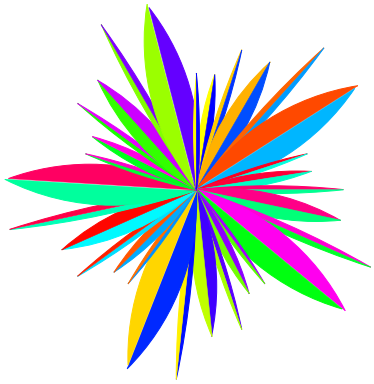


Expander2

https://fldit-www.cs.uni-dortmund.de/~peter/Expander2.html

Peter Padawitz

TU Dortmund

January 18, 2019

(actual version: http://fldit-www.cs.tu-dortmund.de/~peter/Expander2/Manual1.pdf)

Contents

This paper shall become the new manual of Expander2. The old manual can be found here.

1 Overview

Expander2 is a flexible multi-purpose workbench for interactive term rewriting, graph transformation, theorem proving, constraint solving, flow graph analysis and other procedures that build up proofs or computation sequences. Moreover, tailor-made interpreters display terms as two-dimensional structures ranging from trees and graphs to a variety of pictorial representations that include tables, matrices, alignments, partitions, fractals and various tree-like or rectangular graph layouts (see section 8). Proofs and computations performed with Expander2 follow the rules and the semantics of swinging types.

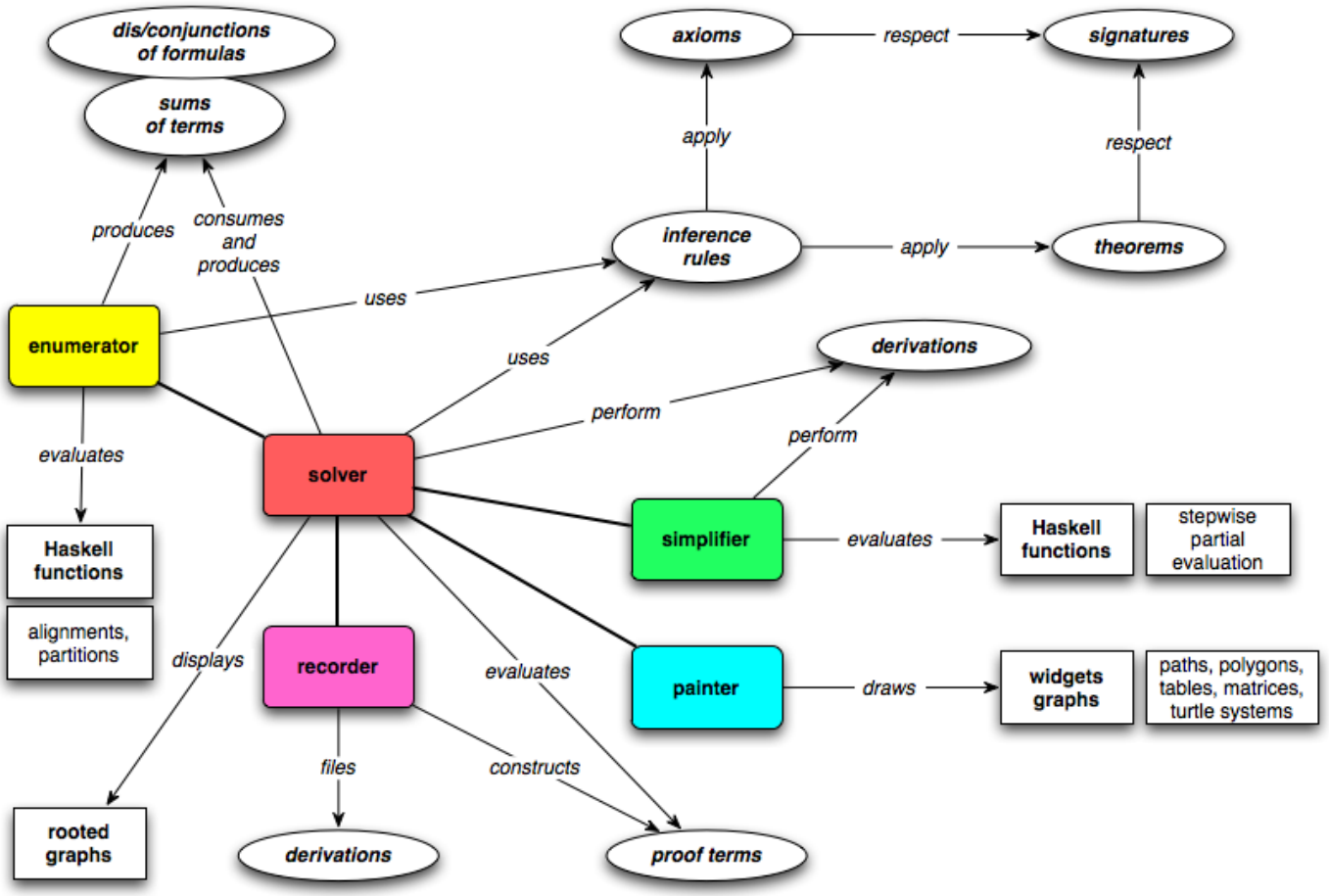
Swinging types are based on many-sorted predicate logic and combine constructor-based types with destructor-based (e.g. state-based) ones. The former come as initial term models, the latter as final models consisting of context interpretations. Relation symbols are interpreted as least or greatest solutions of their respective axioms.

The user may interact with the system at three levels of decreasing control over proofs and computations. At the top level, rules like induction and coinduction are applied locally and step by step. At the medium level, goals are rewritten or narrowed, i.e. axioms are applied exhaustively and iteratively. At the bottom level, built-in rules (some of them executing Haskell programs) simplify, i.e. (partially) evaluate terms and formulas, and thus hide routine steps of a proof or computation. Proofs are automatically translated into proof terms that can be evaluated and modified later. This allows one to design functional-logic programs as proof carrying code that a client can validate by running the proof term evaluator (proof checker).

Expander2 has been written in O'Haskell, an extension of Haskell with object-oriented features for reactive programming and a typed interface to Tcl/Tk. Besides a comfortable GUI the design goals of Expander2 were to integrate testing, proving and visualizing deductive methods, admit several degrees of interaction and keep the system open for extensions or adaptations of individual components to changing demands.

Send comments, bugs, etc. to Peter Padawitz. *Any suggestions for improvements, extensions, applications or project proposals are welcome!*

The main components of Expander2 are the **solver**, the **painter**, the **simplifier**, the **enumerator** and a **recorder** of proofs and computation sequences.



The **solver** is accessed via a window for editing and displaying trees that represents a disjunction or conjunction of logical formulas or a sum of functional terms. A proper (non-singleton) sum results from a computation obtained by nondeterministic rewriting. The solver window has a canvas for the two-dimensional representation of the list of current trees (among which one browses by moving the slider below the window) and a text field for their string representation. With the *parse buttons* one switches between the tree (or graph) and the string representation. Both representations are editable. As the usual cut, copy and paste operate on substrings in the text field, so do corresponding mouse-triggered functions when the cursor is moved over subtrees on the canvas.

After a widget interpreter has been selected from the *pict type menu*, pushing the paint button opens a painter window and the pictorial representations of all interpretable subtrees of the solver's current trees will be shown. Pictures are lists of widgets that can be edited in the painter window and completed to widget graphs. Widgets are built up of path, polygon and turtle action constructors that admit the definition of a variety of pictorial representations ranging from tables and matrices via string alignments, piles and partitions to complex fractals generated by turtle systems [22], which define a picture in terms of a sequence of actions that a turtle would perform when drawing the picture while moving over a canvas. The turtle works recursively in two ways: it maintains a stack of positions and orientations where it may return to, and it may give birth to subturtles, i.e. call other turtle systems. The solver and its associated painter are fully synchronized: the selection of a tree in the solver window is automatically translated to a selection of the tree's pictorial representation in the painter window and vice versa. Hence

rewriting, narrowing and simplification steps can be carried out from either window.

The **enumerator** provides algorithms that enumerate trees or graphs and passes their results both to the solver and the painter. Currently, two algorithms are available: a generator of all sequence alignments [5, 15] satisfying constraints that are partly given by axioms, and a generator of all nested partitions of a list with a given length and satisfying constraints given by particular predicates. The painter displays an alignment in the way DNA sequences are usually visualized. A nested partition is displayed as the corresponding rectangular dissection of a square.

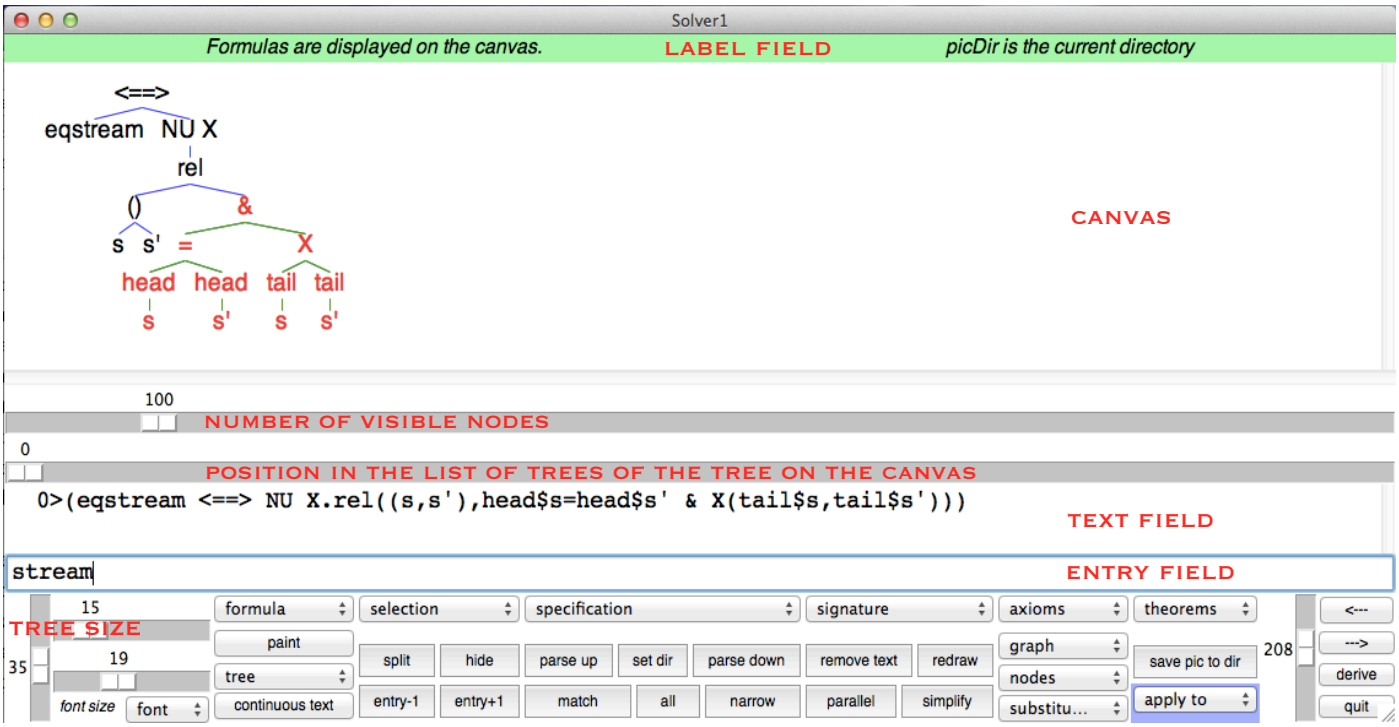


Figure 1. *The solver window*

Expander2 allows the user to control proofs and computations at **three levels of interaction**:

At the high level, analytic or synthetic inference rules or other syntactic transformations are applied individually and locally to selected subtrees (see the *transform-selection menu*). The rules cover single axiom applications, substitution or unification steps, Noetherian, Hoare, subgoal or fixpoint induction and coinduction. Derivations are correct if, in the case of trees representing terms, their sum is equivalent to the sum of their sucessors or, in the case of trees representing formulas, their dis- resp. conjunction is implied by the dis- resp. conjunction of their successors. The underlying models are determined by built-in data types and the least/greatest interpretation of Horn/co-Horn axioms. Incorrect deduction steps are detected and cause a warning. All proper tree transformations are recorded, be they correct proofs or other transformations. Terms and formulas are built up from the symbols of the current signature. For more details on the syntax and semantics of axioms, theorems and goals, see section 5 and [11].

At the medium level, rewriting and narrowing realize the iterated and exhaustive application of all axioms for the defined functions, predicates and copredicates of the current signature. Terminating rewriting sequences end up with **normal forms**, i.e. terms consisting of constructors and variables.

Terminating narrowing sequences end up with the formula True, False or *solved formulas* that represent solutions of the initial formula. Since the axioms are functional-logic programs in abstract logical syntax, rewriting and narrowing agree with program execution. Hence the medium level allows one to test such programs, while the inference rules of the high level provide a "tool box" for program verification. In the case of finite data sets, rewriting and narrowing is often sufficient even for program verification. Besides relations and deterministic functions, non-deterministic transition systems employing structured states, such as Maude programs [3] or algebraic nets [23], may also be axiomatized and verified by Expander2. The latter are executed by applying associative-commutative rewriting or narrowing on bag terms, i.e. multisets of terms.

At the low level, built-in Haskell functions simplify or (partially) evaluate terms and formulas and thereby hide most routine steps of proofs or computations. The functions comprise arithmetic, list, bag and set operations, term equivalence and inequivalence (that depend on the current signature's constructors) and logical simplifications that turn formulas into **nested Gentzen clauses**. Evaluating a function f at the medium level means narrowing upon the axioms for f, Evaluating f at the low level means running a built-in Haskell implementation of f. This allows one to test and debug algorithms and visualize their results. For instance, translators between different representations of Boolean functions were integrated into Expander2 in this way. In addition, an execution of an iterative algorithm can be split into its loop traversals such that intermediate results become visible, too. Currently, the computation steps of Gaussian equation solving, automata minimization [8], OBDD optimization, LR parsing, data flow analysis and global model checking can be carried out and displayed (see section 6).

For an introduction, see also Expander2 as a Prover and Rewriter, sections 1 and 2.

2 Overall code structure

Expander2 consists of six O'Haskell modules:

- **Ecom.hs** configures the GUI and provides all string- or tree-generating, -manipulating or -translating commands that the user may call for carrying out proofs or computations and presenting their results interactively. Multiple tree-shaped results can be displayed and browsed through on the canvas of a solver and in some cases interpreted graphically and displayed in the painter window of a solver. *Ecom* closes with the main program of the system that creates the main objects, partly in a mutually recursive way:

```
main :: TkEnv -> Cmd ()
main tk = do
 mkDir $ userLibDir
 mkDir $ userLib "Pix"
 win1 <- tk.window []
 win2 <- tk.window []
 fix solve1 <- solver tk "Solver1" win1 solve2 "Solver2" enum1 paint1
     solve2 <- solver tk "Solver2" win2 solve1 "Solver1" enum2 paint2
```

```
    paint1 <- painter 820 tk "Solver1" solve1 "Solver2" solve2
    paint2 <- painter 820 tk "Solver2" solve2 "Solver1" solve1
    enum1 <- enumerator tk solve1
    enum2 <- enumerator tk solve2
solve1.buildSolve (0,20)
solve2.buildSolve (20,20)
win2.iconify
```

- **Epaint.hs** provides Haskell functions for parsing terms and formulas and computing and displaying their graphical representations that are built up from Tk canvas widgets. Collections of various pictorial elements can be defined as movements over the plane according to a *turtle interpretation* (see section 8). The reactive components for animating the turtle and displaying graphical objects are gathered in the `painter`, `crawler` and `slowActor` templates (= classes). The `colorFlasher` template animates the error messages appearing in label fields (see below).
- **Esolve.hs** encapsulates translators between string, tree and graphical representations of terms and formulas. *Esolve.hs* also contains the **simplifier** that partially evaluates terms and formulas. Moreover, the basic inference rules for applying axioms and theorems are implemented here. *Esolve.hs* also contains the `enumerator` template that provides a GUI for running tree enumeration algorithms (see Section 9). They are called from the `solver` template, which is part of *Ecom.hs*.
- **Eterm.hs** contains data types and functions for generating, manipulating or checking terms and formulas, such as unification, matching, reduction and expansion of collapsed trees.
- **System.hs** and **Tk.hs** provide the interface to Tcl/Tk.

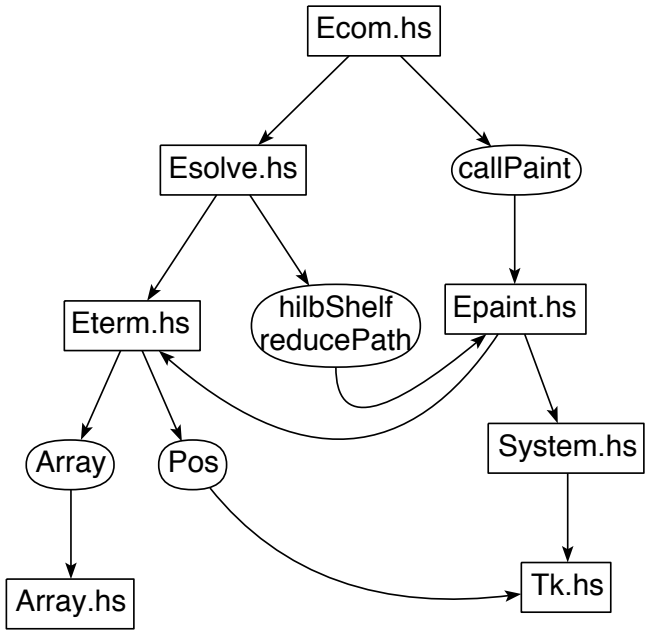


Figure 2. *A part of the Haskell-modules use graph*

3 Menus and commands

A command followed by a letter in round brackets is executed when the corresponding key is pushed

after the keyboard has been activated by placing the cursor over the entry resp. label field and pressing the left mouse button. The keys for specification > add, apply clause, load text and save tree work if the entry field has been activated. The keys for parse up and parse down work if the text field has been activated. The keys for other commands work if the label field has been activated.

4 Terms, formulas and their manipulation

The type of terms (see Eterm.hs)

```
data Term a = V a | F a [Term a] | Hidden Special deriving (Show,Eq,Ord)

data Special = Dissect [(Int,Int,Int,Int)] |
               BoolMat [String] [String] [(String,String)] |
               ListMat [String] [String] (Triples String String) |
               ListMatL [String] (TriplesL String) |
               LRarr (Array (Int,Int) ActLR) |
               ERR deriving (Show,Eq,Ord)

type TermS = Term String
```

See From Modal Logic to (Co)Algebraic Reasoning, section 8.

5 Specifications

The type of signatures (see Eterm.hs)

```
struct Sig = isPred,isCopred,isConstruct,isDefunct,isFovar,isHovar,blocked
                                  :: String -> Bool
             hovarRel             :: BoolFun String
             safeEqs              :: Bool
             simpls,transitions   :: [(TermS,[TermS],TermS)]
             states,atoms,labels  :: [TermS]          -- types of a Kripke model
             trans,value          :: [[Int]]
             transL,valueL        :: [[[Int]]]
```

A sample specification

```
-- PAFL
defuncts: flatten
preds:    part >>
fovars:   s s' p s1
axioms:
```

```
  part([x],[[x]])
& (part(x:y:s,[x]:p) <=== part(y:s,p))
& (part(x:y:s,(x:s'):p) <=== part(y:s,s':p))
& flatten[] == []
& flatten(s:p) == s++flatten(p)
& x:s>>s
& (s >> s' <=== s >> s1 & s1 >> s')
conjects:
part(s,p) ==> flatten(p)=s
```

See also Expander2 as a Prover and Rewriter, section 3, and From Modal Logic to (Co)Algebraic Reasoning, sections 10-22.

6 Simplifications

See also Expander2 as a Prover and Rewriter, section 5, and From Modal Logic to (Co)Algebraic Reasoning, section 8.

7 Narrowing, rewriting and (co)induction

Animation of interactive proofs of the conjecture of PAFL by fixpoint and Noetherian induction, respectively.

See Expander2 as a Prover and Rewriter, sections 6 and 7, and From Modal Logic to (Co)Algebraic Reasoning, section 9.

8 Widgets, pictures and graphs

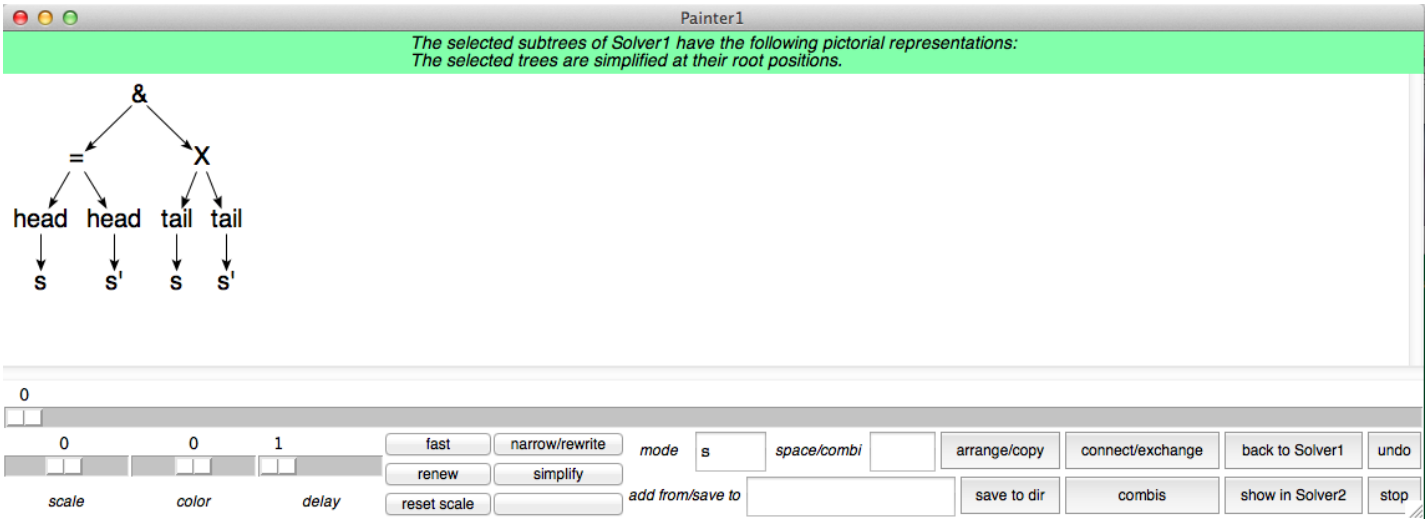


Figure 3. *The painter window*

Painter types (see Epaint.hs)

```
type Point  = (Float,Float)
type Line_  = (Point,Point)
type Lines  = [Line_]
type Path   = [Point]
type State  = (Point,Float,Color,Int) -- (center,orientation,hue,lightness)

type Graph   = (Picture,Arcs)
type Picture = [Widget_]
type Arcs    = [[Int]]

-- ([w1,...,wn],[as1,...,asn]) :: Graph represents a graph with node set
-- {w1,...,wn} and edge set {(wi,wj) | j in asi, 1 <= i,j <= n}.

data Widget_ = Arc State ArcStyleType Float Float | Bunch Widget_ [Int] |
               -- Bunch w is denotes w together with outgoing arcs to the
               -- widgets at positions is.
               Dot Color Point | Fast Widget_ | Gif String Widget_ | New |
               Oval State Float Float | Path State Int Path |
               Path0 Color Int Int Path | Poly State Int [Float] Float |
               Rect State Float Float | Repeat Widget_ | Skip |
               Text_ State Int [String] [Int] |
               Tree State Int Color (Term (String,Point,Int)) |
               -- The center of Tree .. ct agrees with the root of ct.
               Tria State Float | Turtle State Float TurtleActs | WTree TermW
               deriving (Show,Eq)

type TurtleActs = [TurtleAct]
data TurtleAct  = Close | Draw |
                  -- Close and Draw finish a polygon resp. path starting at the
                  -- preceding Open command.
                  Jump Float | JumpA Float | Move Float | MoveA Float |
                  -- JumpA and MoveA ignore the scale of the enclosing turtle.
                  Open Color Int | Scale Float | Turn Float | Widg Bool Widget_
                  -- The Int parameter of Open determines the mode of the path
                  -- ending when the next Close/Draw command is reached;
                  -- see drawWidget (Path0 c i m ps).
                  -- Widg False w ignores the orientation of w, Widg True w
                  -- adds it to the orientation of the enclosing turtle.
                  deriving (Show,Eq)
```

```
type WidgTrans = Widget_ -> Widget_
type PictTrans = Picture -> Picture

type TermW  = Term Widget_
type TermWP = Term (Widget_,Point)
```

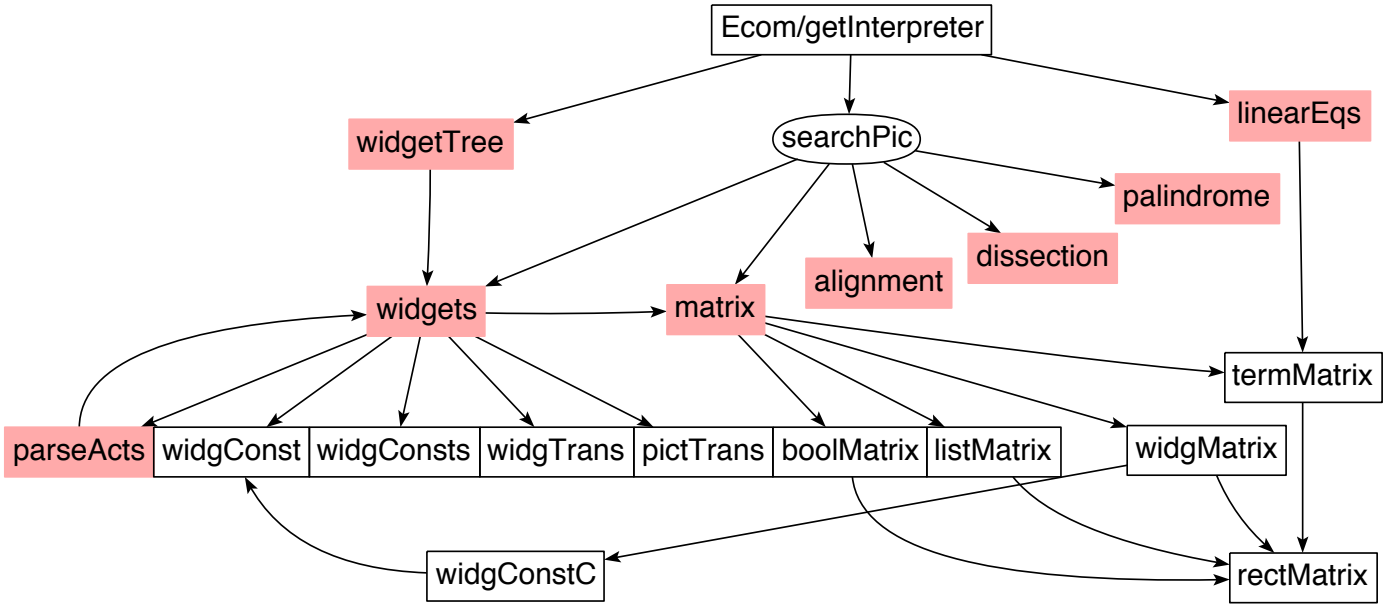


Figure 4. *Eight graphic term interpreters*

Specifications of the Examples directory containing terms processable by a term interpreter

- diagrams (widget trees; see below)
- flowers
- graphs (widget trees)
- grow
- morphs
- overlaps
- polygons
- rainbows
- rotate
- shelves
- shine
- widgets (many examples)

The part of *diagrams* that specifies Figures 2 and 4

```
defuncts: draw
fovars: com solve paint term tk
        egi wit wis mat les ali dis pal bm lm tm wm rm pas wc search
axioms:
draw == wtree $ fun(edge(x),frameS(5)$text$x,
                    red(x),light red$frameF(5)$black$text$x,
                    x,frame(5)$text$x)
conjects:
-- Equations for Figure 2
com = string(Ecom.hs)(solve,string(edge$callPaint)$paint) &
solve = string(Esolve.hs)(term,string(edge$hilbShelf reducePath)$paint) &
paint = string(Epaint.hs)(term,(string$System.hs)$tk) &
term = string(Eterm.hs)(string(edge$Array)$string(Array.hs),
                        string(edge$Pos)$tk) &
tk = string(Tk.hs) &
-- Equations for Figure 4
egi = string(Ecom/getInterpreter)(wit,search,les) &
wit = string(red$widgetTree)(wis) &
wis = string(red$widgets)(pas,wc,widgConsts,widgTrans,pictTrans,mat) &
mat = string(red$matrix)(bm,lm,wm,tm) &
les = string(red$linearEqs)(tm) &
bm = boolMatrix(rm) & lm = listMatrix(rm) & tm = termMatrix(rm) &
wm = widgMatrix(rm,widgConstC$wc) &
rm = rectMatrix &
pas = string(red$parseActs)(wis) &
wc = widgConst &
search = string(edge$searchPic)(wis,mat,string(red$alignment),
                                        string(red$dissection),
                                        string(red$palindrome))
```

How to build, save and load Figure 4:

- enter *diagrams* into the entry field
- press the *return* key: *diagrams* is compiled to a specification
- press (the) *theorems* > *show conjects* (button): the conjectures of *diagrams* are listed in the text field
- cut out the equations for Figure 2 (see above)
- press *parse up*: the equations for Figure 4 are displayed on the canvas
- press *simplify*: the term arguments of *string* are converted to strings
- press *graph* > *show graph* button: the equations are translated to a graph
- for changing node positions, move the tree size sliders

- enter *Figure4S.png* into the entry field
- press the down arrow key: the graph on the solver canvas is saved to *ExpanderLib/Pix/Figure4S.png*
- enter *Figure4S.eps* into the entry field
- move the lower-right corner of the solver window in order to remove empty space below or right to the graph on the canvas
- press the down arrow key: the entire solver canvas is saved to the *ExpanderLib/Pix/Figure4S.eps*
- enter *draw* into the entry field
- press the *tree* button in the menu below *paint*
- press the *paint* button: the graph on the canvas is converted to an object of type *Graph* (see above) and displayed in a *painter* window
- for changing node or (red) edge support point positions, click and move them with the left mouse button
- enter 1 into the *save/combi* field and push *combis*: support points are removed
- enter *Figure4* into the *add from/save to* field
- press the down arrow key: the Haskell code of the figure on the painter canvas is saved to *ExpanderLib/Figure4*
- enter *Figure4.png* into the *add from/save to* field
- press the down arrow key: the graph on the painter canvas is saved to *ExpanderLib/Pix/Figure4.png*
- enter *Figure4.eps* into the *add from/save to* field
- move the lower-right corner of the solver window in order to remove empty space below or right to the graph on the canvas
- press the down arrow key: the entire painter canvas is saved to the *ExpanderLib/Pix/Figure4.eps*
- delete the entry in the *add from/save to* field
- press the up arrow key: the graph on the painter canvas is deleted
- enter *Figure4* into the *add from/save to* field
- press the up arrow key: by executing the Haskell code in *ExpanderLib/Figure4*, Figure 4 is re-displayed on the painter canvas

See also Picture construction and animation with Expander2, sections 3-11, and Das Painter-Manual, sections 3-6.

9 Loading and saving terms, proofs or pictures

10 Building and verifying transition systems

See From Modal Logic to (Co)Algebraic Reasoning, sections 1-7 and 10-17.

11 Enumerators

12 References

[1] S. Antoy, R. Echahed, M. Hanus, A Needed Narrowing Strategy, Journal of the ACM 47 (2000) 776-822

[2] R.E. Bryant, Graph-Based Algorithms for Boolean Function Manipulation, IEEE Transactions on Computers 35 (1986) 677-691

[3] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, C. Talcott, Maude Manual, SRI International 2005, Maude

[4] Frank Drewes, Grammatical Picture Generation, Springer 2006

[5] R. Giegerich, A Systematic Approach to Dynamic Programming in Bioinformatics. Parts 1 and 2: Sequence Comparison and RNA Folding, Report 99-05, Technical Department, University of Bielefeld 1999

[6] R. Giegerich, C. Meyer, Algebraic Dynamic Programming, Proc. AMAST 2002, Springer LNCS 2422 (2002) 249-364

[7] Andrew D. Gordon, Bisimilarity as a Theory of Functional Programming, Theoretical Computer Science 228 (1999) 5-47

[8] J.E. Hopcroft, R. Motwani, J.D. Ullman, Introduction to Automata Theory, Languages, and Computation, 2nd ed., Addison-Wesley 2001

[9] J.B. Kam, J.D. Ullman, Global data flow analysis and iterative algorithms, Journal of the ACM 23 (1976) 158-171

[10] P. Padawitz, Expander2 as a Prover and Rewriter, TU Dortmund 2017

[11] P. Padawitz, From Modal Logic to (Co)Algebraic Reasoning, TU Dortmund 2017

[12] P. Padawitz, Computing Rectangular Dissections, Research Report 536/1994, Dept. of Comp. Sci., University of Dortmund 1994

[13] P. Padawitz, Inductive Theorem Proving for Design Specifications, J. Symbolic Computation 21 (1996) 41-99

[14] P. Padawitz, Swinging Types = Functions + Relations + Transition Systems, Theoretical Computer Science 243 (2000) 93-165

[15] P. Padawitz, Swinging Types At Work

[16] P. Padawitz, Structured Swinging Types

[17] P. Padawitz, Fixpoints, Categories, and (Co)Algebraic Modeling, TU Dortmund 2017

[18] P. Padawitz, Expander2: Program Verification between Interaction and Automation, slides, University of Madrid 2006

[19] P. Padawitz, Dialgebraic Picture Generation: A case Study in Multi-Level Data Abstraction, slides, University of Dortmund 2006

[20] P. Padawitz, Formale Methoden des Systementwurfs, course notes, University of Dortmund 2005

[21] P. Prunsinkiewicz, A. Lindenmayer, *The Algorithmic Beauty of Plants*, Springer 1990

[22] G. Rozenberg, A. Salomaa, eds., *Handbook of Formal Languages, Vol. 3: Beyond Words*, Springer 1997

[23] M.-O. Stehr, J. Meseguer, P.C. Ölveczky, Rewriting Logic as a Unifying Framework for Petri Nets, in: H. Ehrig et al., eds., Unifying Petri Nets, Springer LNCS 2128 (2001)