RelAPS

A Relation Algebraic Proof System

User Manual

Author: Joel Glanfield (glanfield@cs.dal.ca)
Last Updated: August 2008
School: Brock University
Department: Computer Science
Supervisor: Michael Winter (mwinter@brocku.ca)

Contents

Figures

1. Introduction

1.1 Purpose

The purpose of the RelAPS (pronounced “ree-laps”) system is to provide an environment whereby a user may construct a derivation of provable theorems as if the derivation were done by hand. This provides the benefits of having a system ensure that an individual proof-step is executed properly, while it remains the responsibility of the user to complete the proof.

There are several motivations behind the design of RelAPS. Firstly, proof-construction is more intuitive since the user should have some feeling as if he/she were manually constructing a proof just as if it were being done on paper. Secondly, it provides a learning environment for those new to the realm of deductive reasoning; that is, the system provides an environment whereby a user may “practice” generating proofs.

It should be mentioned that it is not the aim of the system to provide automated deduction.

1.2 Brief Overview

The RelAPS system has several aspects that are worth mentioning. First of all, a user may define custom (nullary, unary and binary) operations. These operations may be combined with arbitrary axioms to produce new theories. The base theory of the system is the theory of allegories which consists of the following operations: the identity, converse, intersection and composition (see [1] for details).

The goal of my undergraduate research was to implement the basic functionality mentioned above. Once that goal was achieved, I focused on implementing a decision procedure for the equational theory of allegories. This algorithm was previously described in a PhD thesis by Claudio Gutierrez (PhD) [3]. It was this algorithm that provided motivation for the main focus of my MSc research.

My MSc research was geared towards providing an algorithm that would allow automated proof generation of provable equations in the theory of allegories [1,2]. The algorithm is an extension of the decision procedure mentioned above.

2. Getting Started With RelAPS

2.1 Installation

There really is no installation process for RelAPS. Since it was developed using Java, you simply need to be able to run executable .jar files in order to run the program.

If you don't have Java installed on your machine, you will need at least JRE 1.5 to run RelAPS. Make sure your CLASSPATH (on Windows) is updated as well, unless you can run executable jar files. If you need help setting the CLASSPATH, read more here.

Click here to download the appropriate files. Unzip the files to your preferred location.

Open the RelAPS directory, right click on the ‘RelAPS.jar’ file and select ‘Open With’ and then ‘Choose Program’. If you have Java installed correctly, you should see ‘Java Standard Edition’ in the list (or something like it). Select it and then press ‘OK’. RelAPS should then start.

If your system is already configured to run executable .jar files, then simply double-click the ‘RelAPS.jar’ file included in the download.

2.2 The Startup Screen

The Startup Screen is the first screen you will encounter upon starting RelAPS and is shown in Figure 1.

This screen allows the user to specify which theory should be used while using RelAPS (this can be changed later on). When you start the program for the first time, you will be presented with only one option - namely the (default) theory of allegories.

Under the label ‘Theories:’, you will presented with each theory that is in the system along with the associated operations. For example, the theory of Allegories has four operations: Identity, Converse, Intersection, and Composition.

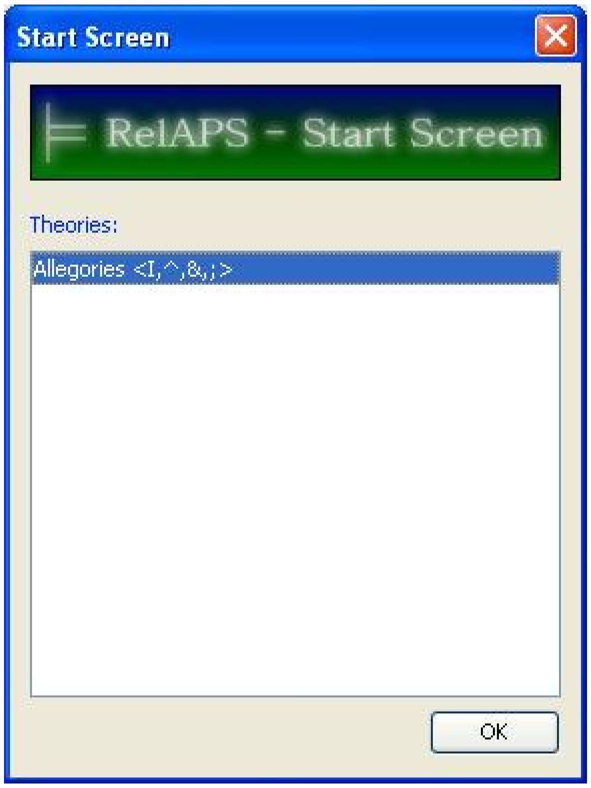


Figure 1: The Startup Screen

Once you learn how to create additional theories and add them to the system, you will then see them in the list (which currently only shows one theory) whenever you start the program.

To continue using the program, simply select the desired theory (choose ‘Allegories’ if it is the only one) and press the ‘OK’ button (or press the Return key, or double-click the theory name - both are shortcuts).

3. The User Interface

3.1 Overview

The main interface of the program is shown in Figure 2 on the next page. The interface if divided into five sub-windows, each of which will be summarized here.

1. Proof Explorer

This window allows the user to view the proofs that are currently being worked on. From this window the user may open/save proofs, add new proofs, remove current proofs, and add proven theorems to the system. If multiple proofs exists, the ‘current’ proof (ie. the proof currently being worked on) will be shown in **bold-type** in this window.

2. Assertions

This window contains the assertions that the user is currently proving. For example, if the user is proving some implication A=>B, then the ‘B’ part (i.e. the *assertion*) will be visible in this window.

3. Assumptions

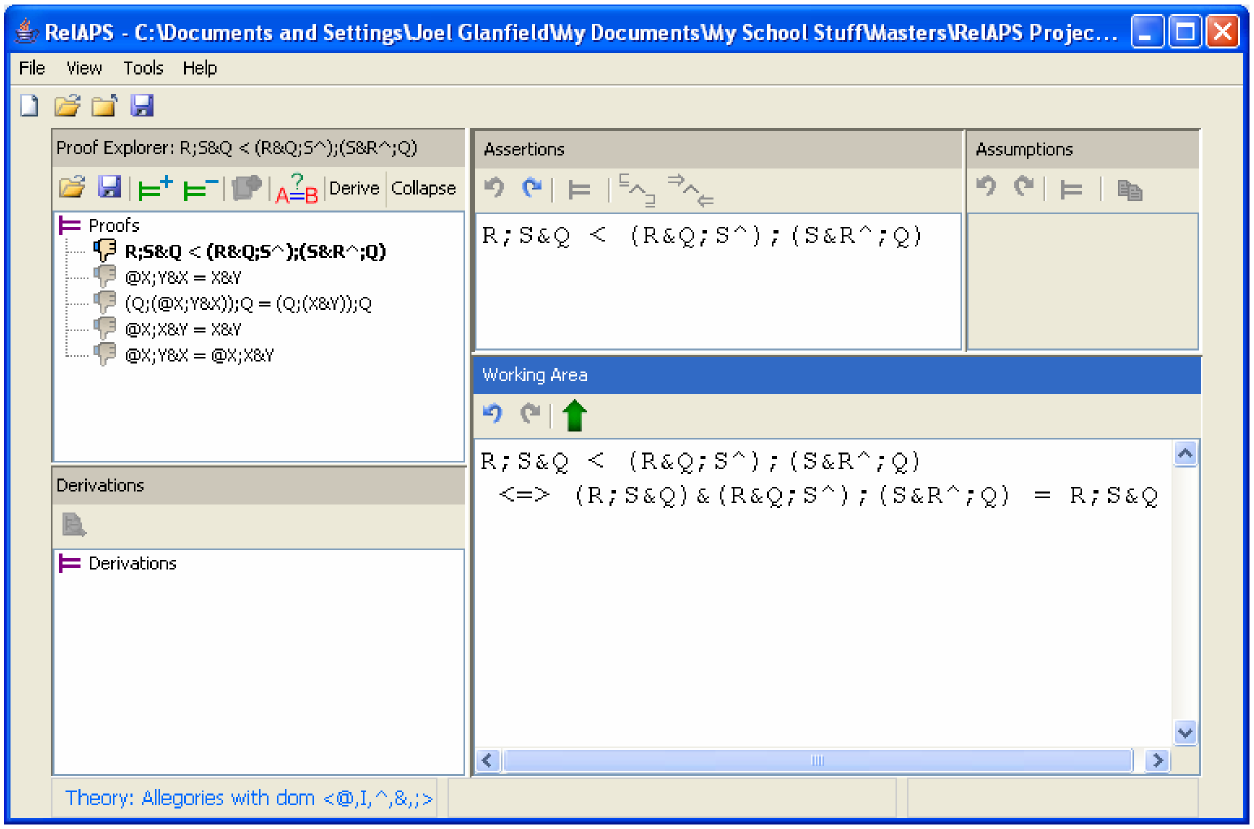
This window contains any assumptions that pertain to the current proof. For example, if the user is proving some implication A&B=>C, then the ‘A’ and ‘B’ (i.e. the *assumptions*) parts will be visible in this window.

4. Derivations

This window keeps track of any completed derivations associated with the current proof.

5. Working Area

This window is arguably the most important part of the interface, and is likely where the user will spend most of his/her effort. This window is where derivations are performed. The user will specify which terms/formulas are to be manipulated, and then within this window will apply certain rules to perform some derivation.


RelAPS - C:\Documents and Settings\Joel Glanfield\My Documents\My School Stuff\Masters\RelAPS Projec...
File
View
Tools
Help
Proof Explorer: R;S&Q < (R&Q;S^);(S&R^;Q)
Derive
Collapse
Assertions
Assumptions
Proofs
Working Area
Derivations
Derivations
Theory: Allegories with dom <@,I,^,&,;>

```
R;S&Q < (R&Q;S^);(S&R^;Q)
@X;Y&X = X&Y
(Q;(@X;Y&X));Q = (Q;(X&Y));Q
@X;X&Y = X&Y
@X;Y&X = @X;X&Y
```

```
R;S&Q < (R&Q;S^);(S&R^;Q)
```

```
R;S&Q < (R&Q;S^);(S&R^;Q)
 <=> (R;S&Q)&(R&Q;S^);(S&R^;Q) = R;S&Q
```

Figure 2: A View of the RelAPS System's Main Interface

3.2 Proof Explorer

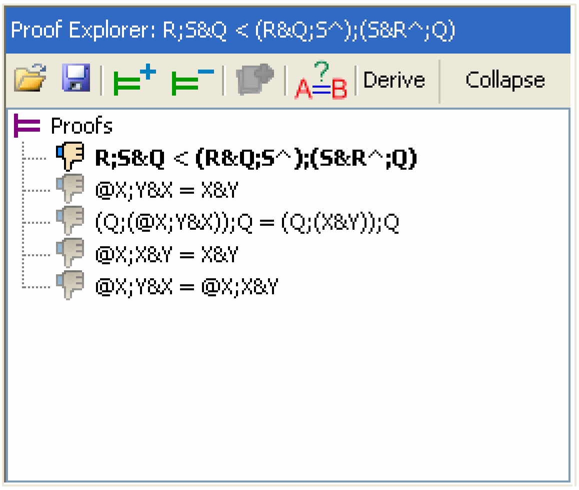
The Proof Explorer window allows the user to view the current proofs being worked on. The screenshot displayed in Figure 3 shows an example where there are several proofs being worked on.

The two main components of this window are: 1) the toolbar, and 2) the tree-view. Both of these will now be discussed.

1. The Toolbar

The toolbar contains eight buttons that have to do with setting up and storing proofs, along with some extended functionality recently implemented.

The first button is the 'Open Proof' button (with an open-folder icon) and allows the user to open a previously saved proof.


Proof Explorer: R;S&Q < (R&Q;S^);(S&R^;Q)

R;S&Q < (R&Q;S^);(S&R^;Q)
@X;Y&X = X&Y
(Q;(@X;Y&X));Q = (Q;(X&Y));Q
@X;X&Y = X&Y
@X;Y&X = @X;X&Y

Figure 3: The Proof Explorer

The second button is the 'Save Proof' button (with a disk icon) and allows the user to save the 'current' proof (recall that the 'current' proof is the one in **bold-type** in the tree-view). The user may save a proof at any state; it does not have to be complete. If opened, it will continue at the same state at which it was saved.

The third button is the 'Start Proof' button, and it allows the user to create a new formula that will be derived (the details of creating a formula will be discussed in Section 4). When a formula is added to the system, it will appear in the tree-view as shown in the screenshot above.

The fourth button is the 'Remove Proof' button, and it allows the user to remove any proof from the tree-view.

The fifth button is the 'Add Theorem' button, and it allows the user to store any derived theorem in the system. Any stored theorem will be associated with the current theory, as selected in the Startup window.

The sixth button executes the decision algorithm for the equational theory of allegories. The implementation of this algorithm was mentioned in Section 1.2. When executed, a message will appear notifying the user whether the formula is provable.

The seventh button (labeled 'Derive') attempts to provide a derivation of a provable theorem in the equational theory of allegories. The algorithm implemented here corresponds to Lemma 72 of Gutierrez' PhD Thesis (see Chapter 5 of [3]).

The eighth button simply collapses all children nodes on the tree-view so only parent nodes are visible.

2. The Tree-View

The tree-view simply lists all of the current proofs being worked on by the user. When proofs are added/removed to/from the system, the tree-view will reflect the change.

There are several icons that reflect the state any of the proofs:

The ‘thumb-down’ icon shows that a proof has not yet been completed, and hence may not be stored as a theorem in the system.

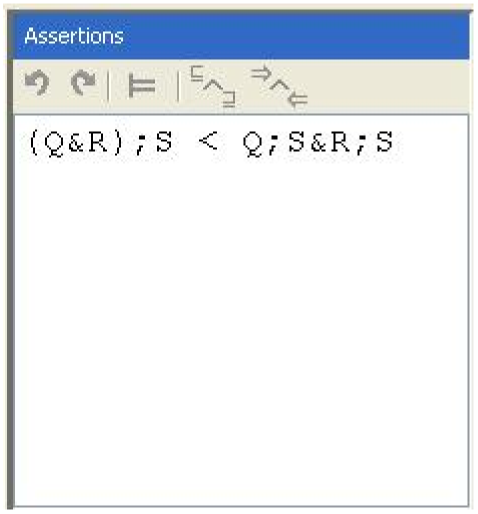
The ‘red-checkmark’ icon shows that a proof has been completed. These proofs may be added to the system (i.e. the ‘add-theorem’ button will be enabled).

The ‘blue-checkmark’ icon shows that a proof-obligation has been proven.

Proof obligations are nested below the main proof to which they belong.

3.3 Assertions Window

The Assertions Window displays the assertion a user wishes to derive. Figure 4 shows an example of such an assertion.



```
(Q&R);S < Q;S&R;S
```

Figure 4: Assertions Window

The Assertions window has a toolbar and a text-area where the actual assertion(s) are displayed. Both of these components will be explained:

1. The Toolbar

The toolbar contains five buttons:

The first button is the ‘Undo’ button. Whenever the user performs a derivation that modifies an assertion, this button allows the user to undo the effects of applying the derivation.

The second button is the ‘Redo’ button. This button allows the user to redo the application of a derivation after it has been undone.

The third button is the ‘Derive’ button. When the user selects (with the mouse) either the left-hand or right-hand side of an assertion, or even the complete assertion, the ‘Derive’ button will then become enabled. This allows the user to move the selection to the ‘Working Area’ in order to attempt a derivation. (See Section 3.5 for details on the Working Area).

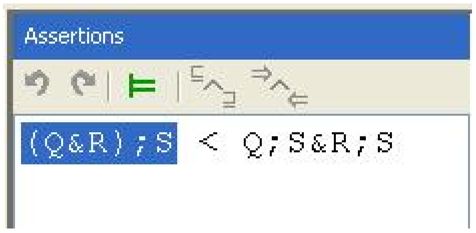
The fourth button allows the user to separate an equation into two separate inclusions. If the assertion is an equation, and the user selects the entire assertion, then this button becomes enabled. If the user presses the button, then two inclusions are created as proof obligations and must be derived before the proof is considered complete. **NOTE:** When the two inclusions are created, the user will immediately notice that both inclusions appear in the Assertions window. However, since you may only work with one assertion at a time, you must specify which inclusion you wish to work with be clicking the corresponding formula in the tree-view of the Proof Explorer window (see Section 3.2 on the Proof Explorer for more details).

The fifth button allows the user the separate an equivalence into two separate implications. The exact same procedure and rules apply here as were discussed in the preceding paragraph (except that we are now talking about two implications and not inclusions).

2. The Text-Area

The text-area simply displays the current state of the assertion being worked with. As was already noted, you may only work with one assertion at a time. This is specified by clicking the appropriate assertion in the tree-view of the Proof Explorer window.

It is in this component that you may specify which part of the assertion you wish to modify. You may do this by either selecting the entire assertion, or by selecting the complete left-hand or right-hand side of the formula. The screenshot in Figure 5 shows how the selection of the left-hand side of the formula activated the ‘Derive’ button.

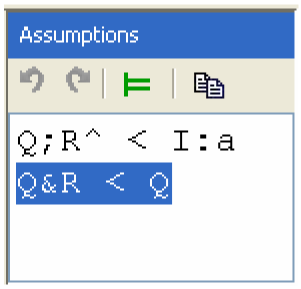


```
(Q&R);S < Q;S&R;S
```

Figure 5: Selecting Part of an Assertion

3.4 Assumptions Window

The Assumptions window displays the assumptions that are associated with the formula a user wishes to derive. Figure 6 shows an example of such a set of assumptions.



```
Q;R^ < I:a
Q&R < Q
```

Figure 6: Assumptions Window

The Assumptions window has a toolbar and a text-area where the actual assumption(s) are displayed. Both of these components are used the same way as the related components in the Assertion window (see the previous section tutorial on the Assertions Window for more details), and the differences will be covered here.

1. The Toolbar

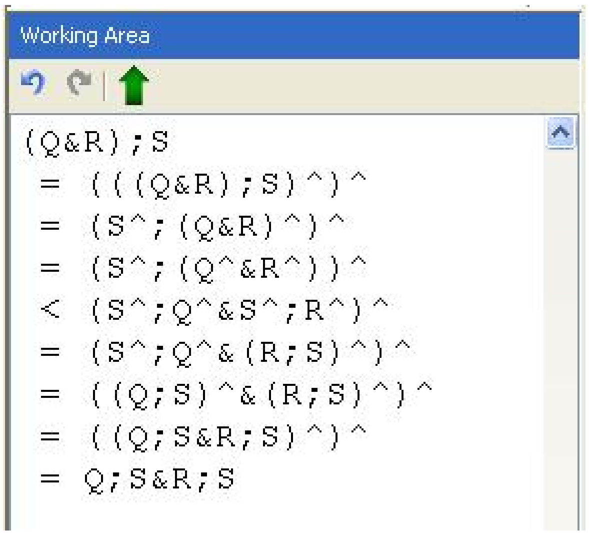
The buttons on the toolbar work in the same manner as the related buttons in the Assertions window, with the exception of the last button (which is called the 'duplicate' button). The 'duplicate' button allows the user to duplicate any selected assumption (the entire assumption much be selected for this button to become enabled).

2. The Text-Area

The text-area allows for the selection of those parts of any assumption that you may wish to modify. The difference in this window (as opposed to the text-area in the Assertions Window) is that multiple assumptions are always in view (if multiple assumptions exist!), and any of them may be selected at any time.

3.5 The Working Area

The Working Area window is where derivations are performed. Figure 7 shows such a derivation.



```
(Q&R);S
 =  (((Q&R);S)^)^
 =  (S^;(Q&R)^)^
 =  (S^;(Q^&R^))^
 <  (S^;Q^&S^;R^)^
 =  (S^;Q^&(R;S)^)^
 =  ((Q;S)^&(R;S)^)^
 =  ((Q;S&R;S)^)^
 =  Q;S&R;S
```

Figure 7: An Example Derivation

The Working Area consists of a toolbar and a text-area, both of which will be explained.

1. The Toolbar

The toolbar consists of three buttons:

The first two buttons allow the user to undo/redo any step of the derivation process.

The last button is the ‘Apply’ button which applies the derivation to the appropriate assertion (or assumption).

2. The Text-Area

The text-area is where the details of the derivation are displayed. It also allows the user to select different terms that will be modified.

When a term is selected, a menu immediately pops up which displays the axioms, assumptions, and theorems that may be applied to the current selection. The following is a screenshot of this process:

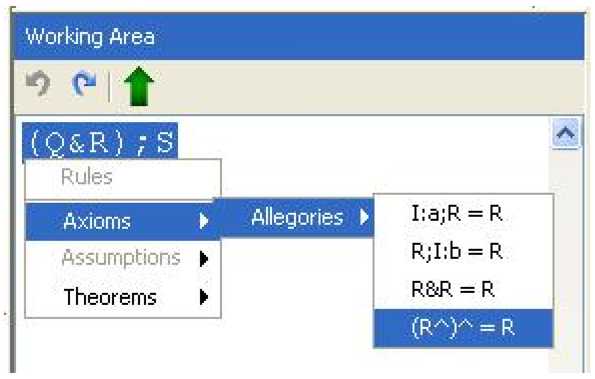
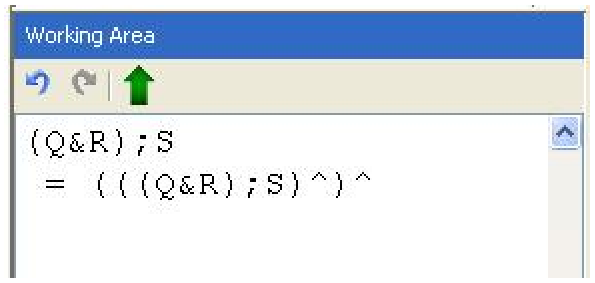


Figure 8: Selecting a Term

In this example, there are four possible axioms that can be applied to the selected term. The user simply selects the desired axiom, and then the next line of the derivation appears:



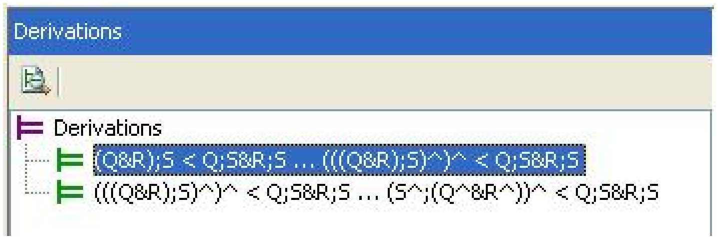
```
(Q&R);S
 = (((Q&R);S)^)^
```

Figure 9: Applying a Rule

3.6 Derivations Window

The Derivations window tracks all of the completed derivations. The toolbar only has one button, which allows you to view the details of any derivation (that is selected in the tree-view). When you view the details of a derivation, _you must click the yellow box that appears in order to hide the details_.

The screenshot displayed in Figure 10 shows the Derivations Window with a couple of completed derivations:


Derivations

Derivations
(Q&R);S < Q;S&R;S ... (((Q&R);S)^)^ < Q;S&R;S
(((Q&R);S)^)^ < Q;S&R;S ... (S^;(Q^&R^))^ < Q;S&R;S

Figure 10: The Derivations Window

4. Creating Formulas

4.1 Overview

Before any derivation can be performed, a syntactically-valid formula must first be provided by the user. This involves creating a formula using ASCII characters representing different quantifiers, variables, and operations. The next few sections will deal with explaining the grammar of valid formulas, as well as the syntax of different types of operations.

Before you can create a new formula, you must bring up the dialog box that will allow you to do so. This is done by pressing the ‘Start Proof’ button contained in the toolbar of the Proof Explorer window (see Section 3.2 for details about the Proof Explorer).

Figure 11 (on the following page) shows the Formula Editor that allows you to input text representing arbitrary formulas.

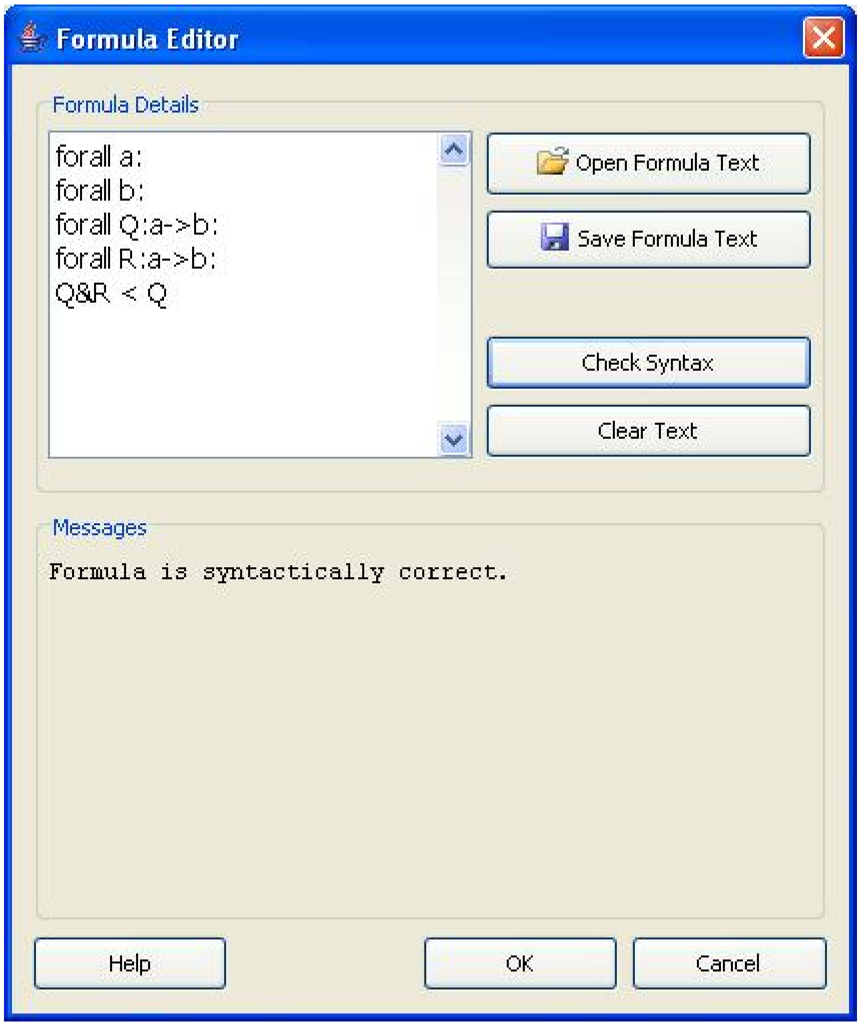
Once you've entered some text (the next section deals with syntax), simply press the ‘Check Syntax’ button to find out whether the formula you've entered is syntactically correct.

4.2 Formulas: Grammar

This section describes the grammar used to create syntactically-valid formulas. One thing you will notice is that there is a restriction as to how valid formulas can be created (e.g. only all-quantifiers can be used at this time). The grammar will be expanded as the system progresses.

The following rule describes a syntactically-valid formula with the restrictions that currently exist in the system:

Formula := OBJVARS RELVARS EXPRESSION



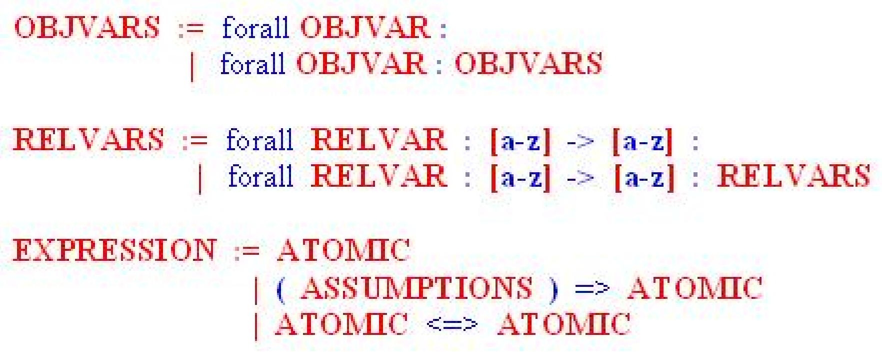
```
forall a:
forall b:
forall Q:a->b:
forall R:a->b:
Q&R < Q
```

```
Formula is syntactically correct.
```

Figure 11: Formula Editor

The above rule states that a valid formula consists of a set of object variables followed by a set of relation variables, which are then followed by some type of expression. The rules for the non-terminals in the above rule are described in Figure 12.

The English-word counterparts of the above (less-obvious) non-terminals are as follows: **OBJVARS** stands for ‘Object Variables’, **RELVARS** for ‘Relation Variables’, and **ATOMIC** for ‘Atomic Formula’.

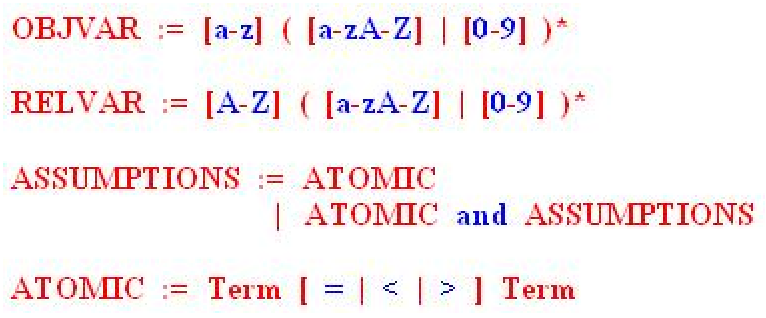


```
OBJVARS := forall OBJVAR :
         |  forall OBJVAR : OBJVARS

RELVARS := forall RELVAR : [a-z] -> [a-z] :
         |  forall RELVAR : [a-z] -> [a-z] : RELVARS

EXPRESSION := ATOMIC
            | ( ASSUMPTIONS ) => ATOMIC
            | ATOMIC <=> ATOMIC
```

Figure 12: Various Rules for Expressing Valid Formulas

Note that whenever a token is colored blue, it is a terminal symbol (see Figure 12). For example, if you are declaring an object variable you must precede it with the keyword 'forall' and follow it with a colon (':') symbol.

The rules in Figure 12 introduce more non-terminals, the rules for which are found in Figure 13.



```
OBJVAR := [a-z] ( [a-zA-Z] | [0-9] )*

RELVAR := [A-Z] ( [a-zA-Z] | [0-9] )*

ASSUMPTIONS := ATOMIC
             | ATOMIC and ASSUMPTIONS

ATOMIC := Term [ = | < | > ] Term
```

Figure 13: More Rules for Expressing Valid Formulas

From the rules described in Figure 13 you will notice that the only difference between an object variable and a relation variable is that an object variable must start with a lowercase letter, whereas a relation variable must start with a capital letter.

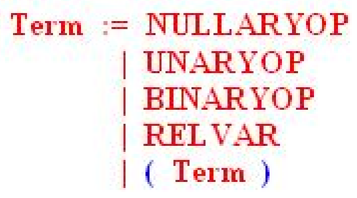
The terminal symbols that make up an atomic formula are equality ('='), less-than/inclusion ('<'), and greater-than ('>').

With respect to the rule for **ATOMIC**, you will notice that we introduce a new component to the formula-grammar, i.e. that of a Term. The next section will deal with the grammar for valid terms.

4.3 Terms: Grammar

This section describes the grammar used to create syntactically-valid terms.

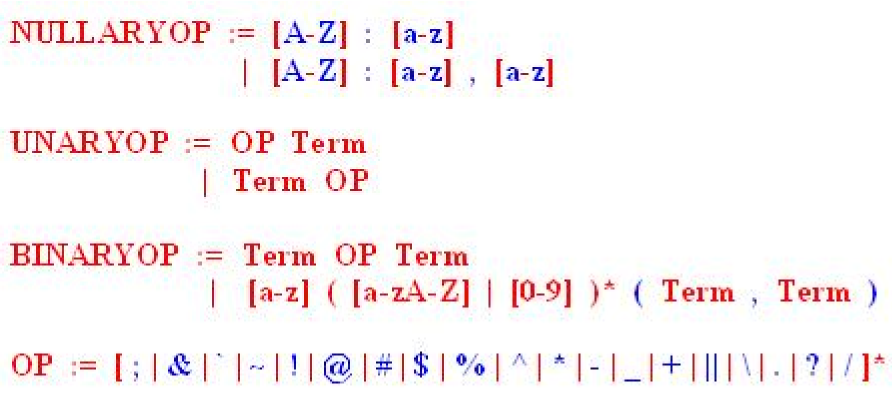
The rule in Figure 14 describes a syntactically-valid term in the current system:



```
Term  := NULLARYOP
       | UNARYOP
       | BINARYOP
       | RELVAR
       | ( Term )
```

Figure 14: Term Rule

From the above rule, it is obvious that a term is made up of operations on relations, or of a single relation itself. The rules for these operations are shown in Figure 15.



```
NULLARYOP := [A-Z] : [a-z]
           | [A-Z] : [a-z] , [a-z]

UNARYOP := OP Term
         | Term OP

BINARYOP := Term OP Term
          | [a-z] ( [a-zA-Z] | [0-9] )* ( Term , Term )

OP := [ ; | & | ` | ~ | ! | @ | # | $ | % | ^ | * | - | _ | + | || | \ | . | ? | / ]*
```

Figure 15: Operation Rules

The rule for **NULLARYOP** (i.e. nullary operations) shows that you can create such an operation one of two ways. The first is a shorthand method where the source and target variables are assumed to be the same (e.g. 'I:a' is interpreted as 'I:a,a'). The second method should be used when the source and target variables are not the same.

The rule for **UNARYOP** (i.e. unary operations) shows that you can either create prefix or postfix unary operations.

The rule for **BINARYOP** (i.e. binary operations) shows that you can either create an infix binary operation, or a prefix binary operation. The main difference between the two is that an infix binary operation is made up of a combination of special symbols (see the rule for **OP**), whereas a prefix binary operation must start with a lowercase letter.

The rule for **OP** shows that an operation symbol is made up of a combination of the symbols listed (e.g. R;R^ is a valid binary operation). If two operations are juxtaposed (e.g. the characters '^' and ';' in the term **R^;R**) then it is essential that they are separated by a space (e.g. **R^ ;R**).

4.4 Examples of Syntactically-Valid Formulas

This section will simply list some valid formulas to give you an idea of how the previous formula and term grammars should be used:

- forall a: forall b: forall Q:a->b: Q < Q;Q^ ;Q
- forall a: forall R:a->a: R; -R = R
- forall a: forall b: forall Q:a->b: forall R:a->b: Q <=> Q&(-R) = O:a,b
- forall a: forall b: forall c: forall Q:a->b: forall R:b->c: forall S:a->c: Q;R&S < (Q&S;R^);R
- forall a: forall b: forall R:a->b: I:a;R = R
- etc...

Many of the operations contained in the above formulas are user-defined. The system starts with some default operations, but any additional must be defined by the user. This process will be explained in Section 6.

5. Creating and Editing Theories

5.1 Overview

In RelAPS, a ‘Theory’ is composed of the following:

- a name
- a set of operations
- a set of dependencies
- a set of axioms
- a set of theorems

The first property is self-explanatory: each theory is given a **name**.

When a theory is first created, the user specifies which **operations** are to be associated with the theory.

A theory may be **dependent** upon other theories in terms of hierarchy. For example, you may wish to add a theory that combines the default theory, ‘Allegories’, with the universal relation. The new theory would then only have one new operation, possibly ‘top’, and also would have the theory 'Allegories' as a **dependency**.

A theory is also given a set of **axioms**. Axioms are simply formulas, the construction of which was described in the previous section.

Whenever a valid **theorem** is derived, it is added to the active theory.

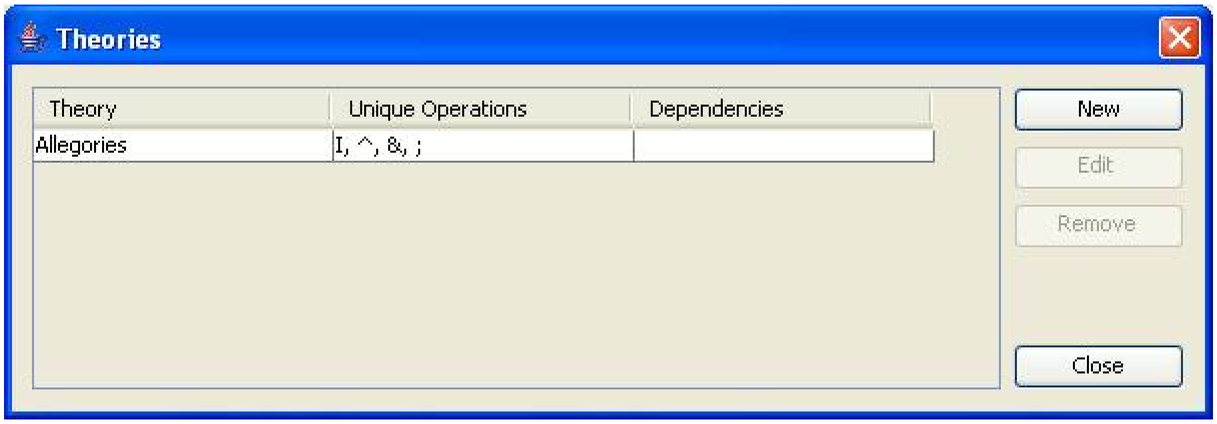
The procedure for accomplishing the above tasks will be explained in the following sections.

5.2 Creating a New Theory

Before any formula can be created and proven, a theory must be created to give context to the formula. Since the default theory ‘Allegories’ is loaded each time the system is started, there is always at least one theory to work with. However, any formula that is created under this theory may only use the operations associated with it.

If the user wishes to create formulas that require additional operations (to those contained in the theory of allegories), new theories must first be created.

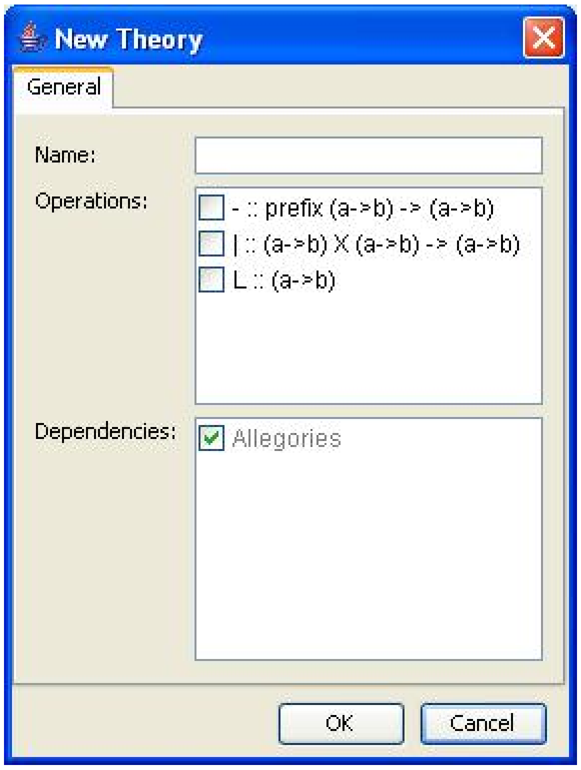
To create a new theory, start by opening the **Tools** menu and drill down to **Theories -> Edit Theories**. You will then be presented with the window shown in Figure 16



Theory	Unique Operations	Dependencies
Allegories	I, ^, &, ;	

Figure 16: Theories Dialog

You will notice that the default theory ‘Allegories’ already exists in the list of available theories. To create a new theory, press the **New** button, and you will then be presented with the window shown in Figure 17.



Name:

Operations:
- [ ] - :: prefix (a->b) -> (a->b)
- [ ] | :: (a->b) X (a->b) -> (a->b)
- [ ] L :: (a->b)

Dependencies:
- [x] Allegories

Figure 17: Theory Details

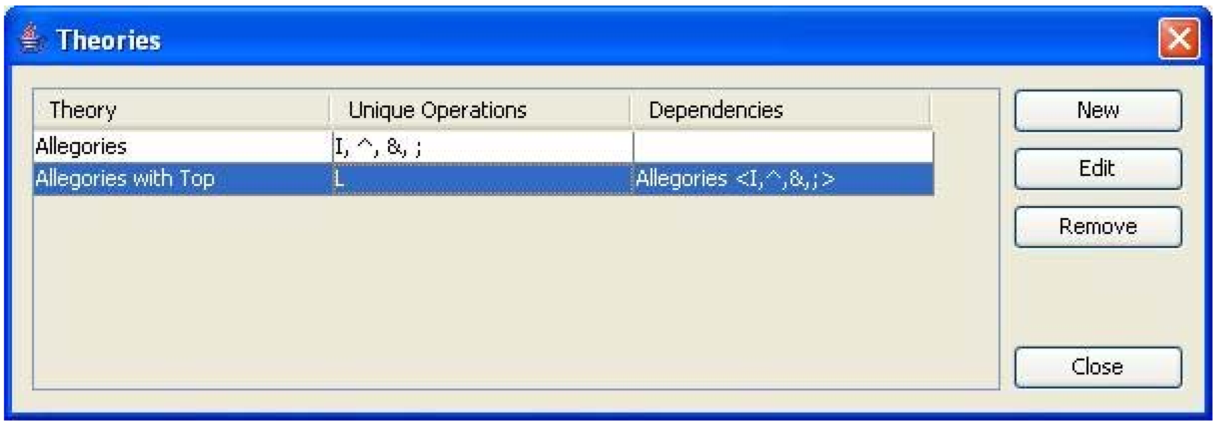
Here you will notice that an available list of operations and potential dependencies is presented to the user. Each new theory must extend from the default theory, which is why 'Allegories' is checked automatically. As for the list of operations, if the user has not created any new operations, then this list will be blank. Creating new operations will be explained in Section 6.

For now, you may create a new theory by simply specifying a name and pressing the **OK** button. You will then be returned to the previous window where your new theory will now be listed. If you want to add axioms to your new theory, then you will need to read the next section.

5.3 Editing a Theory

Editing a theory will allow you to add and/or remove axioms, and view theorems associated with a given theory.

To edit a theory, follow the same steps as you would to create a theory, except that once you see the window with the theories listed, simply select the theory you wish to edit and press the 'Edit' button. Figure 18 displays the Theories Dialog with one theory selected to be edited.

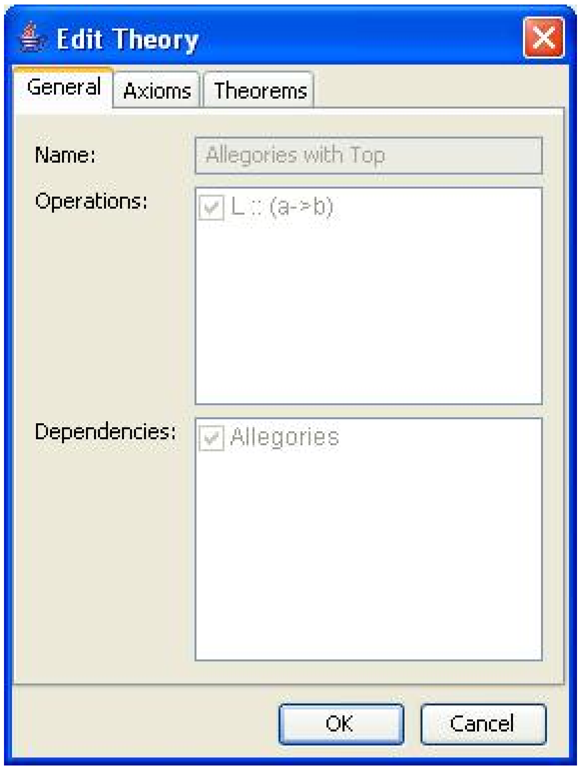


Theory	Unique Operations	Dependencies
Allegories	I, ^, &, ;	
Allegories with Top	L	Allegories <I,^,&,;>

Figure 18: Editing a Theory

Once the edit button is pressed, the ‘Edit Theory’ window appears displaying the details of the theory. This time you will notice that there are two additional tabs, one for adding/removing axioms and the other for viewing theorems. See Figure 19 for a visual.

To add an axiom to the theory simply select the ‘Axioms’ tab and then press the ‘Add’ button. This will display the Formula Editor, and allow you to enter a valid formula representing an axiom (details on creating valid formulas were discussed in Section 4). Entering a valid formula and pressing ‘OK’ will then add the formula as an axiom to the current theory. Figure 20 displays the axioms associated with the theory ‘Allegories’.

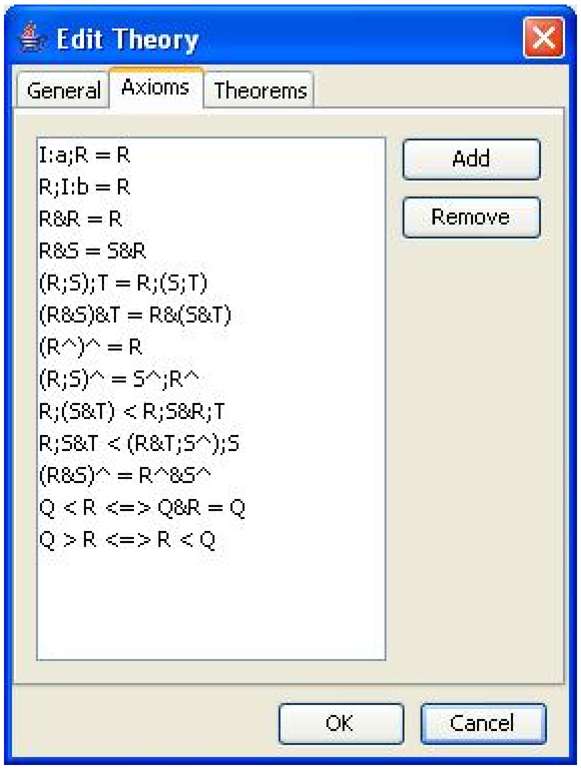


Name: Allegories with Top

Operations: ✓ L :: (a->b)

Dependencies: ✓ Allegories

Figure 19: Edit Theory Dialog



```
I:a;R = R
R;I:b = R
R&R = R
R&S = S&R
(R;S);T = R;(S;T)
(R&S)&T = R&(S&T)
(R^)^ = R
(R;S)^ = S^;R^
R;(S&T) < R;S&R;T
R;S&T < (R&T;S^);S
(R&S)^ = R^&S^
Q < R <=> Q&R = Q
Q > R <=> R < Q
```

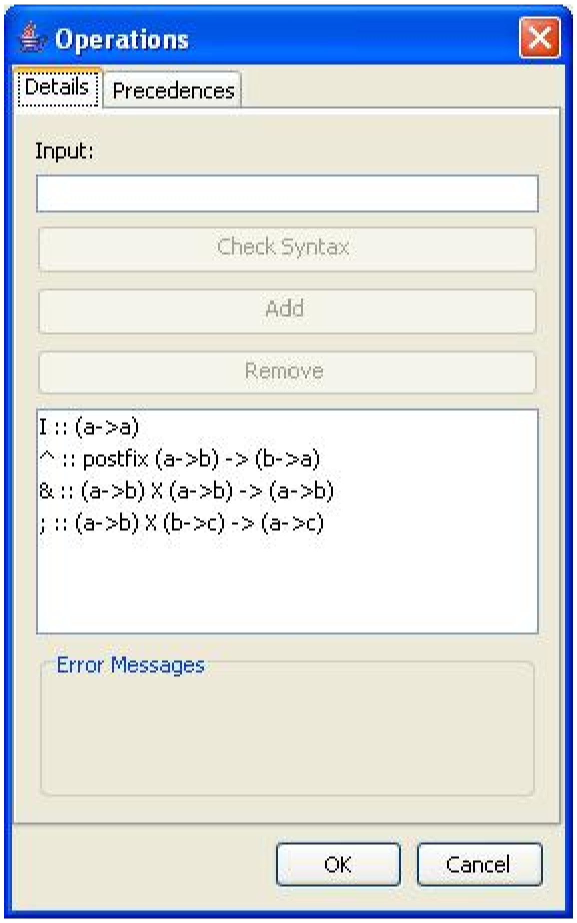
Figure 20: Axioms for the Theory of Allegories

6. Defining Operations

6.1 Overview

Before creating a new theory, you will need to define some new operations that can be used in the theory. This section will focus on how to do so.

To bring up the ‘Operations Editor’, click the ‘Tools’ menu and drill down to **Operations -> Operation Editor**. You will then be presented with the window shown in Figure 21.



```
I :: (a->a)
^ :: postfix (a->b) -> (b->a)
& :: (a->b) X (a->b) -> (a->b)
; :: (a->b) X (b->c) -> (a->c)
```

Figure 21: Operations Editor

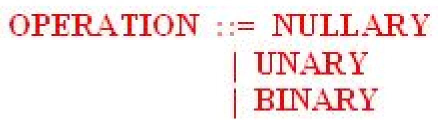
You will notice that the operations that currently exist in the system are listed in this window. They are listed as valid operation-strings that were parsed to create the actual operation itself. To create a new operation, you will need to enter a valid string that can be parsed as a nullary, unary, or binary operation.

Once you have entered a string, simply press the ‘Check Syntax’ button to determine whether it is syntactically correct. If it is, you will then be able to press the ‘Add’ button and add it to the system for use in a new theory.

The next section will focus on the grammar for syntactically-valid operations.

6.2 Operations: Grammar

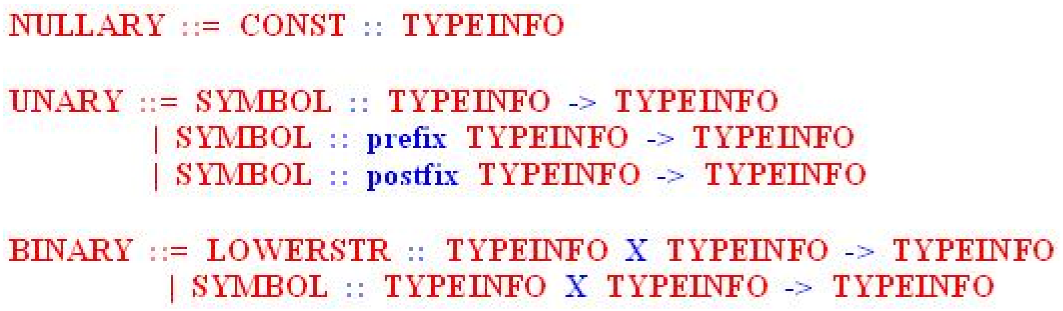
The main rule that gives an overall picture of what operations are available is shown in Figure 22.



```
OPERATION ::= NULLARY
            | UNARY
            | BINARY
```

Figure 22: Rule for Constructing Operations

As is evident from the above rule, the system current allows **Nullary**, **Unary** and **Binary** operations. The rules described in Figure 22 demonstrate the syntactic structure of these operations:



```
NULLARY ::= CONST :: TYPEINFO

UNARY ::= SYMBOL :: TYPEINFO -> TYPEINFO
        | SYMBOL :: prefix TYPEINFO -> TYPEINFO
        | SYMBOL :: postfix TYPEINFO -> TYPEINFO

BINARY ::= LOWERSTR :: TYPEINFO X TYPEINFO -> TYPEINFO
         | SYMBOL :: TYPEINFO X TYPEINFO -> TYPEINFO
```

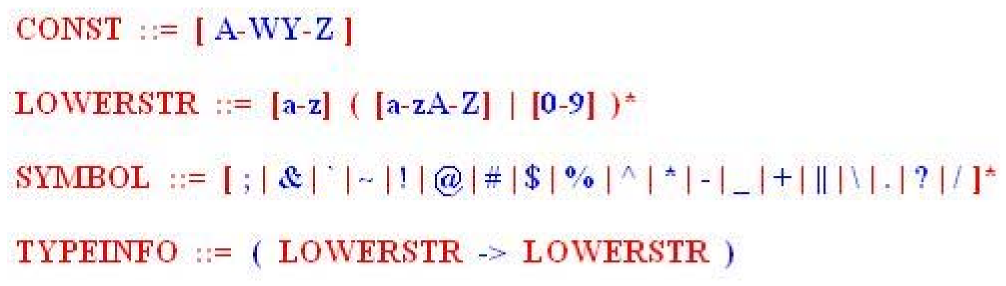
Figure 23: More Rules

Nullary operations consist of a constant (excluding the letter 'X') followed by some type information (the rule for both of these non-terminals is given in Figure 24).

Unary operations consist of a symbol, followed by an optional 'prefix'/'postfix' specification (if left out, the default is 'prefix') and then some typing information.

Binary operations consist of either a string or symbol followed by some type information. If the operation symbol is a 'LOWERSTR', then the binary operation will be prefix and will have a function-like appearance. If the operation is represented by a symbol, then it will be interpreted to be an infix binary operation.

The rules for the remaining non-terminals are given in Figure 24.



```
CONST ::= [ A-WY-Z ]

LOWERSTR ::= [a-z] ( [a-zA-Z] | [0-9] )*

SYMBOL ::= [ ; | & | ` | ~ | ! | @ | # | $ | % | ^ | * | - | _ | + | || | \ | . | ? | / ]*

TYPEINFO ::= ( LOWERSTR -> LOWERSTR )
```

Figure 24: Even More Rules

The rule for **CONST** excludes the letter 'X' since it is used in the rule for binary operations.

The rules for **LOWERSTR** and **SYMBOL** are exactly like those explained in the grammar for formulas.

The rule for **TYPEINFO** describes how you can specify the typing for an operation. Each variable in the typing information is simply a sequence of letters and digits, but must start with a lowercase letter.

Examples of syntactically-valid operations will be given in the next section.

6.3 Examples of Valid Operations

This section contains examples of syntactically-valid operations.

Nullary Operations

- L :: (a->b)
- O :: (a->b)
- I :: (a->a)

Unary Operations

- - :: (a->b) -> (a->b)
- - :: prefix (a->b) -> (a->b)

- ^ :: postfix (a->b) -> (b->a)

Binary Operations

- & :: (a->b) X (a->b) -> (a->b)
- | :: (a->b) X (a->b) -> (a->b)
- cmp :: (a->b) X (b->c) -> (a->c)
- ; :: (a->b) X (b->c) -> (a->c)
- \ :: (a->b) X (a->c) -> (b->c)

7. References

1. Peter J. Freyd and Andre Scedrov. *Categories, Allegories*. North-Holland, 1990.

2. Joel Glanfield. *Towards Automated Derivation in the Theory of Allegories*. MSc thesis, Brock University, 2008.

3. Claudio Gutierrez. *The Arithmetic and Geometry of Allegories – normal forms and complexity of a fragment of the theory of relations*. PhD thesis, Wesleyan University, 1999.