A Propositionlogic-, naturaldeduction-proof app(lication)
Bachelor's Thesis

Tim Selier

Utrecht University

July 24, 2013

Supervisor: Prof. dr. Vincent van Oostrom
Course load: 7.5 ECTS

Abstract

It has been over 2 years ago, my fellow freshmen Cognitive Artificial Intelligence and I learned something relatively new: Propositional Logic. At that point the most of us only knew two kinds of syntaxes: math and natural language, no Propositional Logic. We also had no familiarity with Natural Deduction. In that situation, when constructing a natural deduction proof, it might be hard to know the valid actions.

At this point, Vincent van Oostrom and I developed an ambition to provide students with a tool or framework that would enable the student to construct a valid proof and allow only all the valid actions. Finally we created an implementation of this tool in the form of an iOS-app, which runs on both iPad and iPhone.

The ultimate goal is to enhance the logic skills of students all over the world. We declared our main ideas and philosophy in the article Clickable Proofs [3]. This article grew as the app grew and might be considered as a set of answers to questions we encountered.

This app has already been approved by the Apple staff and will be available in the Apple app store as soon as this document has been reviewed.

Acknowledgements

I see expressing your thankfulness with difficult words you wouldn't use otherwise as one of the biggest clichés of a thesis project. I used to think those poeple were overdrawing the situation by saying "My deepest heartfelt appreciation goes to ...". Now I finished my very first thesis project I have a better insight of what drives people to continue these clichés. It are other people who enable us to reach more, even when their support is not focused on the content.

However, I still do not feel comfortable expressing myself with genteel words. Therefore I will just sum up who I am thankful for and why. It is the significance of someone's contribution, instead of my pretty words for them that really matter!

I would like to start with my supervisor, Vincent. First he came up with this great idea, the app, which fitted my profile perfectly. In addition he was willing to teach me a lot of stuff, both theory and academic skills, which were very useful.

Secondly, I would like to thank my girlfriend, Marit. She understood the importance of this project and accepted I had less attention left for her, even when I spent to much time on it.

Also I would like to thank my father, Theo. He examined my report and came up with a lot of constructive criticism.

My two bosses at work, Flexyz, who have also been patiently to me. However they would love to see me work more for them, it was no big issue for them that I actually worked a little bit less.

I would also like to thank all of the test persons.

At last I would like to thank anyone who has shown any form of interest, enthusiasm or positivism towards this project.

Contents

1 Introduction

One of the starting points was the analogy between this application and Lego. We wanted to build proofs like you build a Lego structure. You can click Lego blocks on top of each other, and you can create structures by first creating small substructures.
This analogy is underlined by figure 1, where you can clearly see a natural deduction proof, aligned over some pieces of Lego.

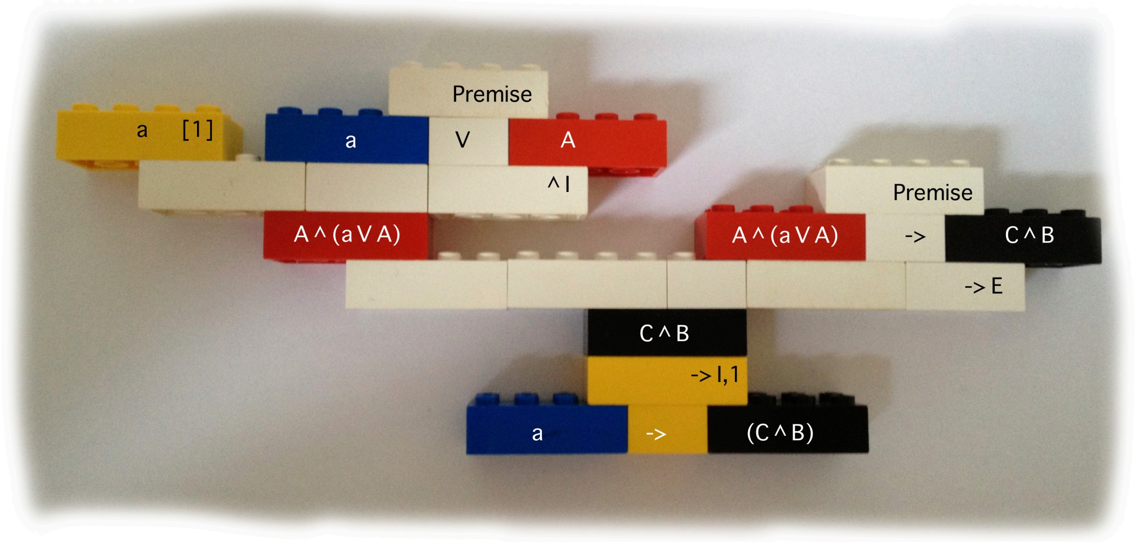


Figure 1: Example of a proof, in lego

1.1 The app

It is important that the App enables users to search with a mix of bottom-up and top-down strategies. Another requirement we imposed ourselves is that a user is allowed to return to a situation made earlier. Furthermore a user will be able to define propositions that are his premises and goal(s) at the beginning, or even during the construction of the proof. The following subsections describe briefly what features are involved by meeting these requirements. In section 3 I will reveal a deeper view of the implementation and the technical choices involved.

1.2 Proofpieces

The app will deal with proofpieces; those elementary proofpieces and the combined proofpieces. A proofpiece consists of a logic n-ary connective and n+1 ports [3, chapter 3]. A proofpiece might also contain up to one reference to a port onto another proofpiece. The latter reference means that the proofpiece is an assumption of the proofpiece and port it refers to. A proofpiece can be of Introducing or Eliminating nature, often indicated with an "E" for elimination or "I" for introduction. Please notice that the "E" or ”I” are just names which makes it easier to distinguish the nature of the proofpieces. A proofpiece always has n input ports for an n-ary connective and one conclusion port, which is charged with the formula spawned by the main connective combined with the values of the input ports. At this point we can choose to make our own connectives and respectively proofpieces. However we will use the proofpieces we already know for natural deduction in propositional logic i.e. via the syllabus [4]. The following figure 2 is a display of all the proofpieces implemented in this app. Notice the fractional notation.

$$\alpha \qquad \frac{\bot}{\alpha}\,\bot\mathrm{E} \qquad \frac{\overset{[\neg\,\alpha]}{\overset{\nabla}{\bot}}}{\alpha}\,\mathsf{RAA} \qquad \frac{\overset{[\alpha]}{\overset{\nabla}{\bot}}}{\neg\,\alpha}\,\neg\mathrm{I} \qquad \frac{\alpha \quad \neg\,\alpha}{\bot}\,\neg\mathrm{E}$$

$$\frac{\alpha \quad \beta}{\alpha\wedge\beta}\,\wedge\mathrm{I} \qquad \frac{\alpha\wedge\beta}{\alpha}\,\wedge\mathrm{EL} \qquad \frac{\alpha\wedge\beta}{\beta}\,\wedge\mathrm{ER}$$

$$\frac{\alpha}{\alpha\vee\beta}\,\vee\mathrm{IL} \qquad \frac{\beta}{\alpha\vee\beta}\,\vee\mathrm{IR} \qquad \frac{\alpha\vee\beta \quad \overset{[\alpha]}{\overset{\nabla}{\gamma}} \quad \overset{[\beta]}{\overset{\nabla}{\gamma}}}{\gamma}\,\vee\mathrm{E}$$

$$\frac{\overset{[\alpha]}{\overset{\nabla}{\beta}}}{\alpha\to\beta}\,{\to}\mathrm{I} \qquad \frac{\alpha \quad \alpha\to\beta}{\beta}\,{\to}\mathrm{E}$$

$$\frac{\overset{[\alpha]}{\overset{\nabla}{\beta}} \quad \overset{[\beta]}{\overset{\nabla}{\alpha}}}{\alpha\leftrightarrow\beta}\,{\leftrightarrow}\mathrm{I} \qquad \frac{\alpha \quad \alpha\leftrightarrow\beta}{\beta}\,{\leftrightarrow}\mathrm{EL} \qquad \frac{\alpha\leftrightarrow\beta \quad \beta}{\alpha}\,{\leftrightarrow}\mathrm{ER}$$

Figure 2: Supported proofpieces, as chosen to be complete for natural deduction on the domain of propositional logic. For further information see Clickable Proofs[3]

2 The graphical user interface

The world of technology is really evolving as we write i.e. the year 2013. A great variety of operating systems and programming languages are available. It is hard to create an application that supports most of the operating systems. Our main targets are touch enabled mobile operating systems, which means Apple’s iOS and Google’s Android at the moment, but windows 8 is also looking promising for mobile devices.

2.1 HTML5

An option for supporting multiple operating systems is using HTML5. The great advantage of HTML5 is that all (serious) operating systems contain at least one HTML

render-engine and JavaScript engine. There is a project, Phonegap, which would be able to provide the HTML5-App with a single API, compatible with multiple platforms.
But there is a downside on HTML5. HTML5 based apps commonly don't feel native. The idea is nice, but the technology, and its support by the platforms is not mature yet.

2.2 iOS

With HTML5 eliminated as an option, the choices were reduced to whether I wanted to develop for android or iOS. Programming for Android means programming in Java with Eclipse, while programming for iOS means Objective-c with Xcode. Eventually I picked iOS for two reasons. A reason of a practical nature was the availability of iPads in our direct environment and it would mean that no extra hardware would have to be acquired. Another reason comes from my own interest: I'm always in for learning new programming languages.

2.2.1 Objective-C

Objective-C is plain C, enriched with objective oriented programming via the Smalltalk syntax. Objective-C is a high level programming language. It differs from other languages like C# and java, where objects implement functions while objective-C implements messages. A message does not simply have a name, but consists of the set of all the names of the input variables.
Consider this small example of code that makes a person object walk to some coordinates for Java and Objective-C.

```
# Java:
person.walk(10,20);

# Objective-C:
[person walkToX: 10 toY: 20];
```

The Objective-C code may take a little more characters, but it is clear what the 10 and the 20 stand for, while in Java this might be problematic.

2.2.2 Cocoa touch

Cocoa touch is a framework that helps implementing a lot of typical touch-based applications. It is the most top layer of the iOS architecture and is approachable via, and implemented in Objective-C.

2.2.3 Cocoa MVC

The separation between the model, view and controller is important for bigger, more complex systems to ensure maintainability and readability. A single model can often be represented in multiple ways. For example: the model of a natural deduction proof can be represented via the Fitch-style, or by trees. Also the input style may differ. For example: touch input verses keyboard input.

There are many versions of MVC's. In each version there is a different set of relationships between those three. In some versions a change in the model automatically implies a change in the view. In other models the view is owner of the data. In the iOS version of MVC none of these latter two is the case. The Controller keeps track of both the views and the models. It also translates input from the views to action in the models. See figure 3. On Apple's online documentation library their philosophy on MVCs is explained more precisely.

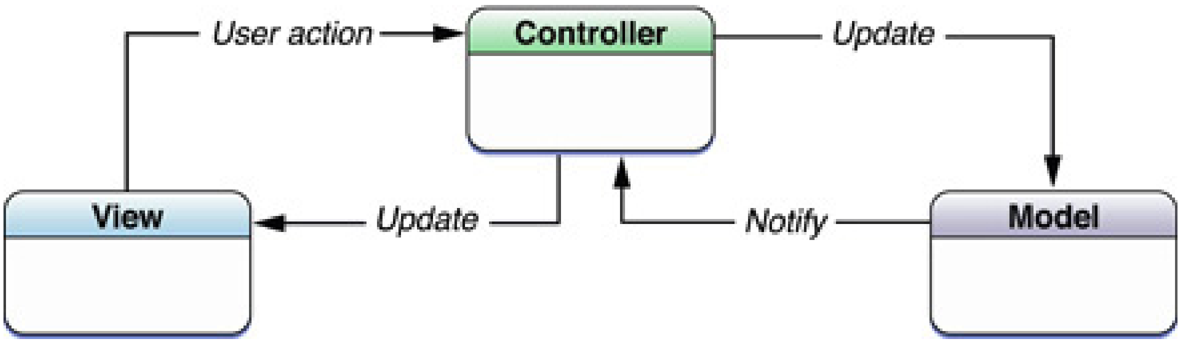


Figure 3: MVC used in iOS systems.
Source: apple.com

2.3 Storyboard

Programming for iOS means having the opportunity to use a storyboard to design your interface. Storyboards are really useful in visualizing the controlflow of the app. It also enables us to easily support multiple devices (running iOS). In figures 4 and 5 two screenshot are displayed of the storyboards of the two versions of the app, the first for the iPad and the second for the iPhone. We can see clearly that the iPad's version of the app has a split view, on top the NaturalDeductionViewController and below some space for the toolbox which enables the user to create goals and targets. While the iPhone has not not enough space on its screen to have a split view, it needs another custom view distribution. For that reason the iPhone uses a UITabBarController, which takes far less space and still enables the user to access the toolbox view, now hidden behind the buttons of the TabBar.
See also section 2.5 for more information about the toolbox.

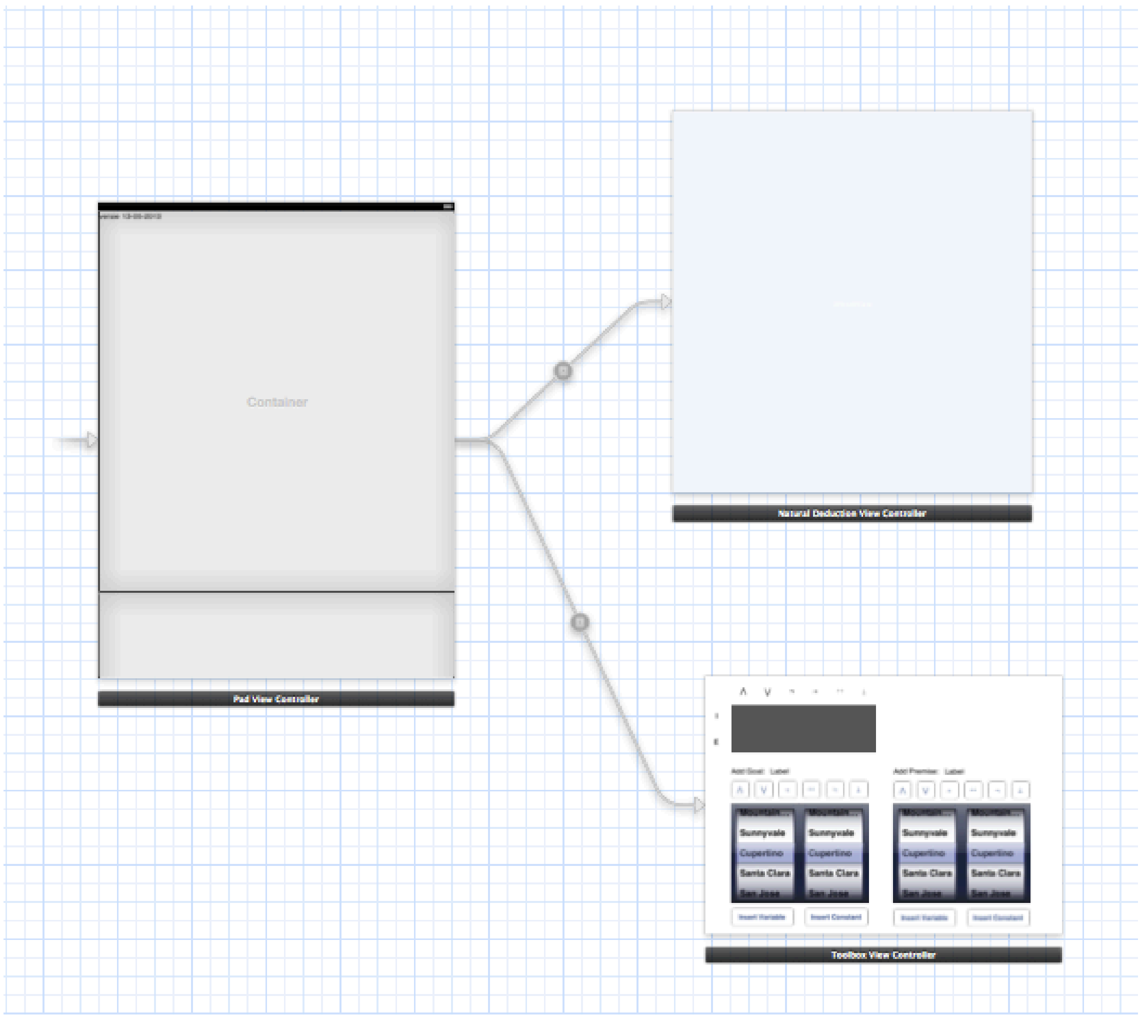


Figure 4: Screenshot of the iPad storyboard

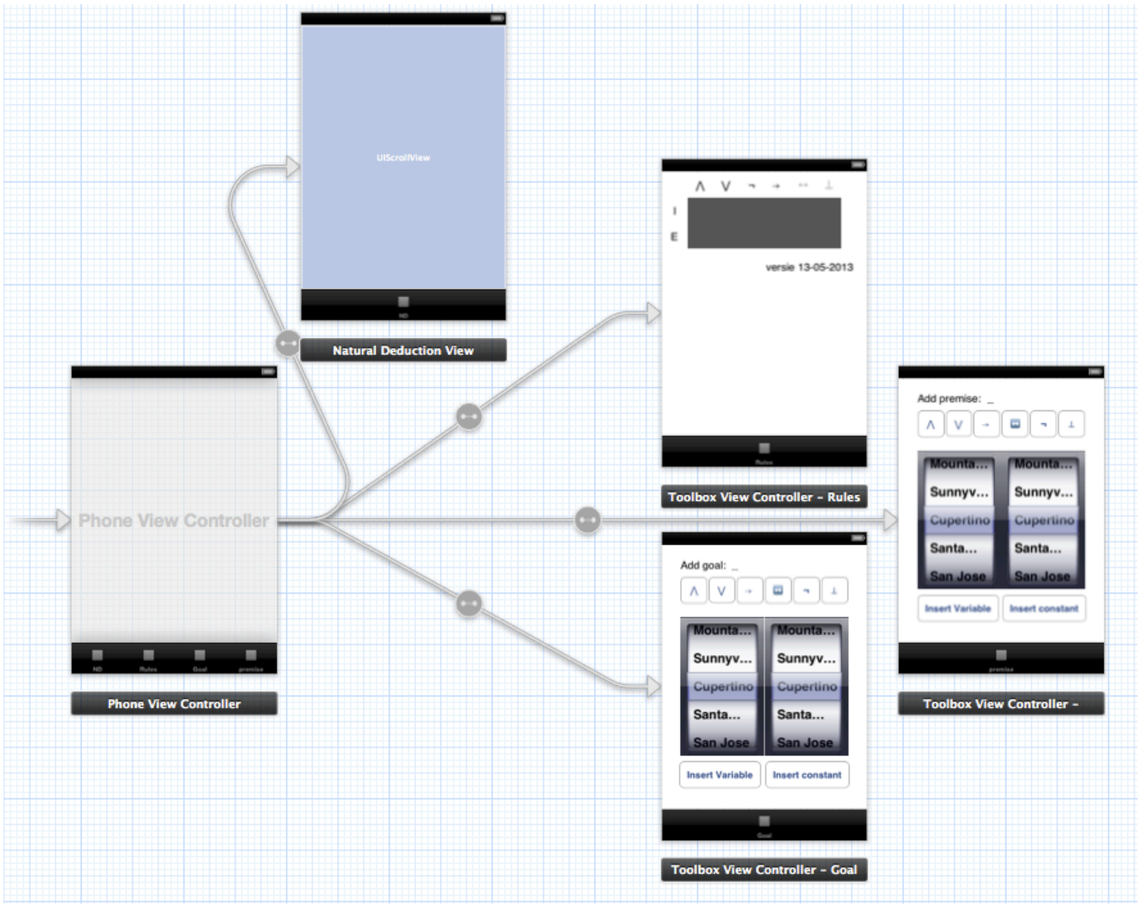


Figure 5: Screenshot of the iPhone storyboard

2.4 TopLevelViewController

Having two devices with a slightly different interface means the necessity of a slightly different behavior and thus code for both devices. This is implemented using a single protocol, with two different implementations. This protocol is called the TopLevelController and is implemented by IPadViewController and IPhoneViewController. Both the TopLevelControllers own an instance of the NaturalDeductionViewController and an instance of the ToolboxViewController. The NaturalDeductionViewController behaves identically for both the iPad and the iPhone while the ToolboxViewController behaves slightly differently for the iPad and the iPhone. See section 2.5 for more details about the ToolboxViewController.

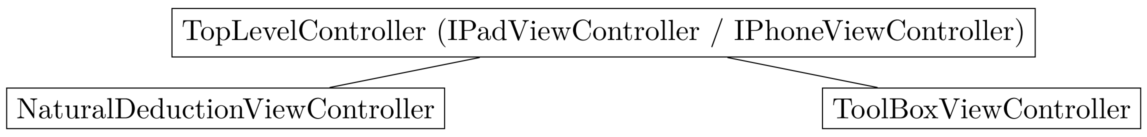

TopLevelController (IPadViewController / IPhoneViewController)
NaturalDeductionViewController
ToolBoxViewController

Figure 6: TopLevelController hierarchy

The TopLevelController protocol is defined as follows:

```objc
@protocol TopLevelController <NSObject>
/*
   Allocate one sound producer.
 */
@property(nonatomic,strong) Musician* musician;
/*
   Whether the device is an iPad. (otherwise: iPhone)
 */
@property BOOL isPad;

@property (strong, nonatomic) NaturalDeductionViewController * naturalDeductionViewController;
@property (strong, nonatomic) ToolboxViewController * toolboxViewController;

/*
   Important for determining when to allocate a new constant.
 */
@property (strong, nonatomic) AtomicProposition * newestConstantAvailable;

/*
   API for ToolBoxViewController to this object.
 */
-(void) addDeductionTreeToNaturalDeductionViewController:(ProofPiece*)deductionTree withBuildingBlock: (
      BuildingBlock*) buildingBlock;
-(void) addGoalBuildingBlockFromTree:(GoalProofPiece*) tree;
-(void) addPremiseBuildingBlockFromTree:(PremiseProofPiece*) tree;
-(Boolean) shouldAddTreeFromToolBoxToMainWithPoint: (CGPoint) location;
-(void) addDeductionTreesToToolBoxFromClasses: (NSArray*) classes;


/*
   API for NaturalDeductionViewController to this object
 */
-(void) boundVariableRemoved: (VariableProposition*) prop;
@end
```

2.5 ToolboxViewController

In order to manipulate a lego structure to alter it to the structure you have in mind you first need to add those basis blocks. The same applies to the app, in order to manipulate proofs you first need to add basic proofpieces. For that reason the ToolBoxViewController is created. The minimum task of this Controller should be the creation of the basic BuildingBlocks, which are all the blocks defined in section 1.2.

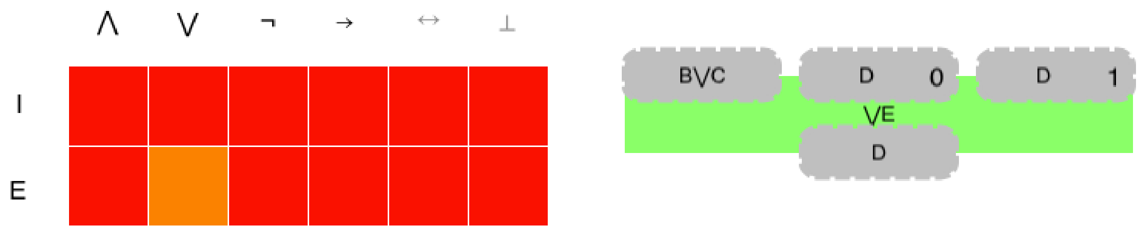


Figure 7: Screenshot of the simple proofpiece builder for iPad

In figure 7 a screenshot of the proofpiece creator is displayed in action. By hovering above the red tiles the users will be served with a preview of the corresponding proofpiece(s) on the right. Subsequently the users can drag that block into the NaturalDeductionViewControllers' view.

In addition we created two extra types of blocks. One being a goal block, which is the only block that has exactly one port and no conclusion, because it is the conclusion itself. The other being the premise block, which also has exactly one port: the conclusion port. Both of these blocks can be created in the ToolboxViewController by creating a (non-atomic) proposition. It is notable that this is the only way to insert propositional constants in the app.

Constants are useful because constants have to property of begin distinct to each other. For example constant "a" will never be unifiable with constant "b". When a student is attributed with the task of proving: "A→B", it is really the intention that he constructs a proof where "A" is not unified with "B" because then he actually proofed that "A→A". So if a student wants to proof "A→B", he would be helped by unifing this proposition with a constant variant, in this case: "a→b".

Consider the use case where a student wants to perform an exercise, he or she can simply add the premises and the goal. Thereafter the student can insert the proofPieces needed to finish the proof. If he or she ends up with a valid proof, with only green buildingBlocks and no unsatisfied dropZones, the proof is complete.

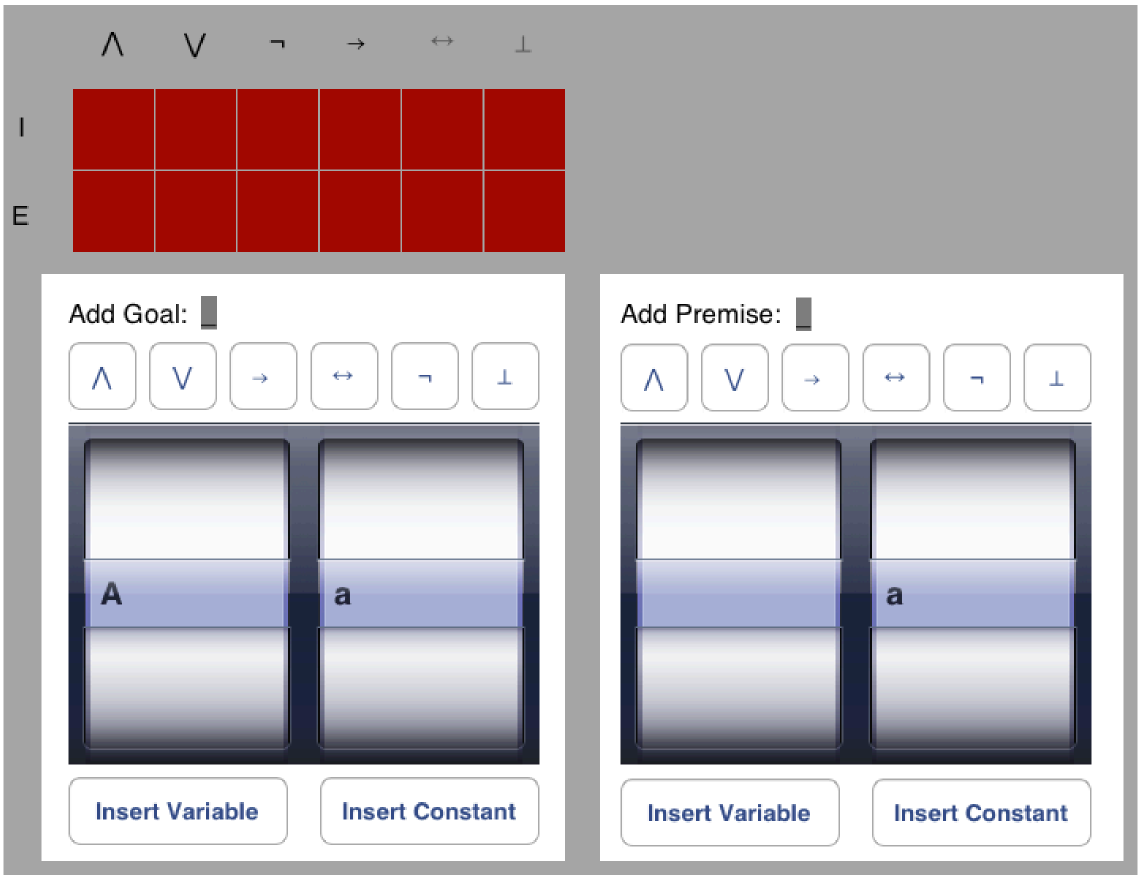


Figure 8: Screenshot of the simple proofpiece builder for iPad

In figure 8 a screenshot of the proof and premise builder is shown. On the iPad you can simply drag the ToolBoxViewControllers view upwards to extend the view. On the iPhone, those builders are under separate tabs.

2.5.1 Building a proposition for a goal or premise

A premise or goal is basicly a proofpiece with a single port and exists of a single proposition. The atoms of such a proposition are variables and constants. The goal proofpiece has its dropzone as input while the premise proofpiece has its dropzone as output. To insert a goal or premise, a user should first define a proposition that the proofpiece will carry. The App will have a panel where a proposition can be built. This panel is provided with some buttons, containing logic-connectives, and two lists, one containing variables and one containing constants. When you press a button, a connective, variable or constant is inserted. This is done according to the Polish notation. For example: If you insert "$\wedge$", the following two inputs will be consumed by this connective. So if you insert: "$\wedge$, a, b" this will result in "$a\wedge b$" and "$\wedge$, a, $\wedge$ b, c" will result in "$a\wedge(b\wedge c)$". Figure 9 is made by inserting in the following order: "$\wedge$, $\vee$, a, c, $\rightarrow$, a, $\vee$".

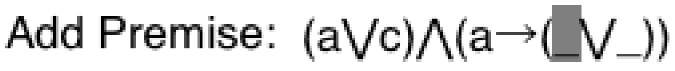

Add Premise: (a⋁c)⋀(a→(_⋁_))

Figure 9: Screenshot of the proposition builder in action

In figure 9 we can see that placeholders, represented with an underscore, indicate where input propositions are expected. When a Proposition contains no underscore it is a valid proposition and will be automatically transformed to a ProofPiece. Also the underscore that is being replaced as first is indicated with a gray background.
The proofpieces that represent the goal and premise blocks are respectively of the type GoalProofPiece and PremiseProofPiece.

3 The implementation

In this section I will show what kind of objects are used in the application and I will shed a light on how these are implemented. I will start with the object representing propositions and proofpieces which live on the model side followed by the BuildingBlock which is a graphical representation of a proofpiece. After that I will explain what valid manipulations are available.

When implementing this app it was a goal to use as much local relationships as possible instead of relationsships defined globally e.g. a top-level array. However I did not focus on proving efficient complexities, the idea of using these kind of relationships is that it will decrease complexity. However complexity is always defined as the runtime of a program, given the amount of input, but in this case the goal was also to increase modularity which ultimately leads to better readability and maintainability.
A Lego structure is also a set of local connections which sums up to a bigger shape.

3.1 Proposition

We want to do natural deduction on propositional logic. That means that we first have to define a mechanism that is able to deal with the propositional syntax and its properties. This mechanism exists of a bundle of Objects. The standard object is ”Proposition” and the other propositions inherit from this class. The most important feature of a Proposition is that it is able to perform an unification.
We can distinguish these Proposition classes for normal proposition logic: AtomicProposition which corresponds to the ”propositievariabele” (proposition variable) in the syllabus [4], VariableProposition which corresponds to the ”metavariabele” (meta variable) in the syllabus [4], PropositionAnd, OrProposition, ImplicationProposition, BiImplicationProposition, NotProposition, FalsumProposition. Finally there is a special class called UndefinedProposition, which represents the underscore mentioned in section 2.5.

The file Proposition.h, listed here below, indicates what messages will be implemented for all propositions.

```objc
// Proposition.h

@interface Proposition : NSObject{
    NSString* conectiveSign;
}
/*
  Get a duplicate of this proposition.
  The structure is copied, and the Variables are fresh. Constants will be original
  For example: duplicate( A ^ ( B ^ A ) ) ^ a => C ^ ( D ^ A ) ^ a
 */
@property( strong, nonatomic ) id duplicate;

/*
  This object that is responsible for managing the variable names.
  It functions as a library.
 */
@property( weak, nonatomic ) id<AlphabethTokenProvider> alphabethTokenProvider;

/*
   This boolean will be set to YES if giveBackTokenToProvider is called.
   It means this object does not longer own the tokens it once was assigned
 */
@property BOOL tokensGivenBackToProvider;

/*
   Standard initilizer
 */
-( id ) initWithAlphabethTokenProvider: ( id<AlphabethTokenProvider> ) tokenProvider;

/*
   Write the formule to a string according to the infix notation.
   This first should not get brackets because that is not necessary.
 */
-( NSString* ) printAsFirst: ( Boolean ) first;

/*
  Unify this proposition with the given proposition.
  The recursive rule for unification: if main connective match, try to match corresponding input propositions
        .
  Cycles are not allowed, and will fail ( return NO ). For example: A and ( A$\land$B ) are not unifiable.

  "-( Boolean ) unify:( Proposition* ) proposition" is a wrapper message which depends on "-( Boolean )
        unifyWith:( Proposition* ) proposition". The main logic is in the latter, but the first is able to
        try, commit or rollback a unfication.
 */
-( Boolean ) unify:( Proposition* ) proposition;
-( Boolean ) unifyWith:( Proposition* ) proposition;

/*
  RollbackTrialUnification and commitTrialUnification are messages called after an unification is finished.
        If the unification failed: rollbackTrialUnification is called. Otherwise commitTrialUnification is
        called.
  These methods can reset some variables.
 */
```

```
-( void ) rollbackTrialUnification;
-( void ) commitTrialUnification;

/*
 If an unification is undone: call this message.
 Be careful: this is no local operation but part of a global operation.
 */
-( void ) unUnify;

/*
 This function creates the same proposition, but the variables are replaced by its containing value, while
        normally an unified variable just has a reference to its counterpart.
 */
-( Proposition* ) deepAssignment;

/*
 Cleanup message.
 */
-( void ) giveBackTokenToProvider;

/*
 Helper function of the "duplicate" propery. If the property is nil, this message is called and stored in
        the "duplicate" variable.
 If a proposition is duplicated twice inside a proposition, these duplicates will be the same. ( this is
        important )
 */
-( id ) duplicateCopy;
/*
 Always call these message after getting a duplicate of a proposition. It deletes the duplicate cache.
 */
-( void ) clearDuplicateCopies;

/*
 Returns a set of al variables that are in this proposition. Even variables that are deeper are returned.
        This is important to do cycle detection.
 */
-( NSSet* ) variables;

/*
 For all variable names in this propostion, the number of references is increased by one.
 This is often done when a variable is initialized
 */
-( void ) increseReferencesToVariableNames;

/*
 A UndefinedProposition is not really a proposition. It is an placeholder for an yet to be defined
        Proposition. A proposition that has UndefinedProposition is typycally found in the goal/premise
        builder and should not occur in the main part of the application.
 This message will replace the first occurance of an UndefinedProposition with a given proposition.
 */
-( Proposition* ) replaceFirstUndefinedPropositionWith:( Proposition* ) proposition;
/*
 Indicates wether a proposition contains one or more UndefinedPropositions.
 */
```

```
-( BOOL ) hasUndefindPropositions;

/*
 Given an connective, return a new instance of that connective. If the connective has input propositions,
       they will be instantiated by UndefinedPropositions.
 */
+( Proposition* ) createUndefinedPropositionByConnectiveName: ( NSString* ) string
      withAlphabethTokenProvider: ( id<AlphabethTokenProvider> ) alphabethTokenProvider;
@end
```

3.1.1 Recursion

A Proposition is defined in a recursive way. A connective often comes together with zero, one or two Propositions, forming a bigger Proposition. For the reason of simplicity the Connective is programmed as the container of the Propositions it comes together with. Sending a message to a Proposition is for that reason often recursively defined. Typically a message is sent to a Connective, and then that connective sends the same message to its children.
In the code below this paradigm is demonstrated for the (and) proposition. These messages are similarly implemented for the other Proposition types.

```
// PropositionAnd.m

@implementation PropositionAnd
@synthesize leftProposition = _leftProposition;
@synthesize rightProposition = _rightProposition;

-( PropositionAnd* ) initWithAlphabethTokenProvider:( id<AlphabethTokenProvider> )tokenProvider withLeft:(
      Proposition * )left withRight:( Proposition * )right{
   self = [self initWithAlphabethTokenProvider:tokenProvider];
   if( self ){
      self.leftProposition = left;
      self.rightProposition = right;
   }
   return self;
}

-( NSString* ) printAsFirst:( Boolean ) first
{
   return [NSString stringWithFormat:@"%@%@^%@%@",
         ( first?@"":@"( " ),
         [self.leftProposition printAsFirst: false],
         [self.rightProposition printAsFirst: false],
         ( first?@"":@" )" )];
}

-( BOOL ) isEqual:( id )object
{
   if( [object isKindOfClass: [PropositionAnd class]] )
   {
      return [self.leftProposition isEqual:[object leftProposition]] && [self.rightProposition isEqual:[object rightProposition]];
   }
   return false;
}
-( Proposition* ) deepAssignment
{
  return [[PropositionAnd alloc] initWithAlphabethTokenProvider:self.alphabethTokenProvider withLeft:[self.leftProposition deepAssignment] withRight:[self.rightProposition deepAssignment]];
}


-( void ) rollbackTrialUnification
{
   [self.leftProposition rollbackTrialUnification];
   [self.rightProposition rollbackTrialUnification];
}
-( void ) commitTrialUnification
{
   [self.leftProposition commitTrialUnification];
   [self.rightProposition commitTrialUnification];
}

-( void ) unUnify{
   [self.leftProposition unUnify];
   [self.rightProposition unUnify];
}
-( Boolean ) unifyWith:( Proposition * )proposition{
   if( [proposition isKindOfClass:[PropositionAnd class]] ){
      return [self.leftProposition unifyWith:( ( PropositionAnd* )proposition ).leftProposition]
      && [self.rightProposition unifyWith:( ( PropositionAnd* )proposition ).rightProposition];
   }else if( [proposition isKindOfClass:[VariableProposition class]] ){
      return [proposition unifyWith:self];
   }
   return NO;
}

-( void ) giveBackTokenToProvider{
   [super giveBackTokenToProvider];
   [self.leftProposition giveBackTokenToProvider];
   [self.rightProposition giveBackTokenToProvider];
}
-( id ) duplicateCopy
{
   return [[PropositionAnd alloc] initWithAlphabethTokenProvider:self.alphabethTokenProvider withLeft:self.leftProposition.duplicate withRight:self.rightProposition.duplicate];
}
-( void ) clearDuplicateCopies
{
   self.duplicate = nil;
   [self.leftProposition clearDuplicateCopies];
   [self.rightProposition clearDuplicateCopies];
}

```

```
-( NSSet* ) variables
{
   return [[self.leftProposition variables] setByAddingObjectsFromSet:[self.rightProposition variables]];
}
-( void ) increseReferencesToVariableNames
{
   [_leftProposition increseReferencesToVariableNames];
   [_rightProposition increseReferencesToVariableNames];
}

-( Proposition* ) replaceFirstUndefinedPropositionWith:( Proposition* ) proposition
{
   Proposition * tryLeft = [self.leftProposition replaceFirstUndefinedPropositionWith:proposition];
   if( tryLeft )
   {
      return [[PropositionAnd alloc] initWithAlphabethTokenProvider:self.alphabethTokenProvider withLeft:
             tryLeft withRight:self.rightProposition];
   }
   Proposition * tryRight = [self.rightProposition replaceFirstUndefinedPropositionWith:proposition];
   if( tryRight )
   {
      return [[PropositionAnd alloc] initWithAlphabethTokenProvider:self.alphabethTokenProvider withLeft:
             self.leftProposition withRight:tryRight];
   }

   // Not for me: maybe an ancestor?
   return nil;
}

-( BOOL ) hasUndefindPropositions{
   return [self.leftProposition hasUndefindPropositions] || [self.rightProposition hasUndefindPropositions];
}

@end
```

3.1.2 VariableProposition

A VariableProposition is a special proposition.

```
// VariableProposition.h

@interface VariableProposition : Proposition
@property(strong, nonatomic) NSString* variableName;
@property(strong, nonatomic) Proposition * assignment;
@property(strong, nonatomic) Proposition * trialAssignment;

@end
```

In propositional logic a variable can be unified with another proposition. This unification means all the occurrences of the variable will be replaced by the proposition, under the condition that variable does not occur in the proposition.
When an unification is done in this application, we have to take into account that we

might want to undo the unification. When we replace all the occurrences of a variable, we would lose information about the original situation and it will become hard to undo an unification. For this reason the decision was made to not remove and substitute the variable, but to maintain the variable and set a reference in that variable to its content. The variable should now behave like the proposition it has assigned, until the variable is told to undo the unification.

Another property of a VariableProposition is that it has a name. That name is displayed while the proposition has no assignment. The VariableProposition is owner of that name for its own lifecycle. When the VariableProposition is removed, the name is given back to the name provider.

To build a big structure in Lego you have to click blocks onto each other. Since the clicking of proofpieces depends on unification and the unification eventually depends on the unification of variables, this is an important piece of code. Eventually this code helps us allowing only the valid actions, just like in Lego, where blocks have to be placed in a way they fit too.

```
// VariableProposition.m
-(Boolean) unifyWith: (Proposition*) proposition
{
// If we are unifing the same variables we are already unified. Skip the rest in that case.
   if([proposition isKindOfClass:[VariableProposition class]])
   {
      if([[self deepAssignment] isEqual:[proposition deepAssignment]]){
         return YES;
      }
   }
      // If this is the case, we are unifing this var with something that contains this var already. bad
           stuff!
      if([[proposition variables] containsObject:self.deepAssignment])
      {
         return NO; // don't cycel on me :(
      }
   Proposition * assignment = self.trialAssignment? self.trialAssignment : self.assignment;
   if(!assignment)
   {
      self.trialAssignment = proposition;
      return YES;
   }
   return [assignment unifyWith:proposition];
}
```

As we can see, not the assignment itself is set, but a temporary value is set. After the unification, if the whole unification has succeeded or failed respectively commitTrialUnification or rollbackTrialUnification is called.

Both are listed below and seem to be straightforward.

```
// VariableProposition.m

-(void) rollbackTrialUnification
{
   [self.trialAssignment rollbackTrialUnification];
   [self.assignment rollbackTrialUnification];

   self.trialAssignment = nil;
}

-(void) commitTrialUnification
{
   if(self.trialAssignment)
   {
      self.assignment = self.trialAssignment;
      self.trialAssignment = nil;
   }
   [self.assignment commitTrialUnification];
}
```

3.2 Proofpieces

In our metaphor, an atomic inference rule corresponds to a lego piece, with the inferred propositions corresponding to the shape of the lego piece (i.e. to its boundary, determinining how it may fit onto other lego pieces). Those pieces, corresponding to instances of inference rules, are called Proofpieces in the code and are earlier described in section 1.2.

There are an infinite number of correct ProofPieces, but we only want to equip the app with a set of minimally necessary ProofPiece types. In our case that turns out to be one or two (two i.e. in the case there is a left and right variant) introduction- and one or two elimination-Proofpieces per connective. These proofpieces correspond to the inference rules found in the syllabus [4].

In the app this results in the classes: AssumptionProofPiece, AndIntroductionConnectiveProofPiece, AndEliminationLeftProofPiece, AndEliminiationRightProofPiece, OrIntroductionLeftProofPiece, OrIntroductionRightProofPiece, OrEliminationProofPiece, ImplicationIntroductionProofPiece, ImplictionEliminationProofPiece, NotIntroductionProofPiece, NotEliminationProofPiece, BiimplicationIntroductionProofPiece, BiImplicationLeftEliminationProofPiece, BiImplicationRightEliminationProofPiece, FalsumEliminationProofPiece, RAAProofPiece, GoalProofPiece, PremiseProofPiece.

ProofPiece.h is listed as follows:

```
// ProofPiece.h

@interface ProofPiece : NSObject<NSCopying>

/*
   An array with the keys of the input ports. The ports are name with a semantical useful name for the sake
         of debugging. This
   For example, if you have an and-connective, this array could be: "proof1","proof2".
 */
@property (nonatomic, strong) NSArray * proofInputVariables;
```

```objc

/*
   This is a dictionary of portnames and references to other proofpieces.
 */
@property (strong, nonatomic) NSMutableDictionary * bindings;

/*
   This is a dictionary of portnames and the variables that represent that ports.
 */
@property (strong, nonatomic) NSMutableDictionary * variableBindings;

/*
   This is a dictionary of portnames and OwnerTuple. An ownertuple is a object that describes the
          relationship between a proofpiece-port and one or more PremiseProofPieces.
 */
@property (strong, nonatomic) NSMutableDictionary * assumptionOwnerships;

/*
   The ID of this proofpiece. A unique ID means a unique proofpieces.
 */
@property (strong, nonatomic) NSNumber * proofpieceSubId;

/*
   The ID of the root proofpiece.
 */
@property (strong, nonatomic) NSNumber * proofpieceRootId;
@property (weak, nonatomic) id<AlphabethTokenProvider> alphabethTokenProvider;
@property (nonatomic) BOOL isRemoved;

/*
   Open assumptions are cached in this set. Don't forget to call "recalculateOpenAssumptions" first.
 */
@property (strong, nonatomic) NSMutableSet * openAssumptions;

/*
   The name of the proofpiece type. For example "and-elimination" or a shorthand for that.
 */
@property (strong, nonatomic) NSString * text;

/*
  Initializer(s) of the ProofPiece. It gets a reference to the token provider.
  A very nice feature is the boolean variable "createAlsoAssumptions". If it is set to true all the
        assumptions that are allowed to depend on this ProofPiece will be created and if possible, attached
        to this proofpiece.
 */
-(id) initWithAlphabethTokenProvider: (id<AlphabethTokenProvider>) alphabethTokenProvider;
-(id) initWithAlphabethTokenProvider: (id<AlphabethTokenProvider>) alphabethTokenProvider
      createAlsoAssumptions:(BOOL) createAlsoAssumptions;

/*
   Helper function for -(id) copyWithZone:(NSZone *)zone
   copyWithZone returns a copy of the proofPiece.
 */
-(void) copyChildrenTo: (ProofPiece*) copyOfTree withZone:(NSZone*) zone;
```

```

/*
   With this method you can apply two proofpieces to each other. It will attach the conclusion of this
         Proofpieces to the port named by the "key" value on the "copiedTree" input var.
 */
-(void) setBinding: (ProofPiece *) copiedTree forKey:(NSString*)key;

/*
   Remove the childtree "tree".
 */
-(NSString*) removeChild: (ProofPiece *) tree;

/*
   This method will undo ALL unification in the tree, and its children.
   This method is ussually called on the root-tree
 */
-(void) undoUnifications;

/*
   This method will redo all the unification in the tree and its childen.
   undoUnifications and reEvaluateUnifications are often called when a single unification needs te be undo.
         First: remove the binding and than undo and reEvaluate the unifcations.
 */
-(void) reEvaluateUnifications;

/*
   Calculates what assumptions are still not covered by its owner.
 */
-(NSMutableSet*) recalculateOpenAssumptions;

-(void) giveBackTokensToProvider;

/*
   This function will create a new proofpiece with the same structure as the current proofpiece, but with
         new variables.
   AssumptionProofpieces that are stil "open" are not duplicated but copied instead.
   For example: if you copy: [A]1 that results to [A]1.
   If you copy: [A]1 ... (A->A)1 that results to [B]1 ... (B->B)1
 */
-(ProofPiece*) duplicateWithOpenAssumptionsOnRoot:(NSMutableSet*) unconnectedOwnerTuples;

/*
   Called after duplicating a ProofPiece. This will remove cached variables created while making a duplicate
          of this ProofPiece.
 */
-(void) clearDuplicates;

/*
   This message will help preventing creating cycles in a proof.
   If this message return YES, the currect ProofPiece contains a cycle and should be considered invalid.
 */
-(BOOL) noCyclesWithTreeIdsSeen: (NSArray*) treeIdsSeen;

/*
```

```
    A message that reports to the tokenprovider that the propositions (variables) are referenced one time
         more.
 */
-(void)claimVariableNames;

@end
```

3.2.1 BuildingBlock, DropZone

Now we got our proofpieces defined, we need a way to interact with them. Before exploring the controller which is responsible for this, we will take a look at the graphical counter object of the Proofpiece; the BuildingBlock. The BuildingBlock has a blockId, which corresponds to the ProofPieceSubId on the ProofPiece. The buildingblock also has an array with so called DropZones. A dropzone represents a port. Since the BuildingBlock and the DropZone are views, they do not implement business logic. Instead the code focuses on the representation and gesture recognition and delegation. The BuildingBlocks will be draggable, and the dropzones are fixedly attached to the BuildingBlock.
A BuildingBlock is typically colored green, but is red when the corresponding Proofpiece depends on an assumption that has not been made yet in the proof.

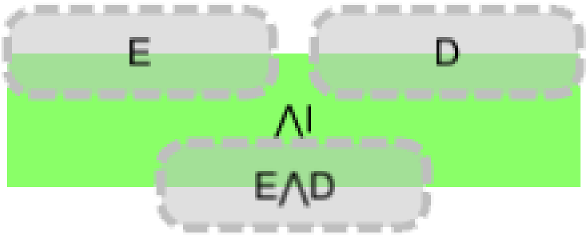
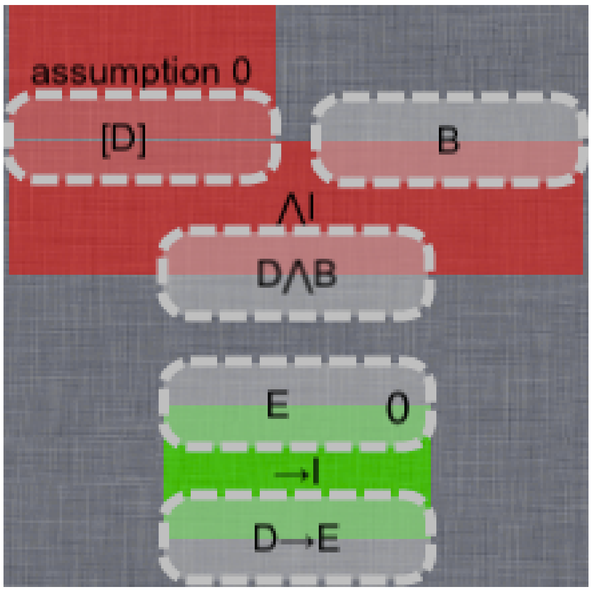


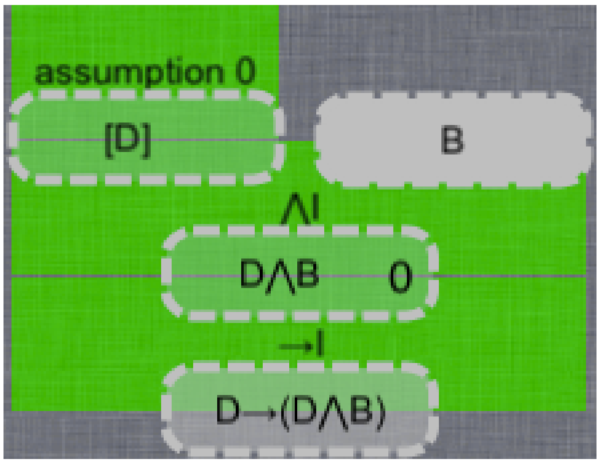
Figure 10: Example of a building block

3.2.2 assumptions

When doing natural deduction on propositional logic, we want to use assumptions. If an assumption is made, it has to be retracted somewhere in the proof. If the assumption remains unretracted the overall proof is not complete. In the application, a proofpiece will be colored red instead of green when it depends on an unretracted assumption. If an invalid tree has a branch that is valid on its own, that branch is colored green. Assumptions have uniquely identifying labels which can also be found on the port that retracts the assumption.



(a) Example: Assumptions 0 is not retracted yet.



(b) Example: Assumptions 0 is here retracted.E is unified by D∧B

Figure 11: Example assumptions

3.2.2.1 OwnerTuple

Ports of inference rules in natural deduction may have a dependency on assumptions, which are programmed as AssumptionProofPiece. In the tool, the inference rule and its ports' assumptions are separate proofpieces between which a connection is maintained. This connection is implemented with an OwnerTuple. The OwnerTuple can have a name which is typically a number. It does not refer to the AssumptionProofPiece but the AssumptionProofPiece does always refer to its OwnerTuple. Also the OwnerTuple does not refer to a single port but it refers to the port name and a list of ProofPieces. These ProofPieces are all hardcopies of the same ProofPiece. The usecase where multiple ProofPieces are around is when a ProofPiece is being connected (clicked) on another proofpiece and a temporary copy of the proofpiece is introduced.

```objc
// OwnerTuple.h

@interface OwnerTuple : NSObject<NSCopying>
/*
   A set of (the same!) proofpieces.
 */
@property(nonatomic, strong) NSMutableSet* proofpieces;
/*
   Used to point out the port on the proofpiece(s)
 */
@property(nonatomic, strong) NSString * key;

@property(nonatomic, weak) id<AlphabethTokenProvider> tokenProvider;
@property NSUInteger numberOfAssumptions;
@property(strong, nonatomic) NSString * assumptionReferenceToken;
/*
   A placeholder for a copy of this ownerTuple. When the tree is being copied this object is copied too.
 */
@property(strong, nonatomic) OwnerTuple * duplicate;

/*
   Straightforward initializers.
 */
-(id) initWithTree: (ProofPiece*) tree withKey: (NSString*) key withTokenProvider: (id<
      AlphabethTokenProvider>) tokenProvider;
-(id) initWithTrees: (NSMutableSet*) trees withKey: (NSString*) key withTokenProvider: (id<
      AlphabethTokenProvider>) tokenProvider;
/*
   Shortcut to the proposition of the port.
 */
-(Proposition*) proposition;
/*
   Name of the relationship (lazy instantiated)
 */
-(NSString*) lazyAssumptionReferenceToken;
-(void) checkForGivingBackToken;
@end
```

3.2.3 NaturalDeductionViewController

The NaturalDeductionViewController is one of the important classes in this app. It basically controls a canvas on which the BuildingBlocks interact. It also can receive messages from its parent ViewController, for example to insert a new proofpiece. This class is also charged with the responsibility of controlling tokens used in the app. It holds a registry of what tokens are already used which makes it possible to recycle some tokens.

Below are all the messages listed that are declared in the NaturalDeductionViewController.

```
/*
   Create three alphabeths for: variables, constants and relationship
 */
-(void) initAlphabeth


/*
   The sender is a buildingblock. Try to find another buildingblock that intersects one of its dropzones
         with a dropzone of the sender.
   If such a buildingblock is found, the message "start" is called to check wether the transaction can
         acctually be started.

 */
- (DroppingTransaction*) findNewTransactionWithSender: (id) sender


/*
   A buildingblock is touched and dragging starts.
 */
- (void) startDragging:(UITouch *)touch sender:(id)sender


/*
   A buildingblock is being dragged.
   Check if transaction invoked on this object is still valid.
   If there are no (valid) transactions for this block: search for a new one. ( findNewTransactionWithSender
          is called).
 */
- (void) dragging:(UITouch *)touch sender:(id)sender


/*
   This function will drag a complete tree (And its subtrees).
 */
- (void) dragSubTrees: (ProofPiece*) tree withOffset: (CGPoint) offset


/*
   This function is called when you stop dragging. If a transaction was open, it will be comitted!
 */
- (void) dropping: (UITouch *)touch sender:(id)sender


/*
   Not used yet
 */
- (void) doubleTapped:(id)sender


/*
```

```
    Add a proofPiece, and its child proofpieces to the main view.
 */
- (NSNumber *) addTree: (ProofPiece *) proofPiece


/*
    Add a proofpiece, but if it was already attached to a BuildingBlock, add that buildingblock instead of
           creating a new one.
 */
- (NSNumber *) addTree: (ProofPiece*) tree withBlock: (BuildingBlock *) block


/*
    Try to start a transaction.
    Check for cycle detection.
    Try to unify the propositions.
    Return whether the starting of the transaction succeeded.
    If this function fails it could start a sound.
    If this function succeeds the ProofPiece that has the conclusion on it will be copied and display as if
           the block was already dropped.
 */
- (BOOL) start: (id)transaction


/*
    When a BuildingBlock is dropped on another BuildingBlock this function is called.

    The temporary block introduced on start will remain, and the original block will be deleted.
 */
- (void) commit:(id)transaction



/*
    Rollback is called when you move a BuilingBlock away from an other BuildingBlock.
    The temporary block is removed.
 */
- (void) rollback:(id)transaction


/*
    Set this Boolean for the buildingblocks corresponding to a particular tree.
 */
- (void) changeTransactionOfTree:(ProofPiece*)tree to: (Boolean) isInTransaction


/*
    Set this Boolean for the buildingblocks corresponding to a particular tree.
 */
- (void) changeLockForBreakingOfTree:(ProofPiece*)tree to: (Boolean) lockForBreaking


/*
    Cut a deductiontree in two parts.
 */
– (void) cutProofpieceOnPiece: (ProofPiece *) tree formatTree: (Boolean) formatTree



/*
   Retrieve all the root–ProofPieces
 */
–(NSMutableArray*) getRootPieces


/* Remove a proofPiece */
– (void) removeProofPiece: (ProofPiece *) proofPiece



/*
   Color the BuildingBlock red or green.
 */
–(void) recolorAssumptions: (ProofPiece*) proofPiece withOpenAssumptions: (NSMutableSet*) openAssumptions


/*
   Reevaluate the strings for the dropzones.
 */
–(void) relabelDropzones: (ProofPiece*) proofPiece
–(void) relabelDropzones: (ProofPiece*) proofPiece fromDropZone: (DropZone*) dropZoneConclusion


/*
   Format the treesizes in two round, a pre–proces round (startFormatTreeSizes) and a finish round (
          finishFormatTreeSizes)
 */
–(void) formatTreeSizes: (ProofPiece *) tree withPoint:(CGPoint)start



/*
   Calculate the widths
 */
– (NSUInteger) startFormatTreeSizes: (ProofPiece *) tree withOffset:(CGPoint) start


/*
   Set the widths calculated in at "startFormatTreeSizes"
 */
– (void) finishFormatTreeSizes: (ProofPiece *) tree withStart: (CGPoint) start

– (void) movePositionOfTree:(ProofPiece*)tree withPosition: (CGPoint) position

/*
   Reorder the blocks. Set every child in front of its parent
 */
– (void) reorderBlockOfTree:(ProofPiece*)tree
```

```


/*
   Check for cycle detection
 */
-(BOOL) noCyclesOnTarget:(ProofPiece*)targetTree withSubjectTree:(ProofPiece*)subjectTree forKey: (NSString
      *) key


/*
 AlphabethTokenProvider protocol implementation
 */
-(NSString*) getNewTokenOfType:(NSString *)type


/*
   Tokenprovider method to decrease (release) a token count
 */
-(void)decreaseReference:(id)token forType:(NSString *)type


/*
   Tokenprovider method to increase (allocate) a token count
 */
-(void)increaseReference:(id)token forType:(NSString *)type


/*
   A ProofPiece with a corresponding BuildingBlock are insertered at a given position.
 */
-(void) insertTreeFromToolbox: (ProofPiece*) tree withBlock: (BuildingBlock*) block onPosition: (CGPoint)
      position


/*
   If a longpressed is recognized, you get the options to copy or remove a proofpiece.
 */
-(void) longPressed: (UILongPressGestureRecognizer*) sender


/*
   Open the modal window, add modifier buttons like: duplicate and remove.
   Idea: Make here a TEX-output thingy, or a share button!
 */
-(void) startEditing: (ProofPiece*) tree


/*
   Bring the buildingblocks corresponding to a proofpiece to the front.
 */
-(void) bringProofPieceToFront: (ProofPiece*) tree


/*
```

```
   Close the modal window.
 */
-(void) cancelEditing


/*
   The copy button was pressed, copy the current editing tree, and make some cool animation to divide them
         from each other.
 */
-(void) duplicateCurrentEditingProofPiece


/*
   "Are you sure you want to delete this proofPiece??"
 */
-(void) confirmDeleteCurrentEditingProofPiece


- (void)alertView:(UIAlertView *)alertView clickedButtonAtIndex:(NSInteger)buttonIndex


/*
   Delete the current editing tree.
 */
-(void) deleteCurrentEditingTree
/*
   Delete a tree (used in deleteCurrentEditingTree)
 */
-(void) deleteTree: (ProofPiece*) tree


/*
   Get access to the musician via its parent
 */
-(Musician*) musician
```

3.2.3.1 Token Provider

A token provider can serve tokens of different types i.e. variablenames, constantnames and ralationnames. The token provider is a protocol, which is adopted by the NaturalDeductionViewController.

The Protocol is listed below:

```
@protocol AlphabethTokenProvider <NSObject>
- (NSString*) getNewTokenOfType:(NSString*)type;
- (void) decreaseReference: (id) token forType: (NSString*) type;
- (void) increaseReference: (id) token forType: (NSString*) type;
- (void) initAlphabeth;

@property(strong,nonatomic) NSMutableArray * boundVariables;
@end
```

3.2.4 Manipulations

We can distinguish five manipulations we can perform on proofpieces, which are completely analogue to the manipulations possible on Lego. First we need to *initialize* a proofpiece. Secondly we want to *combine* some proofpieces. After that we might want to correct a mistake by *splitting* a proof. As an extra we implemented the possibility of *duplicating* a proofpiece. Finally a proof might be *removed*.

3.2.4.1 Initializing a Proofpiece

Every type of inference rule has its own behavior, just like every differt type of lego block has a different shape and interacts in another way to other blocks. The behavior of Proofpiece is determined when it is initialized.

Below a code snippet of the ImplicationIntroductionProofPiece initializer to illustrate the pattern used on initialization.

```
-(id) initWithAlphabethTokenProvider: (id<AlphabethTokenProvider>) alphabethTokenProvider{
    /*
       Send to the main init.
     */
    self = [self initWithAlphabethTokenProvider:alphabethTokenProvider createAlsoAssumptions:NO];
    return self;
}

-(id) initWithAlphabethTokenProvider: (id<AlphabethTokenProvider>) alphabethTokenProvider
     createAlsoAssumptions:(BOOL)createAlsoAssumptions
{
    self = [super initWithAlphabethTokenProvider: alphabethTokenProvider];
    if(self)
    {
       // Set the text of the block, here: implication introduction
       self.text = @"$->$I";

       // What input-ports do we have? one named: consequent.
       _proofInputVariables = [[NSArray alloc] initWithObjects: @"consequent", nil]; // Readonly

       // Store a reference to the tokenprovider.
       self.alphabethTokenProvider = alphabethTokenProvider;

       // Create proposition, used for in- and ouput ports but also assumptionProofPieces.

       // A
       Proposition * antecedent = [[VariableProposition alloc] initWithAlphabethTokenProvider:
              alphabethTokenProvider];
       // B
       Proposition * consequent = [[VariableProposition alloc] initWithAlphabethTokenProvider:
              alphabethTokenProvider];
       // A -> B
       Proposition * implication = [[ImplicationProposition alloc] initWithAlphabethTokenProvider:self.
              alphabethTokenProvider withAntecedent:antecedent withConsequent:consequent];
```

```

      _antecendent = antecedent;

      // Attach the proposition (variable) to the input port "consequent"
      [self.variableBindings setObject:consequent forKey: @"consequent"];

      // Attach the proposition (implication prop.) to the output port.
      [self.variableBindings setObject:implication forKey:@"conclusion"];

      // If we would like to auto create a single instance of assumption that depend on a port in this
            proofpiece, do so.
      if(createAlsoAssumptions)
      {
         // First create an ownertuple (so we can compare ownerships of proofpieces).
         OwnerTuple * ownerConsequent = [[OwnerTuple alloc] initWithTree:self withKey:@"consequent"
               withTokenProvider:self.alphabethTokenProvider];
         // This is how the ownership is stored for a port. (in this case: "consequent")
         [self.assumptionOwnerships setObject:ownerConsequent forKey:@"consequent"];

         // create the assumption.
         AssumptionProofPiece * assumption = [[AssumptionProofPiece alloc] initWithProposition:antecedent
               withOwner:ownerConsequent withAlphabethTokenProvider:self.alphabethTokenProvider];


         // In this case we can connect the created assumption to the proofpiece. If that is not possible
               they should not be (directly) attached.
         [self setBinding:assumption forKey:@"consequent"];
      }

      // tell the tokenprovider what tokens we used.
      [self claimVariableNames];
   }
   return self;
}
```

3.2.4.2 Combining proofpieces

Combining proofpieces happens through unification via the ports. In the simple case: if the proposition of the conclusion port of a proof piece is unifiable with the proposition of an input port of a proof piece, these ports are clickable with each other. If a user decides to click these ports on each other the unification will be kept and the unification is reflected in the propositions of both the proofpieces. Also the proofpieces are now attached to each other and form together one bigger proofpiece.

In the NaturalDeductionViewController the following message is responsible for handeling starting a transaction. It returns TRUE if this transaction is valid:

```
/*
   Try to start a transaction.
   Check for cycle detection.
   Try to unify the propositions.
   Return whether the starting of the transaction succeeded.
```

```
  If this function fails it could start a sound.
  If this function succeeds the ProofPiece that has the conclusion on it will be copied and display as if
       the block was already dropped.
 */
– (BOOL) start: (id)transaction
```

This message is called when a transaction should be committed.

```
– (void) commit:(id)transaction
```

This message is called to undo a started transaction and its unifications.

```
– (void) rollback:(id)transaction
```

3.2.4.2.1 Unification

Unification is done via an algorithm that is comparable to the unification algorithm of Montanari and Martelli [2]. A big difference is that unifications are stored locally, and also the original VariablePropositions remain stored. If you for example unify a Variable ”X” with a constant ”a” than X will yield a reference to a, but it will never be replaced by its containments.

The following code demonstrates how the unify message works. It tries unifyWith, which carries the recursion. When this trial code is ran, some temporary values are set for the proposition and its child propositions. If no problem is encountered, the message commitTrialUnification is called to set the temporary as the final value.
When a problem is encountered; the message rollbackTrialUnification is called to clean up the temporary variables.

```
–(Boolean) unify:(Proposition *)proposition
{
   if([self unifyWith:proposition])
   {
      [self commitTrialUnification];
      [proposition commitTrialUnification];
      return YES;
   }
   [self rollbackTrialUnification];
   [proposition rollbackTrialUnification];
   return NO;
}
```

3.2.4.2.2 Drag and Drop

Drag and Drop is the most obvious paradigm for manipulating your proofpieces. The presentation of a proofpiece is a quadrangle. As discussed earlier: a proofpiece has multiple ports. These ports are represented by a quadrangular pieces on the proofpiece, with rounded and dashed borders. A proofpiece can be dragged around. If a port of proofpiece intersects (in the 2d space) another port, the program will determine whether these ports are able to be attached on each other i.e. whether the proposition on their respective ports are unifiable or not and no loops will occur. If the latter is the case and the user drops the proofpiece, by releasing its finger from the screen, the two proofpieces will become one bigger proofpiece.

In the NaturalDeductionViewController the following messages are responsible for handeling a drag event:

```
/*
   A buildingblock is being dragged.
   Check if transaction invoked on this object is still valid.
   If there are no (valid) transactions for this block: search for a new one. ( findNewTransactionWithSender
          is called).
 */
- (void) dragging:(UITouch *)touch sender:(id)sender

/*
   The sender is a buildingblock. Try to find another buildingblock that intersects one of its dropzones
         with a dropzone of the sender.
   If such a buildingblock is found, the message "start" is called to check wether the transaction can
         acctually be started.

 */
- (DroppingTransaction*) findNewTransactionWithSender: (id) sender
```

3.2.4.3 Splitting

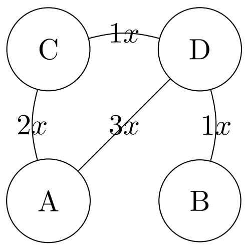
The opposite action of combining, in this case, would be splitting. It is really desirable to be able to split a proofpiece e.g. in the case the user made a wrong connexion and wants to correct it. Because this is the opposite action of combining, we want to undo the unification done when combining.

3.2.4.3.1 Ununification

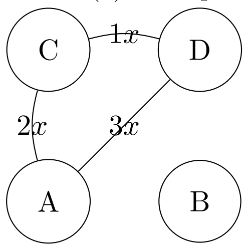
We dealt with a very interesting problem: the undoing of a previously done unification. Since all the (known) unification algorithms do not keep count of the number of the references made, it is impossible to simply release an unification when it's undone. The alternative is to recalculate all the unifications except the unification you want to undo.

Unfortunately we had to choose for the latter. This small subproblem was about to swallow more time than it should.

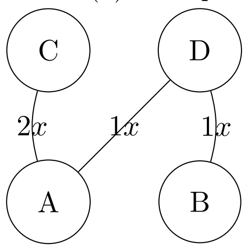
A small introduction to the problem is illustrated at Figure 12. Assume Figure 12a as an environment with four variables: A, B, C and D. A is unified with C two times, A and D three times, C and D one time and D and B are unified one time. This results in a situation where all the variables are unified to the same variable. If we ununify D with B one time, the weight of the edge between them is lowered by one. Since the new weight will be zero, the edge will be removed resulting in figure 12b. We can now say A, C and D are different from B because they are not necessarily unified. It is not always the case that ununfication leads to variables becoming independent. Consider Figure 12c, where altough A and D have been ununified twice and C and D are ununified once, all the variables are still unified to each other.



(a) Example 1



(b) Example 2



(c) Example 3

Figure 12: Three examples of unification environments

3.2.4.3.2 Drag and Split

The action of splitting two directly connected atomic proofpieces is implemented by touching both the proofpieces, with two separate fingers, followed by dragging the pieces away from each other. When the pieces are seperated wide enough from each other, the dividing action is immediate. The latter comparison is made in the "$-(void)dragging : (UITouch*)touchsender : (id)sender$" message

3.2.4.4 Duplicating a proof

Though it is not possible to duplicate a Lego structure in a single action, such a manipulation would be really helpful. This manipulation is therefore considerd as bonus we thought we would make the user happy with.

3.2.4.4.1 Instantiating a duplication

There are multiple ways to look at a duplication. One way is copying the complete proof and the other where only the structure is copied. Since we simply do not want connective proofs te occur more than once, we chose the latter. However there is one exception: an assumption may be used multiple times. Therefore will assumptions that are not yet connected to their assumers be hardcopied, which also means it will contain the same OwnerTuple.

Consider a ∧-introduction atomic proofpiece yielded with the proposition A∧B as conclusion, then duplicating would mean create a proofpiece yielded with a copy of the structure of A∧B as conclusion. In this case the conclusion of the duplicated proofpiece could be: C∧D.

The copying is done by the following message on the ProofPieces:

```
/*
   This function will create a new proofpiece with the same structure as the current proofpiece, but with
         new variables.
   AssumptionProofpieces that are stil "open" are not duplicated but copied instead.
   For example: if you copy: [A]1 that results to [A]1.
   If you copy: [A]1 ... (A->A)1 that results to [B]1 ... (B->B)1
 */
-(ProofPiece*) duplicateWithOpenAssumptionsOnRoot:(NSMutableSet*) unconnectedOwnerTuples;
```

3.2.4.4.2 Context menu

Coping of a proofpiece can be trigger by opening a context menu, as seen in figure 13, on the proofpiece and then simply pressing the duplicate button.

The following methods from the NaturalDeductionViewController are involved with handeling the context menu.

```
/*
   If a longpressed is recognized, you get the options to copy or remove a proofpiece.
 */
-(void) longPressed: (UILongPressGestureRecognizer*) sender



/*
   Open the modal window, add modifier buttons like: duplicate and remove.
   Idea: Make here a TEX-output thingy, or a share button!
 */
-(void) startEditing: (ProofPiece*) tree



/*
   Bring the buildingblocks corresponding to a proofpiece to the front.
 */
-(void) bringProofPieceToFront: (ProofPiece*) tree
```

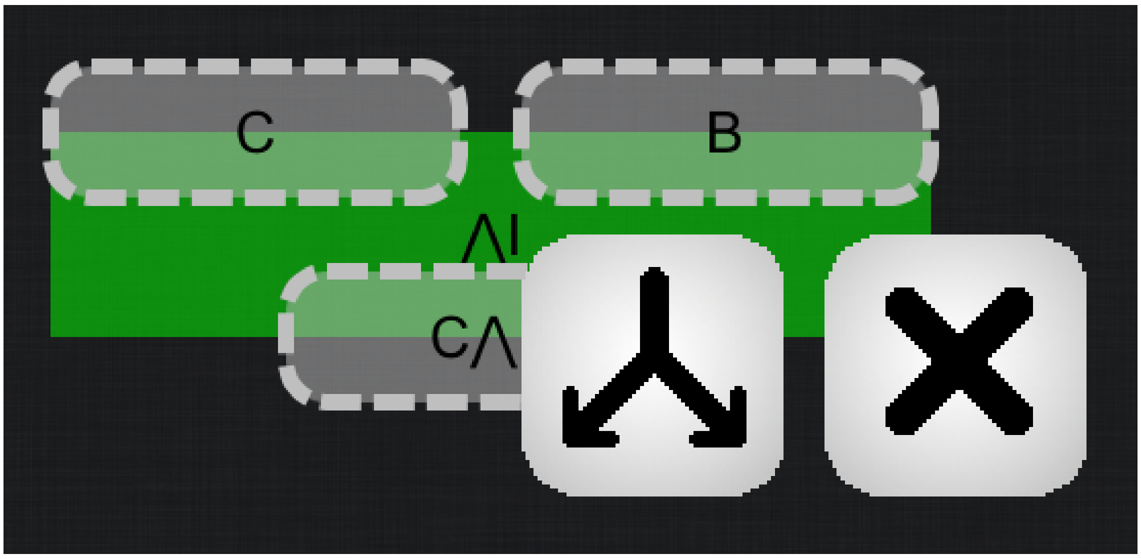


Figure 13: Screenshot of a context menu

```


/*
   Close the modal window.
 */
-(void) cancelEditing

/*
   The copy button was pressed, copy the current editing tree, and make some cool animation to divide them
          from each other.
 */
-(void) duplicateCurrentEditingProofPiece

/*
   Delete the current editing tree.
 */
-(void) deleteCurrentEditingTree
```

3.2.4.5 Removing a proof

The final manipulation is a final manipulation. I mean of course the removal of a proof.

The deletion is done by the following code in the NaturalDeductionViewController:

```
/*
   Delete a tree (used in deleteCurrentEditingTree)
 */
-(void) deleteTree: (ProofPiece*) tree
```

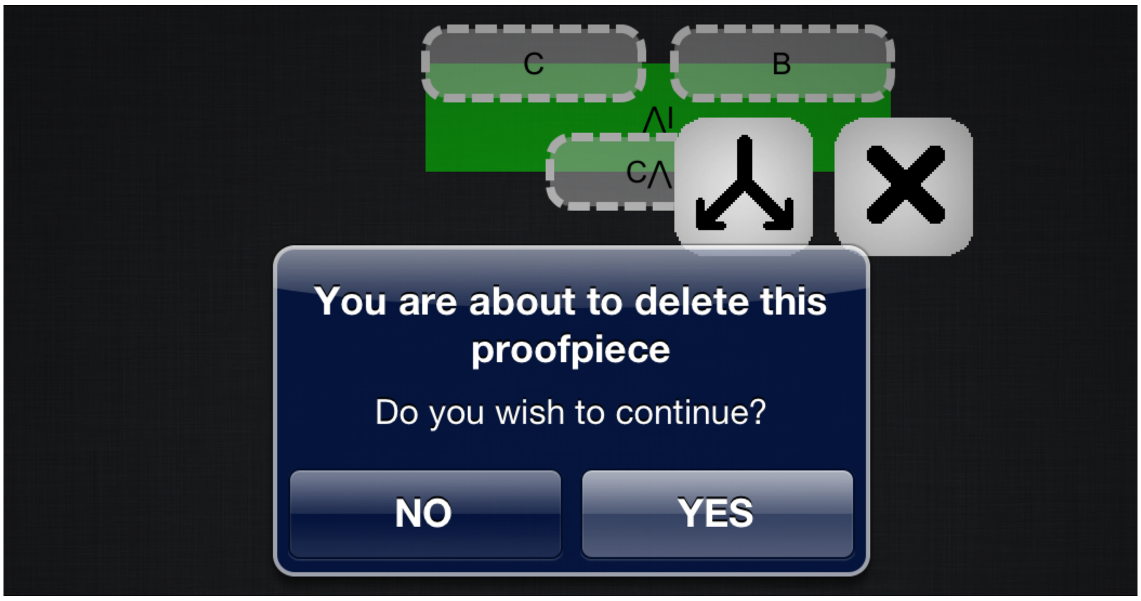


Figure 14

3.2.4.5.1 Context menu

A proof can be removed via the context menu, described in section 3.2.4.4.2, by pressing the delete button. A confirmation conversation will start which which can be seen in figure 14.

The following message of the NaturalDeductionViewController are involved beside the once mentioned in section 13:

```
/*
   "Are you sure you want to delete this proofPiece??"
 */
-(void) confirmDeleteCurrentEditingProofPiece

/*
   Delete the current editing tree.
 */
-(void) deleteCurrentEditingTree
```

4 Future features

As no single software project ever is completed, this app is also not finished yet. The number of bugs are as we speak at an acceptable level for the beta. But there is still a small amount of known bugs, and most probably some bugs that have still not exposed their selves.
There are also some features we are really excited about, and we really wish to implement

them after finishing this project. Below a small overview of some of those features.

4.1 Sharing

It would be a "cool" feature if someone could just post a proof on their Facebook, but there are other ways to share a proof. You might think about a LaTeX-export button, so a person can one tap export a proof to a report. Another nice idea is that students and teachers can send their current state/proof to each other e.g. via internet or bluetooth. A sequel to the latter idea is that a teacher/professor can send exercises to his/her students!

4.2 Saving

Saving a proof would be really useful. A simple example would be that a user saves it state and can simply load that state again. But another example would be saving a complete proof into one new proofPiece, which can be reused. An example of such a proofpiece could be the proof of the excluded middle law! Saving could be done locally but also on iCloud, which enables a user to create a proof on you iPad, and finish it on your iPhone (while sitting in the overcrowded bus to the Uithof).

4.3 Real multiplatform

If this app proofs to be a success we should consider expanding our scope. In other words: Android and pc support. Maintaining and creating three or four code bases would be tough. In that case we could reconsider the use of HTML5.

4.4 Interaction

Ofcourse we could make the app more powerful by adding more ways to interact with it. For example We would like to add the possibility to add an assumption directly via a candidate owner of that assumption.

5 About this project

This project initiated as my bachelor thesis project in November 2012 and was intended to terminate in Februari 2013. The proposed study load was 7.5 European Credit Transfer System points which would mean 210 hours available for the entire project. Eventually it seemed we underestimated the project because the project consited of a lot of small tasks and side issues which all demanded a considerable lot of time, resulting in consuming approximately around 300 hours. I will sum most of the aspects in the following subsections.

5.1 Settings our goal

It all starting when I was looking for a bachelor thesis project and my supervisor introduced me his idea to create a deduction app of propositional logic. He stated that there is an app called "The Logic App SD" which is capable of constructing propositional logic proofs but it does that via the Fitch-style calculus [5]. We did not think this is a really intuitive way of interacting with logic and actually understanding the proofs you make. That is where our ambition comes from to create a deduction app that works with a less static user interface and where deduction is represented by trees.

We immediately came up with analogy of Lego and we insisted on keeping this analogy. In the end this would make it easier to make design choices because we asked ourselves the question "what would Lego do?".

5.2 Learning objective-C

I already had a lot of experience with other programming languages, but objective-C was relatively new for me. Also I had never really used Xcode before. Luckily I had played with the iOS programming tools once so I knew what it meant and how to get resources to learn about the iOS environments.

I started with following online lectures at Standford University called "Developing Apps for iOS" with course code CS193p. The lectures were all recorded in 2011 and uploaded in the "iTunes U".

Also I signed up for the apple development program. It enables me to run and test the app on multiple devices and finaly distribute the app on the Apple app store.

5.3 Formalizing and programming

Especially in the beginning the weekly sessions of myself and my supervisor were about generalizing and formalizing natural deduction for propositional logic. Eventually we wrote a paper on Clickable Proofs [3] where we discussed how proofpieces look like and how they would be interacted to each other. Also we emphasized the role of unification through the ports and we explored the correctness of the system.

While formalizing the system I was programming the app in parallel. While programming I came up with questions we had to answer so we were formalizing on the go.

5.4 Debugging and closed beta testers

In the first stage of development where we had nothing interesting to show the only real testers were my supervisor and me. When we reached a representable status where the idea was clear and no major bugs would occur we decided that some people who were interested in the app would receive a copy of the app. We found approximately ten people who wanted to test the app. Those people received a copy of the app and a certificate to enable their device to run the app. Also I created a small website, found at `http://www.students.science.uu.nl/~3688844/ND-app/`[1], to notify the testers about new downloads, changes made and bugs that were still unsolved.

It was nice to have beta testers around. They always look in a different way to a piece of software than the developer because the developer is too close on the project to maintain an objective view and notice the flaws and bugs.

Thank you testers!

5.5 Last but not least

Finally I wrote the document you are reading right now. I look at this document to justify all the design choices and time spent.

Bibliography

[1] Development website.

[2] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Trans. Program. Lang. Syst.*, 4(2):258–282, April 1982.

[3] Vincent van Oostrom and Tim Selier. Clickable proofs. February 2013.

[4] Hendrik Jan Veenstra, Vincent van Oostrom, Albert Visser, and Jesse Mulder. Parvulae logicales: Propositielogica.

[5] Wikipedia. Fitch-style calculus.