

Expander2

– a formal methods presenter and animator –

Peter Padawitz

May 28, 2017

Expander2 is a flexible multi-purpose workbench for interactive rewriting, theorem proving, constraint solving, flow graph analysis and related procedures that build up proofs or other computation sequences. The associated GUI provides 2D representations of terms and formulas ranging from trees and rooted graphs to a variety of pictorial representations, including tables, matrices, alignments, partitions, fractals and turtle systems (see [Widgets, graphs, and turtle actions](#)).

The user may interact with the system at three levels of decreasing control over proofs and computations. At the first level, rules like induction and coinduction are applied locally and step by step. At the second level, goals are rewritten or narrowed, i.e. axioms are applied exhaustively and iteratively. At the third level, built-in rules (some of them execute Haskell programs) simplify, i.e. (partially) evaluate terms and formulas, and thus hide routine steps of a proof or computation (see [Overview](#)). 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.

Proofs and computations performed with Expander2 follow the rules and the semantics of [swinging types](#). Swinging types combine constructor-based visible types with state-based hidden ones and have unique initial models that interpret relations as the least or greatest solutions of their axioms.

Please email comments, bugs, etc. to [Peter Padawitz](#). **Any suggestions for further applications, improvements, extensions or project proposals are welcome!**




















Contents









1	Main Commands	6
2	Overview	8
3	Overall code structure	10
4	Starting the system	11
5	Solver window	12
6	Solver state variables	13
7	Built-in signature	16
8	Mouse and key events	21
9	Trees menu	22
10	Font menu	24
11	Subtrees menu	25
12	Specification menu	31
13	Signature/map menu	32
14	Axioms menu	33
15	Theorems menu	35
16	Graph menu	36
17	Substitution menu	38
18	Further Buttons	39
19	Grammar	45
20	Axioms and theorems	47
21	Derivations	49
22	Variables	51
23	Simplifications	52
24	Examples	62
25	Widgets, graphs, and turtle actions	65

26 Alignments and palindromes	74
27 Dissections and partitions	77

1 Main 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 *add spec*, *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.

add map from	add spec from file 
apply axioms for symbols	apply clause from entry field left to right  - right to left 
apply axioms in text field	apply clause from text field left to right - right to left
apply coinduction	apply fixpoint induction
apply map	apply strong coinduction
apply strong fixpoint induction	apply substitution
apply to variable:	apply transitivity
build/check	build equations 
build graph 	build relation 
build unifier	call enumerator
check proof term 	clear subtrees
collapse	collapse levelwise
combine trees 	coordinates
copy	create Hoare invariant
create induction hypotheses	create subgoal invariant
decompose atom	decrease current 
enclose/replace by entry	expand
flatten (co-)Horn clause	generalize
hide/show 	Horn axioms for copredicates
increase current 	instantiate
invert axioms for symbols	label graph 
label roots	load text from file 
mark	narrow/rewrite 
negate axioms for symbols	numbers
paint	parse up 
parse down 	polarities
positions	redraw 
remove displayed tree	remove entry&label
remove from entry field	remove other trees
remove spec	remove subtrees
remove text	rename
replace by other sides	replace by tree of Solver1/2
reverse	save spec to file
save proof to file 	save proof term to file 

save tree to file 	save tree in eps format to file 
shift factors	shift summands
set	show axioms for symbols 
show changed	show map
show sig	simplify 
split tree 	stretch conclusion
stretch premise	subsume
turn local def into function application	unify
← unify/match	unify with tree of Solver1/2
unlabel graph 	← / →  / 
+1/-1	

Save commands store into files of the *Examples* subdirectory of your home directory. *If this subdirectory does not exist, nothing will be saved!* File parameters of **add** or **load commands** are first looked for in this subdirectory. Those that do not exist there are searched for in the synonymous system directory.

2 Overview

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

The **solver** is accessed via a window for entering, editing and displaying trees or graphs 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 up** and **parse down** 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* [18], 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 [4, 12] 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.

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 **Subtrees 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 successors 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 (see

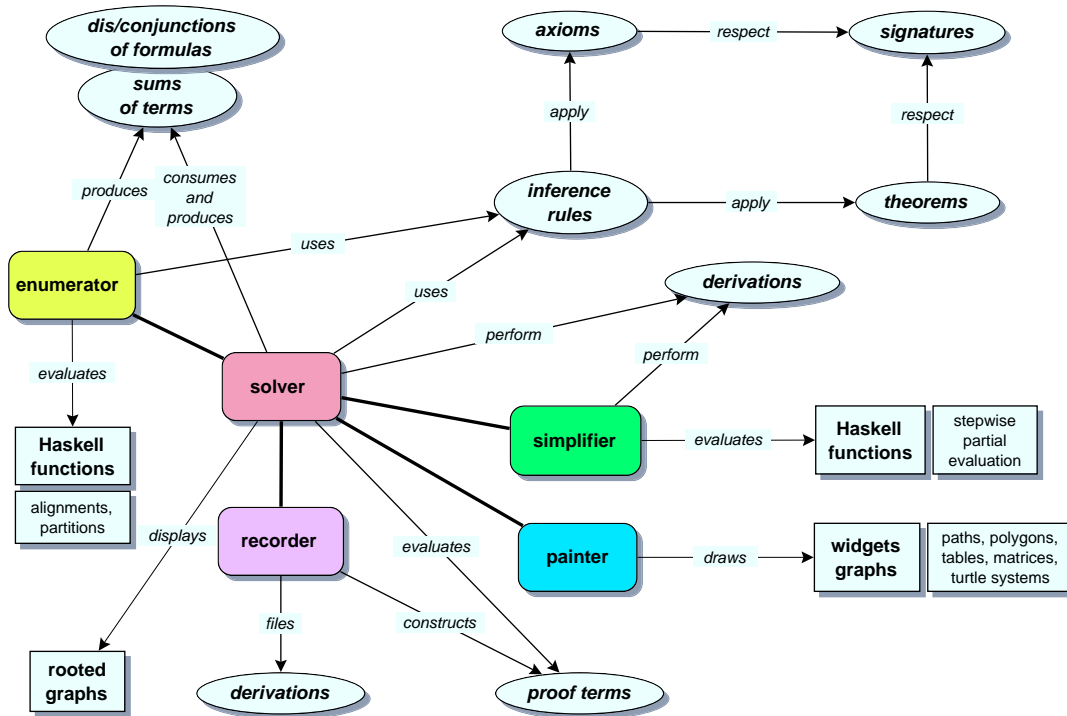


Figure 2.1: Components of Expander2

[Solver state variables](#)). For more details on the syntax and semantics of axioms, theorems and goals, see [Axioms and theorems](#) and [Swinging Types](#).

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 classical relations or deterministic functions, non-deterministic functions (e.g. state transition systems) and "distributed" transition systems like *Maude programs* [3] or algebraic nets [19] 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 [7], OBDD optimization, LR parsing, data flow analysis and global model checking can be carried out and displayed (see [Simplifications](#)).

3 Overall code structure

The code of `Expander2` consists of four O'Haskell modules:

- **Eterm** contains data types and functions for generating, manipulating or checking terms and formulas, such as unification, matching, reduction and expansion of collapsed trees.
- **Epaint** 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 [Widgets, graphs, and turtle actions](#)). 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** encapsulates translators between string, tree and graphical representations of terms and formulas. *Esolve* also contains the **simplifier** that partially evaluates terms and formulas. Moreover, the basic inference rules for applying axioms and theorems are implemented here. *Esolve* also contains the `enumerator` template that provides a GUI for running tree enumeration algorithms (see the sections [Alignments and palindromes](#) and [Dissections and partitions](#)). They are called from the `solver` template, which is part of *Ecom*.
- **Ecom** 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 (see the `paint` button). *Ecom* closes with the main program of the system that creates the main objects, partly in a mutually recursive way:

```
main tk = do
  win1 <- tk.window []
  win2 <- tk.window []
  swing1 <- colorFlasher tk
  swing2 <- colorFlasher tk
  swing3 <- colorFlasher tk
  swing4 <- colorFlasher tk
  fix solve1 <- solver tk "Solver1" win1 solve2 "Solver2" enum1 paint1 swing3
    solve2 <- solver tk "Solver2" win2 solve1 "Solver1" enum2 paint2 swing4
    paint1 <- painter tk swing1 solve1
    paint2 <- painter tk swing2 solve2
    enum1 <- enumerator tk solve1
    enum2 <- enumerator tk solve2
  solve1.configSolve (0,20)
  solve2.configSolve (20,40)
  win2.iconify
```

The `colorFlasher`, `solver`, `painter` and `enumerator` templates make use of the O'Haskell module [Tk.hs](#), which provides the interface to Tcl/Tk (see the [O'Hugs computing environments](#)).

4 Starting the system

Unpack [Ohugs.tar.gz](#) and adapt flags and file paths to your machine. This might apply to CFLAGS and LDFLAGS in *src/Makefile*, the OS flag in *src/prelude.h*, the INSTLIB and INSTBIN paths in *Makefile* and the LD_LIBRARY_PATH in the scripts *ohugs* and *rohugs*. Compile O'Hugs with `make` and `make install`.

The enclosed O'Hugs package contains a special version of the Tk interface module [Tk.hs](#). In contrast to the CVS version, it provides a function that returns the font dependent width of a string. This is needed for the optimal placement of tree node labels. Moreover, three constants in *src/prelude.h* have been increased: `NUM_OFFSETS = 2048`, `NUM_STACK = Pick(1800, 12000, 256000)`, `NUM_ADDRS = Pick(28000, 60000, 640000)`.

Unpack [Expander2.tar.gz](#), call `ohugs -h12000k Ecom` and run `main`. Two solvers and their painters will be created. The window of the first solver is opened. In case of installation problems contact [me](#).

The screenshot shows a window titled "Solver1" containing a tree diagram of a list specification. The root node is "&". It has four children: "=", "<===", "==>", and "===>". Each child node further branches into function names and variables, such as "map", "F", "x", "s", "zipAny", "P", "split", "()", "sort", "merge", "s", "s'", "card", "z", "s'", "z".

Below the tree is a list of axioms:

```

0> map(F) (x:s) = F(x) :map(F) (s)
1>& ( zipAny(P) (x:s) (y:s')
    <=== P(x,y)
      | zipAny(P) (s) (s') )
2>& ( split(s) = (s1,s2)
    ==> sort(x:(y:s)) = merge(sort(x:s1), sort(y:s2)) )
3>& ( s ~ s'
    ==> card(s,z) = card(s',z) )

```

At the bottom is a "SOLVER" control panel with various buttons and sliders. The buttons include "formula", "subtrees", "specification", "signature/map", "axioms", "theorems", "paint", "hide/show (h)", "parse up", "remove text", "parse down", "clear subtrees", "pict type", "graph", "substitution", "font size", "font", "-1", "set", "+1", "narrow (n)", "<-- unify", "simplify (s)", "collapse levelwise", "apply to variable:", "build", and "quit".

A note at the bottom left of the control panel reads: "The entire conjunction is shown above."

Figure 4.1: The solver window shows axioms of a list specification.

5 Solver window

Viewed from top to bottom, a solver window consists of the following widgets:

- a scrollable **canvas**,
- a horizontal **slider** for selecting the tree to be shown on the canvas,
- a scrollable and line-editable **text field**,
- an **entry field** for entering file names, node entries or integers,
- a vertical and a horizontal **slider** for stretching or shrinking the tree horizontally resp. vertically,
- a horizontal **slider** for changing the size of the node label font,
- nine boldface-titled **menus** described below,
- eighteen **framed buttons** described below,
- a vertical **slider** for changing the relative vertical size of the canvas and the text field,
- a **label field** for displaying messages.

6 Solver state variables

- **actions**, **boolFun**, **dissects**, **finals**, **finalsL**, **labels**, **fixPositions**, **matrixU**, **matrixL**, **transitions** and **transitionsL** may occur as function symbols in terms or formulas. Their values are stored by rewriting steps, retrieved and modified by simplification steps and represented pictorially by applying a suitable widget interpreter. The purpose of these state variables is to hide complex function parameters from the screen whose current values are needed by the simplifier needs for evaluating built-in iterative Haskell functions (see [Simplifications](#)).
- The current **axioms** and **theorems** are applied to conjectures/constraints and build up the high- or medium-level steps of a computation or proof. Axioms and theorems are applied by rewriting or narrowing. A narrowing/rewriting step starts with unifying/matching a subtree (the redex) with/against an axiom. Narrowing applies (guarded) Horn or co-Horn clauses, rewriting applies only unconditional, but possibly guarded, equations. The guard of an axiom is a subformula to be solved before the axiom is applied. See also the [Axioms menu](#), [Axioms and theorems](#) and the [narrow/rewrite](#) button.
- **curr** holds the position of the actually displayed tree in the list of current trees.
- **formula** indicates whether the list of current trees represents a disjunction or conjunction of formulas or a sum of terms, respectively. Conditional equations (see [Axioms and theorems](#)) applied to a formula should be valid in the initial model of the underlying swinging type, while conditional equations applied to a term may represent rewrite rules that are not valid equations. The results of the applications of several rewrite rules applied to the same term are combined with $\langle + \rangle$ to a sum of terms (see [Built-in signature](#)).
- The widget interpreter **pictEval** recognizes paintable terms and transforms them into their pictorial representations (see [Widgets, graphs, and turtle actions](#)).
- The current **proof** records the sequence of derivation steps performed since the last initialization of the list of current trees (by [parsing](#) the contents of the text field; see [Derivations](#)). Each element of the current proof consists of a description of a rule application, the resulting list of current trees and the resulting values of *treeMode*, *curr*, *treeposs*, *varCounter*, *solPositions*, *fixPositions*, *substitution* and *subsDom*.
- The current **proof term** represents the current proof as an executable expression for the purpose of later proof checking. It is built up automatically when a derivation is carried out and can be saved to a user-defined file. A saved proof term is loaded by writing its name into the entry field and pushing [check proof term from file](#). This action overwrites the current proof term. The proof represented by the loaded proof term is carried out (and thus checked) on the current tree by pushing the \rightarrow button. Each click triggers a proof step. The proof term is entered into the text field. `POINTER` precedes the rule applied next. If you push the [check](#) button, Expander2 leaves the proof check mode, i.e. the state variables *proof* and *proof term* will be handled as in the (default) proof build mode. This allows you to extend (a prefix of) a stored proof by new rule applications.
- **rule** indicates whether the list of current trees is the result of narrowing steps, rewriting steps,

simplification steps or other rule applications.

- **matching** indicates the current strategy used for narrowing/rewriting (see the [← unify/match](#) button).
- The current **signature** consists of symbols denoting
 - basic specifications (`specs`) consisting of signatures and axioms,
 - predicates (`preds`) interpreted as the least solutions of their (Horn) axioms,
 - copredicates (`copreds`) interpreted as the greatest solutions of their (co-Horn) axioms,
 - constructors (`constructs`) for building up data,
 - defined functions (`defuncts`) specified by (Horn) axioms or implemented as Haskell functions called by the simplifier,
 - coinductive functions (`cofuncts`) specified by (Horn) axioms,
 - first-order variables (`fovars`) that may be instantiated by terms or formulas,
 - higher-order variables (`hovars`) that may be instantiated by functions or (co)predicates.

For more details, see [Built-in signature](#).

- The current **signature map** is a signature morphism from the current signature to the current signature of the other solver. It is initialized as the identity map on strings. Example ([STACK2IMPL](#)):

$$\begin{aligned} &just \rightarrow entry \\ &= \rightarrow \sim \end{aligned}$$

- The list **solPositions** consists of the positions of [solved formulas](#) resp. [normal forms](#) among the current trees.
- The current **substitution** maps the variables of its domain (= actual value of **subsDom**) to terms over the current signature. It is generated, modified and applied by particular buttons (see the [Substitution menu](#) and the [apply to variable:](#) button).
- **treeMode** indicates whether the list **trees** of current trees (or other rooted graphs) is a singleton (`treeMode=tree`) or represents a disjunction of formulas (`treeMode=summand`), a conjunction of formulas (`treeMode=factor`) or a sum (= disjoint union) of terms (`treeMode=term`). *True*, *False* and *()* is the respective zero element (see [Built-in signature](#)). The label of the [trees menu](#) shows the actual tree mode: If `treeMode=tree`, then the label is *term* resp. *formula*. If `treeMode=summand/factor`, then the label tells us how many summands resp. factors the set of current trees consists of. The slider between the canvas and the text field of a solver window allows one to browse among the current trees and to select the one to be displayed on the canvas. For the commands that may change *trees*, see the [Trees menu](#), the [Subtrees menu](#) and the [Graph menu](#).
- The list **treepos** consists of the positions of selected subtrees of the actually displayed tree. Subtrees are selected (and moved) by pushing the left mouse button while placing the cursor over their roots (see [Mouse and key events](#)).

- **varCounter** maps a variable x to the maximal index i such that x_i occurs in the current proof. *varCounter* is updated when new variables are needed.

After each rule application the current proof term is entered into the text field such that the constant `POINTER` precedes the command that will be executed next. If a new step has been performed during the proof check, the proof term is adapted accordingly, i.e., the rest of the proof to be checked is replaced by the new step. The current proof term is set to [] whenever the contents of the text field is **parsed** and thus turned into a new list of current trees.

7 Built-in signature

The built-in signature reads as follows:

preds: <= >= < > » ~/~ ~ ~ == \$ any `disjoint` `gives` `in` Int `not_in` null
Real `shares` zipAny INV

copreds: ~ \$ all zipAll

constructs: <+> () [] {} : 0 bool fun lin pile piles suc

defuncts: + ++ - * ** / ^ bag bisim count dnf drop foldl head init last length
max `meet` min minimize `mod` nerode obdd optimize parse permute postflow
product range reverse sat set stateflow subsflow sum tail take get0 get1
get2... upd0 upd1 upd2... x0 x1 x2...

fovars: i z

hovars:

<=, >=, <, > are predefined on integers, reals, strings and the defined functions x_0, x_1, x_2, \dots (used in OBDDs; see below and [Grammar](#)). <=, `in` and `not_in` denote the subset, membership and non-membership relations, respectively, on **collections**, i.e. terms $C(t_1, \dots, t_n)$ where C is one of the constructors `[]` or `{}` or $C(t_1, \dots, t_n) = t_1 \wedge \dots \wedge t_n$. `in` treats a triple $(i, x, [i_1, \dots, i_n])$ where i, i_1, \dots, i_n are integers and x is any string as the list $[(i, x, i_1), \dots, (i, x, i_n)]$. `gives` denotes the inverse of `in`, but is simplified differently (see [Simplifications](#)).

Int(t) and Real(t) return *True* if t is an integer or real number, respectively. Int(t) and Real(t) return *False* if t is a real or an integer number, respectively.

any(P) (as) return *True* if as contains an element that satisfies P . all(P) (as) return *True* if all elements of as satisfy P . zipAny(P) ($[a_1, \dots, a_n]$) ($[b_1, \dots, b_n]$) return *True* if for some $1 \leq i \leq n$, (a_i, b_i) satisfies P . zipAll(P) ($[a_1, \dots, a_n]$) ($[b_1, \dots, b_n]$) return *True* if for all $1 \leq i \leq n$, (a_i, b_i) satisfies P .

Formulas involving » or INV are generated whenever an induction hypothesis or a (Hoare or subgoal) invariant is created (see [Subtrees menu](#)). For the use of \$, see [enclose/replace by entry](#).

The infix constructor <+> builds **sums** of terms (see [Solver state variables](#)). The simplifier transforms a term of the form $f(\dots, t_1 \langle + \rangle \dots \langle + \rangle t_n, \dots)$ into the sum $f(\dots, t_1, \dots) \langle + \rangle \dots \langle + \rangle f(\dots, t_n, \dots)$ and an atom of the form $p(\dots, t_1 \langle + \rangle \dots \langle + \rangle t_n, \dots)$ into the disjunction $p(\dots, t_1, \dots) \mid \dots \mid p(\dots, t_n, \dots)$. Moreover, given normal forms p_1, \dots, p_n , fun(p_1, t_1) <+> ... <+> fun(p_n, t_n) stands for the abstraction $\lambda p_1.t_1 \mid \dots \mid \lambda p_n.t_n$.

() , [] and {} denote product, list resp. set constructors of arbitrary finite arity. As a nullary constructor, () denotes "undefined" and is neutral with respect to the sum constructor <+>. The sim-

plifier transforms terms containing `()` into `()` and atoms containing `()` into `False`. A term of the form $f((t_1, \dots, t_n))$ is identified with $f(t_1, \dots, t_n)$ provided that f is not a collector. Accordingly, for a variable x , $f(x)$ unifies with $f(t_1, \dots, t_n)$. The constructor `:` appends an element to a list from the left. `0` and `suc` are the natural number constructors. If applied to a number list s , `suc` returns the next permutation of s in reverse lexicographic order. If s is sorted, then $\text{suc}(s) = \text{reverse}(s)$.

`bool` and `lin` embed formulas into terms (see [Graph menu](#)). `pile` and `piles` are widget constructors (see [Widgets, graphs, and turtle actions](#)).

`+`, `-`, `*`, `**`, `/`, ``mod``, `max`, `min` are defined on integer and real numbers. `+`, `-`, `*`, `/` work also for polynomials (`*` and `/` only as scalar operators).

Given finite lists or sets s, s' and integers m, n , $s - s'$ and `range(m, n)` denote the list of elements of s that are not in s' and the interval $[m..n]$ of integers, respectively. Haskell shortcuts of integer lists like $[k, m..n]$ may also be used. `++`, `drop`, `foldl`, `head`, `init`, `last`, `length`, `map`, `null`, `product`, `sum`, `tail`, `take`, `reverse`, `zip` and `zipWith` are defined on collections as the synonymous Haskell functions are defined on lists. Moreover, `length(t1, ..., tn)` is simplified to n .

`any`, `all`, `map`, `foldl`, `zip`, `zipAny`, `zipAll` and `zipWith` also occur in [LIST](#) and [LISTEVAL](#) with (recursive) axioms. The synonymous built-in symbols are interpreted as partial non-recursive functions. For instance, a rewriting step via [LIST](#) transforms the term $\text{map}(\text{suc})(x : s)$ into $x : \text{map}(\text{suc})(s)$, while the simplifier does not modify this term, but would turn $\text{map}(\text{suc})[x, y, z]$ into $[\text{suc}(x), \text{suc}(y), \text{suc}(z)]$. Of course, axioms introduced for built-in symbols should comply with their built-in interpretation that is realized by the simplifier.

`map`, `zip` and `zipWith` do not occur in the above list of built-in symbols because [LIST](#) treats them as defined functions, while [STREAM](#) declares them as constructors.

`count(ts, t)` counts the number of occurrences of t in the list ts . $ts \text{ `disjoint` } us$ checks whether the lists ts and us are disjoint. $ts \text{ `meet` } us$ computes the intersection of the lists ts and us . $ts \text{ `shares` } us$ checks whether the lists ts and us are not disjoint.

As in Haskell, `$` denotes the apply operator whose first argument is a higher-order term t that represents a predicate or function f . The other arguments of `$` are the arguments of f , i.e. $\$(t, t_1, \dots, t_n)$ stands for $t(t_1, \dots, t_n)$.

`get0`, `get1`, `get2`, ... and `upd0`, `upd1`, `upd2`, ... return resp. update the first, second, third, ... component of a tuple or element of a collection. `x0`, `x1`, `x2`, ... are used—besides 0 and 1—as node labels of OBDDs.

`bag` transforms a list into a bag and flattens terms built up with the infix operator `^` (see below). `set` transforms a list or bag into a set. Many functions defined on lists are also defined on other collections. If `count`, ``disjoint``, ``in``, ``meet``, ``not_in``, ``shares``, `++`, `-` and `<=` are applied to such collections where their evaluation needs an equality on the elements of the collections, the simplifier transforms them only if they consist of ground constructor terms!

`obdd` transforms a DNF represented as a list of strings of the same positive length whose characters are 0, 1 or # into an equivalent minimal OBDD. `dnf` transforms OBDDs into equivalent minimal DNFs. If applied to a DNF, `minimize` minimizes the number of summands of a DNF. If applied to an OBDD, `minimize` minimizes the number of nodes of an OBDD according to the two reduction rules for OBDDs [\[17\]](#).

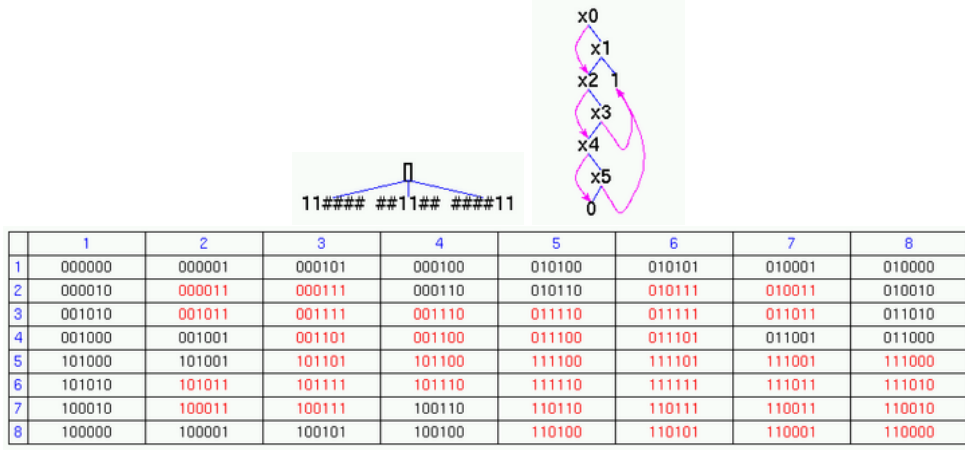


Figure 7.1: A DNF (QUAD2), its minimal OBDD and its Karnaugh diagram

\wedge is an infix operator for building bags and treated by the unification algorithm as an associative and commutative function. When a **bag term** $t_1 \wedge \dots \wedge t_n$ in the displayed tree is to be unified with another bag term u , then the unification succeeds even if only a permutation of $t_1 \wedge \dots \wedge t_n$ unifies with u . If there are several unifiers, those are preferred, which substitute only variables for variables. Among these unifiers those are preferred, which substitute variables only for variables of u .

Axioms of the form

$$\{guard \Rightarrow\} (t_1 \wedge \dots \wedge t_k \wedge t_{k+1} \wedge \dots \wedge t_n = t \quad \{\Leftarrow prem\}) \quad (*)$$

are called **AC equations** because they take into account that \wedge is an associative-commutative function. (*) can be applied to a bag term $u_1 \wedge \dots \wedge u_m$ if a list $L_1 = [u_{i_1}, \dots, u_{i_m}]$ of elements of $L = [u_1, \dots, u_m]$ unifies with $[t_1, \dots, t_n]$ and if the unifier satisfies the guard. At first, a substitution f that unifies $L' = [t_1, \dots, t_k]$ with members of L is looked for. Then f must be extendable to a substitution g that satisfies the guard and unifies $[t_{k+1}, \dots, t_m]$ with a permutation of the list $L_2 = [v_1, \dots, v_{n-k}]$ that consists of all elements of L , which were not unified with elements of L' . The search is performed by traversing the permutations of L_2 in reverse lexicographic order. If a suitable permutation has been found, the elements of L_1 are replaced by the instance of t by g , while the remaining elements of L are replaced with their instances by g . At most 720 permutations of L_2 are checked. If this case is reached without achieving a unifier, then the application of (*) consists of replacing L_2 by the permutation achieved at last. Further permutations may then be tried by re-applying the AC equation.

For instance, repeated applications of the AC equation (see MOD)

$$i \text{ 'mod' } j = 0 \Rightarrow i \wedge j = j$$

(see PRIMS) to

$$2 \wedge 3 \wedge 4 \wedge 5 \wedge 6 \wedge 7 \wedge 8 \wedge 9 \wedge 10 \wedge 11 \wedge 12 \wedge 13 \wedge 14 \wedge 15,$$

sift out the primes and thus end up with

$$2 \wedge 3 \wedge 5 \wedge 7 \wedge 11 \wedge 13.$$

ACCOUNT, HANOI, PUZZLE and the algebraic net specifications PHIL and ECHO also contain AC equations. In contrast to the pure AC unification of bag terms, the AC equation (*) may be applicable to a bag term $u_1 \wedge \dots \wedge u_n$ even if n is greater than $k + m$.

Set brackets used in clauses enclose optional subformulas, i.e. guard and prem in (*) may be empty.

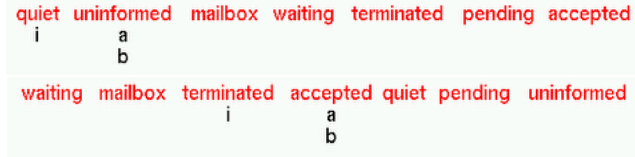


Figure 7.2: Snapshots of a run of the echo algorithm (cf. [19])

The operators $\&$, $|$, $=$, $==$, \neq , \sim , $\sim\sim$, \sim/\sim , $+$, $*$, $\hat{}$, $\{ \}$, $\langle + \rangle$ are interpreted as *permutators*, i.e., the order of their arguments is irrelevant. Consequently, AC unification as described above for $\hat{}$ replaces ordinary unification whenever two argument lists of a permutator are to be unified.

The operators $\hat{}$, $\langle + \rangle$, $++$ are interpreted as associative operators and thus may have an arbitrary finite number of arguments. The parser and the simplifier flatten compositions of the same associative operator. For instance, $t_1 \hat{} ((t_2 \hat{} t_3) \hat{} t_4)$ is turned into a tree with root label $\hat{}$ and four maximal proper subtrees: t_1, t_2, t_3, t_4 . While two terms t_1 and t_2 with root label $\hat{}$ may unify with each other even if the number of maximal proper subtrees of t_1 resp. t_2 is different (see above), this does not hold true for $\langle + \rangle$ and $++$ instead of $\hat{}$. Hence axioms for these operators could not be applied to terms with arities of $\langle + \rangle$ and $++$ that are different from the corresponding arities in the axioms. But such axioms are not needed because the parser and the simplifier already reduce $\langle + \rangle$ - and $++$ -rooted terms to their minimal representations according to the usual meaning of $\langle + \rangle$ and $++$. This includes the idempotency of $\langle + \rangle$, the neutrality of empty collections with respect to $++$ and the axiom $(x : s) ++ s' = x : (s ++ s')$.

\sim , $\sim\sim$ and $==$ are supposed to be congruence relations. \sim is declared as a copredicate because it denotes *behavioral equivalence*. Axioms for the predicate $\sim\sim$ are introduced and used when clauses of the form $prem \Rightarrow t \sim u$ are proved by **coinduction**.

\langle, \rangle , \neq , \sim/\sim are the complements of \geq , \leq , $=$, \sim , respectively. For each other predicate or copredicate P , not_P denotes the complement of P . Axioms for the complement of P are added to the set of current axioms if P is entered into the entry field and the button **negate axioms for symbol** is pushed.

Subformulas involving built-in functions or predicates are (partially) evaluated when the displayed tree is simplified. This includes the stepwise execution of built-in functions with state variable parameters (see **Simplifications**).

Here is a user-defined signature (the one of **OBDD**):

defuncts: restrict forall exists quantor x X Y F and or not

fovvars: u2 u1 u t2 t1 t j i b

hovvars: F{and,or} X{x} Y{x}

$F\{and,or\}$ denotes that the defined functions and and or are the only admissible instances of the higher-order variable F . In general, the list following a higher-order variable F must consist of characters. All terms starting with one of the characters is an admissible instance of F . If F is not followed by a list of characters, all terms are admissible.

Keywords (**specs:**, **preds:**, **copreds:**, **constructs:**, **defuncts:**, **cofuncts:**, **fovvars:** and **hovvars:**) may appear at any place in the list of symbols that builds up a signature. To be recognized as keywords they must be separated from their context by blanks.

7 Built-in signature

User-defined signatures automatically inherit the built-in signature. Symbols that are to be interpreted as infix operators must start and end with the character ``` or consist of characters among

`: + - * < = ~ > / ^ #`

(see [Grammar](#)).

Symbols used in axioms, theorems or conjectures that do not belong to the current signature are interpreted as (undefined) function symbols (see [Grammar](#)). This facilitates certain applications, but may also lead to unexpected unification failures when axioms or theorems shall be applied. To exclude misinterpretations of symbols one should use the respective buttons for showing the current signature, axioms or theorems or even [parse up](#) the latter.

8 Mouse and key events

A subtree t is selected (or deselected if it has already been selected) by clicking on the **left mouse button** while placing the cursor over its root. If the mouse is moved while the button is pressed, t is shifted over the canvas. If the button is released while the root of t is placed over the root of another subtree u , u is replaced by t . If u is an existentially (resp. universally) quantified variable and the scope of u has positive (resp. negative) polarity, then all occurrences of u within the scope are replaced by t (see [Derivations](#)). Replaced subterms are colored in blue.

Subtrees are deselected backwards with respect to the order in which they were selected by moving the cursor away from the displayed tree and pushing the left mouse button. All selected subtrees are deselected simultaneously if the **clear subtrees** button is pushed. If you stop moving a subtree before inserting it into the displayed tree, it will stay at the place where you released the mouse button. Then push the **redraw** button and the subtree will be returned to its previous place within the displayed tree.






If a subtree t has been selected and the move of t is started with a click on the **middle mouse button**, t will be removed from the displayed tree and replaced by the variable z_n where the index n is increased each time a subtree is removed or a new variable is needed when an axiom is flattened (see below). Moreover, the current substitution is extended by the assignment of t to z_n .

By moving the mouse and pushing the middle button outside the root of a selected subtree the entire displayed tree is shifted over the canvas. By pressing the **right mouse button** while placing the cursor over a node x a **pointer** (edge) from x to the root of the last selected subtree t is drawn and all successors of x are removed. The arc is colored in orange if it closes a circle consisting of edges of t . Otherwise the arc is colored in magenta. Subtree replacements and substitutions for variables adapt the pointer values.

A command followed by a letter in round brackets is executed when the key with the letter is pressed after the cursor has been placed over the label field and the left mouse button has been pushed (see [Commands](#)).

9 Trees menu

The commands of the trees menu create or transform the current trees or the current proof.

- **call enumerator** opens a submenu listing tree enumeration algorithms. When you push the button for one of these algorithms, you will be prompted to enter sequences of strings (in the case of the alignment or palindrome enumerator), numbers (in the case of the dissection enumerator) or the length of a list (in the case of the partition enumerator) and certain constraints (see the sections [Alignments and palindromes](#) and [Dissections and partitions](#)). After the "go" button has been pushed, the resulting trees are assigned to Solver1/2 and may be browsed through with the canvas slider.
- **split tree**  decomposes a conjunction, disjunction or sum into its factors, summands and terms, respectively, provided that the list of current trees is a singleton
- **combine trees**  combines all current trees into a disjunction, conjunction or sum, respectively.
- **remove other trees** eliminates all current trees except the current one.
- **remove displayed tree** eliminates the displayed tree.
- **show changed** selects all maximal elements within the set of subtrees that have been modified during the last transformation of the displayed tree.
- **show proof** enters the current proof into the text field.
- **show proof in text field of Solver1/2** opens Solver1/2 and enters the current proof into its text field.
- **save proof to file**  saves the current proof to the file in the entry field.
- **show proof term** enters the current proof term into the text field.
- **show proof term in text field of Solver1/2** opens Solver1/2 and enters the current proof term into its text field.
- **save proof term to file**  saves the current proof term to the file in the entry field.
- **check proof term from file**  assigns the contents of the file in the entry field to the current proof term provided that the contents is a proof term.
- **check proof term from text field** assigns the contents of the text field to the current proof term provided that the contents is a proof term.
- **create induction hypotheses** prepares the displayed tree cl for a proof by Noetherian induction. The command assumes that cl is a formula and that free or universal *induction* variables x_1, \dots, x_n

of cl have been selected, which will be decorated with an exclamation mark. Non-selected free variables are turned into universal variables. If cl has the form $prem \Rightarrow conc$, then the clauses

$$\begin{aligned} conc' &\Leftarrow (x_1, \dots, x_n) \gg (x'_1, \dots, x'_n) \quad \& \quad prem' \\ prem' &\Rightarrow ((x_1, \dots, x_n) \gg (x'_1, \dots, x'_n) \Rightarrow conc') \end{aligned}$$

are added to the set of current theorems. The primed formulas are obtained from the unprimed ones by replacing x_i with x'_i , $1 \leq i \leq n$. If cl is not an implication, then

$$cl' \Leftarrow (x_1, \dots, x_n) \gg (x'_1, \dots, x'_n)$$

is added to the set of current theorems.

- **flatten (co-)Horn clause** assumes that the displayed tree is a Horn or co-Horn clause cl (see [Axioms and theorems](#)). If subterms t_1, \dots, t_n of cl are selected and F is the set of roots of t_1, \dots, t_n , then cl is replaced by an equivalent formula where each $f \in F$ occurs only at the outermost position of the left- or right-hand side of an equation. If no subterms are selected, F is the set of all defined functions of the current signature. For instance, the [LISTEVAL](#)-axiom

$$sort(x : (y : s)) = merge(sort(x : s_1), sort(y : s_2)) \Leftarrow split(s) = (s_1, s_2)$$

is turned into

$$\begin{aligned} sort(x : (y : s)) &= merge(z_0, z_1) \\ &\Leftarrow split(s) = (s_1, s_2) \& sort(x : s_1) = z_0 \& sort(x : s_2) = z_1 \end{aligned}$$



if $F = \{sort\}$ and into

$$\begin{aligned} sort(x : (y : s)) &= z_0 \\ &\Leftarrow split(s) = (s_1, s_2) \quad \& \quad merge(z_1, z_2) = z_0 \\ &\quad \& \quad sort(x : s_1) = z_1 \quad \& \quad sort(y : s_2) = z_2 \end{aligned}$$

if $F = \{sort, merge\}$.

- **turn local def into function application** takes the last local definition (= equation with a normal form on the right-hand side) in the displayed tree, which is supposed to be a conditional equation, and transforms it into an equivalent function application. For instance,

$$t = u \Leftarrow prem \quad \& \quad v = nf \quad \text{becomes} \quad t = fun(nf, u)(v) \Leftarrow prem.$$

- **save tree to file**  saves the string representation of the displayed tree to the file in the entry field.
- **save tree in eps format to file**  saves the current tree in Encapsulated PostScript format to the file in the entry field.
- **save trees to file** saves the disjunction, conjunction or sum, respectively, of the current trees to the file in the entry field.
- **load text from** opens a submenu of files. The term or formula in the selected file is entered into the text field. Signature elements declared in the file are added to the current signature. In this case, the formula must be separated from the term resp. formula by the keyword `conjectures:`. Files containing only signature elements are not allowed here.

10 Font menu

consists of buttons for choosing the font to be used for the text in tree nodes and pictorial term representations. The font size is controlled by a slider (see [Solver window](#)).

11 Subtrees menu

The commands of this menu transform the subtrees that were selected with the left mouse button. If no subtree has been selected, the entire displayed tree is regarded as being selected. Most commands call inference rules and deliver messages that tell us whether or not the executed rule application is sound with respect to the initial model induced by the current signature and axioms (see [Derivations](#)).

- **copy** adds a copy of the subtree selected at last to the children of its parent node.
- **remove** removes all selected subtrees if they are summands/factors of the same disjunction/conjunction with positive/negative polarity. Otherwise the greatest lower bound of the selected subtrees is removed.
- **generalize** combines the last selected subformula cl of the displayed tree with the formula cl' in the entry field. If cl has positive polarity, then $cl \& cl'$ replaces cl . Otherwise $cl \mid cl'$ replaces cl . A generalization of cl may be necessary before cl can be proved by Noetherian induction, fixpoint induction or coinduction.
- **instantiate** assumes the selection of a quantified variable x . If x is existential resp. universal and the scope of x has positive resp. negative polarity (see [Derivations](#)), then all occurrences of x are replaced by the term in the entry field. The replaced variables are colored in blue.
- **subsume** assumes the selection of the premise t and the conclusion u of an implication or two factors t and u of a conjunction or two summands t and u of a disjunction. If t subsumes u , then $t \Rightarrow u$ is replaced by *True* or u is removed from the conjunction or t is removed from the disjunction, respectively.
- **unify** assumes the selection of two factors or summands t and u of a conjunction resp. disjunction. If t and u are unifiable and the unifier instantiates only existential resp. universal variables of the conjunction resp. disjunction, then t is removed and the unifier is applied to the remaining conjunction resp. disjunction.
- **reverse** reverses the list of at least two selected subtrees. The reduct implies the redex if the subtrees have the same direct predecessor x and if x is a *permutator* (see [Built-in signature](#)). If only one subtree t is selected, then the operation is applied to the list of maximal proper subtrees of t .
- **decompose atom** assumes the selection of an atom $t R t'$ with positive polarity such that R is among $=, \sim$ or an atom $t R t'$ with negative polarity such that R is among $\neq, \not\sim$. The selected atom is decomposed in accordance with the assumption that R is compatible with function symbols (see [Built-in signature](#)).
- **apply transitivity** assumes the selection of an atom $t R t'$ with positive polarity or factors $t_1 R t_2, t_2 R t_3, \dots, t_{n-1} R t_n$ of a conjunction with negative polarity (see [Derivations](#)) such that R is among $=, \sim, \sim\sim, \leq, \geq, <, >$. The selected atoms are changed in accordance with the assumption that R is transitive (see [Built-in signature](#)).

- **apply clause from entry field** applies the n -th clause cl in the text field to all selected subtrees provided that the entry field contains the number n . cl may be applied from left to right $\boxed{\text{a}}$ or from right to left $\boxed{\text{b}}$ where the direction refers to the left resp. right argument of the clause's leading symbol ($=$, \Rightarrow or \Leftarrow). If cl is distributed (see [Axioms and theorems](#)), then cl 's atoms must unify componentwise with the selected subtrees. Otherwise cl is applied sequentially to each selected subtree.
- **... and save redex** adds the redex disjunctively/conjunctively to the reduct if the clause is a non-distributed Horn/co-Horn clause. The correctness of this version of the rule does not depend on the polarity of the redex.
- **apply clause from text field** applies the clause in the text field analogously to *apply clause from entry field*.
- **shift factors** shifts to the conclusion of an implication $prem \Rightarrow conc$ all selected factors of the premise (see [10, section 5]). This may be necessary before $prem \Rightarrow conc$ is submitted to a proof by fixpoint induction.
- **shift summands** shifts to the premise of an implication $prem \Rightarrow conc$ all selected summands of the conclusion (see [10, section 5]). This may be necessary before $prem \Rightarrow conc$ is submitted to a proof by coinduction.
- **stretch premise** assumes the selection of a formula of the form (B), (C) or (D). The formula is turned into the corresponding co-Horn clause of the form (B'), (C') resp. (D').
- **stretch conclusion** assumes the selection of a formula of the form (A). The formula is turned into the corresponding Horn clause of the form (A').
- **apply coinduction** assumes the selection of conjectures

$$\begin{aligned} & \{prem_1 \Rightarrow\} p(t_{1_1}, \dots, t_{1_n}) \\ & \& \dots \\ & \& \{prem_k \Rightarrow\} p(t_{k_1}, \dots, t_{k_n}) \end{aligned} \tag{A}$$

about a copredicate p that does not depend on any predicate or function occurring in $prem_i$. (A) is turned into

$$\begin{aligned} p(x_1, \dots, x_n) \Leftarrow & \{prem_1 \&\} x_1 = t_{1_1} \& \dots \& x_n = t_{1_n} \\ & | \dots \\ & |\{prem_k \&\} x_1 = t_{k_1} \& \dots \& x_n = t_{k_n}. \end{aligned} \tag{A'}$$

x_1, \dots, x_n are variables. In fact, only those terms among t_{i_1}, \dots, t_{i_n} are replaced by variables that are not variables or occur more than once among t_{i_1}, \dots, t_{i_n} . Moreover, a new predicate p' is added to the current signature and

$$\begin{aligned} p'(x_1, \dots, x_n) \Leftarrow & \{prem_1 \&\} x_1 = t_{1_1} \& \dots \& x_n = t_{1_n} \\ & | \dots \\ & |\{prem_k \&\} x_1 = t_{k_1} \& \dots \& x_n = t_{k_n} \end{aligned} \tag{*}$$

becomes the axiom for p' . All occurrences of p in the axioms for p are replaced by p' . Then (*) is applied to all occurrences of p' in the transformed axioms for p . The conjunction of the clauses resulting from these applications replaces the original conjecture (A).

The primed version of the copredicate \sim (see **Built-in signature**) is denoted by $\sim\sim$ and regarded as an equivalence relation. This takes into account the equivalence closure involved in the coinduction rule for \sim (see, e.g., [16, Section 9.2]).

If $n > 1$ conjectures of the form (A) have been selected, they are assumed to be factors of the same conjunction and deal with different copredicates p_1, \dots, p_n . The n stretched versions of the form (A') are applied to the axioms for p_1, \dots, p_n .

- **apply strong coinduction** (see [6, 16]) also assumes the selection of conjectures (A) and turns them into (A'). Moreover, a new predicate p' is added to the current signature and

$$\begin{aligned} p'(x_1, \dots, x_n) &\Leftarrow \{prem_1 \& x_1 = t_{1_1} \& \dots \& x_n = t_{1_n} \\ &| \dots \end{aligned} \tag{*}$$

$$\begin{aligned} &\{prem_k \& x_1 = t_{k_1} \& \dots \& x_n = t_{k_n}, \\ p'(x_1, \dots, x_n) &\Leftarrow p(x_1, \dots, x_n) \end{aligned} \tag{**}$$

become the axioms for p' . Each axiom $p(t_1, \dots, t_n) \implies conc$ for p is replaced by

$$p(t_1, \dots, t_n) \implies conc \mid p(t_1, \dots, t_n).$$

All occurrences of p in the original part of the extended axioms for p are replaced by p' . Then (*) and (**) are applied to all occurrences of p' in the transformed axioms for p . The conjunction of the clauses resulting from these applications replaces the original conjecture (A).

- **apply fixpoint induction** assumes the selection of conjectures

$$\begin{aligned} p(t_{1_1}, \dots, t_{1_n}) &\implies conc_1 \\ &\& \dots \\ &\& p(t_{k_1}, \dots, t_{k_n}) \implies conc_k \end{aligned} \tag{B}$$

about a predicate p that does not depend on any predicate or function occurring in $conc_i$ or a conjecture of the form

$$\begin{aligned} f(t_{1_1}, \dots, t_{1_n}) = t_1 &\implies conc_1 \\ &\& \dots \\ &\& f(t_{k_1}, \dots, t_{k_n}) = t_k \implies conc_k \end{aligned} \tag{C}$$

or

$$\begin{aligned} f(t_{1_1}, \dots, t_{1_n}) = t_1 &\{ \& conc_1 \} \\ &\& \dots \\ &\& f(t_{k_1}, \dots, t_{k_n}) = t_k \{ \& conc_k \} \end{aligned} \tag{D}$$

about a defined function f that does not depend on any predicate or function occurring in t_i or $conc_i$. (B) is turned into

$$\begin{aligned} p(x_1, \dots, x_n) &\implies (x_1 = t_{1_1} \& \dots \& x_n = t_{1_n} \implies conc_1) \\ &\& \dots \\ &\& (x_1 = t_{k_1} \& \dots \& x_n = t_{k_n} \implies conc_k). \end{aligned} \tag{B'}$$

(C) is turned into

$$\begin{aligned} f(x_1, \dots, x_n) = x &\implies (x_1 = t_{1_1} \& \dots \& x_n = t_{1_n} \& x = t_1 \implies conc_1) \\ &\& \dots \\ &\& (x_1 = t_{k_1} \& \dots \& x_n = t_{k_n} \& x = t_k \implies conc_k). \end{aligned} \tag{C'}$$

(D) is turned into

$$\begin{aligned} f(x_1, \dots, x_n) = x \implies & (x_1 = t_{1_1} \& \dots \& x_n = t_{1_n} \Rightarrow x = t_1 \{ \& conc_1 \}) \\ & \& \dots \\ & \& (x_1 = t_{k_1} \& \dots \& x_n = t_{k_n} \Rightarrow x = t_k \{ \& conc_k \}). \end{aligned} \quad (D')$$

x_1, \dots, x_n, x are variables. In fact, only those terms among t_{i_1}, \dots, t_{i_n} are replaced by variables that are not variables or occur more than once among t_{i_1}, \dots, t_{i_n} . Moreover, a new predicate p' resp. f' is added to the current signature and

$$\begin{aligned} p'(x_1, \dots, x_n) \implies & (x_1 = t_{1_1} \& \dots \& x_n = t_{1_n} \Rightarrow conc_1) \\ & \& \dots \\ & \& (x_1 = t_{k_1} \& \dots \& x_n = t_{k_n} \Rightarrow conc_k) \end{aligned} \quad (*)$$

resp.

$$\begin{aligned} f'(x_1, \dots, x_n, x) \implies & (x_1 = t_{1_1} \& \dots \& x_n = t_{1_n} \& x = t_1 \Rightarrow conc_1) \\ & \& \dots \\ & \& (x_1 = t_{k_1} \& \dots \& x_n = t_{k_n} \& x = t_k \Rightarrow conc_k) \end{aligned} \quad (*)$$

resp.

$$\begin{aligned} f'(x_1, \dots, x_n, x) \implies & (x_1 = t_{1_1} \& \dots \& x_n = t_{1_n} \Rightarrow x = t_1 \{ \& conc_1 \}) \\ & \& \dots \\ & \& (x_1 = t_{k_1} \& \dots \& x_n = t_{k_n} \Rightarrow x = t_k \{ \& conc_k \}). \end{aligned} \quad (*)$$

becomes the axiom for p'/f' . All occurrences of p/f in the (flattened) axioms for p/f are replaced by p'/f' . Replacing f actually means replacing equations $f(t) = u$ by logical atoms $f'(t, u)$. Then (*) is applied to all occurrences of p'/f' in the transformed axioms for p/f . The conjunction of the clauses resulting from these applications replaces the original conjecture (B)/(C)/(D)

If $n > 1$ conjectures of the form (B)/(C)/(D) have been selected, they are assumed to be factors of the same conjunction and deal with different predicates or defined functions p_1, \dots, p_n . The n stretched versions of the form (B')/(C')/(D') will be applied to the (flattened) axioms for p_1, \dots, p_n .

- **apply strong fixpoint induction** (see [6, 16]) also assumes the selection of conjectures (B), (C) or (D) and turns them into (B')/(C')/(D'). Moreover, a new predicate p' resp. f' is added to the current signature and

$$\begin{aligned} p'(x_1, \dots, x_n) \implies & (x_1 = t_{1_1} \& \dots \& x_n = t_{1_n} \Rightarrow conc_1) \\ & \& \dots \\ & \& (x_1 = t_{k_1} \& \dots \& x_n = t_{k_n} \Rightarrow conc_k), \\ p'(x_1, \dots, x_n) \implies & p(x_1, \dots, x_n) \end{aligned} \quad \begin{array}{l} (*) \\ (**) \end{array}$$

resp.

$$\begin{aligned} f'(x_1, \dots, x_n, x) \implies & (x_1 = t_{1_1} \& \dots \& x_n = t_{1_n} \& x = t_1 \Rightarrow conc_1) \\ & \& \dots \\ & \& (x_1 = t_{k_1} \& \dots \& x_n = t_{k_n} \& x = t_k \Rightarrow conc_k), \\ f'(x_1, \dots, x_n, x) \implies & f(x_1, \dots, x_n) = x \end{aligned} \quad \begin{array}{l} (*) \\ (**) \end{array}$$

resp.

$$f'(x_1, \dots, x_n, x) \implies (x_1 = t_{1_1} \& \dots \& x_n = t_{1_n} \Rightarrow x = t_1 \{ \& conc_1 \})$$

$$\& \dots \tag{*}$$

$$\& (x_1 = t_{k_1} \& \dots \& x_n = t_{k_n} \Rightarrow x = t_k \{ \& conc_k \}),$$

$$f'(x_1, \dots, x_n, x) \implies f(x_1, \dots, x_n) = x \tag{**}$$

become the axioms for p'/f' . Each axiom $p(t_1, \dots, t_n) \Leftarrow prem$ for p is replaced by

$$p(t_1, \dots, t_n) \Leftarrow prem \& p(t_1, \dots, t_n).$$

Each axiom $f(t_1, \dots, t_n) = t \Leftarrow prem$ for f is replaced by

$$f(t_1, \dots, t_n) = t \Leftarrow prem \& f(t_1, \dots, t_n) = t.$$

All occurrences of p/f in the original part of the (flattened) axioms for p/f are replaced by p'/f' . Replacing f actually means replacing equations $f(t) = u$ by logical atoms $f'(t, u)$. Then (*) and (**) are applied to all occurrences of p'/f' in the transformed axioms for p/f . The conjunction of the clauses resulting from these applications replaces the original conjecture (B)/(C)/(D).

- **create Hoare invariant** assumes the selection of a conjecture of the form (C) or (D) such that $k = 1$ and f has a single axiom of the form

$$f(x_1, \dots, x_n) = loop(u_1, \dots, u_k) \tag{E}$$

(C)/(D) is turned into (C')/(D'). (C')/(D') and (E) are transformed into the following conjectures, which characterize INV as a Hoare invariant:

$$INV(x_1, \dots, x_n, u_1, \dots, u_k) \tag{INV1}$$

$$loop(y_1, \dots, y_k) = z \& INV(x_1, \dots, x_n, y_1, \dots, y_k) \Rightarrow conc_1 \tag{INV2}$$

INV2 may be provable by fixpoint induction.

- **create subgoal invariant** works like the preceding command except that the following conjectures are created, which characterize INV as a subgoal invariant:

$$INV(u_1, \dots, u_k, z) \Rightarrow conc_1 \tag{INV1}$$

$$loop(y_1, \dots, y_k) = z \Rightarrow INV(y_1, \dots, y_k, z) \tag{INV2}$$

Again, INV2 may be provable by fixpoint induction.

- **enclose/replace by entry** \boxed{e} assumes the selection of either a single subtree t or several subtrees t, t_1, \dots, t_n such that t encloses t_1, \dots, t_n . In the first case, the entry field must contain a tree u with at most one occurrence of \S . If u contains exactly one occurrence of \S , then t is replaced by $u[t/\S]$. Otherwise the displayed tree v is replaced by $v[u/t]$. In the second case, u must contain exactly n occurrences of \S . Then these occurrences are replaced by t_1, \dots, t_n , respectively, and the resulting tree replaces t in the displayed tree.
- **replace by other sides** assumes that the subtree t selected at first is an implication and the other selected subtrees t_1, \dots, t_n are subterms of the conclusion of t . For each $1 \leq i \leq n$, *replace by term* searches for an equation eq in the premise of t whose left- or right-hand side agrees with t_i . t_i is then replaced by the other side of eq .
- **replace by tree of Solver1/2** replaces the subtree selected at last by the displayed tree of Solver1/2.

- **unify with tree of Solver1/2** unifies the subtree selected at last with the displayed tree of Solver1/2.
- **build unifier** assumes the selection of two subtrees. If they are unifiable, the most general unifier is assigned to the current substitution. Otherwise the reason for the failure is reported.

12 Specification menu

- **remove** removes the set of current axioms. The current signature is reduced to the **built-in symbols**, the current signature map is set to the identity map and the widget interpreter *pictEval* to *matrix* (see **Widgets, graphs, and turtle actions**).
- **save to** saves the current specification to the file in the entry field.
- **load text from** opens a submenu of files. The contents of the selected file is entered into the text field.
- **add from** opens a submenu of files. Signature elements declared in the file are added to the current signature, while formulas are added to the set of current axioms. The formulas must be entered as a disjunction of at most two conjunctions of guarded Horn or co-Horn clauses (see **Axioms and theorems**). The first conjunction is regarded as a set of eager axioms, the second conjunction as a set of lazy axioms (see the **narrow/rewrite** button).

13 Signature/map menu

- **remove map** reduces the current signature map to the identity.
- **show sig** enters the current signature into the text field.
- **show map** enters the current signature map into the text field.
- **apply map** applies the current signature map to the current tree and displays the result in the other solver.
- **load text from file** loads to the text field the file whose name is in the entry field.
- **add map from** opens a submenu of files whose contents is compiled into an extension of the current signature map when the respective menu button is pushed.

14 Axioms menu

- **remove all** removes the set of current axioms.
- **remove from entry field** removes the n -th clause in the text field from set of current axioms provided that the entry field contains the number n .
- **apply axioms in text field** performs at most 5 **narrow/rewrite** steps upon the axioms in the text field. If subtrees have been selected, then each selected subtree is transformed only once and only at the root.
- **apply axioms for symbols** writes the axioms for the symbols in the entry field into the text field and performs at most 5 **narrow/rewrite** steps upon these axioms. If subtrees have been selected, then each selected subtree is transformed only once and only at the root.
- **negate axioms for symbols** adds axioms for the complements of the roots of the subtrees selected at last (or the (co)predicates in the entry field if no subtrees have been selected) to the set of current axioms. The current signature automatically includes symbols for the complements of all its predicates and copredicates (see **Built-in signature**).
- **invert axioms for symbols** transforms the axioms for the roots of the subtrees selected at last (or the (co)predicates in the entry field if no subtrees have been selected) into a single (co-)Horn clause that represents the inverse of the axioms. The clause expresses the *least/greatest* fixpoint semantics of the predicates and is thus added to the current theorems. For instance, Horn axioms for *sorted* (see **LIST**) read as follows:

$$\begin{aligned} &sorted[] \\ &sorted[x] \\ &x \leq y \Rightarrow (sorted(x : (y : s)) \Leftarrow sorted(y : s)) \end{aligned}$$

The equivalent Horn axiom is:

$$\begin{aligned} sorted(z) \implies z = [] \\ | \text{Any } x : z = [x] \\ | \text{Any } xys : (z = x : (y : s) \&x \leq y \&sorted(y : s)) \end{aligned}$$

The clause resulting from inverted axioms is indeed a theorem with respect to the underlying least/greatest fixpoint semantics of predicates/copredicates.


- **Horn axioms for copredicates** transforms the co-Horn axioms for the roots of the subtrees selected at last (or the copredicates in the entry field if no subtrees have been selected) into equivalent Horn axioms (see [11, Def. 2.10]). For instance, co-Horn axioms for *unsorted* (see **LIST**) read as follows:

$$\begin{aligned} unsorted[] \implies False \\ unsorted[x] \implies False \\ x \leq y \Rightarrow (unsorted(x : (y : s)) \implies unsorted(y : s)) \end{aligned}$$

The equivalent Horn axioms are:

$$\begin{aligned}
 & \text{unsorted}(z) \Leftarrow \text{All } i : \text{unsortedLoop}(i, z) \\
 & \text{unsortedLoop}(0, z) \\
 & \text{unsortedLoop}(\text{suc}(i), z) \\
 & \Leftarrow z \neq [] \\
 & \quad \& \text{All } x : z \neq [x] \\
 & \quad \& \text{All } xys : (z = x : (y : s) \& x \leq y \Rightarrow \text{unsortedLoop}(i, y : s))
 \end{aligned}$$





The old axioms for the copredicates are deleted and the copredicates are turned into predicates.

- **show axioms for symbols**  enters the axioms for the roots of the subtrees selected at last (or the symbols in the entry field if no subtrees have been selected) into the text field.
- **show** enters all current axioms into the text field.
- **show in text field of Solver1/2** enters all current axioms of Solver 1/2 into the text field of Solver1/2.
- **load text from** opens a submenu of files. The contents of the selected file is entered into the text field.
- **add from** opens a submenu of files. Signature elements declared in the file are added to the current signature, while formulas are added to the set of current axioms. The formulas must be entered as a disjunction of at most two conjunctions of guarded Horn or co-Horn clauses (see [Axioms and theorems](#)). Files containing only signature elements are not allowed here.

15 Theorems menu

- **remove all** removes the set of current theorems.
- **remove from entry field** removes the n -th clause in the text field from set of current theorems provided that the entry field contains the number n .
- **show** enters all current theorems into the text field.
- **show in text field of Solver1/2** enters all current theorems of Solver 1/2 into the text field of Solver1/2.
- **save to file** saves the current theorems to the file in the entry field.
- **load text from** opens a submenu of files. The contents of the selected file is entered into the text field.
- **add from** opens a submenu of files. Signature elements declared in the file are added to the current signature, while formulas are added to the set of current theorems. The formulas must be entered as a conjunction of Horn or co-Horn clauses (see [Axioms and theorems](#)). Files containing only signature elements are not allowed here.

16 Graph menu

- **redraw**  redraws the displayed tree. This removes junk from the canvas (see [Mouse and key events](#)).
- **expand** dereferences all pointers of the displayed tree or the selected subtrees, respectively. If the entry field contains a positive number n (default: $n = 0$), each circle in the trees is unfolded n times.
- **expand leaves** dereferences all pointers to leaves of the displayed tree or the selected subtrees, respectively.
- **collapse** \rightarrow and **collapse** \leftarrow identify all common subtrees of the displayed tree or the selected subtrees, respectively. If there is a number n in the entry field, cycles are unfolded n times. Otherwise cycles are not unfolded. *collapse* \rightarrow creates pointers to the right, *collapse* \leftarrow produces pointers to the left.
- **build equations**  assumes that the subtree t selected at last is a list of pairs consisting of an integer (a node) and a list of integers (the direct successors) or a list of triples consisting of an integer (a node), a term (an edge label) and a list of integers (the direct successors). *build equations* transforms t into an equivalent conjunction $x_1 = t_1 \& \dots \& x_n = t_n$ of regular equations, i.e. x_1, \dots, x_n are variables and t_1, \dots, t_n are terms that may contain these variables.
- **build graph**  assumes that the subtree t selected at last is a collection of pairs consisting of an integer and a list of integers or a collection of triples consisting of an integer, a term and a list of integers or a conjunction of regular equations. *build graph* transforms t into an equivalent (edge-labelled) graph. Edge labels are turned into node labels so that the graph is actually a bipartite one. The graph is constructed in a depth-first manner starting out from the first element of the list or conjunction. Hence only pairs, triples or regular equations, respectively, that are "reachable" from this element are considered!
- **label graph**  labels the nodes of a graph representing an (edge-)labelled or unlabelled transition system on integer state by the labels stored in the state variable *labels* (see [Simplifications](#)).

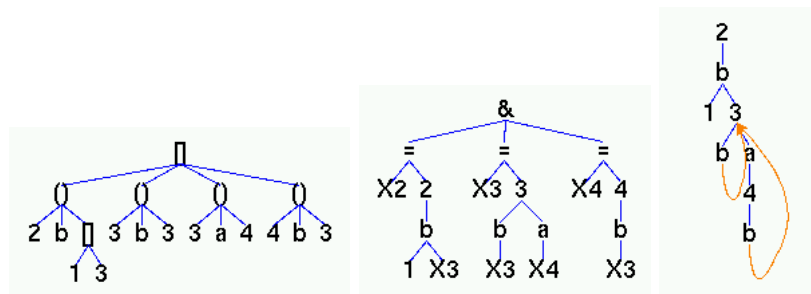




Figure 16.1: A labelled transition system (see [TRANS1](#)) as a list of (node,label,successors)-triples, a conjunction of regular equations and a bipartite graph.

- **unlabel graph**  restores the nodes of a (node-)labelled graph representing an (edge-)labelled or unlabelled transition system on integer states.
- **build relation**  reverses the application of *build graph*, i.e. an (edge-labelled) graph is transformed into an equivalent list of pairs consisting of an integer (a node) and a list of integers (the direct successors) or an equivalent list of triples consisting of an integer (a node), a term (an edge label) and a list of integers (the direct successors).
- **greatest lower bound** colors the root of the greatest lower bound of the selected subtrees in green.
- **predecessors** colors the predecessors of the roots of the selected subtrees in green.
- **successors** colors the successors of the roots of the selected subtrees in blue.
- **variables** colors the variables of the selected subtrees in blue.
- **free variables** colors the free variables of the selected subtrees in blue.
- **label roots with entry** labels the roots of the selected subtrees with the string in the entry field. The root types (F or V) are preserved. If the replacing string or the replaced string belong to the current signature, both strings must be of the same type. The changed labels are colored in blue.
- **polarities** colors the roots of all subtrees of the displayed tree. A root is colored in green if the subtree has positive polarity. Otherwise it is colored in red.
- **positions** replaces the nodes of the displayed tree by their tree positions. Each pointer position is labelled in red with the position of its target node.
- **numbers** replaces the labels of the nodes of the displayed tree by their positions in the breadthfirst node ordering.
- **coordinates** shows the coordinates of the node labels of the displayed tree.

17 Substitution menu

- **add from text field** adds to the current substitution the substitution that is given by the conjunction of equations in the text field.
- **apply** applies the current substitution to the selected subtrees of the displayed tree (or to the entire tree if no subtrees have been selected) and sets the current substitution to the empty one.
- **rename** assumes that the entry field contains a conjunction of equations $x = y$ between variables. All occurrences of x in the selected subtrees are replaced by y .
- **remove** clears the current substitution.
- **show** enters the equations that represent the current substitution into the text field.
- **show in text field of Solver1/2** enters the equations that represent the current substitution of Solver2/1 into the text field of Solver1/2.
- **show on canvas of Solver1/2** displays the equations that represent the current substitution of Solver2/1 on the canvas of Solver1/2. The equations become the current trees of Solver1/2.
- **show solutions** writes the positions of the **solved formulas** resp. **normal forms** among the current trees into the label field.

18 Further Buttons

Pictorial term representations consist of **widgets**. A list of widgets is called a **picture**. A picture becomes a **widget graph** if some of its widgets are connected by directed arcs. Widgets comprise circles, paths, polygons, text entries, node-labelled trees, and sequences of **turtle actions** that admit the hierarchical construction of pictures insofar as the drawing of a picture (without arcs) is also a turtle action. For the complete list of widgets, see [Widgets, graphs, and turtle actions](#). The actual widget interpreter is selected from the **pict type** menu. Some built-in axiom files (see [Examples](#)) and enumerators are automatically associated with a widget interpreter.

- **paint** opens a *Painter* window that consists of the following widgets:
 - A scrollable canvas.
 - A slider for selecting the graph to be displayed on the canvas.
 - The **size** slider sets the scaling factor of the displayed widgets.
 - The **speed** slider and the **fast** button select the painting speed.
 - The **renew** button calls *paint* from the painter window.
 - The **narrow/rewrite** button and the **simplify** button trigger the synonymous actions in the associated solver and display their pictorially representable results on the canvas of the painter.
 - The **combine:** entry field recognizes numbers describing the source and layout of pictures to be copied from the list of current graphs and added to the displayed one. *combine:* is activated by pushing the *Up* key while the cursor is in the entry field.
 - The **replace by:/save to:** entry field may contain either
 - * the name of a file containing a picture term or, if the file does not exist (neither in the user's *Examples* directory nor in the synonymous system directory), a picture term that shall be inserted into the displayed graph, or
 - * the name of a file where the displayed graph shall be saved to, in eps format or as an object of the Haskell type $(Nodes, Arcs)$ for widget graphs (see [Widgets, graphs, and turtle actions](#)).

The first action is executed when the *Up* key is pushed while the cursor is in the entry field. The second action is executed when the *Right* key is pushed while the cursor is in the entry field.

- The **connect/enclose** button switches to or from a state in which arcs between widgets of the displayed graph may be drawn (if the left or right mouse button is pressed) or widgets

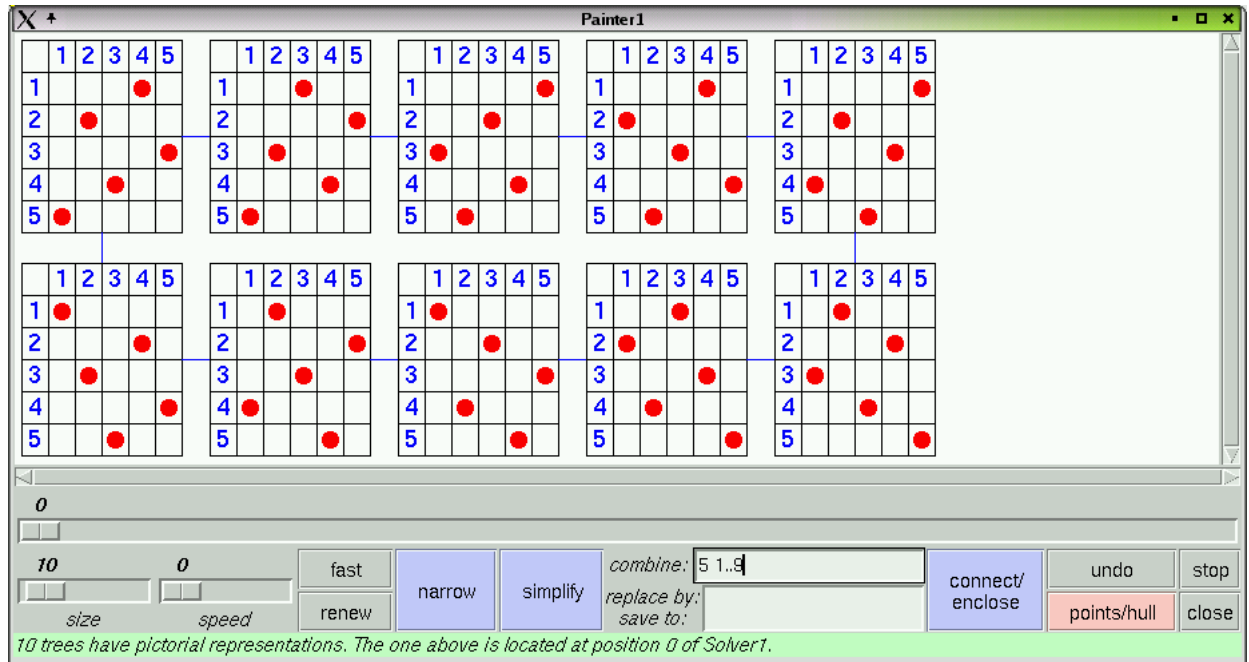


Figure 18.1: The painter window shows a graph consisting of the ten solutions of $queens(5, X)$, obtained from applying axioms of **QUEENS**. The "5" in the add field denotes that (at most) 5 pictures are listed in one line. "1..9" says that the pictures at (slider) positions 1, ..., 9 are added to the displayed one (at position 0).

may be grouped for admitting simultaneous modifications.

- The **undo** button revokes the immediately preceding action on the displayed graph.
- The **points/hull** button hides/shows the (arc shaping) point widgets of the displayed graph or draws the convex hull of all displayed widgets, respectively. The widgets of the hull may be numbered counter-clockwise.
- The **stop/go** button interrupts/resumes the painting of the displayed graph.
- The **close** button finishes painting and closes the painter window, but keeps the actual size and speed values. *For closing a painter window, use only this button!*
- A label field for displaying messages.

A complete description of the features for creating and editing widget graphs is given in the section [Widgets, graphs, and turtle actions](#).

If subtrees have been selected, **paint** combines the widget interpretations of all pictorially representable selected subtrees and displays the resulting picture on the canvas. With the middle mouse button, the individual widgets, which are usually displayed on top of each other, can be drawn horizontally and vertically away from each other.

If no subtrees have been selected, then for each element t of the list of current trees, **paint** combines the widget interpretations of all maximal pictorially representable subtrees of t . The resulting picture that corresponds to the tree displayed on the solver canvas is drawn on the painter canvas, while the pictures derived from other elements of the list of current trees are assigned to other

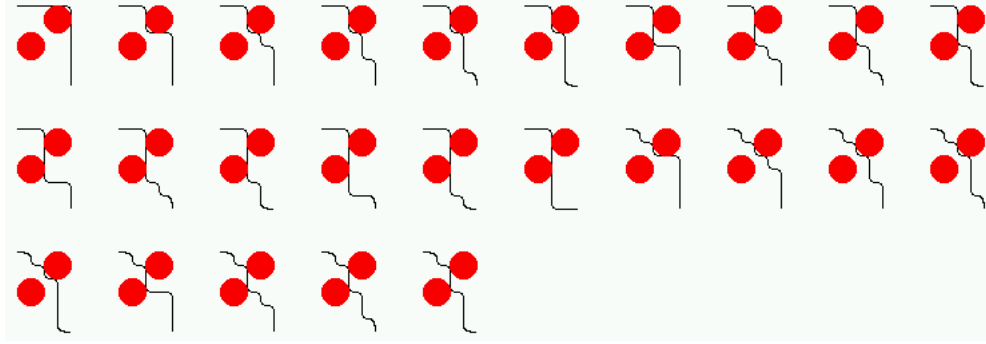


Figure 18.2: Pictorial representations of solutions obtained by applying axioms of **ROBOT**

positions in the **list of current pictures**. One may browse among the pictures by moving the graph selecting slider (see above).


Expander2 provides several widget interpreters and combinators thereof. The actual one depends on the current axioms, but can also be set by selecting from the *pict type* menu. For instance, Fig. 18.2 shows solutions of the formula

$$\begin{aligned} \text{Any } PS : (\text{robot}([(2, 6), (6, 2)], PS) \\ \& \text{Acts} = [\text{pathS}(PS), O, J(2), R, J(6), \\ \text{circ}(2, \text{red}), B, J(4), R, J(4), \text{circ}(2, \text{red}), C]), \end{aligned}$$


obtained by applying axioms of **ROBOT**, were generated by the interpreter *polygon solution* that looks for solved formulas and applies the interpreter *polygon* to the solving terms in these formulas. A solved formula looks as follows:

$$\begin{aligned} \text{Any } Z_1 : x_1 = t_1 \& \dots \& \text{Any } Z_k : x_k = t_k \\ \& \text{All } Z_{k+1} : x_{k+1} \neq t_{k+1} \& \dots \& \text{All } Z_n : x_n \neq t_n. \end{aligned}$$

x_1, \dots, x_n are different free variables, t_1, \dots, t_n are normal forms and the transitive closure of $\{(i, j) \mid t_i \text{ contains } x_j\}$ is acyclic.

- **hide/show**  hides all unhidden selected subtrees by replacing them with the character @ and restores all selected subtrees that are hidden, i.e. had been replaced by @. If the entry field contains a natural number n and no subtrees were selected, then *hide/show* hides all unhidden subtrees at level n and restores the hidden ones. If the entry field does not contain a natural number and no subtrees were selected, then *hide/show* restores all hidden subtrees.

Trees with hidden subtrees can also be simplified, narrowed or rewritten. This saves the time for displaying reducts and is particularly useful when simplification, rewriting or narrowing steps are carried out from the painter because a pictorial representation of reducts is preferred to the tree representation.

- **+1/-1** increases/decreases the natural number in the entry field. If the entry field does not contain a natural number, then *+1* writes 0 into it, while *-1* clears it.
- **set** selects the subtree whose position in the breadthfirst node ordering agrees with the number in the entry field.
- **parse up**  parses the string in the text field according to the grammar given below, initializes



the list of current trees and the tree mode and displays the first element of the list on the canvas. Trees are coded as objects of the instance `Term String` of the Haskell type

```
data Term a = V a | F a [Term a] | state variable constructors
```

The first constructor precedes first-order variables, the second one logical and non-logical function symbols and higher-order variables. For the state variable constructors, see [Simplifications](#).

parse up expects a term or formula built up of logical operators, signature symbols and further strings a that are regarded as function symbols. If the term is a sum or the formula is a disjunction or conjunction, the respective summands or factors become the list of current trees. Otherwise this set is a singleton consisting of the entire parsed tree. If the entry field contains an index i of a list of axioms or theorems or a substitution in the text field, *parse up* parses and displays the i -th element of the list or the i -th equation of the substitution, respectively.

Given natural numbers n_1, \dots, n_k , strings of the form *pos* $n_1 \dots n_k$ are interpreted as pointers (see above). Only first-order variables and pointers are turned into objects built up with the constructor `V`. Subtrees whose root starts with the character `@` is not printed (see the [hide/show](#) button).

- **remove text** clears the text field.
- **remove entry&label** clears the entry and the label field.
- **parse down**  computes the textual representation of the displayed tree resp. selected subtrees, connects them with the symbol in the entry field and writes the result into the text field provided that all selected subtrees are either terms or formulas.
- **narrow/rewrite**  performs narrowing/rewriting steps and, between each two steps, at most 100 simplification steps, from top to bottom and from left to right, first to the displayed tree and then to other current trees, until,
 1. *no subtrees have been selected and the entry field does not contain a positive natural number*: at most one narrowing/rewriting step has been performed,
 2. *if the current trees are terms/formulas, no subtrees have been selected and the entry field contains a positive natural number n* : all current trees are non-rewritable/non-narrowable or the number of successive rewriting/narrowing steps exceeds n ,
 3. *if subtrees have been selected and the entry field does not contain a positive natural number*: at most one rewriting/narrowing step has been performed on each selected subtree (if the subtree is a term t and the current trees are formulas, then the step is performed on the t enclosing atom),
 4. *if subtrees have been selected and the entry field contains a positive natural number n* : the number of successive rewriting/narrowing steps performed on subtrees of the subtree selected at last has reached n .
- The \leftarrow **unify/match** button switches between the (default) **unification mode**, the **matching mode** and the **greedy versions** of these modes when formulas are narrowed or terms are rewritten. The unification mode admits the instantiation of redex variables, the matching mode does not. Usually, all applicable axioms are applied in parallel and for each guarded axiom, each solution of

the guard leads to a reduct. In a greedy mode, only the first applicable axiom is applied to a given reduct, but, again, if the axiom is guarded, each solution of the guard leads to a reduct.





In case (1), (2) or (4), non-narrowable logical atoms $P(t)$ with normal form t are simplified to *False* if P is a predicate and by *True* if P is a copredicate. Analogously, non-narrowable terms $f(t)$ with normal form t are simplified to the undefinedness constant $()$ (see **Built-in signature**). *This transformation may lead to undesired results if some function or (co)predicate occurring in a (simplified) conjecture has not been specified completely or if narrowing/rewriting is used in a matching or greedy mode!* For instance, the simplifier turns a negated formula $Not(F)$ into a formula without negation, which may contain unspecified complement predicates. This is one reason why explicit negation should be avoided!

Narrowing upon a predicate or copredicate p generalizes **linear resolution** to the simultaneous application of all axioms for p . Narrowing also generalizes rewriting from terms to formulas and admits the instantiation of reduct variables by non-variable terms. These substitutions are supposed to build up solutions of the formula at the beginning of a narrowing sequence. Since each narrowing step applies the definition (axioms) of a predicate, copredicate or defined function, we have also used the term **unfolding** for narrowing steps [15].

Applying all applicable (Horn) axioms for a predicate p or a defined function f simultaneously results in the replacement of the reduct by the *disjunction* of their premises and equations representing the computed unifiers. Applying all applicable (co-Horn) axioms for a copredicate p simultaneously results in the replacement of the reduct by the *conjunction* of their conclusions. Some equational axioms may be only partially unifiable with the reduct. These are applied as well, but contribute to the reduct not with their premises resp. conclusions, but with equations representing the partial unifiers. This extension is called **needed narrowing** [1, 14] and ensures that the iteration of narrowing steps proceeding from supertrees to subtrees leads to all solutions of the current trees.

The set of current axioms splits into the set of **eager axioms** and the set of **lazy axioms**. A rewriting or narrowing step consists in the simultaneous application of all axioms for the leading predicate, copredicate or defined function of the maximal subtree to which eager axioms apply or, if no eager axiom applies, lazy axioms apply. Common candidates for lazy axioms are AC equations (see **Built-in signature**) that have several overlapping redices and thus may prevent other axioms from being applied to subredices.

- **simplify** s performs simplification steps, from top to bottom and from left to right, first to the displayed tree and then to other current trees, until,
 1. *if no subtrees have been selected and the entry field contains a positive natural number n* : all current trees are simplified, i.e. no further simplification rule is applicable, or the number of successive simplification steps exceeds n ,
 2. *if no subtrees have been selected and the entry field does not contain a positive natural number*: at most one simplification step has been performed,
 3. *if subtrees have been selected and the entry field does not contain a positive natural number*: at most one simplification step has been performed on each selected subtree,
 4. *if subtrees have been selected and the entry field contains a positive natural number n* : the number of successive simplification steps performed on subtrees of the subtree selected at last has reached n .

- **clear subtrees** deselects all selected subtrees.
- **collapse levelwise** identifies the common subtrees of the current tree level by level.
- **apply to variable:** opens a menu of all variables in the domain of the current substitution f . If the button for a variable x is pressed, all occurrences of x in the subtree selected at last are replaced by $f(x)$. If they are bound by an existential/universal quantifier and the respective quantified subformula t has positive/negative polarity, then all occurrences of x in t are replaced by $f(x)$ and x is removed from the quantifier.
- \leftarrow/\rightarrow  /  proceed one step backward/forward in the current proof and display the corresponding list of current trees. As soon as a rule is applied to the list, its previous successors are removed from the proof.
- **build/check** switches between the (default) proof build mode and the proof check mode. In the latter, you can evaluate a loaded proof term step by step by pushing the \rightarrow button. Hence the proof term will not be modified in this mode, even if you apply new rules to the current tree, In the proof build mode, however, new rule applications will be added to the proof term. More precisely, the proof term will be kept until the actual position of the proof term pointer and from there extended by the new rule applications.
- **increase/decrease current**  /  proceed to the next/previous element of the list of current trees if the Up resp. Down key is pushed after the label field has been activated. If a Painter window is in the foreground, then the next/previous picture is displayed.
- **quit** quits Solver1/2.

19 Grammar

according to which **parse up** translates a string in the text field into a term or formula. Boldface-typed symbols are terminal.

implication	→ disjunct ==> disjunct disjunct ===> disjunct disjunct <=== disjunct	
disjunct	→ conjunct conjunct disjunct	
conjunct	→ enclosedFactor enclosedFactor & conjunct	
enclosedFactor	→ (implication) factor	
factor	→ True False Not enclosedFactor Any vars : enclosedFactor All vars : enclosedFactor infixAtom prefixAtom singleTerm ^ singleTerm moreBag	
vars	→ var var vars	
var	→ noBlanks	<i>noBlanks must derive to a first- or higher-order variable of the current signature.</i>
infixAtom	→ term infixPred term noBlanks	<i>noBlanks must derive to a first-order variable of the current signature.</i>
prefixAtom	→ noBlanks moreTerms	<i>noBlanks must derive to a predicate, a copredicate or a higher-order variable of the current signature.</i>
moreTerms	→ (relTerms) moreTerms enclosedTerms moreTerms enclosedTerms	
enclosedTerms	→ (terms) [terms] {terms} (infixFun)	
relTerms	→ relTerm relTerm , relTerms	
relTerm	→ relChars relChars (relTerms) prefixAtom term	<i>relChars must derive to =, =/=, a predicate or a copredicate of the current signature.</i>
infixPred	→ infixChars infixWord	<i>infixPred must derive to =, =/=, a predicate or a copredicate of the current signature.</i>
term	→ singleTerm termrest	
terms	→ term term , terms term .. terms	
termrest	→ bagop term moreBag(bagop) moreInfix	
moreBag(bagop)	→ bagop term moreBag(bagop) empty	
bagop	→ ^ <+> ++ list set boolTerm double int int curryrest string fovar pos treepos -singleTerm () (term) enclosedTerms noDelims noDelims curryrest	<i>Nested bag terms are flattened: t1^ t2 ^ t3^...</i>
singleTerm	→	<i>noDelims must not derive to =, =/=, a logical symbol, a predicate (except for §), a copredicate (except for §) or a first-order variable of the current signature.</i>
curryrest	→ enclosedTerms curryrest empty	
moreInfix	→ infixFunL singleTerm moreInfix infixFunR term empty	
infixFunL	→ + -	<i>Functions derived from infixFunL are left-associative: (((t1+t2)-t3)+t4)-t5.</i>
infixFunR	→ infixChars infixWord	<i>infixFunR must derive to a constructor or defined function o of the current signature, but not to <+> or ^. o is right-associative: t1 o (t2 o (t3 o (t4 o t5))).</i>
boolTerm	→ bool (implication) cond (disjunct, terms) lin (conjunct)	
list	→ [] [terms]	

set	→ { } {terms}	
int	→ any constant of the Haskell type int	
double	→ any constant of the Haskell type double	
string	→ any string	
fovar	→ noBlanks	fovar must derive to a first-order variable of the current signature.
		any finite list of natural numbers separated by blanks
treepos	→	For instance, pos 2 1 3 denotes a pointer to the node at position [2,1,3].
infixWord	→ `any string that does not contain back quotes `	
infixChars	→ any string except ^ and <+> that consists of characters among	
	: + - * < = ~ > / ^ #	
noDelims	→ any string that does not contain a character among	
	() [] { } , ` & : + - * < = ~ > / ^ # \t \n	
noBlanks	→ any string that neither contains a blank nor an element of the preceding character list	
relChars	→ any string that neither contains a blank nor a character among	
	() , \t \n	

noDelims need not derive to a symbol of the current signature. Any string derived from noDelims is turned into a node with the Term constructor F (see the [parse up](#) button). Moreover, noDelims may derive to a string with blanks. This permits the use of symbols consisting of several words separated by blanks.

Integers, reals and (quoted) strings are automatically interpreted as (not always nullary!) constructors. This admits, for instance, the use of natural numbers in the tree representations of nested partitions (see [Dissections and partitions](#)).

x_0, x_1, x_2, \dots are interpreted as defined functions (see [Built-in signature](#).) An important technical reason for declaring a function symbol as a defined function is the fact that the leading non-equational symbol of each axiom must be a predicate, a copredicate or a defined function.

Newline characters followed by a dot must be avoided because this because such a string is interpreted in a particular way. When a line with more than 80 characters is entered into the text field, it is split into several lines each of which starts with a dot. This ensures that decomposed lines are recognized as single ones when the contents of the text field is parsed.

Line suffixes starting with -- are regarded as comments.

20 Axioms and theorems

Axioms and theorem to be applied in derivations are **Horn clauses**

$$(1) \{guard \Rightarrow\}(f(t_1, \dots, t_n)) = u \{\Leftarrow prem\} \quad \text{or}$$

$$(2) \{guard \Rightarrow\}(p(t_1, \dots, t_n) \quad \{\Leftarrow prem\}) \quad \text{or}$$

$$(3) t = u \{\Leftarrow prem\} \quad \text{or}$$

$$(4) q(t_1, \dots, t_n) \{\Leftarrow prem\},$$

co-Horn clauses

$$(5) \{guard \Rightarrow\}(f(t_1, \dots, t_n)) = u \Longrightarrow conc) \quad \text{or}$$

$$(6) \{guard \Rightarrow\}(q(t_1, \dots, t_n) \Longrightarrow conc) \quad \text{or}$$

$$(7) t = u \Longrightarrow conc \quad \text{or}$$

$$(8) p(t_1, \dots, t_n) \Longrightarrow conc,$$

distributed Horn clauses

$$(9) a_{t_1} \mid \dots \mid a_{t_n} \quad \{\Leftarrow prem\} \quad \text{or}$$

$$(10) a_{t_1} \& \dots \& a_{t_n} \quad \{\Leftarrow prem\}$$

or **distributed co-Horn clauses**

$$(11) a_{t_1} \mid \dots \mid a_{t_n} \Longrightarrow conc \quad \text{or}$$

$$(12) a_{t_1} \& \dots \& a_{t_n} \Longrightarrow conc$$

Curly brackets enclose optional parts. f , p and q denote a defined function, a predicate and a copredicate, respectively, of the current signature. If the current trees are terms, then the reducts must be terms and thus only premise-free clauses of the form (1) can be applied.

Axioms are of type (1), (2) or (6). The *step functions* (or *consequence operators*) induced by axioms must be monotone [15, 11]. Usually, f , p resp. q agree with the root of the redex to which a clause is applied. The redex is unified with the head of the clause and the instance of the body ($prem$ resp. $conc$) by the unifier is the reduct that replaces the redex. In case (3), the redex must be unifiable with the term t . In case (7), the redex must be unifiable with the equation $t = u$. The co-Horn clause that is created when fixpoint induction is called is of type (7) and applied to a redex $t' = u'$ only if neither t nor t' is a variable.

Defined functions need not be total and not even deterministic! Undefinedness is expressed by the constructor constant $()$. Multiple values are combined with the sum constructor $\langle + \rangle$ (see [Built-in signature](#)).

For applying a distributed clause, select n atoms $a'_{t_1}, \dots, a'_{t_n}$ in a disjunction/conjunction F of the displayed tree such that for all $1 \leq i \leq n$, a'_{t_i} is unifiable with a_{t_i} . The summand/factor of F where a'_{t_i} is selected from must consist of disjunctions, conjunctions and existential/universal quantifiers only (see [16, 10]). a'_{t_i} is replaced by the corresponding instance of *prem/conc*. The resulting reducts are combined conjunctively in the case of a Horn clause and disjunctively in the case of a co-Horn clause.

A clause with a guard is applied only if the guard is solvable. The solution becomes part of the unifier that is generated when the clause is applied. For instance, the axiom

$$\textit{split}(s) = (s_1, s_2) \Rightarrow \textit{sort}(x : (y : s)) = \textit{merge}(\textit{sort}(x : s_1), \textit{sort}(y : s_2)),$$

for \textit{sort} (taken from [LISTEVAL](#)) is guarded, while the logically equivalent axiom

$$\textit{sort}(x : (y : s)) = \textit{merge}(\textit{sort}(x : s_1), \textit{sort}(y : s_2)) \Leftarrow \textit{split}(s) = (s_1, s_2)$$

(taken from [LIST](#)) is unguarded. On the one hand, guarded axioms are needed for evaluating ground terms efficiently. On the other hand, axioms and theorems used as lemmas in step by step derivations (see below) must be unguarded. Otherwise the search for a solution of the guard may block the derivation process.

21 Derivations

A proof with Expander2 is a sequence of successive values of the state variable `trees`. It is documented and stored in the state variable `proof`. The values of `proof` and `current trees` are initialized whenever `parse up` parses the contents of the text field and displays the resulting tree t on the canvas. Then `trees` is set to $[t]$, `tree` is set to t , and `proof` is set to the initial values of its components.

A proof step is correct if the transformed disjunction (or conjunction or sum) of the current trees implies (or is equivalent to) the original one. In the case of possible incorrectness Expander2 delivers the warning

CAUTION: This step may be incorrect.

Such steps are not stored in the current proof term.

If the current trees are formulas, a proof ending up with `True` or `False` yields a proof resp. refutation of the conjunction/disjunction of the initial trees. Other final results are given by `solved formulas` that represent solutions of the original conjecture in their free variables.

If the current trees are terms, only rewriting steps may be applied. If a rewriting step leads to several reducts, the applied axioms specify a non-deterministic function: each reduct is a possible value and will become an element of the `sum` of the current trees.

The correctness of a proof step depends on the **polarity** of the redex with respect to its position within the displayed tree. The polarity is *positive* if the number of preceding negation symbols or premise positions is even. Otherwise the polarity is *negative*. Fixpoint induction, coinduction, summand removal, summand unification, applications of Horn clauses, instantiations of existential variables and term replacements (see `replace by other sides`) are correct if the redex has positive polarity because here the reduct implies the redex. Atom composition, factor removal, factor unification, applications of co-Horn clauses and instantiations of universal variables are sound if the redex has negative polarity because here the redex implies the reduct. Simplifications, rewriting, narrowing, splitting, flattening and stretching may be applied to any possible redex within the displayed tree because here the redex and the reduct are equivalent. The restrictions guarantee that the transformed displayed tree implies the original one.

The theorem-proving features of Expander2 do not aim at fully automatic proofs. Expander2 favors natural deduction in contrast to many other provers that submit a conjecture to Skolemization and other extensive normalizations before the proof can start. Avoiding this enhances the readability and thus the controllability of the generated proofs significantly. This is particularly necessary when induction or coinduction steps are involved, which are at the heart of any non-trivial program verification. Fortunately, the axioms, the theorems and, to some extent, the conjectures we are faced with in program verification already come as Horn or co-Horn clauses and thus can indeed be handled by Expander2 in their original form.

It also complies with a natural proof process that Expander2 avoids negation symbols. The simplifier drives them innermost until they directly precede (co)predicates and can be removed completely

by transforming the (co)predicates into their complements. Axioms for the complement not_P of a (co)predicate P can usually be constructed from those for P . If P is a predicate, then not_P is a copredicate. If P is a copredicate, then not_P is a predicate. Axioms without negation ensure that the induced *step functions* (or *consequence operators*) are monotone. Hence they have least resp. greatest solutions, which interpret predicates resp. copredicates in the initial model [15, 11].

22 Variables

Variables of a clause that are introduced into the current tree when the clause is applied to the tree are renamed by increasing the number suffixes of the variables. Variables that the tree shares with the applied clause are renamed in the same way. Since variable renaming affects the state variable *varCounter* (see [Solver state variables](#)), it is not performed during simplification.

Variables of a Horn or co-Horn clause are turned into existential resp. universal variables. The scope of these variables is the respective reduct.

If a free variable x of a redex is instantiated by a term t during a rewriting or narrowing step, then the equation $x = t$ is added to the reduct.

All integers starting with a backslash are regarded as variables. They are used for the new variables introduced by the formula transformations *create Hoare/subgoal invariant*, *stretch premise/conclusion*, *apply fixpoint induction*, *apply coinduction* and *flatten (co-)Horn clause* (see [Subtrees menu](#)) and when a subtree is cut out of the displayed tree (see [Mouse and key events](#)).

23 Simplifications

Narrowing removes predicates, copredicates and defined functions of the current signature from the current trees. The simplifier does the same with logical operators, constructors and symbols of the built-in signature. Simplifications realize the highest degree of automation and the lowest level of interaction (see [Overview](#)). The reducts of rewriting or narrowing steps are simplified automatically. Pushing the [simplify](#) button admits step-by-step simplification of the current trees.

The simplifier turns formulas into minimal *nested Gentzen clauses* of the form

$$\text{All } x_1 \dots x_i : (\text{Any } y_1 \dots y_j : (t_1 \& \dots \& t_m) \Rightarrow \text{All } z_1 \dots z_k : (u_1 \mid \dots \mid u_n)).$$

from the current trees. “Nested” means that the clause is derivable by the following grammar:

$$\begin{aligned} S &\longrightarrow A \\ S &\longrightarrow B \\ A &\longrightarrow A_1 \\ A &\longrightarrow C \\ B &\longrightarrow B_1 \\ B &\longrightarrow C \\ A_1 &\longrightarrow \text{Any } x_1 \dots x_k : (B \& \dots \& B) \\ B_1 &\longrightarrow \text{All } x_1 \dots x_k : (A \mid \dots \mid A) \\ C &\longrightarrow \text{All } x_1 \dots x_k : (A_1 \Rightarrow B_1) \\ C &\longrightarrow \text{atom} \end{aligned}$$

The rules applied repeatedly by the simplifier are equivalence transformations. The main ones are given in [\[10\]](#). They employ not only logical equivalences, but also the semantics of constructors, equality, inequality and other [built-in symbols](#). For instance, an implication *prem* \Rightarrow *conc* is reduced to *True* if *prem* **subsumes** *conc* [\[10\]](#), a disjunction is reduced to its minimal summands, a conjunction to its maximal factors. Here are some examples:

$$\text{Any } xyz : (x = f(y) \& Q(z)) \Rightarrow \text{Any } x' y' z' : (Q(z') \& f(y') = x')$$

reduces to *True*.

$$\text{Any } x : Q(x) \& Q(\text{suc}(y)) \& \text{All } x : R(x) \& R(y + z) \& \text{Any } x : x = \text{suc}(y) \& \text{suc}(y) = y + z$$

reduces to $Q(\text{suc}(y)) \& \text{All } x : R(x) \& \text{suc}(y) = y + z$.

$$\begin{aligned} &\text{Any } x : (x = f(h(y, z), z) \& P(x, y)) \\ &\& \text{All } x : (x \neq f(h(y, z), z) \mid P(x, y)) \\ &\& \text{All } x : (x = f(h(y, z), z) \& P(x, y) \Rightarrow Q(x)) \\ &\& \text{All } x : (P(x, y) \Rightarrow x \neq f(h(y, z), z) \mid Q(x)). \end{aligned}$$

reduces to $P(f(h(y, z), z), y) \& Q(f(h(y, z), z))$.

$$P(x, y) \& Q(z) \& (P(x, y) \Rightarrow R(x, y, z))$$

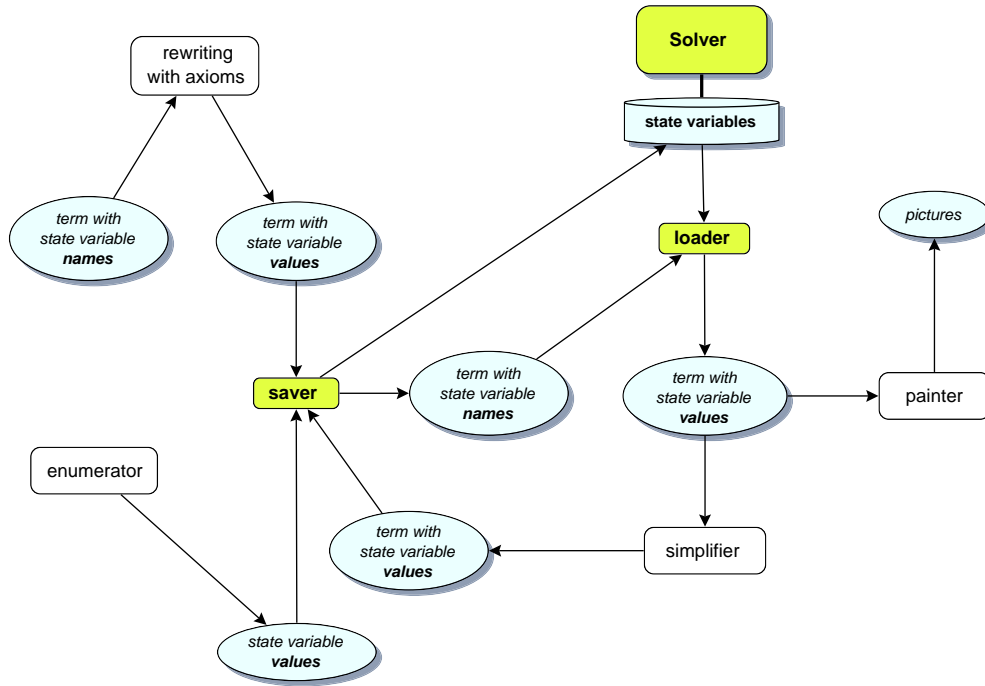


Figure 23.1: How the solver processes terms with state variable names

reduces to $P(x, y) \& Q(z) \& R(x, y, z)$.

$$P(x, y) \Rightarrow (Q(y) \Rightarrow R(x, y, z)) \mid P(y, z)$$

reduces to $P(x, y) \& Q(y) \Rightarrow R(x, y, z) \mid P(y, z)$.

$$2 \text{ 'in' } [1, 2, 3]$$

reduces to *True*.

$$[1, x, 4] \text{ 'gives' } y$$

reduces to $y = 1 \mid y = x \mid y = 4$. While 'in' may only be applied to constructor terms, the arguments of 'gives' may be arbitrary terms. The example shows that an atom of the form $ts \text{ 'gives' } t$ where ts and t are constructor terms is simplified to a **solved formula** and can thus be used in the guard of an axiom.

$$[2, 3] + + [5 \text{ 'mod' } 2, 1] \text{ <+> } 78 \text{ <+> } \{ \} \wedge \{9, 5, 5\} \wedge \{9, 9, 5\}$$

reduces to $[2, 3, 1, 1] \text{ <+> } 78 \text{ <+> } \{ \} \wedge \{5, 9\} \wedge \{5, 9\}$.

$$[1, 2, 3] - 2$$

reduces to $[1, 3]$.

$$\text{zipAny}(=)[1, x, 3, 4][5, 2, y, 6]$$

reduces to $x = 2 \mid 3 = y$.

$$\text{zipAny}(=)[1, x, 3, 4][1, 2, y, 6]$$

reduces to *True*.

$$\text{zipAll}(=)[1, x, 3, 4][1, 2, y, 4]$$

reduces to $x = 2 \& 3 = y$.

Some simplifications depend on **solver state variables** that hold actual parameters for the function to be simplified. For this purpose, certain terms can be rewritten into *state variable names* the serve as

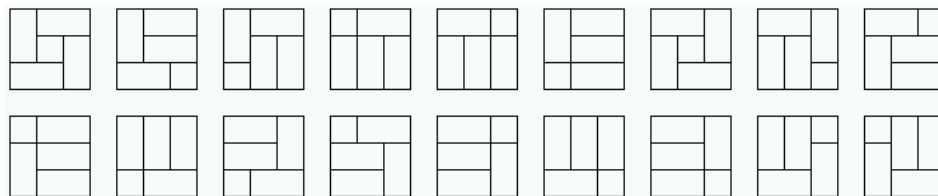


Figure 23.2: All dissections of a 3x3-rectangle that satisfy $area(2)$: each dissection consists of $\lceil 9/2 \rceil = 5$ subrectangles.

placeholders for the dynamically changing values of state variables. If the simplifier encounters a state variable constant c in the tree to be simplified, the actual value of c is loaded from the solver and, after the simplification step using this value has been performed, the modified value of c is saved by the solver. This allows the simplifier to evaluate Haskell functions directly on Haskell data that otherwise would have to be translated from and into — often complex — terms over the current signature. The above diagram shows that besides simplification also rewriting and widget interpretation have an interface to state variable values: certain axioms produce reducts that involve state variable values; certain state variable values can be translated into pictures. The latter feature allows one to make state changes visible in terms of representations that are not the symbolic ones given by formulas or algebraic terms. Currently, the following state variables are available. Each of them is associated with a constructor of Term a (see the [parse up](#) button):

- `actions :: Int -> String -> ActLR` (constructor `Actions`) holds a function from states and strings to actions of an LR(1) parser.
- `boolFun :: (String -> Bool, Int)` (constructor `Perm`) holds a Boolean function and the number n of its Boolean arguments. `Perm` has two arguments: `boolFun` and a permutation of $[1..n]$.
- `dissects :: [[(Int, Int, Int, Int)]]` (constructor `Dissect`) holds a list of lists of quadruples of integers. `Dissect` has two arguments: an index and an element of `dissects`. This state variable is used by the dissection enumerator for storing dissections such that the painter can access them directly instead of interpreting their (large) term representations.
- `finals :: Int -> Bool` (constructor `Finals`) holds a Boolean function on states.
- `finalsL :: Int -> String -> Bool` (constructor `FinalsL`) holds a Boolean function on states and constructor constants.
- `labels :: Int -> [String]` (constructor `Labels`) holds a function from states to sets of constructor constants.
- `fixPositions :: [[Int]]` (constructor `Fix`) holds a list of tree positions.
- `matrixU :: Int -> Int -> [(Int, Int)]` (constructor `Matrix`) holds a square matrix of lists of pairs of states. `Matrix` has two arguments: `states` and `matrixU`.
- `matrixL :: Int -> Int -> [[Int], [Int]]` (constructor `MatrixL`) holds a square matrix of lists of pairs of lists of states. `MatrixL` has two arguments: `states` and `matrixL`. Similar to `dissects`, `matrixU` and `matrixL` allow the painter to access their values directly instead of interpreting their (large) term representations.

- `theorems :: [Term a] (constructor Theorems)` holds the current set of theorems.
- `transitions :: Int -> [Int]` (constructor `Trans`) holds the transition function (mapping a state to its successor states) of an unlabelled transition system. `Trans` has two arguments: `states` and `transitions`.
- `transitionsL :: Int -> String -> [Int]` (constructor `TransL`) holds the transition function (mapping a state and a label to their successor states) of labelled transition system. `TransL` has two arguments: `states` and `transitionsL`.

Each of the following Haskell functions evaluated by the simplifier employs some of the above state variables:

- `gauss`: the Gaussian algorithm for solving linear equations;
- `bisim`: computation of bisimilar states of a labelled transition system by table filling [7];
- `nerode`: computation of behaviorally equivalent states of a deterministic Moore or Mealy automaton by table filling [7];
- `optimize`: optimization of an OBDD by enumerating (in reverse lexicographic order) its variable orderings;
- `parse`: running an LR(1) parser with goto and action tables;
- `permute`: enumeration of the variable orderings of a DNF and adaptation of the DNF to the orderings;
- `postflow`: *program verification* by the backward propagation of a postcondition through the flowgraph of an iterative program,
- `stateflow`: *global model checking* by the backward propagation of state sets through the flowgraph that represents a μ -calculus formula (in contrast, *local* model checking amounts to proofs by induction and coinduction, see [LOCMODAL](#) and [LOCTRANS](#));
- `subsflow`: *program interpretation* by the forward propagation of sets of substitutions through the flowgraph of an iterative program.

Here are the details on how these functions are called and executed:

- `bisim`: Enter an axiom $trans = (t, u)$ for the state set and the transition function of a labelled transition system. t must be a list of integers. u must be a list of triples consisting of a state, a constructor constant denoting a label and the list of respective successor states (see [COIN1](#), [COIN2](#), [CYCLE](#)). Enter `bisim(trans)` and rewrite `trans`. Deselect selected subtrees. Simplify the entire displayed tree step by step. Since the simplification steps lead to pictorially representable results (here: triangular matrices), they may be executed from the painter window so that the change of results may be viewed directly in terms their pictorial representations.
- `gauss`: Select the widget interpreter *linear equations* (see [Widgets, graphs, and turtle actions](#)), enter a term of the form $lin(t)$ where t is a conjunction of linear equations and simplify $lin(t)$ step by step.

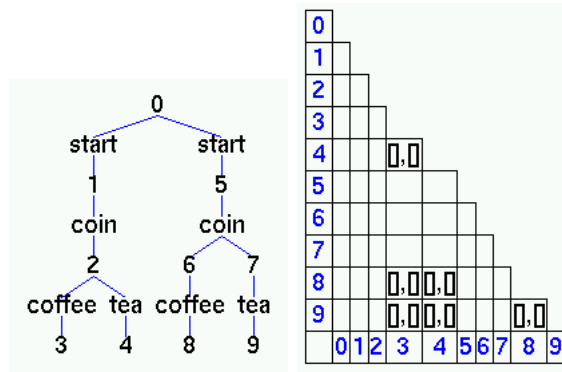


Figure 23.3: A run of *bisim* on the left LTS (see [COIN1](#)) results in the state equivalence on the right: only the sink states 3,4,8 and 9 are bisimilar.

	x	y	z	=
1	10.0	5.0	-2.0	1.0
2	3.0	-8.0	-1.0	9.0
3	1.0	-1.0	5.0	12.0

	x	y	z	=
1	1.0			0.999994
2		1.0		-0.999999
3			1.0	2.0

Figure 23.4: Two snapshots of a run of *gauss* on the conjunction $(10x) + (5y) - (2z) = 1 \wedge (3x) - (8y) - z = 9 \wedge x - y + (5z) = 12$ of linear equations (see [GAUSS1](#))

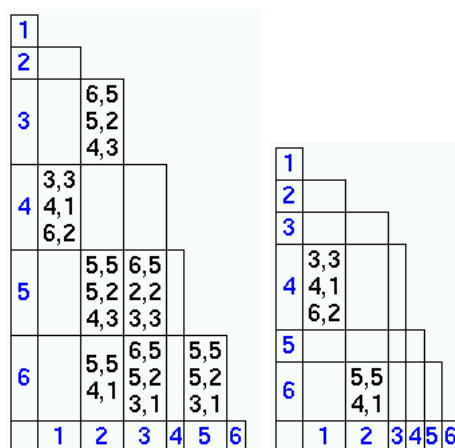


Figure 23.5: Two snapshots of a run of *nerode* on [AUTO1](#)

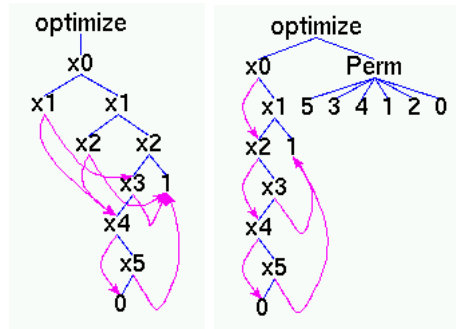


Figure 23.6: Two snapshots of a run of *optimize* on **OBDD6**

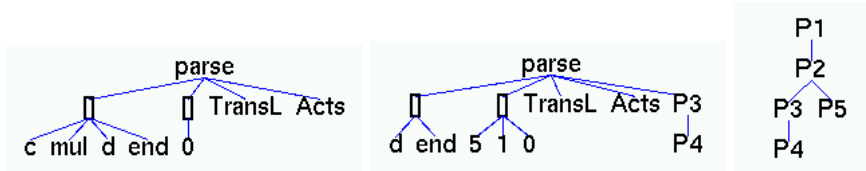


Figure 23.7: Three snapshots of a run of *parse* on **TRANSACTS**. The three arguments denote the input, the LR stack and the generated syntax tree.

- *nerode*: Enter axioms $trans = (t, u)$ for the state set and the transition function and $finals = v$ for the output function of a deterministic Moore or Mealy automaton. t must be a list of integers. u must be a list of triples consisting of a state, a constructor constant denoting an input symbol and the respective successor state. v must be a list of states (in the case of a Moore automaton) or pairs consisting of a state and a constructor constant denoting an input symbol (in the case of a Mealy automaton; see **AUTO1**). Enter $nerode1(trans, finals)$ (in the case of a Moore automaton) or $nerode2(trans, finals)$ (in the case of a Mealy automaton), rewrite $trans$ and $finals$ and proceed as in the case of *bisim*.
- *optimize*: Enter a term of the form $optimize(t)$ where t is an OBDD. Simplify the entire displayed tree step by step. t is changed towards an optimal OBDD. The argument of *Perm* (see above) denotes the actual variable ordering of the OBDD.
- *parse*: Enter axioms $trans = (t, u)$ for the state set and the goto table and $acts = v$ for the action table of an LR(1) grammar. t must be a list of integers. u must be a list of triples consisting of a state, a constructor constant denoting a grammar symbol and the respective successor state. v must be a list of triples consisting of a state, a constructor constant denoting a terminal symbol and a list of constructor constants denoting a grammar rule (**TRANSACTS**). Given a list *input* of terminal grammar symbols (constructor constants), enter $parse(input, [0], trans, acts)$, rewrite $trans$ and $acts$ and proceed as in the case of *bisim*.
- *permute*: Enter a term of the form $permute(t)$ where t is a DNF (see above). Simplify the entire displayed tree step by step. $permute(t)$ is changed to $permute(t, u, t')$ where t' is the DNF obtained

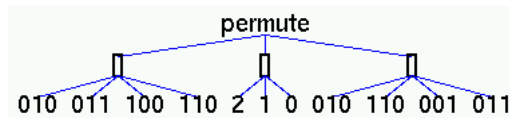


Figure 23.8: A snapshot of a run of *permute* on **DNF1**

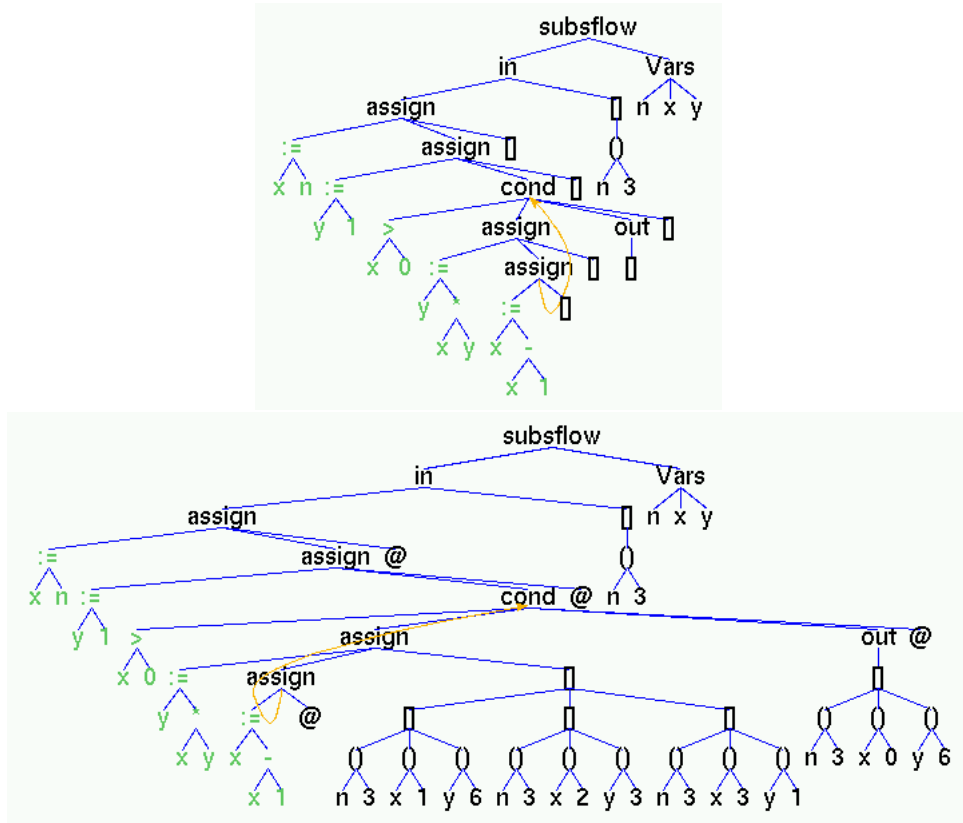


Figure 23.9: Two snapshots of a run of *postflow* on the factorial program: $x := n; y := 1; \text{while } x > 0 \ \& \ \text{fact}(n) = \text{fact}(x) * y \ \text{do } y := x * y; x := x - 1 \ \text{od}$. The invariant $\text{fact}(n) = \text{fact}(x) * y$ had to be added to the loop condition for ensuring that *postflow* terminates.

from t by rearranging the variables according to the permutation u .

- *postflow*: Enter axioms $\text{flow} = \text{bool}(X_1 = t_1 \ \& \ \dots \ \& \ X_n = t_n)$ for the flowgraph F of an iterative program P and $\text{post} = \text{bool}(t)$ for a postcondition of P . $X_1 = t_1 \ \& \ \dots \ \& \ X_n = t_n$ must be a set of regular equations representing F . t must be a formula over the current signature (see **FACTPOST**). Enter $\text{postflow}(\text{flow}, \text{post})$, rewrite flow and post and simplify the entire displayed tree step by step. The commands and tests in F are colored in green. In the snapshots depicted above, some intermediate postconditions are hidden behind @. Having starting out from the postcondition post of P (= valuation of the *out* node of F), *postflow* will — if it terminates — end up with a formula pre at the *in* node of F such that pre implies post .

The postcondition valuation (of the nodes) of F changes with each simplification step. The valuation is stable when F cannot be simplified any more. This simplification includes the reduction of intermediate postconditions by applying current theorems in the non-greedy matching mode (see the **← unify/match** button). Hence *postflow* depends on the set of current theorems and, conversely, the set can be modified between simplification steps and thus tailored to their respective results. The incorporation of theorem application into simplification is necessary here. Otherwise *postflow* may not notice that the postcondition valuation of F has become stable and thus would not terminate. The point is that a valuation is stable if it is equal to the preceding one, but, in the case of postconditions, equality means logical equivalence! The application of tailor-made theorems to intermediate postconditions may turn them into “normal forms” so that logical equivalence amounts to syntactic equality. For running **FACTPOST**, we used the theorems of **FACTTHS**.

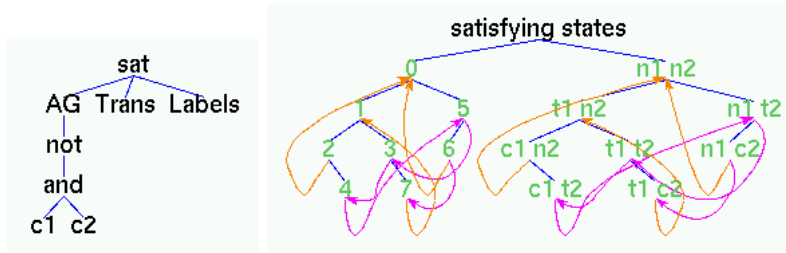


Figure 23.10: Two snapshots of a run of *sat* on **CTLMUTEX1**

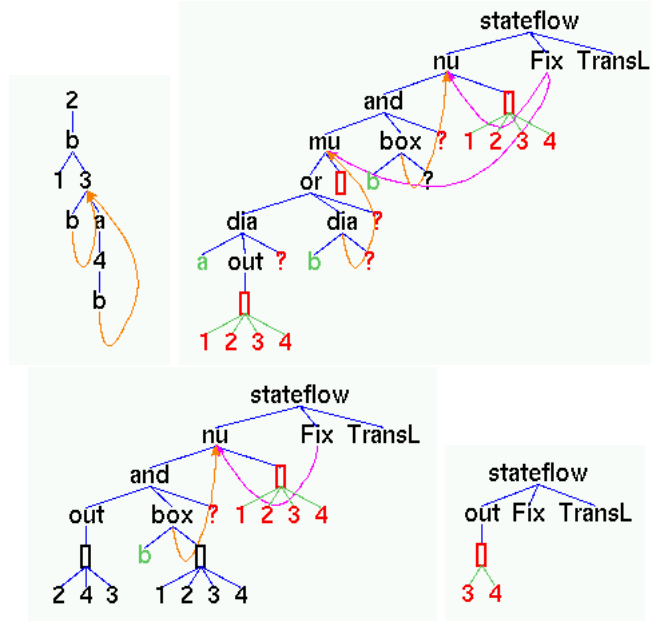


Figure 23.11: Three snapshots of a run of *stateflow* on the transition system **TRANS1** (see upper left corner) and the modal formula $\nu x.(\mu y.(\langle a \rangle true \vee \langle b \rangle y) \wedge [b]x)$

- *sat*: Enter axioms $ctl = v'$ for a CTL formula, $trans = (t, u)$ for the state set and the transition function of an unlabelled transition system and $labels = v$ for the labelling of states with atomic formulas. t must be a list of integers. u must be a list of pairs consisting of a state and the list of respective successor states. v must be a list of pairs consisting of a state and a list of constructor constants (see **CTLMUTEX1** and **CTLMUTEX2**). Enter $sat(ctl, trans, labels)$, rewrite ctl , $trans$ and $labels$ and simplify the entire displayed tree. The result is a term of the form *satisfying states*(g_1, g_2) where g_1 and g_2 are graph representations of the transition function u . If a node of g_1 is labelled with a state s , the corresponding node of g_2 is labelled with the atomic formulas satisfied by s . If the node is colored in green, s satisfies ctl , otherwise s does not satisfy ctl .
- *stateflow*: Enter axioms $flow = bool(X_1 = t_1 \& \dots \& X_n = t_n)$ for the flowgraph F of a μ -calculus formula, $trans = (t, u)$ for the state set and the transition function of a labelled transition system and $labels = v$ for the labelling of leaves of F with atomic formulas. $X_1 = t_1 \& \dots \& X_n = t_n$ must be a set of regular equations representing F . t must be a list of integers. u must be a list of triples consisting of a state, a constructor constant denoting an action and the list of respective successor states. v must be a list of pairs consisting of a state and a list of constructor constants (see **TRANS1** and **TRANS2**). Enter $stateflow(flow, trans, labels)$, rewrite $flow$, $trans$ and $labels$ and simplify the entire displayed tree step by step. The actions in F are colored in green.

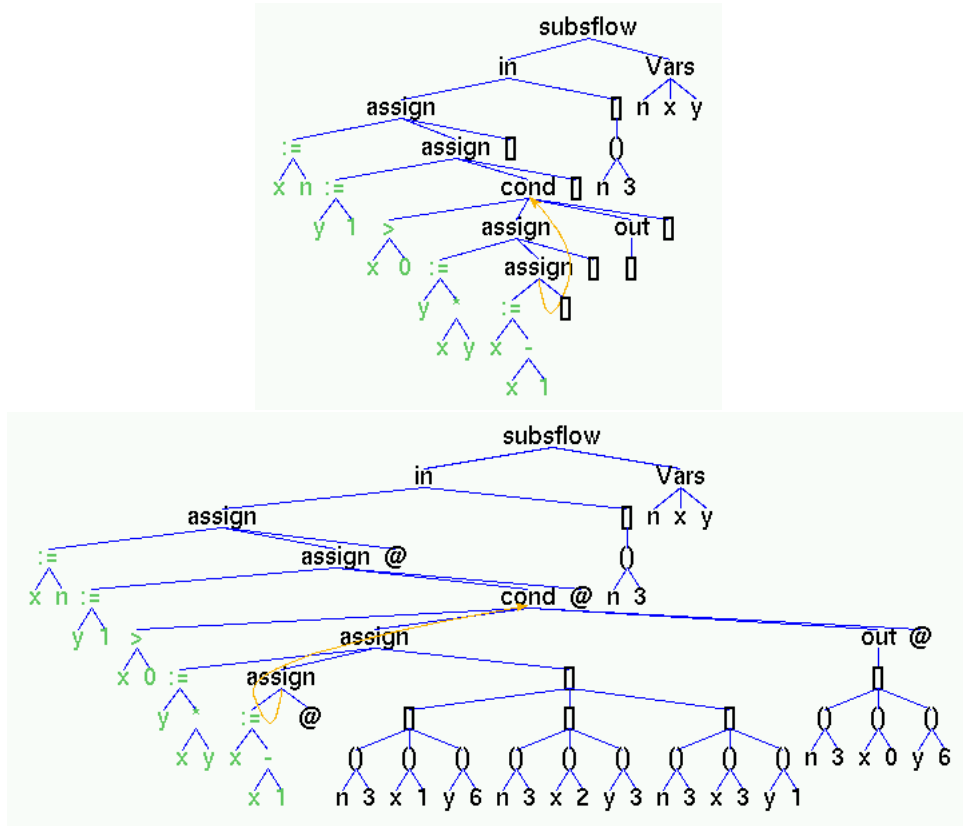


Figure 23.12: Two snapshots of a run of *subsflow* on the factorial program:

$$x := n; y := 1; \text{ while } x > 0 \text{ do } y := x*y; x := x-1 \text{ od}$$

The state set valuation (of the nodes) of F changes with each simplification step. The valuation is stable when F cannot be simplified any more. The arcs emanating from Fix point to the (fixpoint) subformulas whose state set value is not yet stable. Once such a formula obtains a stable value, it is replaced by an *out* node carrying the value.

- *subsflow*: Enter axioms $flow = bool(X_1 = t_1 \& \dots \& X_n = t_n)$ for the flowgraph F of an iterative program P and $subs = t$ for an initial set of substitutions of program variables. $X_1 = t_1 \& \dots \& X_n = t_n$ must be a set of regular equations representing F . t must be a list of pairs consisting of a (program) variable x and a term over the current signature (see **FACTSUB**). Enter $subsflow(flow, subs)$, rewrite $flow$ and $subs$ and simplify the entire displayed tree step by step. The commands and tests in F are colored in green. In the snapshots depicted above, some intermediate postconditions are hidden behind @. The substitution set valuation (of the nodes) of F changes with each simplification step. The valuation is stable when F cannot be simplified any more. (see [8]).

The following rewriting steps among the above-mentioned ones transform state variable names into state variable values (see Fig. 23.1):

- If t can be simplified to a list of triples consisting of an integer, a constructor constant and a list of constructor constants, then the application of the axiom $acts = t$ to $acts$ results in the reduct $Actions f$ where $f :: Int \rightarrow String \rightarrow ActLR$ represents t .
- If t can be simplified to a list of integers, then the application of the axiom $finals = t$ to $finals$ results in the reduct $Finals f$ where $f :: Int \rightarrow Bool$ represents t .

- If t can be simplified to a list of pairs consisting of an integer and a constructor constant denoting an input symbol, then the application of the axiom $finals = t$ to $finals$ results in the reduct $FinalsL f$ where $f :: Int \rightarrow String \rightarrow Bool$ represents t .
- If t can be simplified to a list of pairs consisting of an integer and a list of constructor constants, then the application of the axiom $labels = t$ to $labels$ results in the reduct $Labels f$ where $f :: Int \rightarrow [String]$ represents t .
- If t can be simplified to a list of integers and u can be simplified to a list of pairs consisting of an integer and a list of integers, then the application of the axiom $trans = (t, u)$ to $trans$ results in the reduct $TransL f$ where $f :: Int \rightarrow [Int]$ represents t .
- If t can be simplified to a list of integers and u can be simplified to a list of triples consisting of an integer, a constructor constant and a list of integers, then the application of the axiom $trans = (t, u)$ to $trans$ results in the reduct $TransL f$ where $f :: Int \rightarrow String \rightarrow [Int]$ represents t .

24 Examples

	axioms	theorems	conjectures	derivation or proof term
alignments	ALIGN			
arithmetic and simplifications	NAT WIRTH MOD	NATTHS	NATCONJS EVEN POLY GAUSS1 GAUSS2 GAUSS3 REGS1 REGS2 SET1 SIMPL1 SIMPL2 SIMPL3 SIMPL4 SIMPLTERM WIRTH2 PRIMS	ASSOCPROOF COMMPROOF DIVPROOF EVPROOF proof term EVODPROOF EXPPROOF FIBPROOF POTPROOF proof term WIRTHPROOF proof term
binary trees	BTREE REPMIN		BTREE1 REPMIN1	REPMIN1PROOF
Boolean functions	OBDD SWAP		DNF1 DNF2 DNF3 PARITY QUAD1 QUAD2 QUAD3 OBDD1 OBDD2 OBDD3 OBDD4 OBDD5 OQUAD1 OQUAD2 OQUAD3	
concept formation	FRUIT	FRUITTHS	FRUITCONJS	FRUIT1PROOF FRUIT2PROOF FRUIT3PROOF FRUIT4PROOF
finite lists	LIST LISTEVAL PARTN	LISTTHS	LISTCONJS FILTER MAP SPLIT MERGE SORT FLATTEN ZIP1 ZIP2 ZIP3 ZIP4	PARTPROOF proof term PARTPROOF2 proof term SORTPROOF PERMPROOF MERGEPROOF PARTNPROOF LGOK
imperative programs	FACTPOST FACTSUBS	FACTTHS	POSTFLOW SUBSFLOW PROG1 PROG2 PROG3	


	axioms	theorems	conjectures	derivation or proof term
infinite sequences	STREAM	STREAMTHS	STREAMCONJS	FAIRBLINK proof term BLINKZIP proof term EVENSZIP proof term ITERITER2 proof term ITERLOOP proof term ODDSEVENS proof term MAPFACT proof term MAPITER1 proof term MAPLOOP proof term MAPLOOP0 proof term NATLOOP proof term REVREV proof term ZIPODDS proof term
modal formulas	CTLMUTEX1 CTLMUTEX2 TRANS1 TRANS2 LOCTRANS LOCMODAL		CTLSAT STATE- FLOW LOC- MOD1	LOCMOD1PROOF
needed narrowing	NEED		NEED1	NEEDPROOF
parser	TRANSACTS LR		PARSE PARSERUN	LRRUN
permutations and partitions	LOG		PART1 PART2 PART3 PART4 PERM1 PERM2 PERM3	PERM3PROOF
pictures			RTC TURTLE SEXTAGON OCTAGON LEAF PIX SIX STAR STAR2 VIBE	
relational algebra	REL		REL1	
stacks	STACK STACKIMPL STACKIMPL2	STACKTHS	STACKCONJS	TOPEMPTY TOPPUSH POPEMPTY POPPUSH PUSHCOMP PUSHCOMP2 UPDEQ
state equivalence	AUTO1 COIN1 COIN2 CYCLE		NERODE BISIM	

	axioms	theorems	conjectures	derivation or proof term
transition systems	ACCOUNT BOTTLE BOTTLEF BOTTLEAC ECHO HANOI KNIGHT MUTEX PHIL PUZZLE QUEENS QUEENSF ROBOT ROBOTF ROBOTACTS ROBOTACTSF		ACCOUNT1 BOTTLE1 BOTTLEAC1 ECHO1 ECHO2 HANOI1 KNIGHT1 MUTEX1 PHIL1 PHIL2 PUZZLE1 QUEENS1 QUEENSF1 ROBOT1 ROBOTF1 ROBOTACTS1 ROBOTACTSF1	ACCOUNTSOL BOTTLESOLS ECHO2PROOF KNIGHTSOLS MUTEX1PROOF PHIL2PROOF QUEENS4 QUEENS5 ROBOTSOL ROBOTSOLS ROBOTSOLS2 ROBOT2PROOF

25 Widgets, graphs, and turtle actions

Built on top of the Tk interface module `tk`, the module `Epaint` provides features for creating and editing pictorial term representations. These are displayed in the painter window of a solver when the **paint** button is pushed. The scroll region of this window is adapted automatically to the displayed picture.

`Expander2` provides several widget interpreters and combinators thereof, which recognize paintable terms and transform them into pictorial representations. The actual widget interpreter can be selected from the *pict type* menu. Some built-in axiom files (see **Examples**) and enumerators are automatically associated with a widget interpreter: `ALIGN` and the alignment and the palindrome enumerator with *alignment*, the dissection enumerator with *rectangles*, the partition enumerator with *partition*, `QUEENS` with *matrix solution*, `BOTTLE`, `BOTTLEF`, `ROBOTF` and `ROBOTACTSF` with *polygon* and `ROBOT` and `ROBOTACTS` with *polygon solution*. The default interpreter is *matrix*.

The basic elements of pictorial term representations are called **widgets**. A **picture** is a list of *widgets*. A **widget graph** G is a pair consisting of a list of *widgets* $= [w_1, \dots, w_n]$ and a list *arcs* $= [s_1, \dots, s_n]$ of sublists of $[1, \dots, n]$. k is element of s_i iff an arc leads from w_i to w_k . For editing widget graphs, see Fig. 25.3 and the **paint** button. A displayed graph is saved to a file F (in eps format or as a Haskell object of type $(Nodes, Arcs)$; see below) by writing F into the *save to:* field and pushing the  key while the cursor is in this field.

Widgets are Haskell objects of the following type:

```
data Widget_ = Circ Color Float Float Float | File_ String |
             Gif String Float Float | Path Color [(Float,Float)] |
             PathS Color [(Float,Float)] | Point Color Float Float |
             Poly Color [(Float,Float)] |
             PolyC Color Float Float [Float] [Float] Float |
             Rect Color Float Float Float Float Float |
             Snow Color Int Float Float Float Float |
             Text_ Color Float Float [String] |
             Tree Color Color (Term (String,Float,Float)) |
             Tria Color Float Float Float Float |
             Turtle Color Float Float Float [TurtleAct]

type Nodes = [Widget_]

type Arcs = [[Int]]

data TurtleAct = Move Float | Jump Float | Turn Float | Pict Nodes |
               Open Color | OpenP Color | OpenS Color | Close
```

The constructor `Pict` makes a picture into a **turtle action**. Given terms t_1, \dots, t_n that represent turtle actions, the widget interpreter *polygon* translates the list term $[t_1, \dots, t_n]$ into a single widget of the form

```
Turtle c x y a [a1, ..., an]
```

(see below). Each widget interpreter is a Haskell function of type

```
Term String -> Maybe Picture.
```

In the sequel, we list, for each interpreter, the terms it recognizes and describe the pictures displayed on the canvas of a painter.

Given a file F that contains a Haskell object G of type $(Nodes, Arcs)$, all widget interpreters translate the term $file(F)$ into the corresponding widget graph denoted by G .

Given a file F and a picture term t , simplifying the term $file(F, t)$ means translating t into a Haskell object of type $(Nodes, Arcs)$, saving it to F and replacing $file(F, t)$ by $file(F)$.

alignment recognizes syntax trees generated by the grammar G_1 or G_2 of section [Alignments and palindromes](#) are displayed as horizontal alignments.

linear equations: A term of the form $p_1 = r_1 \& \dots \& p_n = r_n$ where p_1, \dots, p_n are polynomials and r_1, \dots, r_n are real numbers is interpreted as a system of linear equations and displayed as the corresponding matrix of coefficients. The variables occurring in the equations must be part of the current signature (see [GAUSS1](#)).

matrix:

- `Matrix f ss` and `MatrixL f ss` are displayed as triangular matrices (see [Simplifications](#)).
- **Unlabelled transition graphs** are terms generated by the grammar

```
stateTerm -> state(stateTerm) | position of a state node
state      -> integer number
```

They are displayed as the corresponding adjacency matrices.

- **Labelled transition graphs** are terms generated by the grammar

```
stateTerm -> state(labelTerm) | position of a state node
labelTerm -> label(stateTerm)
state      -> integer number
label      -> constructor of the current signature
```

They are displayed as the corresponding adjacency matrices.

- A term of the form $pix((x_1, y_1), \dots, (x_n, y_n))$ where $x_1, \dots, x_n, y_1, \dots, y_n$ are integers is displayed as a matrix of pixels. Pixels at positions $(x_1, y_1), \dots, (x_n, y_n)$ are red. Pixels at other positions are invisible.
- A term of the form $pix((x_1, y_1, c_1), \dots, (x_n, y_n, c_n))$ where $x_1, \dots, x_n, y_1, \dots, y_n$ are integers and c_1, \dots, c_n are pixels is displayed as a matrix of pixels. The pixel at position (x_i, y_i) , $1 \leq i \leq n$, has color c_i . Pixels at other positions are invisible.

- A term of the form $[b_1, \dots, b_n]$ where b_1, \dots, b_n are words over $\{0,1,\#\}$ of the same length is interpreted as a DNF and displayed as the equivalent Karnaugh diagram.
- A collection $C((x_1, y_1), \dots, (x_n, y_n))$ (see **Built-in signature**) where x_1, \dots, x_n are integers and y_1, \dots, y_n are integers or constructor constants is displayed as the matrix that represents the Boolean function, which maps (x_i, y_i) , $1 \leq i \leq n$, to *True* and other pairs to *False*.
- A collection $C((x_1, y_1, t_1), \dots, (x_n, y_n, t_n))$ where x_1, \dots, x_n are integers, y_1, \dots, y_n are integers or constructor constants and t_1, \dots, t_n are single terms or lists of terms is displayed as a matrix that represents the partial function, which maps (x_i, y_i) , $1 \leq i \leq n$, to t_i . Several elements of t_i are listed vertically.
- A collection $C(t_1, \dots, t_n)$ such that all t_i are neither collections nor tuples is displayed as a horizontal listing of t_1, \dots, t_n . If t_i is of the form $f(t)$, then f is regarded as a column name and written in red. t is displayed below f . If t is a collection $C(u_1, \dots, u_k)$, then the terms u_1, \dots, u_k are listed vertically below f_i . For instance, f_1, \dots, f_n may represent places of an algebraic Petri net. Then $f_1(t_1) \dots f_n(t_n)$ denotes a marking of f_1, \dots, f_n (see **Built-in signature**). If $f(t)$ is of the form $pile(x, h)$ with natural numbers x and h , then the interpretation of $f(t)$ by *polygon* (see below) is displayed at the respective list position.

matrices recognizes the maximal subtrees of the displayed tree that are interpretable by *matrix* and combines the resulting pictures into a single one.

matrix solution recognizes each **solved formula**

$$\text{Any } Z_1 : x_1 = t_1 \& \dots \& \text{Any } Z_k : x_k = t_k \& \text{All } Z_{k+1} : x_{k+1} \neq t_{k+1} \& \dots \& \text{All } Z_n : x_n \neq t_n$$

and submits the terms t_1, \dots, t_n to *matrix*.

partition interprets the displayed tree t as a nested partition (see **Dissections and partitions**) and draws t as a dissection of a square that consists of colored rectangles each of which represents a leaf of t . Leaves at the same level of t correspond to equally colored rectangles.

polygon:

- $bush(n)$ is displayed as a bush fractal with depth n .
- $bush(n, c)$ is displayed as a bush fractal with depth n and outermost branch color c .
- $circ(r)$ is displayed as a circle with radius r .
- $circ(r, c)$ is displayed as a c -colored circle with radius r .
- $fern(n, d, r)$ is displayed as a fern fractal with depth n , apical delay d and internode elongation rate r (see [17], Section 5.3).
- $fern(n, d, r, c)$ is displayed as a fern fractal with depth n , apical delay d , internode elongation rate r and color c .
- $gif(f)$ is displayed as the contents in the file f (must be in gif format).
- $gras(n)$ is displayed as a gras fractal with depth n .

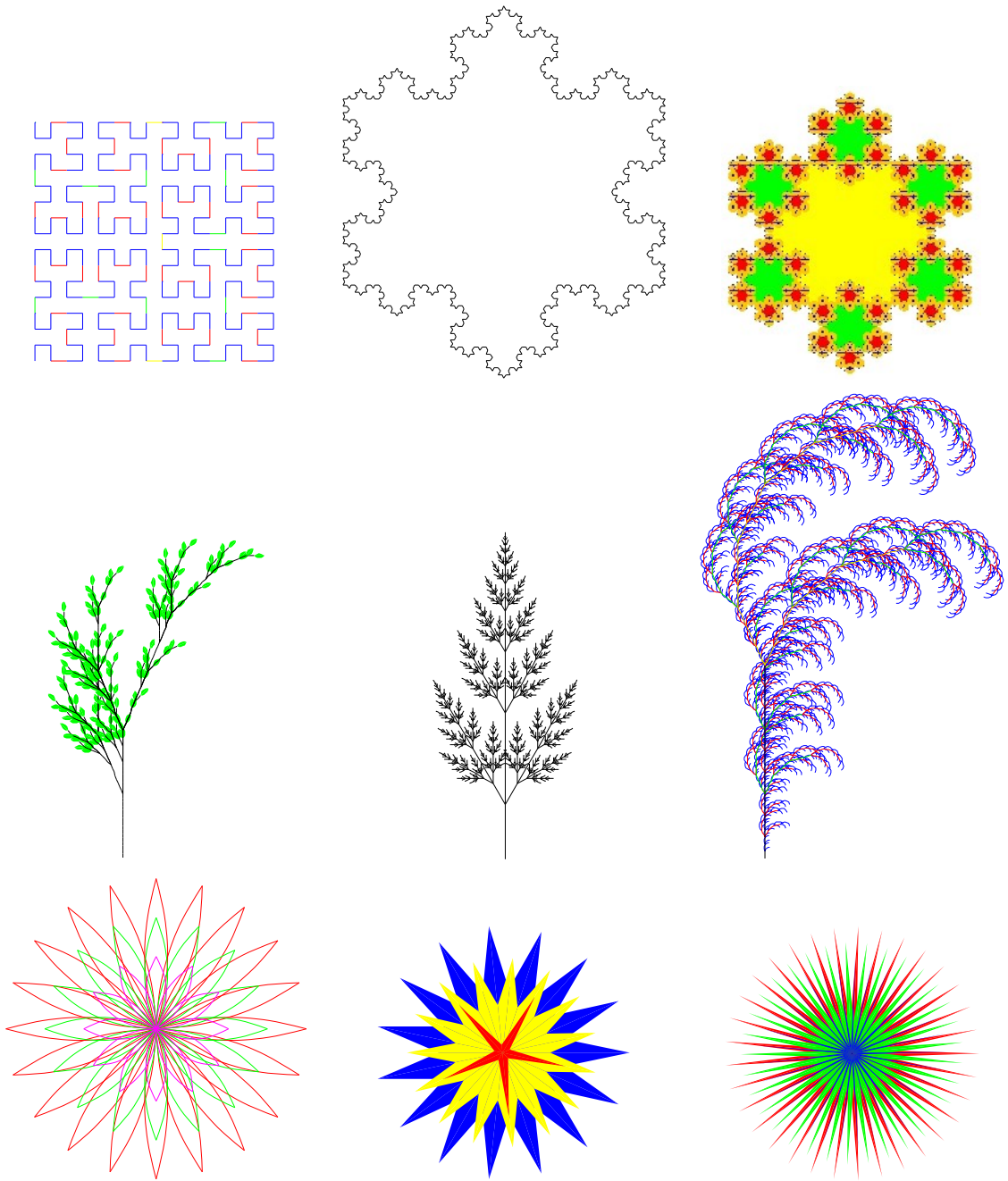


Figure 25.1: Pictorial representations of the terms $hilb(4, red)$, $koch(4)$, $snow(4, 6, yellow)$, $gras(4)$, $fern(12, 1, 1.5)$, $bush(4)$, $[leaf(20, magenta, 4), leaf(16, green, 3), leaf(12, blue, 2)]$, $[star(15, 8, 4, blue), T(12), star(15, 6, 3, yellow), star(5, 4, 0.5, red)]$ and $[star(33, 8, 2, red), T(5), star(66, 7, 2, green), T(5), star(33, 3, 0.7, blue)]$

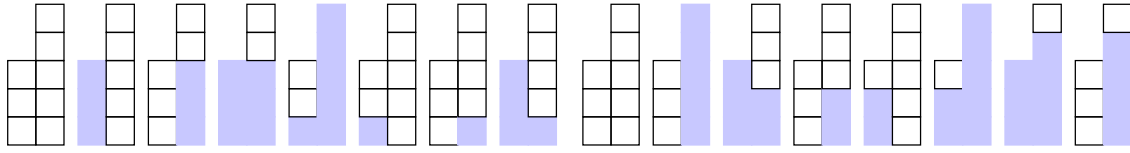


Figure 25.2: The two solutions of $bottle(0,0,[],EPS)\&PS = piles(reverse(EPS),3,5)$ obtained from applying axioms of **BOTTLE** (cf. [12, Section 3.3])

- $gras(n, c)$ is displayed as a gras fractal with depth n , green leaves and trunk color c .
- $hilb(n)$ is displayed as a smooth Hilbert curve of depth n .
- $hilb(n, c)$ is displayed as a Hilbert curve of depth n . The lines that build up the curve are colored differently, starting with color c for the lines at level 0. Colors are ordered. The lines at level $i \leq n$ are drawn in the i -th successor color of c .
- $koch(n)$ is displayed as the outline of a Koch snowflake with depth n .
- $koch(n, c)$ is displayed as the c -colored outline of a Koch snowflake with depth n .
- $leaf(n)$ is displayed as a blossom consisting of n identically shaped outlines of a leaf of length 1.
- $leaf(n, c)$ is displayed as a c -colored blossom consisting of n identically shaped leaves of length 1.
- $leaf(n, c, d)$ is displayed as a c -colored blossom consisting of n identically shaped leaves of length d .
- $path[(x_1, y_1), \dots, (x_n, y_n)]$ is displayed as a path that consists of lines connecting the points $(x_1, y_1), \dots, (x_n, y_n)$ in the given order.
- $pathS[(x_1, y_1), \dots, (x_n, y_n)]$ is displayed as a smooth path that consists of lines connecting the points $(x_1, y_1), \dots, (x_n, y_n)$ in the given order.
- $pile(x, h)$ is displayed as a container of height h that is filled with x units.
- $piles([(x_1, y_1), \dots, (x_n, y_n)], h_1, h_2)$ is displayed as n states of two containers c and d of height h_1 and h_2 , respectively. For all $1 \leq i \leq n$, (x_i, y_i) represents the state where c is filled with x_i units and d is filled with y_i units.
- $point(x, y)$ is displayed as a red point at position (x, y) . If two widgets w_1 and w_2 are connected by an arc while the right mouse button is pushed, a point p is introduced between w_1 and w_2 and a smooth path is drawn from w_1 via p to w_2 (see below).
- $poly([a_1, \dots, a_n][r_1, \dots, r_n])$ is displayed as a polygon with n vertices p_1, \dots, p_n such that for all $1 \leq i \leq n$, p_i is the point that is reached after the point $(r_i, 0)$ has been moved counter-clockwise $a_1 + \dots + a_i$ degrees around the center of the polygon.
- $poly([a_1, \dots, a_n][r_1, \dots, r_n], c)$ is displayed as a c -colored polygon with n vertices p_1, \dots, p_n such that for all $1 \leq i \leq n$, p_i is the point that is reached after the point $(r_i, 0)$ has been moved counter-clockwise $a_1 + \dots + a_i$ degrees around the center of the polygon.

- $rect(b, h)$ is displayed as a rectangle with breadth b and height h .
- $rect(b, h, c)$ is displayed as a c -colored rectangle with breadth b and height h .
- $snow(n, r)$ is displayed as a Koch snowflake with depth n and radius r .
- $snow(n, r, c)$ is displayed as a colored Koch snowflake with depth n and radius r . The triangles that build up the snowflake are colored differently, starting with color c for the two outermost triangles. Colors are ordered. The triangles at level $i \leq n$ are drawn in the i -th successor color of c .
- $star(n, r, r')$ is displayed as a star such that the maximum of r and r' is the peak radius r of the star and the minimum is the valley radius.
- $star(n, r, r', c)$ is displayed as a c -colored star such that the maximum of r and r' is the peak radius r of the star and the minimum is the valley radius.
- $text(s)$ is displayed as the string s .
- $text(s, c)$ is displayed as the c -colored string s .
- $text(s)$ is displayed as the string s .
- $text(s, c)$ is displayed as the c -colored string s .
- $textR(s)$ is displayed as the string s and enclosed in a rectangle with grey outlines.
- $textR(s, c)$ is displayed as the c -colored string s and enclosed in a rectangle with c -colored outlines.
- $tree(t)$ is displayed as the tree that represents t . t should not contain pointers. They will not be translated correctly.
- $tria(r)$ is displayed as an equilateral triangle with outer radius r .
- $tria(r, c)$ is displayed as a c -colored equilateral triangle with outer radius r .
- $[act_1, \dots, act_n]$ is displayed as the picture that a turtle draws when it starts at point $(0, 0)$ with an orientation of 0 degrees and executes the actions act_1, \dots, act_n sequentially. The turtle maintains a stack of states of the form (c, x, y, a, n) where c is a color, (x, y) is a position, a is an orientation and p is a number, which indicates whether (x, y) belongs to a path, a polygon or a smooth path. The terms that represent turtle actions read as follows:
 - B : The turtle move backwards, i.e. turns by 180 degrees.
 - C : The turtle pops the stack and returns to the new stack top, say s . Moreover, it repeats its moves from s to the popped state, say s' , and draws a c -colored path that connects the visited points. If $n = 2$, it closes the path by drawing a line from s' to s and fills the resulting polygon. If $n = 3$, it does not close, but smoothes the path. The same action takes places before a J -, O -, OP - or OS -action is performed.
 - $J(d)$: The turtle jumps d units.

- *L*: The turtle turns left by 90 degrees.
- *M*(*d*): The turtle moves *d* units.
- *O*: Given the stack top $s = (c, x, y, a, n)$, the turtle pushes (*black*, $x, y, a, 1$) on top of the stack.
- *O*(*c*): Given the stack top $s = (c', x, y, a, n)$, the turtle pushes ($c, x, y, a, 1$) on top of the stack.
- *OP*(*c*): Given the stack top $s = (c', x, y, a, n)$, the turtle pushes ($c, x, y, a, 2$) on top of the stack.
- *OS*(*c*): Given the stack top $s = (c', x, y, a, n)$, the turtle pushes ($c, x, y, a, 3$) on top of the stack.
- *R*: The turtle turn right by 90 degrees.
- *T*(*a*): The turtle turns right by *a* degrees.
- any term that is recognized by *polygon*: The turtle draws the resulting picture in its current orientation at its current position, which coincides with the center of each widget the picture consists of. While drawing the picture the turtle may leave this position, but when it is done, it will return to the starting point.

Most number arguments mentioned above are supposed to be integers. Only $a, a_1, \dots, a_n, d, r, r', r_1, \dots, r_n, x, x_1, \dots, x_n, y, y_1, \dots, y_n$ may be real numbers.

Terms representing colors must be of the form *t* or *light t* where *t* is of the form RGB *r g b* with numbers r, g, b between 0 and 255 or *t* is among *black, blue, cyan, green, grey, magenta, orange, red, white, yellow*.

polygon recognizes the maximal subtrees of the displayed tree that are interpretable by *polygon* and combines the resulting pictures into a single one.

polygon solution recognizes each **solved formula**

$$\text{Any } Z_1 : x_1 = t_1 \& \dots \& \text{Any } Z_k : x_k = t_k \& \text{All } Z_{k+1} : x_{k+1} \neq t_{k+1} \& \dots \& \text{All } Z_n : x_n \dots t_n$$

and submits the terms t_1, \dots, t_n to *polygon*.

rectangles interprets a term of the form $[(x_1, y_1, b_1, h_1), \dots, (x_n, y_n, b_n, h_n)]$ as a collection of rectangles r_1, \dots, r_n such that for all $1 \leq i \leq n$, (x_i, y_i) is the top-left corner, b_i the breadth and h_i is the height of r_i .

All currently available graph editing features and their state dependencies are shown in Fig. 25.3.

A displayed graph is always aligned to the top and the left of the painter's canvas. Hence a picture consisting of a single widget cannot be moved!

For drawing an arc from a widget to itself at least one other widget must be passed.

The remaining two sections deal with the alignment, palindrome, dissection and partition enumerators that can be called from the solver's **Trees menu**. Other enumeration algorithms may be added accordingly.

26 Alignments and palindromes

The alignment enumerator and the palindrome enumerator compute alignments between two string sequences [4] or within a single sequence [5], respectively. A development of the Haskell program for the former can be found in [12, Section 2.4].

After two sequences xs and ys of strings separated by blanks have been entered into the text field, the **alignment enumerator** asks for a constraint. There are three possibilities:

- **dna**. The alignment enumerator computes all syntax trees for $xs\#\text{reverse}(ys)$ according to the following grammar G_1 :

equal	: start	→ a start a	<i>for all strings a</i>
compl	: start	→ a start compl(a)	<i>for all strings a</i>
insert	: start	→ match s	<i>for all nonempty sequences s of strings</i>
delete	: start	→ s match	<i>for all nonempty sequences s of strings</i>
#	: start	→ #	
equal	: match	→ a start a	<i>for all strings a</i>
compl	: match	→ a start compl(a)	<i>for all strings a</i>

Again, *compl* is a function on strings that is defined by the actual axioms of **ALIGN**.

- **match**. The alignment enumerator computes all syntax trees according to G_1 that contain a maximal number of *equal*- or *compl*-nodes and a minimal number of *insert*- or *delete*-nodes.
- **local**. The alignment enumerator computes all syntax trees according to G_1 that contain a maximal local alignment, i.e. a path consisting of *equal*- or *compl*-nodes, and a minimal number of *insert*- or *delete*-nodes.

Following the assignment of complementary DNA bases, the function *compl* maps a to t , t to a , c to g , g to c and all other strings to $\#$. Corresponding axioms are loaded when the alignment or palindrome enumerator is called from the solver.

Given the sequences

$s_1 = actactgct,$	$s_2 = agatag,$
$s_3 = adfaaaaaa,$	$s_4 = aaaaaadfa,$

the trees in Fig. 26.1 are: (1) the only derivation tree of $s_1\#\text{reverse}(s_2)$ that meets the *match*-constraint, (2) the only derivation tree of $s_3\#\text{reverse}(s_4)$ satisfying the *match*-constraint and (3) the only derivation tree also of $s_3\#\text{reverse}(s_4)$ that meets the *local*-constraint.

After a sequence xs of strings separated by blanks has been entered into the text field, the **palindrome enumerator** computes syntax trees for xs according to the following grammar G_2 :

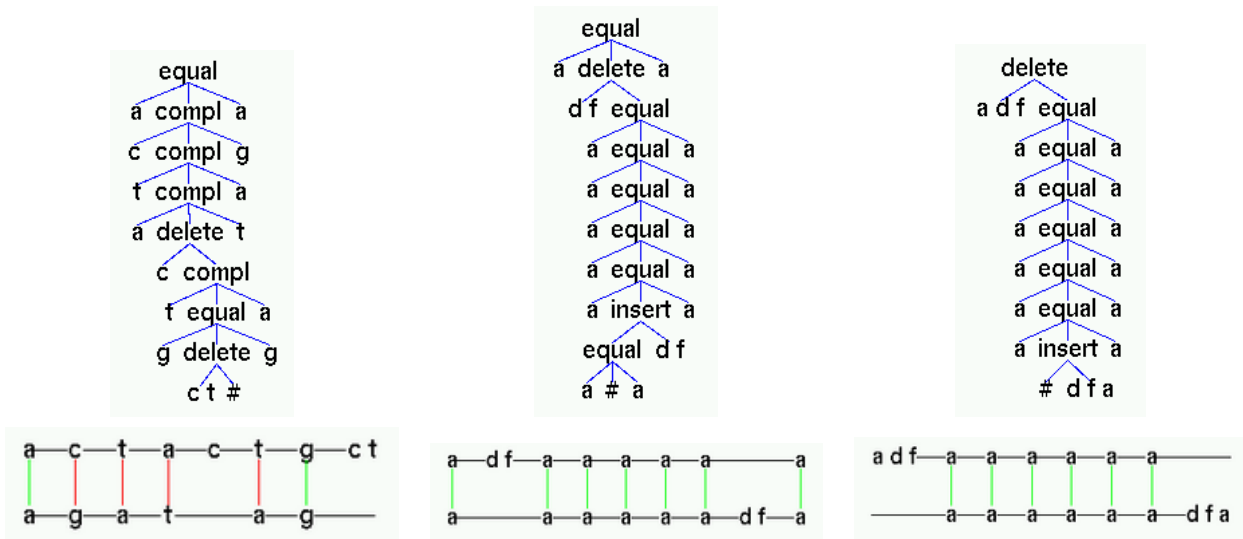


Figure 26.1: Alignment terms and their pictorial representations

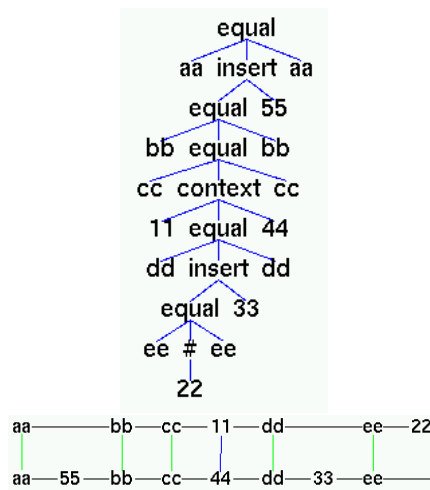


Figure 26.2: A palindrome term and its pictorial representation

equal	: start	→ a start a	for all strings a
compl	: start	→ a start $compl(a)$	for all strings a
context	: start	→ s match t	for all nonempty sequences s,t of strings
insert	: start	→ match s	for all nonempty sequences s of strings
delete	: start	→ s match	for all nonempty sequences s of strings
#	: start	→ s	for all sequences s of strings
equal	: match	→ a start a	for all strings a
compl	: match	→ a start $compl(a)$	for all strings a

$compl$ is a function on strings that is defined by the actual axioms of **ALIGN**. Only syntax trees with a maximal number of *equal*- or *compl*-nodes and a minimal number of *context*-nodes are returned.

27 Dissections and partitions

The **dissection enumerator** computes dissections of a rectangle and represents them directly without a detour via term representations. The underlying algorithm creates and modifies a triple of lists of top, left and inner subrectangles, respectively, such that dissection elements violating certain given constraints are discarded as early as possible (see [13, Section 4]).

Constraints. The dissection enumerator returns dissections of a given rectangle with breadth b and height h that satisfy one of the following atomic constraints or disjunctive or conjunctive combinations thereof:

constraint	holds true for all dissections
area(n)	consisting of $\lceil (b * h) / n \rceil$ subrectangles that cover at most n square units
area(m, n)	consisting of subrectangles that cover at least m and at most n square units
brick	consisting of subrectangles r such that for all $(x, y, b, h), (x, y', b', h')$ in r , $x = 0$, $y' \neq y + hy'$ or $y = y' + h'$ (see below)
eqarea(n)	consisting of n subrectangles that cover the same number of square units
factor(p)	consisting of subrectangles such that the breadth b and the height h of each subrectangle satisfy $b = p * h$ or $h = p * b$
hori	consisting of subrectangles whose height does not exceed the breadth
sizes(ns)	consisting of n in ns subrectangles
True	
vert	consisting of subrectangles whose breadth does not exceed the height

Formulas built up of atomic constraints are parsed according to the **Grammar**. Each constraint is translated into a triple of the Haskell type

$$((Int, Int, Int, Int) \rightarrow Bool, [Int], [(Int, Int, Int, Int)] \rightarrow [(Int, Int, Int, Int)] \rightarrow Bool).$$

The first component is a Boolean function that checks individual rectangles each of which is represented as a quadruple (x, y, b, h) where (x, y) is the top-left corner, b the breadth and h the height of the rectangle. The second component lists the admissible cardinalities of a dissection. The third component is a Boolean function that checks a relation between two parts of a dissection. Such a Boolean function is needed for expressing the brick constraint.

The **partition enumerator** computes nested partitions of a list and represents them as trees whose nodes are labelled with the nesting degrees of the respective subpartitions. Partitions with singleton subpartitions are not constructed.

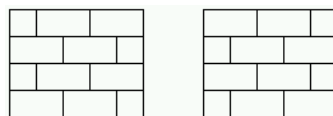


Figure 27.1: All dissections of a 5x4-rectangle that satisfy $area(1, 2) \& brick \& hori$: each dissection satisfies brick and consists of subrectangles covering 1 or 2 square units and satisfying hori (see below).

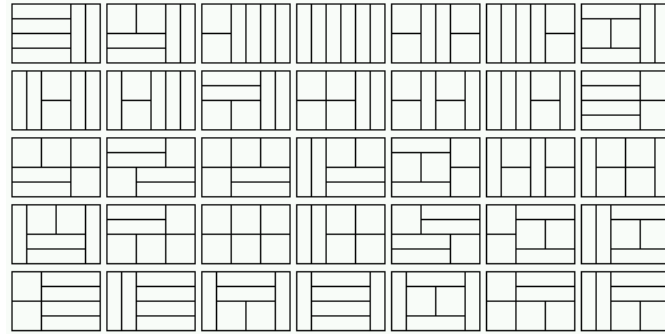


Figure 27.2: All dissections of a 6x4-rectangle that satisfy $eqarea(6)$: each dissection consists of 6 sub-rectangles that cover the same area.

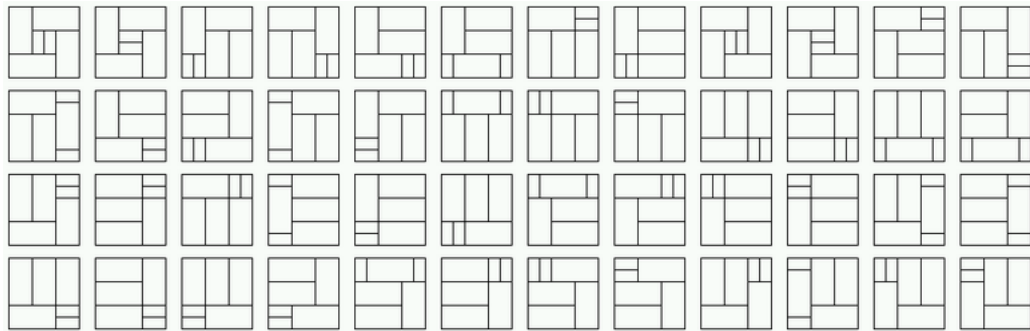


Figure 27.3: All dissections of a 6x6-rectangle that satisfy $sizes[6]\&factor(2)$: each dissection consists of 6 subrectangles and the breadth b and the height h of each subrectangle satisfy $b = 2h$ or $h = 2b$.

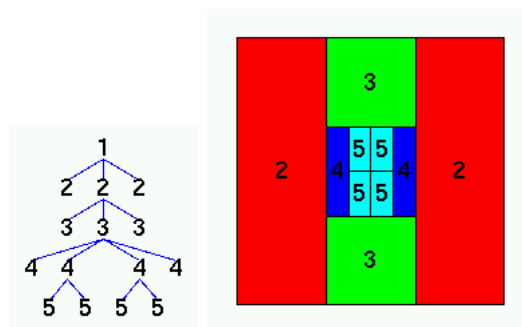


Figure 27.4: A nested partition satisfying $eqout\&sym$ of a list with 10 elements and its pictorial representation

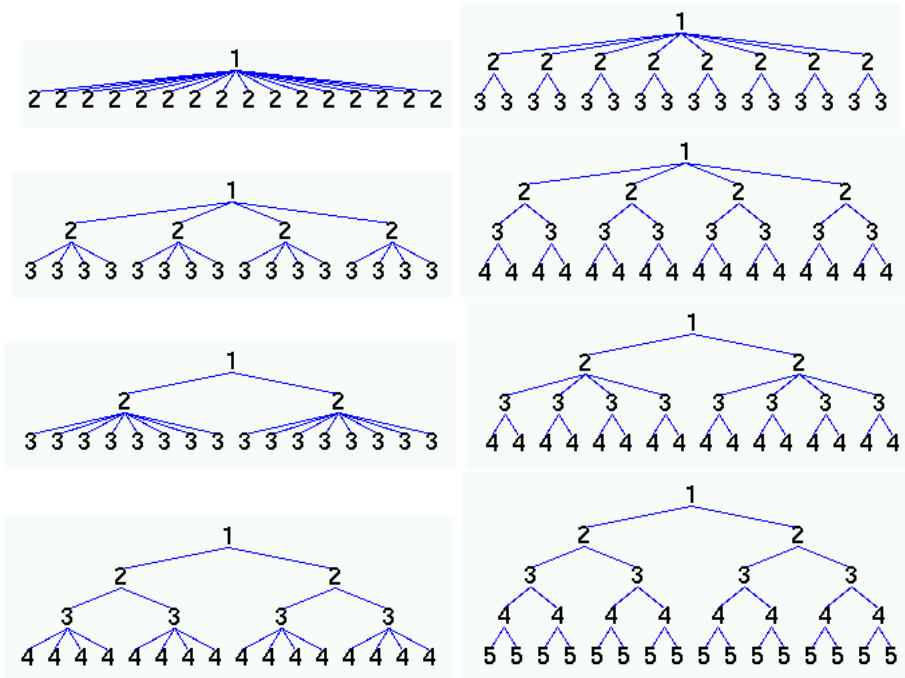


Figure 27.5: The nested partitions satisfying *balanced* of a list with 16 elements

Let s be a set with n elements and $parts(s)$ be the set of non-nested partitions of s with at least two elements. A Haskell program that computes the cardinality $f(n)$ of $parts(s)$ reads as follows:

```
f 0 = 1
f n = sum (map g [0..n-1])
      where g i = (fact n / (fact (n-i) * fact i)) * f i
fact i = product [1..i]
```

For the number $h(n)$ of *nested* partitions of s we obtain:

```
h 2 = 1
h n | n > 2 = sum [product [h (length p) | p <- ps] | ps <- parts s]
```

Hence, without meeting additional constraints, the number of trees representing nested partitions increases combinatorially with the number of leaves:

number of leaves	number of trees
5	45
6	197
7	903
8	4279
9	20793
10	103049

Constraints. The partition enumerator returns nested partitions that satisfy one of the following atomic constraints or disjunctive or conjunctive combinations thereof.

constraint	holds true for all trees
alter	whose nodes at even (odd) positions of a list s of all nodes with the same direct predecessor are leaves (inner nodes) unless s consists of leaves
bal	that are balanced
eqout	whose inner nodes with the same direct predecessor have the same outdegree
hei(n)	whose height is at most n
levmin	whose inner nodes at level n have an outdegree of at least n
levmax	whose nodes at level n have an outdegree of at most $\max(2, n)$
sym	that are vertically symmetric
out(m, n)	whose inner nodes at level $n > 1$ have an outdegree between m and n
True	

Formulas built up of atomic constraints are parsed according to the [Grammar](#). Each constraint is translated into a Boolean function of the Haskell type

$$Int \rightarrow [Term Int] \rightarrow Bool$$

that checks the constraint for each subtree st of a tree t in terms of the level of st within t (`Int` parameter) and the list of maximal proper subtrees of st (`[Term Int]` parameter).

Bibliography

- [1] S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. *Journal of the ACM*, 47: 776–822, 2000.
- [2] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, 35:677–691, 1986.
- [3] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. *A Maude Tutorial*. SRI International, 2000. URL <http://maude.csl.sri.com>.
- [4] R. Giegerich. A Systematic Approach to Dynamic Programming in Bioinformatics. Parts 1 and 2: Sequence Comparison and RNA Folding. Technical Report Report 99-05, University of Bielefeld, Technical Department, 1999.
- [5] R. Giegerich and C. Meyer. Algebraic Dynamic Programming. In *AMAST 2002*, pages 349–364. Springer LNCS 2422, 2002.
- [6] Andrew D. Gordon. Bisimilarity as a Theory of Functional Programming. *Theoretical Computer Science*, 228:5–47, 1999.
- [7] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 2nd edition, 2001.
- [8] J. B. Kam and J. D. Ullman. Global data flow analysis and iterative algorithms. *Journal of the ACM*, 23:158–171, 1976.
- [9] M. Müller-Olm, D. Schmidt, and B. Steffen. Model checking: A Tutorial Introduction. In *Proc. SAS '99*, pages 330–354. Springer LNCS 1694, 1999.
- [10] P. Padawitz. Expander2: Towards a Workbench for Interactive Formal Reasoning, . URL <http://ls5-www.cs.uni-dortmund.de/~peter/Expander2/Chiemsee.ps>.
- [11] P. Padawitz. Structured Swinging Types, . URL <http://ls5-www.cs.uni-dortmund.de/~peter/SST.ps.gz>.
- [12] P. Padawitz. Swinging Types At Work, . URL <http://ls5-www.cs.uni-dortmund.de/~peter/BehExa.ps.gz>.
- [13] P. Padawitz. Computing Rectangular Dissections. *J. Symbolic Computation*, 21:41–99, 1996.
- [14] P. Padawitz. Inductive Theorem Proving for Design Specifications. *J. Symbolic Computation*, 21: 41–99, 1996.
- [15] P. Padawitz. Swinging Types = Functions + Relations + Transition systems. *Theoretical Computer Science*, 243:93–165, 2000.

Bibliography

- [16] P. Padawitz. Formale Methoden des Systementwurfs, Course Notes, 2003. URL <http://ls5-www.cs.uni-dortmund.de/~peter/TdP96.ps.gz>.
- [17] P. Prunsinkiewicz and A. Lindenmayer. *The Algorithmic Beauty of Plants*. Springer, 1990.
- [18] G. Rozenberg and A. Salomaa, editors. *Beyond Words*, volume 3 of *Handbook of Formal Languages*. Springer, 1997.
- [19] M.-O. Stehr, J. Meseguer, and P. C. Ölveczky. Rewriting Logic as a Unifying Framework for Petri Nets. *Lecture Notes in Computer Science*, 2128:250+, 2001.