worked on by the system. Most functions will use this administration only. This administration is also used to fill the administration window. When a new node is selected the current-proof-administration is changed too.

Now some of the major commands' implementations will be described. Some are so trivial they can be easily be understood by reading the code. Those will not be explained here.

Both beginning functions "New proof" and "Open proof" are quite simple. The only interesting details are the two handles sent to the LEGO-shell. The begin-marker is the very first thing the system send to the LEGO-shell. This is to make sure everything can be deleted when an entirely new context is needed. This marker is also set after jumping with the refinement-tree screen. The other marker, the file-marker, is a marker set to stepping back. When a step back is performed to the node to which the file-marker belongs, LEGO-shell only has to forget everything up to the file-marker. It is also used in the interrupting of a proof-session, because of a refinement.

"Close proof" is another interesting command. The important parts here are the checks on proof status and the updating of the refinement-tree. The status of a proof is checked in two stages. First it is checked for being a relative proof. This is done by searching in the LEGO-shell. If both the starting of the proof and the saving of the proof can be found, it is assumed the proof is indeed relatively proven. This might not be the best check possible. It can be abused, but that is only done on purpose and the final files that will be delivered will most certainly not pass the final LEGO-check.

Next the check for local correctness is made. This is done by checking the statuses of all refinements (in the current-proof-adm). If these are all local the current proof is too. Remember that the proof was already relatively proven.

Updating the refinement-tree is done by going up the tree step-by-step until no changes are made anymore or until the top is reached, whichever comes first. The check for local correctness is the only check performed here. Notice that the position of a proof can be lowered if one of its refinements' statuses is lowered. This will probably not happen very often.

The "Insert refinement"-command is a true hack. The system checks if the file exists. If it exists The system fits it into the current-proof-adm and redisplays the administration-window. Then it searches the place to insert it into the working-window. It searches by means of the "(** Refinement **)"-headers. Lastly it updates the files of the inserted refinement. The appearances of the parent file in those files are updated.

"Build LEGO-file" is the command used to make LEGO-readable files out of $LEGO_r$-files. It can only be given if the entire proof is absolute. It searches which file is the top file and then it goes on and replaces every refinement with an Include of the necessary file. It will also remove the Includes and Forgets in the beginning of the refinement files. All files will be stored in a new directory named final. The files are handled depth-first when the refinement-tree is concerned. After all files have been rewritten the top file is checked one last time. If it is indeed absolute that message is given and the system restarts. Otherwise an error is signaled, but the system still restarts.

The refinement-tree moves made by the user all need the highlighting of the target refinement with the cursor. The system needs to know which refinement is selected and does so by reading the line that is highlighted. The different move-commands are quite similar to each other and their main techniques are the same as that of the two starting commands. The main difference is the use of the LEGO-shell markers.

To take one step deeper into the refinement-tree, all statements in the context already in the LEGO-shell may be used. That is, if the statements are placed before the declaration of the refinement moved to. The LEGO-shell is told to forget everything starting from that declaration to the end of the context. Next the file (if it exists) is read and send to the LEGO-shell. The entire view of the current statement is now in place.

One step back uses the file-marker of the parent-node. As only parts of the context contained in the parent-node still remain in the current context, it is simpler to just reread the context from the beginning of the proof of the parent-node than trying to paste in the missing parts. Thus everything is forgotten back to and including the file-marker of the parent node. Next the parent-node's file is reread and the right view is established.

When making big jumps, most of the context has to be removed. Often the fastest and almost the simplest way is to rebuild the entire context. The LEGO-shell is told to forget everything and the context is rebuild from top to the selected node. By inserting two lines in every refinement-file this can be done quite rapidly. The first line "Includes" the parent-node-file (and that file starts by including the next parent-node-file and so on to the top-file) and the second line forgets everything back to and including the declaration of the refinement. This results in very simple forward stepping from the top to the node selected. It also delivers the right refinement view.

Chapter 5

Discussion

In this document we have explained the concept of refinements. The refinements are a natural way of working, not implemented so naturally in LEGO and similar systems. Refinements are statements which may formulated and used before they are proven. Use of refinements both helps the user with a more natural way of proving and they result in a better structuring of proofs. This structuring was explained by the notion of refinement-trees. It has been proven that LEGO can be extended to accept refinements without adding anything to the power of LEGO. The implementation of refinements in LEGO is done in the system $LEGO_r$.

The working with refinements goes quite smoothly and the "extra" function of the refinement-tree screen has been added, although it was not planned originally. As the proof is cut into smaller parts the rereading of the proof can be very fast compared to the LEGO-system. And if one doesn't feel like using refinements one can always refrain from using them. However, one will probably have to use the word-processor facilities again.

The view has been implemeted securely too. The vertical structuring requires some getting used to, but it does result in a nicely structured proof. This goal has been accomplished very well.

The $LEGO_r$-system we build works well, but there are a few points worth discussing.

The test $LEGO_r$ does on the relativeness of a proof, is not the most secure. Anyone knowing how it works can fool the system. The check is done by searching for the evidence of a succesfull save in the LEGO-shell. If the user inserts this himself the system will be fooled. The final test will tackle this abuse. Better would be if there was a check on the messages LEGO sends to Emacs.

Another point is the axiomized refinements. As axioms many things can be done with them, that should't be done normally. Think about Discharging them in LEGO and similar actions. A check on those commands might prevent this.

It is possible to define more than one theorem by hacking, but it would be nice to add a level to $LEGO_r$ so this can be done by the system. A so-called “book”-level.

Lastly, but quite importantly, the check on local correctness is not complete. It only checks if the children a refinement are locally correct. The elder siblings are NOT checked. This may very well give an refinement an local marker, when it is not local. That’s why only the top-node is provable in $LEGO_r$. It would be very nice if this checking was done.

Appendix A

Elaborate example

Here we will proof a lemma of the fixed-point theorem of Knaster-Tarski. To explain this theorem we will have to give some definitions. We will do this by giving the LEGO variant of those definitions and then explain them.

[U:Type];

We will look at an arbitrary set U. We will define subsets of U by predicates on U. These form the power-set of U.

[Power = U→Prop];

If A:Power and x:U then A x means x $\in$ A. We also need an inclusion relation like $\subseteq$.

[Incl [A,B:Power]=x:U(A x)→(B x)];

Now Incl A B means that for every x:U if A x then B x. That is precisely $\subseteq$. The fixed-point theorem is concerned with sets of elements of the power-set of U. We need to define the notion of those sets.

[Class = Power→Prop];

If K:Class and A:Power then K A means A $\in$ K. We need a intersection operation on Classes.

[Inter [K:Class]=[x:U]A:Power(K A)→(A x):Power];

If K:Class and x:U then (Inter K) x means that for every A:Power and A $\in$ K, x $\in$ A. We will now go to the fixed-point theorem of Knaster-Tarski.

[f:Power→ Power];
[monotonous:A,B:Power(Incl A B)→(Incl (f A) (f B))];

We will look at an arbitrary function f from the power-set of U to the power-set of U. f is monotonous, its values never drop if its input rises. A fixed-point of f is:

[fixedpoint [A:Power]=and (Incl (f A) A) (Incl A (f A)):Prop];

Or all A:Power so that f(A) = A. We will now define the class of so-called semi-fixed-points of f.

[SF = [A:Prop](Incl (f A) A):Class];

Or in math style SF = $\{A|f(A) \subseteq A\}$.

[INTER = Inter SF];

This set INTER is the intersection of all semi-fixed-points of f. We need to define the relation lower bound of SF:

[lowerbound [P:Power]=A:Power(SF A)→(Incl P A):Prop];

Meaning that if P:Power and lowerbound P then P is included in every A:Power ∈ SF. Lastly we will define the well known "and" relation.

[and [A,B:Prop]=C:Prop(A→B→C)→C:Prop];

The fixed-point theorem of Knaster-Tarski is:

INTER is a (normal) fixed-point of f: f(INTER) = INTER. Moreover it is the smallest fixed-point.

We will not proof the entire theorem, but just an important lemma:

For all fixed-points A: INTER is included in A.

We prove this by proving the following lemmas:

- INTER is a lower-bound of SF.
- For every A in SF: f(INTER) is included in f(A).
- f(INTER) equals INTER.

We'll formalize these lemmas (and use two sublemmas) to proof Knaster-Tarski in $LEGO_r$.

We will now give the workings of $LEGO_r$ on the proof of the theorem of Knaster-Tarski. We will give LEGO commands as **command**, $LEGO_r$-specific commands as *command*, subscripts of the file at hand will be in

```
subscript
```

and in between we will explain in normal text. We will prove Knaster-Tarski from scratch, so the proof given below may be confusing to read. Remember that this is not the way it will be presented in the file. This is the way the proof is constructed. No comments will be given on the proof itself. This section is to show how $LEGO_r$ works not how to proof in LEGO. It may be helpful to try it out and see what happens though.

Let's assume the $LEGO_r$-system has been started. We will now begin:

- *New proof*
  We want to start a new proof. The system will ask for the name and we will give "knastar" for example. The statement will be A:Power (fixedpoint A)→Incl INTER A. This doesn't mean anything yet but it will soon. The administration-window will appear showing the name and statement followed by the status TODO and the dash-line.

- Now give the above mentioned definitions in that particular order. These definitions make sure the previously given statement makes sense. Let's start proving.

- *Begin proof*
  The system will now insert the line

  ```
  Goal knastar:{A:Power} (fixedpoint A)$\rightarrow$Incl INTER A;
  ```

  . It will send this line to the LEGO-shell.

- **Intros;Refine H;Intros;Refine H2;**
  This will get the proof started. We will now need our first lemma.

- *Give refinement*
  We will call the lemma finterunderfa. The statement is A:Power(SD A)→Incl (f DOOR) (f A). The administration-window will register it (with status TODO) and in the knastar.l-file

  ```
  (** Refinement **)
  ```

  will be inserted, followed by

  ```
  [finterunderfa:{A:Power}(SD A)->Incl (f DOOR) (f A)];
  ```

. This is the normal procedure for refining. We can now continue our proof.

- **Refine finterunderfa;Intros;Refine H2;Immed;**
  Now we need a second refinement.

- *Give refinement*
  This refinement will be called fequalsinter and its statement is and (Incl (f INTER) INTER) (Incl INTER (f INTER)).

- **Refine fequalsinter;Intros;Refine H5;Immed;**
  LEGO will show the proof is finished. Now we need to save it in LEGO.

- **Save;**
  LEGO will have saved the file under the name knastar. To change the administration window let's save the proof ourselves.

- *Close proof*
  The file knastar.l will be saved. The administration window will now show a status of REL in stead of TODO behind the knastar-statement. Let's now proof the refinements.

- *Go to refinement*
  Note that one first has to highlight the refinement. We will proof finterunderfa. We move to the administration window and click on the finterunderfa statement. Now we give the commando to go to the refinement. We will be asked if we want knastar.l saved, which we won't. So we answer by typing "n". Then the administration-window will show the finterunderfa above the dash-line and nothing beneath it. The finterunderfa.l file will appear in the main window. Remark the two lines already in the file.

  ```
  Include knastar;
  ```

  and

  ```
  Forget finterunderfa;
  ```

  . These lines are needed to read this file into normal LEGO, so we will leave them where they are. No more axioms are necessary, so we will start the proof.

- *Begin proof*
  This will have the usual consequences.

- **Intros;Refine monotonous;Refine DOOR;**
  Now we need a new refinement. This goes exactly as before.

- *Give refinement*
  This refinement is called lowerinter and its statement is lowerbound INTER.

- **Refine lowerinter;Immed;Save;**
  Now the proof of finterunderfa is ready and we go deeper into the refinement-tree once more.

- *Go to refinement*
  Highlight lowerinter and give the command. Now remember to say "y" when asked for saving finterunderfa.

- *Begin proof*
  No axioms here too. The proof is quite straight-forward so it will not be handled in detail.

- **Intros;Refine H1;Immed;Save;**
  Now we can't go anywhere but up into the tree. Let's do it by jumping this time.

- *Show refinement-tree*
  Remember this command works in the administration-window only. The question will be if we want to see the whole tree or just a part of it. As the current subtree isn't very exiting (it's only the lowerinter node) we will go for the top. We answer "top" and the administration window will be replaced by the refinement-tree screen. This will have knastar on top (with status REL) followed by finterunderfa (status REL). Next will finterunderfa's refinement lowerinter (status still TODO, because we haven't saved it yet) and last will be fequalinter (status TODO). This last one we're interested in, so we highlight it.

- *Jump to refinement*
  We will be asked if lowerinter should be saved and we answer "y". Next all goes exactly like the *Go to refinement* command.

- *Begin proof*
  We will jump to the giving of the refinements immediately. Let's say we have a hunch what they might be.

- *Give refinement*
  This one is named finterininter and its statement is Incl (f INTER) INTER.

- *Give refinement*
  The second one is named interinfinter and its statement is, as could be guessed, Incl INTER (f INTER).

- **Intros;Refine finterininter;Refine interinfinter;Save;**
  This is all there is to this proof so let's move on quickly to the next refinement. We will choose the second one now, so we highlight interinfinter in the administration window.

- *Go to refinement*
  No axioms again so the proof is started.

- *Begin proof*

- **Intros;Refine H;Intros;Refine monotonous;Refine (f INTER); Refine finterininter;Immed;Save;**
  Now here's something interesting. We use the refinement finterininter, given in the parent of this refinement, to proof this refinement. This is correct in ideas of the view of interinfinter, because finterininter is interinfinter's elder sibling. We can go to proving that lemma now. This time we don't jump but step back and forth.

- *Step back*
  We answer "y" to the save question and the result is as if we had jumped to the parent fequalsinter. Notice now that the administration window will signal interinfinter as ABS or locally proven. This doesn't mean that it does not rely on any refinements, but it relies on refinements that are not its children. This may be confusing at first, but it will be clear after some use.

- *Go to refinement*
  Highlight finterininter and give the command. Saving is not necessary so the answer is "n" this time.

- *Begin proof*

- **Intros;Refine H1;Refine monotonous;Refine INTER;**
  Now we come to a problem. We need a refinement we have already given, but it is outside the present refinement's view. We need lowerinter. It is so simple we can just redefine it here. But let us give lowerinter the place it apparently deserves. We jump to finterunderfa.

- *Show refinement-tree*
  We choose "top", to see the entire tree.

- *Jump to refinement*
  We will want to save finterininter, even though it's not finished yet. So we answer "y" to that question. Now to replace lowerinter.

- *Delete refinement*
  Highlight lowerinter and give the command. Lowerinter will be removed from both administration window and main window. Now to a higher level.

- *Step back*
  We want to save finterunderfa and answer "y". Now finterunderfa will become TODO as the refinement no longer exists. We will get back to it.

- *Insert refinement*
  When asked for the refinement to be inserted, we answer lowerinter and to where we answer finterunderfa, as it should be before in finterunderfa's view. Now it will be inserted in the right place in both the administration window and the main window. It will appear as ABS of course. Now lowerinter becomes elder sibling of finterunderfa.

- *Go to refinement*
  Just visit finterunderfa and everything is OK again. We jump to finterininter and we don't forget to save finterunderfa.

- *Show refinement-tree*
  We want to see the entire tree.

- *Jump to refinement*
  Highlight finterininter and save. We can continue proving where we left of.

- **Refine lowerinter;Immed;Save;**
  We can now use lowerinter. Now we want to see what to do.

- *Show refinement-tree*
  We want to see the whole tree and we notice no more TODO's. Now we want to end this proof so we jump to the top. Highlight knastar.

- *Jump to refinement*
  We want to save finterininter and answer "y". Then we see in the changed administration window that indeed knastar is ABS or locally proven.

- *Build LEGO-file*
  We now make a file LEGO itself can read, without any refinements. We will be shown the message "This proof is absolutely proven." and we

press the OK button. Now the administration window has disappear and the LEGO-shell too. The main window will no longer show a $LEGO_r$-file. The system is ready for a new proof. The final files will be in the directory final.

Appendix B

The key-sequences to the $LEGO_r$-commands

The main window

$C-l-C-n$	$New\ proof$
$C-l-C-o$	$Open\ proof$
$C-l-C-s$	$Close\ proof$
$C-l-C-b$	$Begin\ proof$
$C-l-C-r$	$Give\ refinement$
$C-l-C-i$	$Insert\ refinement$
$C-l-C-m$	$Build\ LEGO-file$
$C-l-C-l$	$recentre\ (this\ was\ previously\ under\ C-l\ in\ Emcas)$

The administration window

$C-l-C-v$	$Go\ to\ refinement$
$C-l-C-d$	$Delete\ refinement$
$C-l-C-t$	$Step\ back$
$C-l-C-z$	$Show\ refinement-tree$

The refinement-tree screen

$C-l-C-c$	$Close\ refinement-tree$
$C-l-C-j$	$Jump\ to\ refinement$

Bibliography

[1] Robert Pollack, *The Theory of LEGO*, Thesis, University of Edinburgh, 1994.

[2] Zhaohui Luo & Robert Pollack, *LEGO Proof Development System : User's Manual*, Department of Computer Science, University of Edinburgh, May 7, 1992.

[3] Henk Barendregt, *Lambda Calculi with Types*, in D.M. Gabbai, S. Abramsky & T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 1, pages 117-309, Oxford University Press, 1992.

[4] G. Huet, G. Kahn & C. Paulin-Mohring, *The Coq proof assistent - a tutorial*, Rapport Technique 078, Projet Coq-INRIA, Rocquencourt/ENS Lyon, 1995.

[5] Richard Stallman, *GNU Emacs Manual*, Free Software Foundation, 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA, 1995.

[6] Thomas Schreiber, Emacs support, a www-page at the University of Edinburgh, `www.dcs.ed.ac.uk/packages/lego/html/emacs.html`.

[7] Bil Lewis, Dan LaLiberte, Richard Stallman & the GNU Manual Group, *GNU Emacs Lisp Reference Manual*, Free Software Foundation, 675 Massachusetts Avenue, Cambidge, MA 02139 USA, 1995.

[8] Anthony Bailey, Peter Aczel & David Carlisle LaTeXLEGO, a www-page, `www.cs.man.ac.uk/ baileya/LaTeXLEGO.html`

[9] GIPE: Generation of Interactive Programming Environments, a www-page, `www.cwi.nl/ gipe/`

[10] Michael J. Hannah, Sandia National Laboratories, HTML Reference Manual, a www-page, `www.sandia.gov/sci_compute/html_ref.html`