

Quickstart Guide to Expander2/Expander3

Richard Stewing¹ and Felix Laarmann²

¹*Department of Computer Science, University of Dortmund , richard.stewing@udo.edu*

²*Department of Computer Science, University of Dortmund , felix.laarmann@udo.edu*

March 30, 2019

Contents

1	What are Expander2 and Expander3?	2
1.1	Installation	2
2	Starting Screen	2
3	The first Term	3
4	Simple Functions	3
4.1	Common Term Functions	3
4.2	Node	4
4.3	Graph	4
5	Pointers	6
6	Specifications	6
6.1	Design Graphs	7
6.2	Functions	7
6.3	Predicates	7
6.4	Induction	8
7	Check Proof Mode	10
8	Painter	10
8.1	Combinations	10
8.2	Saving a picture	11
8.3	Save Modification Sequence	12
8.4	Widgets	12
9	Commands	13

1 What are Expander2 and Expander3?

Expander2 and Expander3 are term rewriter and theorem prover with the ability to transform and analyze graphs, solve constraints as well as other procedures to build up proofs. Furthermore customized functions can be provided by the user to display the resulting terms in a pleasing way. Expander2 has been developed in O'Haskell an extension to the Haskell ¹ while Expander3 is a Haskell native version. Both should be equal in functionality even though Expander3 can be expected to have better performance. This guide should work on both of them. A complete Manuel can be found here.

1.1 Installation

1.1.1 Expander2

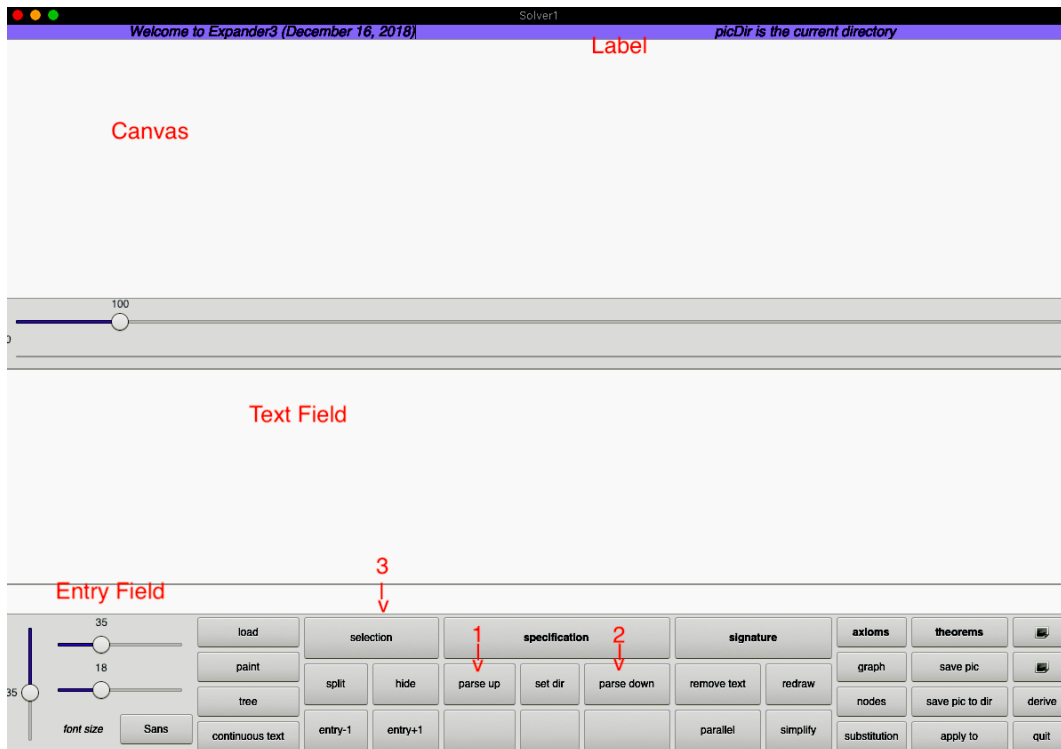
Expander2 requires the *rohugs* O'Haskell compiler to be itself compiled with support for tcl and tk. The source is included in the archive provided here. In order to compile rohugs correctly it is advised to edit the appropriate lines in Makefile and prelude.h for your platform. If your platform is not listed in either of them and you happend to be able to compile rohugs it would be greatly appreciated if you would inform Peter Padawitz or Jos Kusiek.

1.1.2 Expander3

Expander3's source can be downloaded from here. Follow that page for instructions how to install Expander3.

2 Starting Screen

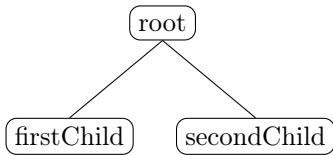
There are a number of very important interface elements which are furthmore referred to by name. There is the **Entry Field** for loading files, the **Text Field** for direct text input, the **Canvas** where the terms are displayed graphically, and the **Label** where response messages are presented.



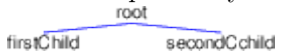
¹<https://www.haskell.org/>

3 The first Term

To enter a single term into Expander, the term can be entered in the **Text Field**. Terms are represented as trees in Expander and have to be translated into a two dimensional text form. In that representation a leaf is just the content of the leaf and an inner node is the content of the node followed by a ordered list of child nodes in parentheses. So the Tree



can be translated to $root(firstChild, secondChild)$ and then be entered into Expander. So enter $root(firstChild, secondChild)$ into the **Text Field**. The term is parsed by Expander upon pressing the **parse up** button (See 2 #1). This gives you the following result.



You know created your first term using Expander.

4 Simple Functions

In this chapter a selection of simple functions to modify terms are introduced. All modifications that are made to a term can be displayed by

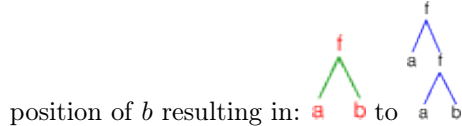
1. **term > show proof term**

This will also hold true in the future, when the focus shifts to more proof oriented work with Expander.

4.1 Common Term Functions

Create the Term $f(a, b)$ in Expander, refer to 3 if you are unsure how to do that. ²

1. Copying by Dragging: Now by dragging the f to the b you are able to copy the tree with the root at f to the



2. Copying by Command: Copying can also be accomplished by using command from the **selection** menu (See 2 #2).

- (a) Click on the tree to copy, it should be highlighted in red
- (b) **selection > copy (c)**
- (c) Click on the node that should be replaced
- (d) **selection > insert/replace by text (i)**

This results in same thing as before.

3. Reverse Tree: To reverse the order of the childtrees, select the subtree you want to work on.

- (a) **selection > reverse (v)**



4. Remove: To remove a seleted subtree:

²Rules for top level interaction can be found at [2], Chapter 6

- (a) Select subtree
- (b) **selection > remove (r)**

For example $f1(f2(a, b), f3(a, b))$ becomes $f1(f2(a, b))$ when you select the subtree $f3(a, b)$ and choose the **remove (r)** button.

5. Remove Node: To remove a single node and move there children one level up the tree:

- (a) Select node to remove
- (b) **selection > remove node (o)**

Used on our previous example with $f1(f2(a, b), f3(a, b))$ with $f3(a, b)$ selected results in $f1(f2(a, b), a, b)$.

6. Replace by Tree of SolverX: Assuming you, have a tree is loaded into the **Solver1** one window, a subtree can be replaced by the tree loaded into the **Solver2**.

- (a) **selection > replace by tree of Solver2**

For example, assuming $f(a, b, c)$ is loaded into **Solver1** and you just did some term rewriting for a in **Solver2** to $g(x)$, you might want transfer the result from **Solver2** to **Solver1**. You can select the subtree a in **Solver1** and than choose **replace by tree of Solver2** resulting in $f(g(x), b, c)$. The reverse way is possible by using **replace by tree of Solver1** in the **Solver1** window.

7. Saving modified Terms: Once a term is modified, you might want to save the term for later use. In order to save it, the term must be translated back into the linear representation aka into text form.

- (a) **parse down** (See 2 #3)

This outputs the text representation of the term currently displayed in the **Canvas** down in the **Text Field**. From there it be copied into a file for later use.

8. Loading Term from File: A term can be loaded from a file to the text field by using built in commands.

- (a) Enter a filename in the **Entry Field**
- (b) **load or term > load from text file**

The filename is relative to the **ExpanderLib** which usually is set to be $\sim/ExpanderLib$ but it might be different.

- (a) Notes about **ExpanderLib**: **ExpanderLib** is the directory from which your own specifications are going to be loaded where Expander is saving pictures and other files it's trying to save to the hard drive. The location is configured in *System.hs* for Expander2 and in *System/Expander.hs* for Expander3.

4.2 Node

Many informations about a tree can be displayed by the expander.

1. Depth: *Expander* can display the depth of nodes in a tree:

- (a) **nodes > level numbers**

2. Prefix Order: When the prefix order of the nodes is needed:

- (a) **nodes > preorder numbers**

3. Heap Order: The same can be done for the heap order:

- (a) **nodes > heap numbers**

4.3 Graph

Graphs are a common structure in proofs and *Expander* offers ways to interact with them. The most common ways are display and modification of graphs, which are presented next.

4.3.1 Show

There is more than one way to look at the graph. In the following section a few ways are exemplified.³ Add

```
states == [1..4] &
1 -> 2 &
2 -> 3 &
3 -> 4 &
4 -> 2 &
3 -> 1
```

to the axioms by clicking the **add** button in the **axiom** menu on the right side, they can be added either **from file** or **from text field**. This code tells you, that there are the states 1 through 4 and the transitions $1 \rightarrow 2$, $2 \rightarrow 3$, etc. For a more detailed description of **Specifications** refer to Chapter 6.

- Graph of transitions: If you added these axioms, you can see the resulting graph by the following steps:

- specification > build Kripke model**
- graph > of transitions > here**

This should result in the following:



- Graph of Kripke model: Even if you created a more complex Kripke Model, it can be displayed:

- specification > build Kripke model**
- graph > of Kripke model > here**

In this case, the resulting graph stays the same.

- Boolean matrix: The transitions can also be displayed in form of a boolean matrix.

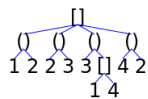
- specification > build Kripke model**
- graph > show Boolean matrix > of transitions**

	1	2	3	4
1				
2				
3				
4				

Resulting in:

- Transitionrelation: To see the transitionrelation, a set of pairs of states and sets of state ($State \times \mathcal{P}(State)$), you can also use the graph menu.

- specification > build Kripke model**
- graph > show binary relation > of transitions**



Resulting in a tree like:

- Iterative Equations: The iterative equations for a graph can be displayed by the following steps

- specification > build Kripke model**
- graph > build iterative equations**

Iterative equations are a way to describe the transitions of a graph.

³A more involved example can be found at [1], Model checking as simplification, trans0



4.3.2 Modifications

Graphs can be modified in many different ways. To get a first impression how *Expander* handles modifications consider the following examples.

1. Collapse of Nodes (1):



(a) **graph > collapse ->**

Equal nodes can be collapsed together. The "->" signals that only the right most node remains and all nodes that are equal to that node are replaced with a pointer to that node. These pointers are arrows to the node.

For example $f(a, b, a)$ turns into $f(pos\ 2, b, a)$. In pictures  turns into . The pointer is marked by *pos* followed by descriptive coordinates for the position. Summarized positions are counted from left to right and top to bottom. See 5 for a more detailed description.

2. Collapse for Nodes (2):

(a) **graph > collapse <-**

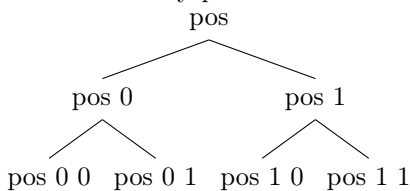
In comparison **collapse ->**, the left most node remains. So $f(a, b, a)$ turns into $f(a, b, pos\ 0)$ or  to .

3. Remove Cycles: The cycles in a graph can be removed from it. Note that this step might not be semantically correct. For such operations *expander* informs you about that in the **Label** field with the Message "..CAUTION: This step may be semantically incorrect!..". If you have a graph loaded in the canvas, cycles can be removed by:

(a) **graph > split cycles**

5 Pointers

Pointers are the way nodes can be referred to by other nodes in the tree. They are identified by the position in the tree. Starting with "pos", for each child an index is added from left to right. For a node identified with p the children are identified by $p\ i$ for each child from left to right, starting with 0. Consider the following tree as an example.



Pointers are displayed differently to normal tree edges. Are they pointing to a node on the same level, they are displayed in a purple color, pointers pointing to a node on a higher level are drawn in yellow. If they are pointing to a node on a lower level, they are colored red.

If you are unsure which node in a tree has which position, you can replace all nodes with there positions by

1. **nodes > positions**

6 Specifications

Up until know, only simple functions were described that worked on terms and graphs directly, but didn't require any involved knowledge what kind of graph and what it represents is needed. Now we shift to more interesting examples. These examples more often than not require encapsulated descriptions of the model. These descriptions are called **Specifications**. To load a **Specification** it needs to be stored in the *ExpanderLib/* directory.

6.1 Design Graphs

You already have seen a **Specification** that represents a graph with its transitions. The difference was, that it wasn't loaded as a **Specification** but as the axioms directly. Now we want to have a separate file for it. This is a simple example, but still useful when a graph is needed. The only difference for this example is, that you specify which blocks of the file are what. In this case only a single block, an *axioms* block, is needed. A block is introduced by its identifier and terminated either by a different identifier or by the end of file. Axioms are connected with one another with the \mathcal{E} operator.

```
axioms:
states == [1..4] &
1 -> 2 &
2 -> 3 &
3 -> 4 &
4 -> 2 &
3 -> 1
```

6.2 Functions

Functions are defined in the *axioms* block in a specification. Consider this example.

```
-- example 1
defuncts: repeat length
fovars: x xs a n

axioms:

repeat(0, a) == [] &
repeat(n, a) == a : repeat(n-1, a) &

length([]) == 0 &
length(x:xs) == 1+length(xs)
```

The function names defined in a specification are listed in the *defuncts* block at the beginning of a specification. Even though not strictly necessary, variables should also be listed at the top of the specification, because behavior differs if the *Expander* knows that a symbol is used as a variable. Variables are separated into two groups. First order variables (*fovars*), the most common variables you will encounter and higher order variables (*hovars*). A function is defined by the way it input relates to its output. For example the function *repeat* relates the input $(0, a)$ for any a to the empty list and the input (n, a) for any n or a to the list with a at the beginning and the tail *repeate* $(n - 1, a)$. Generally, the left side (the inputs) are separated from the outputs by a `==`. Load this specification in the *Expander* by creating a file in your *ExpanderLib* and copy the contents into it. Then enter the file name in the **Entry Field** and press enter. If the specification is loaded successfully The **Label** is set to "The axioms in example1 have been added." where "example1" is replaced by whatever filename you chose. To evaluate a function:

1. Load a term with the function into the canvase
2. Push the **simplify** button. ^{4, 5}
3. The term will be simplified ie. functions are evaluated by applying axioms 100 times.
4. This number can be changes by inserting the desired number in the **Entry Field**

6.3 Predicates

Similarly, predicates can be defined. Predicates define relations, as such functions and predicates can express the same ideas. But, depending on the circumstances, one or the other can be more effective. Especially then proofs are the

⁴The Evaluation Strategy ([1], Chapter 8) can be changed with the button on the left.

⁵Simplification rules can be found at [2], Chapter 5

main concern. Predicates are defined by describing which inputs relate to an output. For example, *lengthP*, the first line of the definition defines that [] and 0 are in the relation *lengthP*. As is a list $x : xs$ and n if xs and $n - 1$ in the relation *lengthP*. The "execution" of a term like *lengthP*([1, 2, 3, 4], 4) happens from left to right, this means that *lengthP*([1, 2, 3, 4], 4) is "executed" to *lengthP*([2, 3, 4], 4 - 1). The intuitiv idea behind this is that you try to complete the implication to proof the statement *lengthP*([1, 2, 3, 4], 4). So what you try to derive is *lengthP*([1, 2, 3, 4], 4) \Leftarrow *lengthP*([2, 3, 4], 3) \Leftarrow *lengthP*([3, 4], 2) \Leftarrow *lengthP*([4], 1) \Leftarrow *lengthP*([], 0) \Leftarrow *True*. *Expander* can do this "execution" and does so when you press the **narrow** button. Note though that the you may need to use the **simplify** button inbetween of narrowing steps because the function (-) is used and *Expander* does not perform simplification in between narrowing steps on its own. The here defined predicates use Horn Clauses. There are also coHorn Clauses ⁶ which can also define logic programms.

```
-- example 1
preds: repeatP lengthP
fovars: x xs a n

axioms:

lengthP([], 0) &
(lengthP(x:xs, n) <=== lengthP(xs, n-1)) &

repeatP(0, a, []) &
(repeatP(n, a, a:xs) <=== repeatP(n-1, a, xs))
```

6.4 Induction

Induction is one of the most famous proof principle. It's based on the fact that many statements can be proven by showing the statement for a base case and then derive the statement for bigger cases under the assumption that the statement hold for the smaller case. Consider the specification of 6.2. It seems rather obvious that the following statement hold: "For all $n \in \mathbb{N}$ if $x = repeat(n, a)$ then $length(x) = n$ ". This statement can be proven by hand but it can also be automated by using the Expander. For better understanding of the induction principle, first the "by hand method" will be considered. As outlined before, the base case is considered first. In case of the natural number 0 is the smallest number and therefore the base case.

Proof. Base $n = 0$: $x = repeat(n, a) = repeat(0, a) = []$, therefore $length(x) = length([]) = 0$
Induction Hypothesis: $\forall n' \leq n. x = repeat(n', a) \Rightarrow length(x) = n'$
Induction Step $n \rightarrow n + 1$: $x = repeat(n + 1, a) = a : repeat(n, a)$, therefore $length(x) = length(a : repeat(n, a)) = 1 + length(repeat(n, a)) \stackrel{InductionHypothesis}{=} 1 + n$. \square

As you might noticed, there is nothing interesting going on. The only thing that is happening, is that the definitions of *repeat* and *length* are used and the induction hypothesis is used. And in fact, the same steps are taken in most proofs by induction and can therefore be automated away to make life easier.

To use a proof automator like the Expander, the statement needs to be translated into the representation used by the tool in question. In case of Expander it's a fairly straight forward translation. $x = repeat(n, a) ==> length(x) = n$. Variables like n and a are all-quantified ⁶. To proof the statement follow these steps:

1. Load the specification into Expander
2. Load the term $x = repeat(n, a) ==> length(x) = n$ into the **Canvas**
3. Choose **selection > induction**

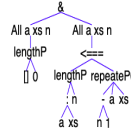
This should derive to the value *True*. This tells the user that the induction finished in one induction step, as the proof by hand, and derived that the statement is true. But notes should be taken. For one, this is a fairly simple proof and sometimes custom steps like simplification might be needed or more than one induction step is needed. Second, in the

⁶A description of these can be found at [1] Chapter 9.

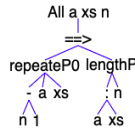
proof by hand, a specialised induction is used called natural induction over natural number. Expander, by default, uses a more general form of induction called fixpoint induction.⁷ Other forms are available as well.⁸

Of course the same proposition can be proven for the predicates. The proof is a good example of a proof that needs more manual work. This is because of the nature of predicates. Use the specification in 6.3 for the following proof.

1. Load the specification into Expander
2. Load the term $repeateP(n, a, xs) ==> lengthP(xs, n)$



3. Choose **selection** > **induction**, resulting in
4. Choose **narrow**



5. Choose **simplify**, resulting in
6. Choose **narrow**
7. Choose **narrow**
8. Choose **simplify**, resulting in *true*

A third type of induction that should be mentioned in noetherian induction.⁹ In order for noetherian induction to work, a order needs to be defined. More specifically >> needs to be defined. Consider the following specification.

```
-- gauss
defuncts: sumTo
fovars: n
preds: >>

axioms:

sumTo(0) == 0 &
sumTo(n) == n+sumTo(n-1) &
(x >> y <== x > y)
```

It defines >> as the normal *greater than* order on natural numbers. How the order needs to be defines depends on the conjecture that you want to proof. To use noetherian induction follow these steps:

1. Load the specification into Expander
2. Load the term $((n * (n + 1))/2) = x ==> sumTo(n) = x$
3. ??

6.4.1 Saving and Demonstrating your Proof

As you can see, the proof involves custom **narrow** and **simplify** steps. The steps are not directly obvious so you might want to write them down to share or to replay. Both of those can be handled by the Expander. To save a finished proof, enter a name *name* into the **Entry Field** and choose **formula** > **save proof to file (s)**. This creates

⁷Rules for a fixpoint induction can be found at [2], Chapter 6

⁸See [2] for a full list.

⁹Rules for a Noetherian induction can be found at [2], Chapter 6

two files, *nameP* and *nameT*. *nameP* is a *human readable* version of the proof giving the steps and their results. *nameT* is a so called proof term. It is used to replay the proof.

A proof is not worth much if you can't easily convince someone else that it is correct. If the proof is saved, Expander can show anyone how the proof works. To replay a proof follow these steps:

1. Load the specification
2. Load the term with which the proof starts
3. Enter the name of file containing the proof term into the **Entry Field** ¹⁰
4. Choose **formula > check proof term from file**

Note that the proof term only contains the step taken. This means that the proof replayed with the wrong term in the canvas might not crash Expander but will probably create unexpected results. Refer to 7 for more Information about Check Proof Mode.

The *human readable* proof created by Expander might look like the following.

```
0. Derivation of
  repeate(n,a) = xs ==> length(xs) = n
  All simplifications are admitted.
  Equation removal is safe.
1. Adding
  (repeate0(n,a,xs) ==> length(xs) = n)
  to the axioms and applying FIXPOINT INDUCTION wrt
  at positions
  []
  of the preceding trees leads to
  True
```

This is the proof which we did by hand before. Depending on what you are expecting one might be easier to understand, but the advantage is obvious, you do not need to write it by hand.¹¹

7 Check Proof Mode

Check Proof Mode is the mode Expander uses when in a proof replay. It doesn't allow normal functions like *copy* to be used but it makes it possible to go through the different results in the proof. To get into Check Proof Mode start replaying a proof as in 6.4.1. This replays the proof to completion. Click **stop run** in the bottom right. This allows you to go back and forward in the proof by using the <- and -> respectively. To start the replay process again, click **run proof**. To exit Check Proof Mode use the **quit check** button. When **stop run** is activated, it is possible to use **paint** to enter the Painter-Window. See 8 for more details. All other functions may result in undocumented behavior.

8 Painter

Painter is a subsystem of Expander. It can be entered by clicking the **paint** button on the left side in the Expander window. The term currently displayed in the canvas is copied over to the Painter window for modification. It allows aesthetic modifications to the terms. The main tool here is the mouse. You can use it to move nodes in the canvas and rearrange them however you like.

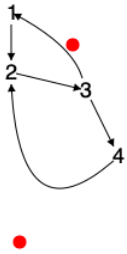
8.1 Combinations

The Term is displayed in different modes called combinations. You can cycle through them by using the **combis** button in the center of the button area. The most useful combinations are discussed now.

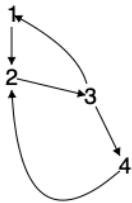
¹⁰Without the T. So when checking *proofT proof* needs to be entered.

¹¹More Sample Proofs can be found at [2], Chapter 7

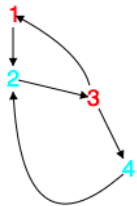
1. **Combination 0** displays the term in a simple fashion. The main the feature here is, that you can move each edges support point. This makes it possible to not only move nodes but edges as well. The support points are displayed as red dots.



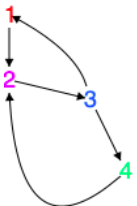
2. **Combination 1** simply removes the support point to save the picture.



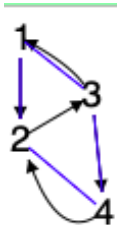
3. **Combination 2** colors the level in the term in two alternating colors. From level to level the color switches. It does **not** color the nodes according to the coloring problem.



4. **Combination 3** colors the nodes according to there level through the whole color circle.



5. **Combination 4** adds the convex cover of the graph.



8.2 Saving a picture

1. Saving to Haskell Code: The graph can be saved as the corresponding Haskell Code by entering the name in the text field in the lower center of the window and pressing the down arrow key.
2. Saving to Picture: To save the graph as a picture enter in the same text field the name of the file **with** the file extension of the format you want. For example *test.png* creates a PNG file. Supported formats are:

- (a) Portable Network Graphics with the .png extension
- (b) Encapsulated PostScript with the .eps extension
- (c) Graphics Interchange Format with the .gif extension

8.3 Save Modification Sequence

When you want a sequence of modifications to be saved Expander can do that. Enter a directory name into the entry field and choose **set dir**. When ever **save pic to dir** either in a Painter or Expander window a new picture is created that directory with a new number. Also in *ExpanderLib* a .html file with the name of the directory is created. This file contains a site that allows to go through the pictures in the corresponding directory.

8.4 Widgets

Widgets allow the user to add frames and colors to nodes. This is done using *wtree* function. Consider this example `drawS == wtree $ fun(sat$st,frame$text$st)`. The argument to *wtree* is a λ -function. This particular function does a pattern match and frames any node that has a *sat* constructor in front of it. Further examples can be taken from the *widget*-specification distributed with Expander.

9 Commands

Command	Prerequisites	Effect
selection > copy	selected subtree	subtree is saved in a clipboard
selection > insert/replace by text (i)	tree in clipboard selected subtree	selected subtree is replaced by the tree in the clipboard
selection > reverse (v)	selected subtree	reverses order of children
selection > remove	selected subtree	removes the subtree
selection > remove node (o)	selected subtree	only removes the root of the subtree, children are moved up one level
selection > replace by tree of SolverX	selected subtree	replaces selected subtree with tree from SolverX
parse down	term in canvas	turns term in canvas into the linear text from of the term
load > load from text file	file name in the Entry Field	loads the term from the file into the canvase
node > level numbers	tree in canvas	replaces nodes with their depth in the tree
nodes > preorder numbers	tree in canvas	replaces nodes with their preorder number
nodes > heap numbers	tree in canvas	replaces nodes with their heap number
specification > build Kripke model	specification loaded	builds the Kirpke model of the transition system in the specification
graph > of transitions > here	Kripke model	displays graph of the transition model
graph > of Kripke model > here	Kripke model	displays graph of the kripke model
graph > show Boolean matrix > of transitions	Kripke model	displays transitions in form of an adjacency matrix
graph > show binary relation > of transitions	Kripke model	displays the transition relation
graph > build iteratiove equations	Kripke model	displays the iterativ equations for the graph
graph > collapse ->	graph in canvas	collapses equal nodes into each other keeping the rightmost occurrence
graph > collapse <-	graph in canvas	collapses equal nodes into each other keeping the leftmost occurrence
graph > split cycles	graph in canvas	removes cycles by unfolding them.
simplify	term in canvas	tries to evaluate the term by using the function in scope
narrow	term in canvas	tries to derive term by using the predicates in scope
selection > induction	term in canvas	tries to derive term by using the principle of induction and the predicates in scope
paint	term in canvas not in proof run	opens Painter-Window for further modifications
stop run / run proof	in Check Proof Mode	stop/restart proof run at current position
quit proof	in Check Proof Mode	exit Check Proof Mode
<-	-	undo
->	-	redo
combis	in Painter window	cycles through the combinations available
set dir	name in entry field	sets the directory value
save pic to dir	set dir	creates new picture in directory

References

- [1] P. Padawitz, “From modal logic to (co)algebraic reasoning,” jan 2019. [Online]. Available: <https://fdit-www.cs.tu-dortmund.de/~peter/CTL.pdf>
- [2] —, “Expander2 as a prover and rewriter,” jan 2019. [Online]. Available: <https://fdit-www.cs.uni-dortmund.de/~peter/Expander2/Prover.pdf>