

# *From Modal Logic to (Co)Algebraic Reasoning*

Peter Padawitz, TU Dortmund, Germany

June 12, 2021

(actual version: <http://fdit-www.cs.tu-dortmund.de/~peter/CTL.pdf>)

Webseiten zu den zugehörigen Lehrveranstaltungen:

Funktionales und regelbasiertes Programmieren

Einführung in den logisch-algebraischen Systementwurf

## Contents

1. Kripke models and modal formulas	5
2. Kripke models in Expander2	19
3. Model checking as simplification <b>modal</b>	28
4. Model checking as term evaluation	33
5. Model checking as simplification	28
6. Specifications that import <b>modal</b> <b>micro</b>	37
<b>trans0</b>	43
7. Model checking as data flow analysis Examples	50
57	
8. Acceptors and regular expressions	69
9. State equivalence and minimization Examples	95
	101

10.	Simplifications	111
	evaluate	114
	base	123
11.	(Co)resolution and (co)induction on (co)predicates	136
12.	Man-wolf-goat-cabbage problem	142
13.	Railway crossing	150
14.	Mutual exclusion	161
15.	N-queens problem	193
16.	Robots	203
17.	Filling problem	207
18.	Elevator	214
19.	Towers of Hanoi	215
20.	Natural numbers	226
21.	Lists	234
22.	Streams	244
23.	Binary trees	265

24. Lazy evaluation	278
25. Further examples	287
26. Beispiele aus [13]	288
27. Literatur	390

## Kripke models and modal formulas

A **Kripke model**  $K = (Q, Lab, At, trans, transL, value, valueL)$  consists of

- a set  $Q$  of **states**,
- a set  $Lab$  of **labels** (actions, input, etc.),
- a set  $At$  of **atoms**,
- **transition relations**  $trans : Q \rightarrow \mathcal{P}(Q)$  and  $transL : Q \times Lab \rightarrow \mathcal{P}(Q)$ ,
- **value relations**  $value : At \rightarrow \mathcal{P}(Q)$  and  $valueL : At \times Lab \rightarrow \mathcal{P}(Q)$ .

## State and path formulas

are generated by the following CFG rules: Let  $S = \{sf, pf\}$  and  $V$  be an  $S$ -sorted set of variables.

$$sf \rightarrow at \mid val(at, lab) \quad \begin{matrix} \text{for all } at \in At \\ \text{and } lab \in Lab \end{matrix} \quad (1)$$

$$sf \rightarrow true \mid false \mid \neg sf \mid sf \vee sf \mid sf \wedge sf \mid sf \Rightarrow sf \quad (2)$$

$$sf \rightarrow EX\ sf \mid AX\ sf \mid \langle lab \rangle sf \mid [lab]sf \quad \text{for all } lab \in L \quad (4)$$

$$sf \rightarrow x \mid \mu x. sf \mid \nu x. sf \quad \text{for all } x \in V \quad (5)$$

$$sf \rightarrow EF\ sf \mid AF\ sf \mid EG\ sf \mid AG\ sf$$

$$sf \rightarrow sf\ EU\ sf \mid sf\ AU\ sf \mid sf\ EW\ sf \mid sf\ AW\ sf$$

$$sf \rightarrow sf\ ER\ sf \mid sf\ AR\ sf$$

$$pf \rightarrow at \mid at(lab) \quad \begin{aligned} &\text{for all } at \in At \\ &\text{and } lab \in L \end{aligned} \quad (6)$$

$$pf \rightarrow true \mid false \mid \neg pf \mid pf \vee pf \mid pf \wedge pf \mid pf \Rightarrow pf \quad (7)$$

$$pf \rightarrow next\ pf \mid \langle lab \rangle pf \mid [lab]pf \quad \text{for all } lab \in L \quad (8)$$

$$pf \rightarrow x \mid \mu x. pf \mid \nu x. pf \quad \text{for all } x \in V \quad (9)$$

$$pf \rightarrow F\ pf \mid G\ pf \mid pf\ U\ pf \mid pf\ W\ pf \mid pf\ R\ pf$$

$$pf \rightarrow pf \rightsquigarrow pf$$

## Assumptions

Both the local and global semantics of modal formulas defined below require that formulas resp. transition relations meet the following constraints:

- For every free variable  $x$  of a subformula  $\psi$  of a formula  $\varphi$ ,  $\varphi$  encloses a smallest abstraction  $\mu x.\vartheta$  or  $\nu x.\vartheta$  (see rules (5) and (9)) that encloses  $\psi$ .  
 $\implies$  *Abstractions can be evaluated hierarchically.*
- Every free occurrences of a variable  $x$  in  $\mu x.\varphi$  resp.  $\nu x.\varphi$  has **positive polarity**, i.e., the number of negation symbols of  $\mu x.\varphi$  resp.  $\nu x.\varphi$  above the occurrence is even.  
 $\implies$  *All logical operators are monotone.*
- $\rightarrow$  is **image finite**, i.e., for all  $q \in Q$  and  $lab \in Lab$ ,  $trans(q)$  and  $transL(lab)(q)$  are finite.  
 $\implies$  *All logical operators are  $\omega$ -bicontinuous.*  
 $\implies$  *Fixpoints can be computed incrementally as Kleene closures of the accompanying step functions (see below).*

## Local semantics

*The set of all paths of K*

$$\begin{aligned} \text{path}(K) =_{\text{def}} & \{p \in Q^{\mathbb{N}} \mid \forall i \in \mathbb{N} : p_{i+1} \in \text{trans}(p_i) \\ & \vee \exists \text{lab} \in \text{Lab} : p_{i+1} \in \text{transL}(p_i)(\text{lab}) \\ & \vee (\text{trans}(p_i) = \emptyset \wedge \\ & \forall \text{lab} \in \text{Lab} : \text{transL}(p_i)(\text{lab}) = \emptyset \wedge \\ & \forall k > i : p_i = p_k)\} \end{aligned}$$

*The set of all paths of K that start with  $q \in Q$*

$$\text{path}(K, q) =_{\text{def}} \{p \in \text{path}(K) \mid p_0 = q\}$$

Let  $q \in Q$ ,  $\text{at} \in \text{At}$ ,  $\text{lab} \in \text{Lab}$ ,  $b : V \rightarrow \mathcal{P}(Q)$  and  $c : V \rightarrow \mathcal{P}(\text{path}(K))$ .

*Validity of state formulas*

$$\begin{aligned} q \models \text{at} & \iff_{\text{def}} q \in \text{value}(\text{at}) \\ q \models \text{at}(\text{lab}) & \iff_{\text{def}} q \in \text{valueL}(\text{at}, \text{lab}) \\ q \models \text{true} & \quad \text{for all } q \in Q \cup \text{path}(K) \\ q \models \text{false} & \quad \text{for no } q \in Q \cup \text{path}(K) \\ q \models \neg \varphi & \iff_{\text{def}} q \not\models \varphi \end{aligned}$$

$q \models \varphi \vee \psi$	$\iff_{def} q \models \varphi \text{ or } q \models \psi$
$q \models \varphi \wedge \psi$	$\iff_{def} q \models \varphi \text{ and } q \models \psi$
$q \models \varphi \Rightarrow \psi$	$\iff_{def} q \models \neg\varphi \vee \psi$
$q \models x$	$\iff_{def} q \in b(x)$
$q \models \mu x.\varphi(x)$	$\iff_{def} q \in \text{least solution of } x = \varphi(x)$
$q \models \nu x.\varphi(x)$	$\iff_{def} q \in \text{greatest solution of } x = \varphi(x)$
$q \models EX\varphi$	$\iff_{def} \exists q' \in \text{trans}(q) : q' \models \varphi$
$q \models AX\varphi$	$\iff_{def} \forall q' \in \text{trans}(q) : q' \models \varphi$
$q \models \langle lab \rangle \varphi$	$\iff_{def} \exists q' \in \text{transL}(q)(lab) : q' \models \varphi$
$q \models [lab]\varphi$	$\iff_{def} \forall q' \in \text{transL}(q)(lab) : q' \models \varphi$
$q \models EF\varphi$	$\iff_{def} \exists p \in \text{path}(K, q) \exists i \in \mathbb{N} : p_i \models \varphi$
$q \models AF\varphi$	$\iff_{def} \forall p \in \text{path}(K, q) \exists i \in \mathbb{N} : p_i \models \varphi$
$q \models EG\varphi$	$\iff_{def} \exists p \in \text{path}(K, q) \forall i \in \mathbb{N} : p_i \models \varphi$
$q \models AG\varphi$	$\iff_{def} \forall p \in \text{path}(K, q) \forall i \in \mathbb{N} : p_i \models \varphi$
$q \models \varphi EU \psi$	$\iff_{def} \exists p \in \text{path}(K, q) \exists i \in \mathbb{N} : p_i \models \psi \wedge \forall j < i : p_j \models \varphi$
$q \models \varphi AU \psi$	$\iff_{def} \forall p \in \text{path}(K, q) \exists i \in \mathbb{N} : p_i \models \psi \wedge \forall j < i : p_j \models \varphi$

$$\begin{aligned}
 q \models \varphi \text{ } EW \text{ } \psi &\iff_{\text{def}} \exists p \in \text{path}(K, q) \forall i \in \mathbb{N} : p_i \models \psi \Rightarrow \forall j < i : p_j \models \varphi \\
 q \models \varphi \text{ } AW \text{ } \psi &\iff_{\text{def}} \forall p \in \text{path}(K, q) \forall i \in \mathbb{N} : p_i \models \psi \Rightarrow \forall j < i : p_j \models \varphi \\
 q \models \varphi \text{ } ER \text{ } \psi &\iff_{\text{def}} \exists p \in \text{path}(K, q) \forall i \in \mathbb{N} : p_i \models \psi \vee \exists j < i : p_j \models \varphi \\
 q \models \varphi \text{ } AR \text{ } \psi &\iff_{\text{def}} \forall p \in \text{path}(K, q) \forall i \in \mathbb{N} : p_i \models \psi \vee \exists j < i : p_j \models \varphi
 \end{aligned}$$

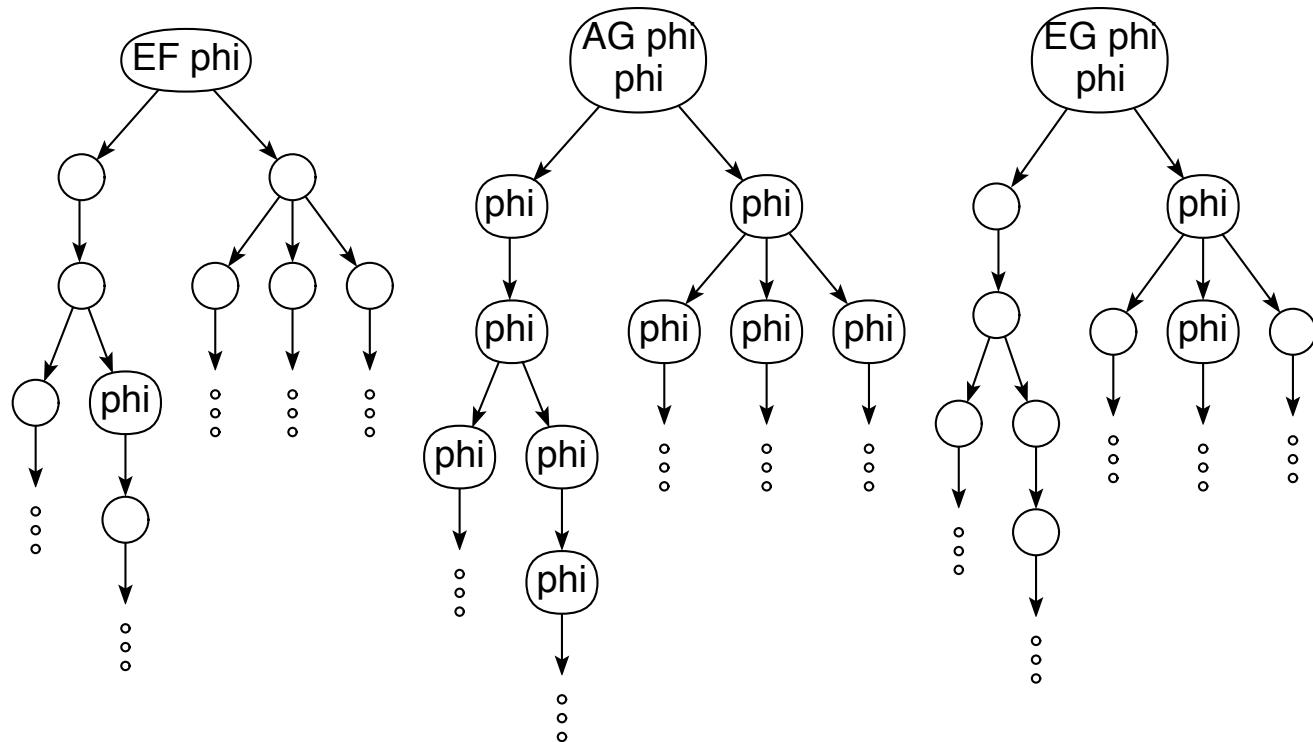


Fig. 1. Transition graphs whose roots satisfy  $EF\varphi$ ,  $AG\varphi$  and  $EG\varphi$ , respectively

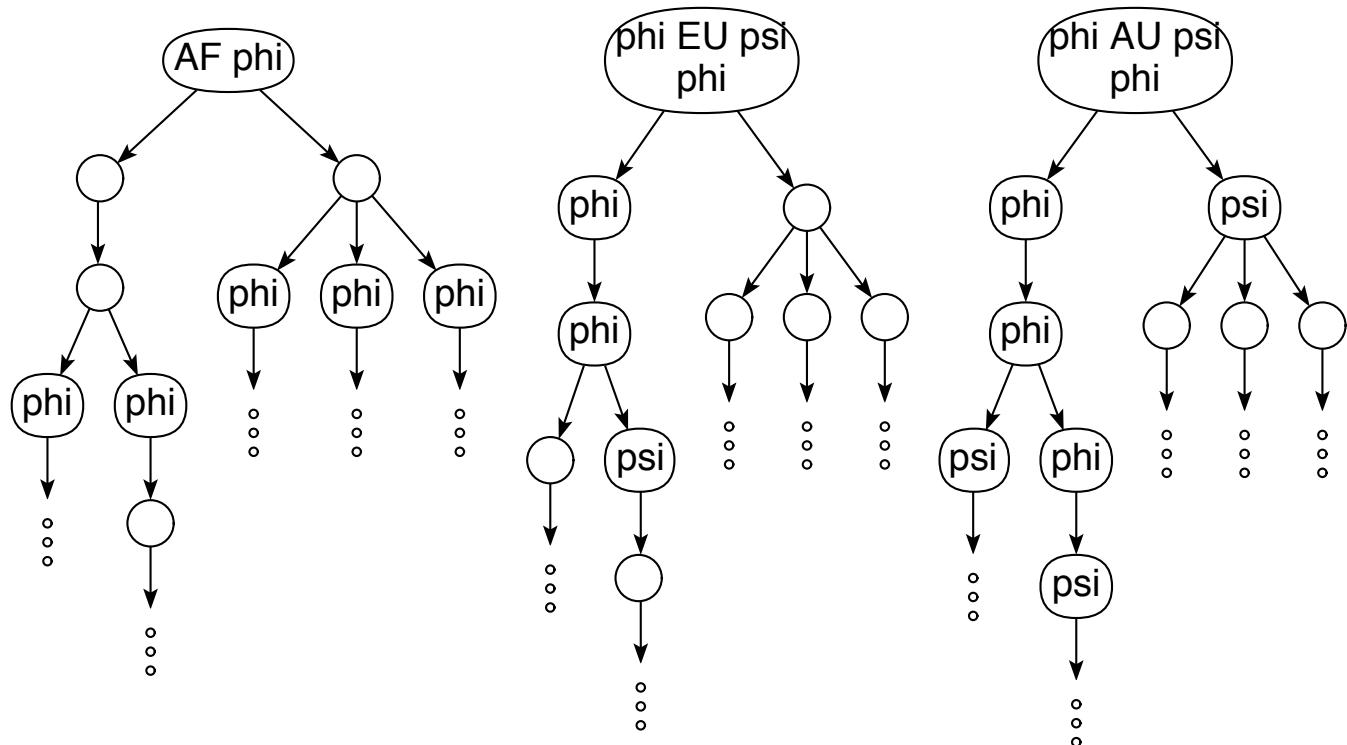


Fig. 2. Transition graphs whose roots satisfy  $AF\varphi$ ,  $\varphi EU\psi$  and  $\varphi AU\psi$ , respectively

## Validity of path formulas

$p \models at$	$\iff_{def} p_0 \in value(at)$
$p \models at(lab)$	$\iff_{def} p_0 \in valueL(at, lab)$
$p \models true$	for all $p \in S \cup path(K)$
$p \models false$	for no $p \in S \cup path(K)$
$p \models \neg\varphi$	$\iff_{def} p \not\models \varphi$
$p \models \varphi \vee \psi$	$\iff_{def} p \models \varphi \text{ oder } q \models \psi$
$p \models \varphi \wedge \psi$	$\iff_{def} p \models \varphi \text{ und } q \models \psi$
$p \models \varphi \Rightarrow \psi$	$\iff_{def} p \models \neg\varphi \vee \psi$
$p \models x$	$\iff_{def} s \in c(x)$
$p \models \mu x \varphi$	$\iff_{def} p \in \text{least solution in } x \text{ of the equation } x = \varphi$
$p \models \nu x \varphi$	$\iff_{def} p \in \text{greatest solution in } x \text{ of the equation } x = \varphi$
$p \models next\varphi$	$\iff_{def} \lambda i.p_{i+1} \models \varphi$
$p \models F\varphi$	$\iff_{def} \exists i \in \mathbb{N} : \lambda k.p_{i+k} \models \varphi$
$p \models G\varphi$	$\iff_{def} \forall i \in \mathbb{N} : \lambda k.p_{i+k} \models \varphi$
$p \models \varphi U \psi$	$\iff_{def} \exists i \in \mathbb{N} : \lambda k.p_{i+k} \models \psi \wedge \forall j < i : \lambda k.p_{j+k} \models \varphi$
$p \models \varphi W \psi$	$\iff_{def} \forall i \in \mathbb{N} : \lambda k.p_{i+k} \models \psi \Rightarrow \forall j < i : \lambda k.p_{j+k} \models \varphi$
$p \models \varphi R \psi$	$\iff_{def} \forall i \in \mathbb{N} : \lambda k.p_{i+k} \models \psi \vee \exists j < i : \lambda k.p_{j+k} \models \varphi$
$p \models \varphi \sim \psi$	$\iff_{def} p \models G(\varphi \Rightarrow F\psi)$

## Global semantics

The abstract syntax of the above grammar generating state and path formulas – except rules (5) and (9) – is given by a constructive signature  $\Sigma = (S, OP)$  with sort set  $S = \{sf, pf\}$ .

The  $sf$ - or  $pf$ -constructors of  $\Sigma$  are usually called **modal** or **temporal operators**, respectively.

Every state or path formula is represented by a  $\Sigma$ -term over  $V$ .

The **global semantics** of state and path formulas is given by the following  $\Sigma$ -Algebra  $\mathcal{A} = (A, Op)$  (see [18]):

$$\begin{aligned} A_{sf} &= \mathcal{P}(Q) \\ A_{pf} &= \mathcal{P}(\text{path}(K)) \end{aligned}$$

For all  $qs, qs' \subseteq Q$  and  $f : Q \rightarrow \mathcal{P}(Q)$ ,

$$\begin{aligned} \text{imgsShares}(qs)(f)(qs') &=_{\text{def}} \{q \in qs \mid f(q) \cap qs' \neq \emptyset\}, \\ \text{imgsSubset}(qs)(f)(qs') &=_{\text{def}} \{q \in qs \mid f(q) \subseteq qs'\}. \end{aligned}$$

For all  $at \in At$ ,  $lab \in Lab$ ,  $qs, qs' \subseteq Q$  and  $ps, ps' \subseteq \text{path}(K)$ ,

$$\begin{aligned} at^{\mathcal{A}} &= \text{value}(at) \\ at(lab)^{\mathcal{A}} &= \text{valueL}(at, lab) \end{aligned}$$

$$\begin{aligned}
true^{\mathcal{A}} &= Q \\
false^{\mathcal{A}} &= \emptyset \\
\neg^{\mathcal{A}}(qs) &= Q \setminus qs \\
\vee^{\mathcal{A}} &= \cup \\
\wedge^{\mathcal{A}} &= \cap \\
qs \Rightarrow^{\mathcal{A}} qs' &= (Q \setminus qs) \cup qs' \\
EX^{\mathcal{A}} &= \textcolor{red}{imgsShares}(Q)(trans) \\
AX^{\mathcal{A}} &= \textcolor{red}{imgsSubset}(Q)(trans) \\
\langle lab \rangle^{\mathcal{A}} &= \textcolor{red}{imgsShares}(Q)(\text{flip}(\text{transL})(lab)) \\
[lab]^{\mathcal{A}} &= \textcolor{red}{imgsSubset}(Q)(\text{flip}(\text{transL})(lab)) \\
true^{\mathcal{A}} &= path(K) \\
\neg^{\mathcal{A}}(ps) &= path(K) \setminus ps \\
ps \Rightarrow^{\mathcal{A}} ps' &= (path(K) \setminus ps) \cup ps' \\
next^{\mathcal{A}}(ps) &= \{p \in path(K) \mid \lambda i.p_{i+1} \in ps\}
\end{aligned}$$

$\text{flip} : (a \rightarrow b \rightarrow c) \rightarrow (b \rightarrow a \rightarrow c)$  denotes the Haskell function that exchanges the arguments of a cascaded binary function.

The remaining modal and temporal operators are reduced to  $\mu$ - resp.  $\nu$ -abstractions:

$$\begin{aligned}
 EF\varphi &= \mu x.\varphi \vee EX\ x \\
 AF\varphi &= \mu x.\varphi \vee (AX\ x \wedge EX\ true) \\
 EG\varphi &= \nu x.\varphi \wedge (EX\ x \vee AX\ false) \\
 AG\varphi &= \nu x.\varphi \wedge AX\ x \\
 \varphi\ EU\ \psi &= \mu x.\psi \vee (\varphi \wedge EX\ x) \\
 \varphi\ AU\ \psi &= \mu x.\psi \vee (\varphi \wedge AX\ x) \\
 \varphi\ EW\ \psi &= \nu x.\psi \vee (\varphi \wedge EX\ x) \\
 \varphi\ AW\ \psi &= \nu x.\psi \vee (\varphi \wedge AX\ x) \\
 \varphi\ ER\ \psi &= \nu x.\psi \wedge (\varphi \vee EX\ x) \\
 \varphi\ AR\ \psi &= \nu x.\psi \wedge (\varphi \vee AX\ x)
 \end{aligned}$$

$$\begin{aligned}
 F\varphi &= \mu x.\varphi \vee next\ x \\
 G\varphi &= \nu x.\varphi \wedge next\ x \\
 \varphi\ U\ \psi &= \mu x.\psi \vee (\varphi \wedge next\ x) \\
 \varphi\ W\ \psi &= \nu x.\psi \vee (\varphi \wedge next\ x) \\
 \varphi\ R\ \psi &= \nu x.\psi \wedge (\varphi \vee next\ x) \\
 \varphi \rightsquigarrow \psi &= G(\varphi \Rightarrow F\psi)
 \end{aligned}$$

$\Sigma$ -terms representing  $\mu$ - or  $\nu$ -abstractions are evaluated in  $\mathcal{A}$  as follows:

Since  $A_{sf}$  and  $A_{pf}$  are powersets and thus  $\omega$ -bicomplete carrier sets – with partial order  $\subseteq$ , least element  $\emptyset$  and greatest element  $Q$  resp.  $path(K)$  (see [18]) –, **Kleene's Fixpoint Theorem** (see [12, 18]) provides us with the least fixpoint

$$lfp(f) = \bigsqcup_{n \in \mathbb{N}} f^n(false^{\mathcal{A}})$$

and the greatest fixpoint

$$gfp(f) = \bigsqcap_{n \in \mathbb{N}} f^n(true^{\mathcal{A}})$$

of every  $S$ -sorted  $\omega$ -bicontinuous function  $f : A \rightarrow A$ .

If  $A_{sf}$  und  $A_{pf}$  are finite, then

$$lfp(f) = fixpt(\leq)(f)(false^{\mathcal{A}}) \quad \text{and} \quad gfp(f) = fixpt(\geq)(f)(true^{\mathcal{A}})$$

where

$$\textcolor{red}{fixpt} : (A \rightarrow A \rightarrow \textit{Bool}) \rightarrow (A \rightarrow A) \rightarrow (A \rightarrow A)$$

is defined as follows:

$$fixpt(\leq)(f)(a) = \text{if } f(a) \leq a \text{ then } a \text{ else } fixpt(\leq)(f)(f(a))$$

Let  $\varphi$  be a modal formula satisfying the above assumptions and  $g : V \rightarrow A$  be an  $S$ -sorted function.

The extension  $\mathbf{g}^* : T_\Sigma(V) \rightarrow \mathcal{A}$  of  $g$  to  $\Sigma$ -terms is inductively defined as follows:

For all  $x \in V$ ,  $s \in \{sf, pf\}$ ,  $op : s^n \rightarrow s \in OP$ ,  $\varphi_1, \dots, \varphi_n \in T_\Sigma(V)_s$  and  $\varphi \in T_\Sigma(V)$ ,

$$\begin{aligned} g^*(x) &= g(x), \\ g^*(op(\varphi_1, \dots, \varphi_n)) &= op^{\mathcal{A}}(g^*(\varphi_1), \dots, g^*(\varphi_n)), \\ g^*(\mu x. \varphi) &= lfp(\lambda a. g[a/x]^*(\varphi)), \quad (1) \\ g^*(\nu x. \varphi) &= gfp(\lambda a. g[a/x]^*(\varphi)). \quad (2) \end{aligned}$$

Under the above assumptions on formulas and transition relations, the function

$$\lambda a. g[a/x]^*(\varphi) : A \rightarrow A$$

is  $\omega$ -bicontinuous and thus (1) is the least and (2) the greatest fixpoint of  $\lambda a. g[a/x]^*(\varphi)$ .

Two formulas  $\varphi$  and  $\psi$  are  **$\mathcal{A}$ -equivalent**, written:  $\varphi \sim \psi$ , if for all  $g : V \rightarrow A$ ,

$$g^*(\varphi) = g^*(\psi).$$

For instance,

$$\begin{aligned}\neg F\varphi &\sim G(\neg\varphi) \\ \neg(\varphi R \psi) &\sim \neg\varphi U \neg\psi \\ F\varphi &\sim \text{true} U \varphi \\ G\varphi &\sim \varphi W \text{false} \\ \varphi U \psi &\sim (\varphi W \psi) \wedge F\psi \\ \varphi W \psi &\sim (\varphi U \psi) \vee G\varphi\end{aligned}$$

In the following chapters 3 to 7 we only deal with state formulas. Path formulas (and temporal logic) are treated in chapter 22 in the context of the specification **stream**.

## Kripke models in Expander2

The command *specification > build Kripke model* constructs a Kripke model

$$K = (Q, At, Lab, trans, transL, value, valueL)$$

from parts of the actual specification *SP*. The sets *Q*, *At* and *Lab* are stored in the constants

```
states, atoms, labels :: [Term String]
```

where **Term String** is the **String**-instance of the datatype

```
data Term a = V a | F a [Term a] | Hidden Special
type TermS = Term String
leaf a = F a []
```

*SP* should contain an axiom **states == inits** where **inits** evaluates to a list of initial states. *build Kripke model* adds to **states** all states that are reachable from **inits** via the **transition axioms** of *SP*, which have the following form:

$$\begin{array}{ll} \varphi \implies st \rightarrow st', & \varphi \implies st \rightarrow \text{branch\$sts}, \\ \varphi \implies (st, lab) \rightarrow st', & \varphi \implies (st, lab) \rightarrow \text{branch\$sts} \end{array}$$

`st`, `lab`, `st'` and `sts` are supposed to evaluate to a state or atom, a label, a state or a list of states, respectively.

The transition axioms of  $SP$  are converted into the transition and value relations of  $K$  and stored as the lists

`trans, value :: [[Int]]`

`transL, valueL :: [[[Int]]]`

such that for all  $i, j, k \in \mathbb{N}$ ,

$$i \in \text{trans}!!j \iff \text{states}!!i \in \text{trans}(\text{states}!!j),$$

$$i \in \text{transL}!!j!!k \iff \text{states}!!i \in \text{transL}(\text{states}!!j)(\text{labels}!!k),$$

$$i \in \text{value}!!j \iff \text{atoms}!!i \in \text{value}(\text{atoms}!!j),$$

$$i \in \text{valueL}!!j!!k \iff \text{atoms}!!i \in \text{valueL}(\text{atoms}!!j)(\text{labels}!!k).$$

### *Signatures in Expander2* (see `Eterm.hs`)

Symbols, simplification axioms and transition axioms are stored in the signature of  $SP$  that is an object of the following O'Haskell type.

```

struct Sig = isPred,isCopred,isConstruct,isDefunct,isFovar,isHovar,blocked
                           :: String -> Bool
hovarRel           :: String -> String -> Bool
simpls,transitions :: [(Term String,[Term String],Term String)]
states,atoms,labels :: [Term String]
trans,value        :: [[Int]]
transL,valueL      :: [[[Int]]]
safeEqs            :: Bool

out,parents :: Sig -> [[Int]]
out sig      = invertRel sig.atoms sig.states sig.value
parents sig = invertRel sig.states sig.states sig.trans

invertRel :: [a] -> [b] -> [[Int]] -> [[Int]]
invertRel as bs iss = map f $ indices_ bs where f i = searchAll (i `elem` iss

outL,parentsL :: Sig -> [[[Int]]]
outL sig      = invertRelL sig.labels sig.atoms sig.states sig.valueL
parentsL sig = invertRelL sig.labels sig.states sig.states sig.transL

invertRelL :: [a] -> [b] -> [c] -> [[[Int]]] -> [[[Int]]]
invertRelL as bs cs iss = map f $ indices_ cs
                           where f i = map g [0..length as-1] where
                                 where g j = searchAll h iss where
                                       h iss = i `elem` iss!!j

```

Commands of the solver template (see `Ecom.hs`) access components of the current signature via the following command:

```
getSignature = do
    let (ps,cps,cs,ds,fs,hs) = symbols
        (sts,ats,labs,tr,trL,va,vaL) = kripke
        isPred      = (`elem` ps) ||| projection
        isCopred    = (`elem` cps) ||| projection
        isConstruct = (`elem` cs) ||| just . parse int |||
                      just . parse real ||| just . parse quoted |||
                      just . parse (strNat "inj")
        isDefunct   = (`elem` ds) ||| projection
        isFovar     = shares [x,base x] fs
        isHovar     = shares [x,base x] $ map fst hs
        hovarRel x y = isHovar x &&
                        case lookup (base x) hs of
                            Just es@(_:_)
                                -> isHovar y || y `elem` es
                            _ -> not $ isFovar y
        (block,xs) = constraints
        blocked x = if block then z `elem` xs else z `notElem` xs
                     where z = head $ words x
        simpls = simplRules; transitions = transRules
        states = sts; atoms = ats; labels = labs; trans = tr
        transL = trL; value = va; valueL = vaL; safeEqs = safe;
        return $ struct ..Sig
```

```

projection :: String -> Bool
projection = just . parse (strNat "get")

```

## *Embedding of trans, transL, out and outL into the simplifier*

```

mkStates,mkLabels,mkAtoms :: Sig -> [Int] -> TermS
mkStates sig = mkList . map (sig.states!!)
mkLabels sig = mkList . map (sig.labels!!)
mkAtoms sig = mkList . map (sig.atoms!!)

simplifyS sig (F "trans" state @_ _) =
    if null sig.states then Just $ mkList $ simplReducts sig True st
                           else do i <- search (== st) sig.states
                                    Just $ mkStates sig $ sig.trans!!i
    where st = mkTup state

simplifyS sig (F $" [F "transL" state @_ _,lab]) =
    if null sig.states then Just $ mkList $ simplReducts sig True
                           $ mkPair st lab
    else do i <- search (== st) sig.states
            k <- search (== lab) sig.labels
            Just $ mkStates sig $ sig.transL!!i!!k
    where st = mkTup state

```

```

simplifyS sig (F "out" state @_:_)) =
    if null sig.states then Just $ mkList $ filter f sig.atoms
                           else do i <- search (== st) sig.states
                                         Just $ mkAtoms sig $ out sig!!i
    where st = mkTup state
          f = elem st . simplReducts sig True

simplifyS sig (F "$" [F "outL" state @_:_ , lab]) =
    if null sig.states then Just $ mkList $ filter f sig.atoms
                           else do i <- search (== st) sig.states
                                         k <- search (== lab) sig.labels
                                         Just $ mkAtoms sig $ outL sig!!i!!!k
    where st = mkTup state
          f = elem (mkPair st lab) . simplReducts sig True

```

*Further functions for analyzing the Kripke model stored in the actual signature*

*All successors of state*

```

simplifyS sig (F "succs" state) = do i <- search (== st) sig.states
                                      Just $ mkStates sig $ fixpt subset f [i]
                                      where st = mkTup state
                                            f is = is `join` joinMap g is
                                            ks = indices_ sig.labels
                                            g i = sig.trans!!i `join`
                                                joinMap ((sig.transL!!i)!!) ks

```

## *Direct predecessors of state*

```
simplifyS sig (F "parents" state) = do i <- search (== st) sig.states
                                         Just $ mkStates sig $ parents sig!!i
                                         where st = mkTup state

simplifyS sig (F "$" [F "parentsL" state,lab])
            = do i <- search (== st) sig.states
                  k <- search (== lab) sig.labels
                  Just $ mkStates sig $ parentsL sig!!i!!k
                  where st = mkTup state
```

## *All predecessors of state*

```
simplifyS sig (F "preds" state) = do i <- search (== st) sig.states
                                         Just $ mkStates sig $ fixpt subset f [i]
                                         where st = mkTup state
                                               f is = is `join` joinMap g is
                                               ks = indices_ sig.labels
                                               g i = parents sig !!i `join`
                                                     joinMap ((parentsL sig !!i) !!) ks
```

## All state/label sequences from state1 to state2

```
simplifyS sig (F "$" [F "traces" state,st']) =
    do i <- search (== st) sig.states
       j <- search (== st') sig.states
       jList $ map (mkStates sig) $ traces f i j
    where st = mkTup state
          ks = indices_ sig.labels
          f i = sig.trans!!i `join`  
          joinMap ((sig.transL!!i)!!) ks

simplifyS sig (F "$" [F "tracesL" state,st']) =
    do i <- search (== st) sig.states
       j <- search (== st') sig.states
       jList $ map (mkLabels sig) $ tracesL ks f i j
    where st = mkTup state
          ks = indices_ sig.labels
          f i k = (sig.transL!!i)!!k

traces :: Eq a => (a -> [a]) -> a -> a -> [[a]]
traces f a = h [a] a where
    h visited a c = if a == c then [[a]]
                           else do b <- f a`minus`visited
                                   trace <- h (b:visited) b c
                                         [a:trace]
```

```
tracesL :: Eq a => [lab] -> (a -> lab -> [a]) -> a -> a -> [[lab]]
tracesL labs f a = h [a] a where
    h visited a c = if a == c then []
                    else do lab <- labs
                                b <- f a lab`minus`visited
                                trace <- h (b:visited) b c
                                [lab:trace]
```

## Model checking as simplification

The following specification provides

- simplification axioms for reducing modal operators such that they can be evaluated in *ctlAlg* with *eval/G* (see above),
- simplification axioms for reducing state or path formulas to *True* or *False*,
- (co-)Horn clauses for proving state or path formulas by (co)resolution.

The constant **noProcs** is set by pressing the *specification > set number of processes* button.

### modal

```
defuncts: noProcs procs states labels atoms valid
preds:    X sat satL -> true false not \/ /\ `then` or and alloutL
          allanytrans allanytransL EX XE <> >< EF AF FE FA `EU` `AU`
copreds: ~ eqstate AX XA # ## AG EG GE GA `EW` `AW` `ER` `AR`
fovars:   s s' st st' lab at
hovars:  X

axioms:
```

```

procs == [0..noProcs-1]      -- used in many specifications that import modal

-- state equivalence relations

& (eqstate <==> NU X.rel((st,st'),out$st=out$st' &
                           alloutL(st,st') &
                           allanytrans(X)(st,st') &
                           allanytrans(X)(st',st) &
                           allanytransL(X)(st,st') &
                           allanytransL(X)(st',st)))

& (st ~ st' ===> out$st=out$st' &
   alloutL(st,st') &
   allanytrans(~)(st,st') & allanytrans(~)(st',st) &
   allanytransL(~)(st,st') & allanytransL(~)(st',st))

& (alloutL(st,st') <==> all(rel(lab,outL(st)$lab=outL(st')$lab))$labels)

& (allanytrans(P)(st,st') <==> allany(P)(trans$st,trans$st'))
& (allanytransL(P)(st,st') <==> all(rel(lab,allany(P)(transL(st)$lab,
                                         transL(st')$lab)))$labels)

-- simplification of first-order state formulas

```

```

& (sat(at)$st <==> at `in` out$st)
& (satL(at,lab)$st <==> at `in` outL(st)$lab)

& (EX(P)$st <==> any(P)$concat$trans(st):map(transL$st)$labels)
& (AX(P)$st <==> all(P)$concat$trans(st):map(transL$st)$labels)
& ((lab<>P)$st <==> any(P)$transL(st)$lab)
& ((lab#P)$st <==> all(P)$transL(st)$lab)

& (XE(P)$st <==> any(P)$parents$st)
& (XA(P)$st <==> all(P)$parents$st)
& ((lab><P>$st <==> any(P)$parentsL(st)$lab)
& ((lab##P)$st <==> all(P)$parentsL(st)$lab)

& (true$st <==> True)
& (false$st <==> False)
& (not(P)$st <==> Not(P$st))
& ((P/\Q)$st <==> P$st & Q$st)
& ((P\Q)$st <==> P$st | Q$st)
& ((P`then`Q)$st <==> (P$st ==> Q$st))

-- simplification of second-order state formulas

& (or <==> foldl(\/)$.false)
& (and <==> foldl(/\$)$.true)

```

```

& (EF$P <==> MU X.(P\EX$X))          -- forward finally on some path
& (FE$P <==> MU X.(P\XE$X))          -- backwards finally on some path
& (AF$P <==> MU X.(P\/(AX(X)\EX$true))) -- forward finally on all paths
& (FA$P <==> MU X.(P\/(XA(X)\XE$true))) -- backwards finally on all paths
& (EG$P <==> NU X.(P\/(EX(X)\AX>false))) -- forward generally on some path
& (GE$P <==> NU X.(P\/(XE(X)\XA>false))) -- backwards generally on some path
& (AG$P <==> NU X.(P\AX$X))           -- forward generally on all paths
& (GA$P <==> NU X.(P\XA$X))           -- backwards generally on all paths

& ((P`EU`Q) <==> MU X.(Q\/(P\EX$X))) -- until
& ((P`AU`Q) <==> MU X.(Q\/(P\AX$X))) -- weak until
& ((P`EW`Q) <==> NU X.(Q\/(P\EX$X)))
& ((P`AW`Q) <==> NU X.(Q\/(P\AX$X)))
& ((P`ER`Q) <==> NU X.(Q\/(P\EX$X))) -- release
& ((P`AR`Q) <==> NU X.(Q\/(P\AX$X)))

-- narrowing of first-order state formulas

& (EF(P)$st <==> P$st | EX(EF$P)$st) -- forward finally on some path
& (AG(P)$st ==> P$st & AX(AG$P)$st)   -- forward generally on all paths
& (AF(P)$st <==> P$st | AX(AF$P)$st & EX(true)$st) -- forward finally on all paths
& (EG(P)$st ==> P$st & (EX(EG$P)$st | AX(false)$st)) -- forward generally on some path
& (FE(P)$st <==> P$st | XE(FE$P)$st)   -- backwards finally on some path

```

```

& (GA(P)$st ===> P$st & XA(GA$P)$st)           -- backwards generally on all paths
& (FA(P)$st <==> P$st | XA(FA$P)$st & XE(true)$st)
                                                -- backwards finally on all paths
& (GE(P)$st ===> P$st & (XE(GE$P)$st | XA(false)$st))
                                                -- backwards generally on some path

& ((P`EU`Q)$st <==> Q$st | P$st & EX(P`EU`Q)$st)      -- until
& ((P`AU`Q)$st <==> Q$st | P$st & AX(P`AU`Q)$st)
& ((P`EW`Q)$st ===> Q$st | P$st & EX(P`EW`Q)$st)      -- weak until
& ((P`AW`Q)$st ===> Q$st | P$st & AX(P`AW`Q)$st)
& ((P`ER`Q)$st ===> Q$st & (P$st | EX(P`ER`Q)$st))    -- release
& ((P`AR`Q)$st ===> Q$st & (P$st | AX(P`AR`Q)$st))

```

## Model checking as term evaluation

Given the representation of a Kripke model  $K$  produced by *build Kripke model* (see chapter 2), the following Haskell code implements the global semantics of state formulas w.r.t.  $K$  (see chapter 1).

### A signature for state formulas

```
struct Modal a label = true, false :: a
                    neg :: a -> a
                    or, and :: a -> a -> a
                    ex, ax, xe, xa :: a -> a
                    dia, box, aid, xob :: label -> a -> a
```

### The Modal-algebra of state sets

```
ctlAlg :: Sig -> Modal [Int] Int
ctlAlg sig = struct true = sts
            false = []
            neg = minus sts
            or = join
            and = meet
            ex = imgsShares sts $ f sig.trans sig.transL
            ax = imgsSubset sts $ f sig.trans sig.transL
            xe = imgsShares sts $ f (parents sig) (parentsL sig)
            xa = imgsSubset sts $ f (parents sig) (parentsL sig)
            dia = imgsShares sts . flip (g sig.transL)
```

```

box    = imgsSubset sts . flip (g sig.transL)
aid    = imgsShares sts . flip (g $ parentsL sig)
xob    = imgsSubset sts . flip (g $ parentsL sig)
where sts = indices_ sig.states
      f trans transL i = join (trans!!i) $ joinMap (g transL i)
                                $ indices_ sig.labels
      g transL i k = transL!!i!!k

```

## *The interpreter of state formulas in ctlAlg*

```

type States = Maybe [Int]

foldModal :: Sig -> TermS -> States
foldModal sig = f $ const Nothing where
  alg = ctlAlg sig
  f :: (String -> States) -> TermS -> States
  f g (F x []) | just a      = a where a = g x
  f _ (F "true" [])        = Just alg.true
  f _ (F "false" [])       = Just alg.false
  f g (F "not" [t])        = do a <- f g t; Just $ alg.neg a
  f g (F "\\" [t,u])       = do a <- f g t; b <- f g u
                                Just $ alg.or a b
  f g (F "/\\" [t,u])      = do a <- f g t; b <- f g u
                                Just $ alg.and a b
  f g (F "`then`" [t,u])   = do a <- f g t; b <- f g u
                                Just $ alg.or (alg.neg a) b

```

$f\ g\ (F\ "EX"\ [t])$	= do a <- f g t; Just \$ alg.ex a
$f\ g\ (F\ "AX"\ [t])$	= do a <- f g t; Just \$ alg.ax a
$f\ g\ (F\ "XE"\ [t])$	= do a <- f g t; Just \$ alg.xe a
$f\ g\ (F\ "XA"\ [t])$	= do a <- f g t; Just \$ alg.xa a
$f\ g\ (F\ "<>"\ [lab,t])$	= do a <- f g t; k <- searchL lab Just \$ alg.dia k a
$f\ g\ (F\ "#" \ [lab,t])$	= do a <- f g t; k <- searchL lab Just \$ alg.box k a
$f\ g\ (F\ "><"\ [lab,t])$	= do a <- f g t; k <- searchL lab Just \$ alg.aid k a
$f\ g\ (F\ "##"\ [lab,t])$	= do a <- f g t; k <- searchL lab Just \$ alg.xob k a
$f\ g\ (F\ ('M':'U':':x)\ [t])$	= fixptM subset (step g x t) alg.false
$f\ g\ (F\ ('N':'U':':x)\ [t])$	= fixptM supset (step g x t) alg.true
$f\ _\ (F\ "$"\ [at,lab])$	= do i <- searchA at; k <- searchL lab Just \$ sig.valueL!!i!!k
$f\ _\ at$	= do i <- searchA at; Just \$ sig.value!!i

```

searchA,searchL :: Term String -> Maybe Int
searchA at = search (== at) sig.atoms
searchL lab = search (== lab) sig.labels

```

```

step :: (String -> States) -> String -> TermS -> [Int] -> States
step g x t a = f (upd g x $ Just a) t

```

`f(g) :: Term String -> Maybe [Int]` implements the (modal part of the) extension  
 $g^* : T_\Sigma(V) \rightarrow \mathcal{A}$  (see Kripke models in Expander2).

### Embedding of `foldModal(sig)` into the simplifier

$$\frac{\text{eval}(\varphi)}{\varphi^A} \qquad \frac{\text{evalG}(\varphi)}{\text{transition graph of } K \text{ with all } q \in \varphi^A \text{ colored green}}$$

```
simplifyS (F "eval" [phi]) sig = do sts <- foldModal sig phi
                                         Just $ mkStates sig sts

simplifyS (F "evalG" [phi]) sig = do sts <- foldModal sig phi
                                         let f st = if st `elem` map (strs!!) sts
                                                 then "dark green$"++st else st
                                         Just $ mapT f $ eqsToGraph [] eqs
where [strs,labs] = map (map showTerm0) [sig.states,sig.labels]
      (eqs,_) = if null labs then relToEqs 0 $ mkPairs strs strs sig.trans
                 else relLToEqs 0 $ mkTriples strs labs strs sig.transL
```

## Specifications that import modal

### *Actions for building and verifying a specification involving a Kripke model*

- enter the name of the specification into the entry field
- press the *return* key: the specification is parsed
- press the *specification > build Kripke model* button

A Kripke model is constructed from axioms of the specification

- press *graph > show graph of Kripke model > here*, and a graph comprising the states, atoms and transitions of the Kripke model is displayed on the solver canvas
- press *paint*, and the graph is displayed on the painter canvas (see below)
- customize the graph structure by moving nodes and red arc support points
- press *combis*, and the red arc support points are removed

**micro** (see [3], section 4.1)

```
specs:          modal
constructs:    start close heat error
```

```
axioms:
```

```

states == [1]           -- initial states
& atoms == [start,close,heat,error]

& 1 -> branch[2,3] & 2 -> 5 & 3 -> branch[1,6]
& 4 -> branch[1,3,4] & 5 -> branch[2,3] & 6 -> 7 & 7 -> 4

& start -> branch[2,5,6,7]
& close -> branch[3..7]
& heat -> branch[4,7]
& error -> branch[2,5]

```

conjects: -- proof files

EF(Sat\$heat)\$5	--> True	micro1P
& EG(Sat\$heat)\$7	--> True	micro2P, micro2cycleP
& AF(Sat\$heat)\$6	--> True	(16 parallel simplifications)
& all(EF\$Sat\$heat)\$states	--> True	micro3P
& all(AF\$Sat\$heat) [4,6,7]	--> True	(26 parallel simplifications)

terms:

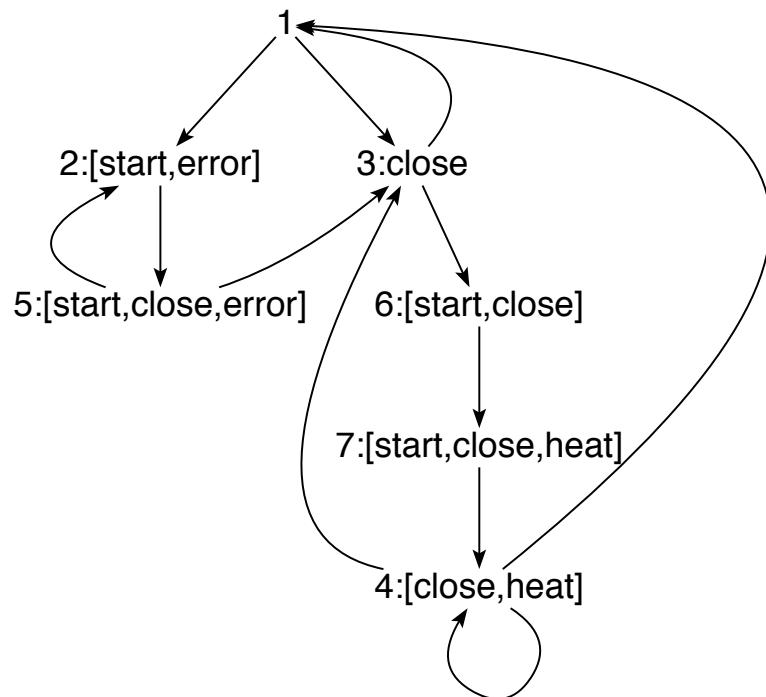
eval\$EF\$heat	--> states
<+> eval\$EG\$heat	--> [4,7]
<+> eval\$AF\$heat	--> [4,6,7]
<+> eval\$start`then`heat	--> [1,3,4,7]

```

<+> eval$EX$start`then`heat           --> [1,3,4,5,6,7]
<+> eval$error`then`AG$not$heat      --> [1,3,4,6,7]

```

After the above actions the following graph appears on the painter canvas:



Four of the five conjectures of `micro` can be proved by applying simplification axioms and *selection > permute subtrees*. Due to the fact that the second conjecture involves the *greatest* fixpoint EG (see `modal`) is led into a cycle with simplification and subtree permutation (see `micro2cycleP`). Applying *selection > coinduction*, however, leads to a successful proof (see `micro2P`). It contains *two* coinduction steps because  $EG(Sat\$heat)\$7$  can only be shown along with  $EG(Sat\$heat)\$4$  (see chapter 9 below or [18], chapter 17). Here is the proof as recorded by Expander2 in `micro2P`:

## 0. Derivation of

`EG(Sat(heat))$7`

All simplifications are admitted.

Equation removal is safe.

## 1. Adding

`(EG0(z0)$z1 <== z0 = Sat(heat) & z1 = 7)`

to the axioms and applying COINDUCTION wrt

`(EG(P)$st ==> P(st) & (EX(EG(P))$st | AX(false)$st))`

at positions [] of the preceding trees leads to

$\text{EG0}(\text{Sat}(\text{heat}))\$4$

The reducts have been simplified in parallel.

## 2. Adding

$(\text{EG0}(z2)\$z3 \leq z2 = \text{Sat}(\text{heat}) \ \& \ z3 = 4)$

to the axioms and applying COINDUCTION wrt

$(\text{EG}(P)\$st \implies P(st) \ \& \ (\text{EX}(\text{EG}(P))\$st \mid \text{AX}(\text{false})\$st))$

at positions [] of the preceding trees leads to

$\text{EG0}(\text{Sat}(\text{heat}))\$1 \mid \text{EG0}(\text{Sat}(\text{heat}))\$3 \mid \text{EG0}(\text{Sat}(\text{heat}))\$4$

The reducts have been simplified in parallel.

## 3. NARROWING the preceding trees (one step) leads to

$\text{EG0}(\text{Sat}(\text{heat}))\$3 \mid \text{EG0}(\text{Sat}(\text{heat}))\$4$

The axioms have been MATCHED against their redices.

The reducts have been simplified in parallel.

4. NARROWING the preceding trees (one step) leads to

EG0(Sat(heat))\$4

The axioms have been MATCHED against their redices.

The reducts have been simplified in parallel.

The formula coincides with no. 1

5. NARROWING the preceding trees (one step) leads to

True

The axioms have been MATCHED against their redices.

The reducts have been simplified in parallel.

## trans0

```
specs:           modal
constructs:     is less
defuncts:       drawI drawS
```

axioms:

```
states == [0,22]
& atoms == map($)$prodL[[is,less],states]

& (st < 6 ==> st -> st+1<+>ite(st`mod`2 = 0,st,()))
& 6 -> branch$[1..5]++[7..10]
& 7 -> 14
& 22 -> 33<+>44

& is$st -> st
& less$st -> valid(<st)

-- widget interpreters:

& drawI == wtree $ fun(st,ite(st`in`states,
                                color(index(st,states),length$states)$circ$11,
                                frame(3)$blue$text$st))

& drawS$F == wtree $ fun(x,ite(F$x,blue$frame(5)$F$text$x,text$x))
```

conjects:

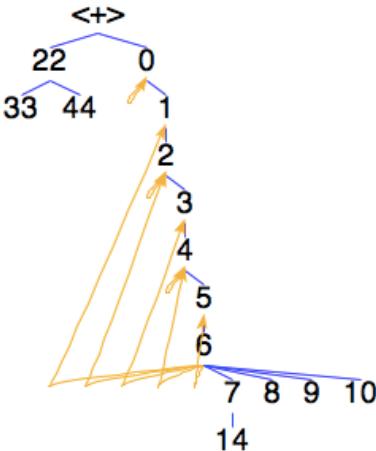
```
(=14)$3          --> False
& AG(<14)$9      --> True
& all(EX(=4))[3,4,6] --> True
& sat(is$4)$3      --> False
& sat(is$4)$4      --> True
& x^4^y->st        -- narrow&simplify --> 5^x^y = st | 4^x^y = st
```

terms:

```
eval$is$4          --> [4]
<+> eval$less$4      --> [0,1,2,3]
<+> eval(EX$less$4)  --> [0,1,2,6]
<+> eval(EF$less$4)  --> [0..6]
<+> eval(AF$less$4)  --> [0..3]
<+> eval(EG$less$4)  --> [0..2]
<+> eval(AG$less$4)  --> []
<+> eval(AG$less$14) --> [8..10]
<+> eval(is(4) `then` EF$less$2)      --> [0..10,14,22,33,44]
<+> eval(is(4) `then` EF$is$0)        --> [0..3,5..10,14,22,33,44]
<+> eval(AG $ is(4) `then` EF$is$0)  --> [7..10,14,22,33,44]
<+> trans(6) `meet` eval(less$2)       --> [1]
<+> f(x^4^y)           -- rewrite&simplify --> f(5^x^y) <+> f(4^x^y)
```

## Graphical representations of the transitions of `trans0`

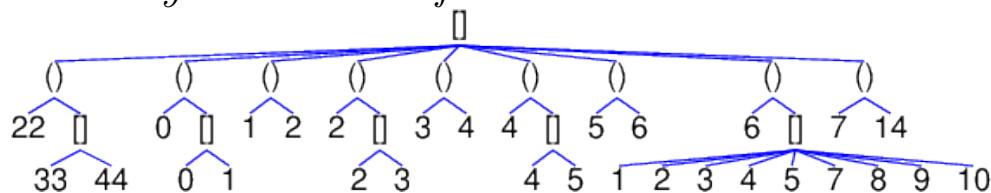
- press `graph > show graph of transitions > here`



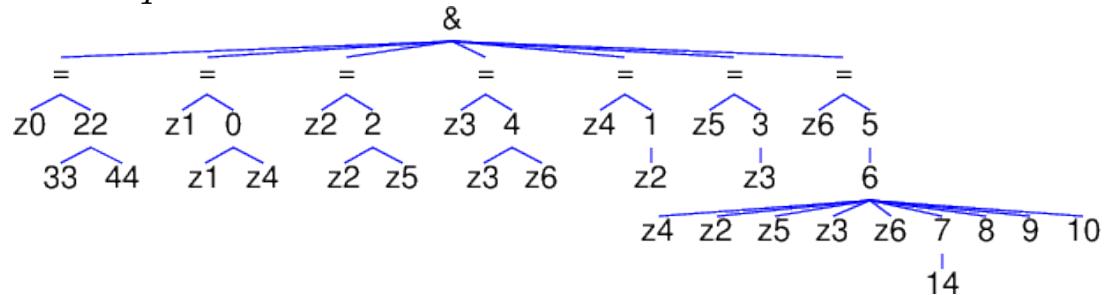
- press `graph > show Boolean matrix > of transitions`

	0	1	2	3	4	5	6	7	8	9	10	14	33	44
0	●	●												
1			●											
2				●	●									
3					●									
4						●	●							
5							●							
6	●	●	●	●	●	●	●	●	●	●	●			
7								●						
22												●	●	●

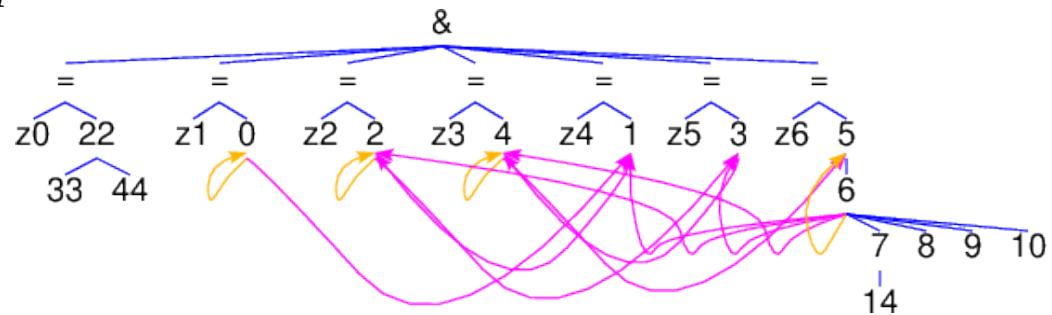
- press `graph > show binary relation > of transitions`



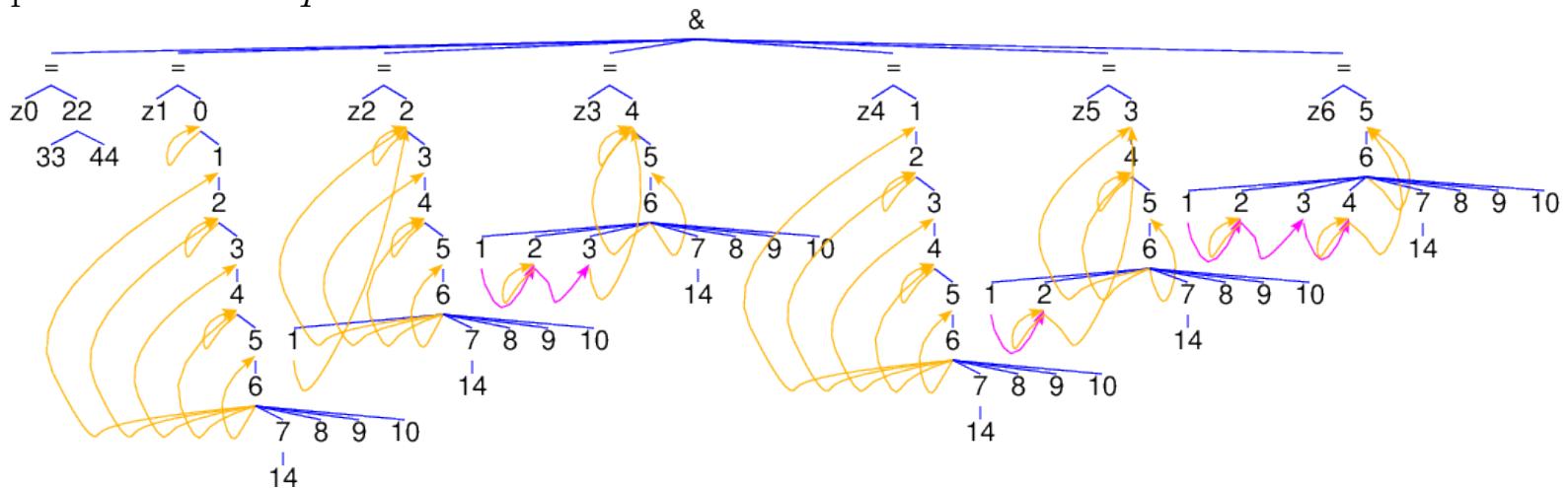
- press *graph > show graph of transitions > here*  
 press *build iterative equations*



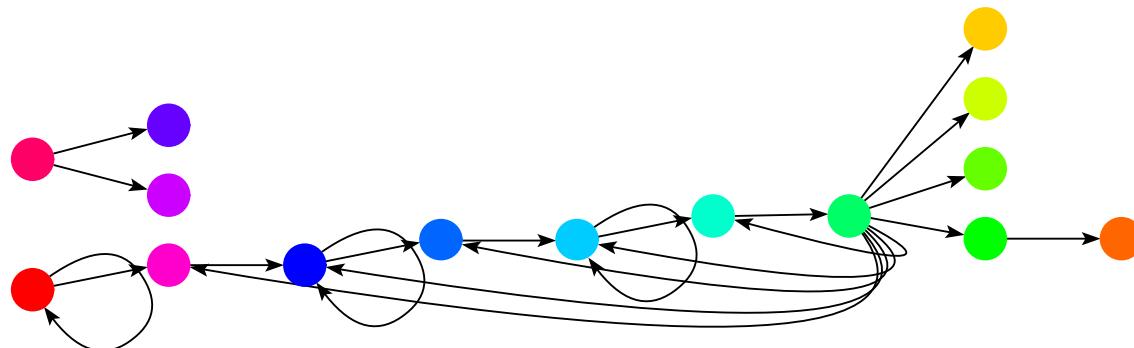
- press *graph > show graph of transitions > here*  
 press *build iterative equations*  
 press *connect equations*



- press *graph > show graph of transitions > here; build iterative equations*  
press *connect equations* two times

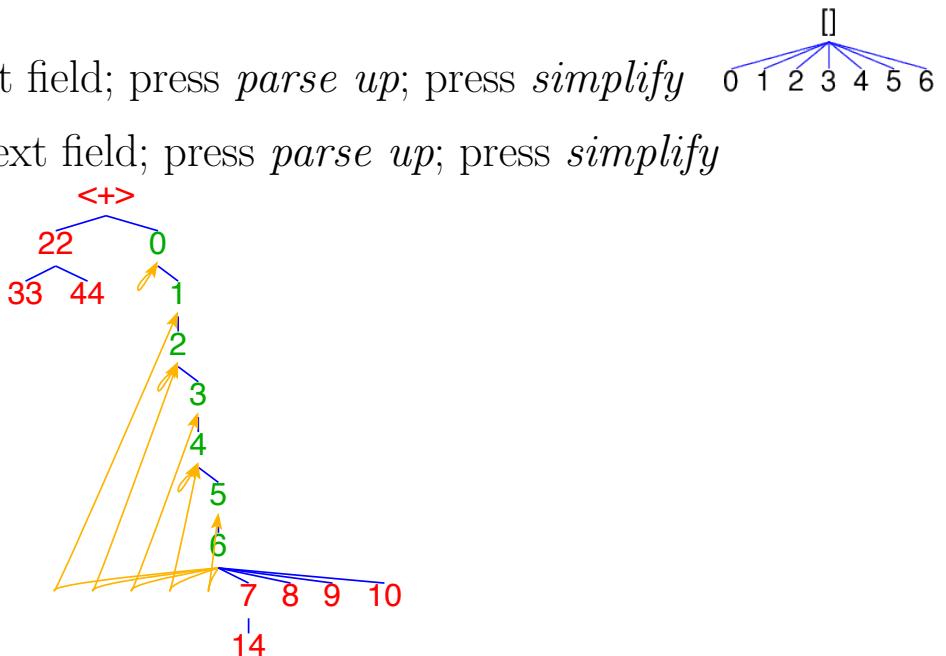


- press *graph > show graph of transitions > here; enter drawI into the entry field*  
press *tree > tree*; press *paint*; enter *t1-90* into the mode field  
press *arrange*; adapt red support points and press *combis* to remove them

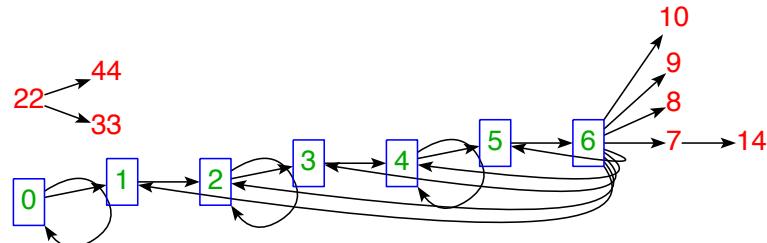


## Graphical representations of the set of states satisfying the modal formula $EF(\text{less}(4))$

- enter `eval$EF$less$4` into the text field; press *parse up*; press *simplify*
- enter `evalG$EF$less$4` into the text field; press *parse up*; press *simplify*



- enter `evalG$EF$less$4` into the text field; press *parse up*; press *simplify*
- enter `drawS$dark green` into the entry field; press *tree > tree*; press *paint*
- enter `t1-90` into the mode field; press *arrange*; press *combis*



The following specifications use **procs** and **noProcs** (see [modal](#)). The value of **noProcs** is to be set by writing a positive natural number into the entry field and pressing *signature > set number of processes*.

\*\*\*\*\*

## Model checking as data flow analysis

*Transformation of a modal formula into a flow graph*

```
simplifyS sig (F "stateflow" [t]) | just u = u where u = initStates sig t

simplifyS sig (F "stateflow" [t]) = initStates True sig $ mkCycles [] t

mkCycles :: [Int] -> TermS -> TermS
mkCycles p t@(F x _) | isFixF x = addToPoss p $ eqsToGraph [0] eqs
                      where (_,eqs,_) = fixToEqs t 0
mkCycles p (F x ts)           = F x $ zipWithSucs mkCycles p ts

data RegEq = Equal String (Term String) | Diff String (Term String)

fixToEqs :: Term String -> Int -> (Term String, [RegEq], Int)
fixToEqs (F x [t]) n | isFixF x = (F z [], Equal z (F mu [u]):eqs,k)
                      where mu:y:_ = words x
                            b = y `elem` foldT f t
                            f x xss = if isFixF x then joinM xss `join1` y
                                       else joinM xss
                                         where _:y:_ = words x
                            z = if b then y++show n else y
                            (u,eqs,k) = if b then fixToEqs (t>>>g) $ n+1
```

```

                                else fixToEqs t n
                                g = F z [] `for` y
fixToEqs (F x ts) n = (F x us,eqs,k)
    where (us,eqs,k) = f ts n
          f (t:ts) n = (u:us,eqs++eqs',k')
                          where (u,eqs,k) = fixToEqs t n
                                         (us,eqs',k') = f ts k
          f _ n       = ([],[],n)
fixToEqs t n      = (t,[],n)

```

unit = leaf "()"

```

initStates :: Sig -> Term String -> Maybe (Term String)
initStates sig = f
    where sts = mkList sig.states
          f (t@(V x))      = do guard $ isPos x; Just t
          f (F "true" [])   = Just sts
          f (F "false" [])  = Just mkNil
          f (F "`then`" [t,u]) = f (F "\\" [F "not" [t],u])
          f (F "MU" [t])     = do t <- f t; jOP "MU" "[]" [t]
          f (F "NU" [t])     = do t <- f t; jOP "NU" (showTerm0 sts) [t]
          f (F "EX" [t])     = do t <- f t; jOP "EX" "()" [t]
          f (F "AX" [t])     = do t <- f t; jOP "AX" "()" [t]
          f (F "<>" [lab,t]) = do p lab; t <- f t; jOP "<>" "()" [lab,t]
          f (F "#" [lab,t])   = do p lab; t <- f t; jOP "#" "()" [lab,t]

```

```

f (F "not" [t])      = do t <- f t; jOP "not" "()" $ [t]
f (F "\\" ts)        = do ts <- mapM f ts; jOP "\\" "()" ts
f (F "/\\" ts)       = do ts <- mapM f ts; jOP "/\\" "()" ts
f at                  = do i <- search (== at) sig.atoms
                           Just $ mkList $ map (sig.states!!)
                                         $ sig.value!!i
p lab = guard $ just $ search (== lab) sig.labels
jOP op val = Just . F (op++":":++val)

```

## *Flow graph transformation step*

```

simplifyS sig t | just flow = do guard changed; Just $ mkFlow True sig ft id
                  where flow = parseFlow True sig t $ parseVal sig
                        (ft,changed) = evalStates sig t $ get flow

data Flow a = Atom a | Neg (Flow a) a | Comb Bool [Flow a] a |
              Mop Bool (Term String) (Flow a) a | Fix Bool (Flow a) a |
              Pointer [Int]

parseFlow :: Sig -> Term String -> (Term String -> Maybe a) -> Maybe (Flow a)
parseFlow sig t parseVal = f t where
  f (F x [u]) | take 5 x == "not::"
                = do g <- f u; val <- dropVal 5 x; Just $ Neg g val
  f (F x ts)  | take 4 x == "\\\\"::"
                = do gs <- mapM f ts; val <- dropVal 4 x
                      Just $ Comb True gs val

```

```

f (F x ts)  | take 4 x == "/\\::"
             = do gs <- mapM f ts; val <- dropVal 4 x
               Just $ Comb False gs val
f (F x [u]) | take 4 x == "EX::"
             = do g <- f u; val <- dropVal 4 x
               Just $ Mop True (leaf "") g val
f (F x [u]) | take 4 x == "AX::"
             = do g <- f u; val <- dropVal 4 x
               Just $ Mop False (leaf "") g val
f (F x [lab,u]) | take 4 x == "<>::"
                  = do g <- f u; guard $ lab `elem` sig.labels
                     val <- dropVal 4 x; Just $ Mop True lab g val
f (F x [lab,u]) | take 3 x == "#::"
                  = do g <- f u; guard $ lab `elem` sig.labels
                     val <- dropVal 3 x; Just $ Mop False lab g val
f (F x [u])     | take 4 x == "MU::"
                  = do g <- f u; val <- dropVal 4 x; Just $ Fix True g val
f (F x [u])     | take 4 x == "NU::"
                  = do g <- f u; val <- dropVal 4 x; Just $ Fix False g val
f (V x)         | isPos x
                  = do guard $ q `elem` positions t
                     Just $ Pointer q where q = getPos x
f val           = do val <- parseVal val; Just $ Atom val
dropVal n x    = do val <- parse (term sig) $ drop n x; parseVal val

```

```

parseVal sig (t@(F "()" [])) = Just t
parseVal sig t = do F "[]" ts <- Just t; guard $ ts `subset` sig.states; Just t

evalStates :: Sig -> Term String -> Flow (Term String) -> (Flow (Term String),Bool)
evalStates sig t flow = up [] flow
  where ps = maxis (<<=) $ fixPositions t
    up p (Neg g val)
      | val1 == unit || any (p<<) ps = (Neg g1 val, b)
      | True = if null ps then (Atom val2, True)
                else (Neg g1 val2, b || val /= val2)
                where q = p++[0]; (g1,b) = up q g
                      val1 = fst $ getVal flow q
                      val2 = mkList $ minus sig.states $ subterms val1
    up p (Comb c gs val)
      | any (== unit) vals || any (p<<) ps = (Comb c gs1 val, or bs)
      | True = if null ps then (Atom val1, True)
                else (Comb c gs1 val1, or bs || val /= val1)
                where qs = succsInd p gs
                      (gs1,bs) = unzip $ zipWith up qs gs
                      vals = map (fst . getVal flow) qs
                      val1 = mkList $ foldl1 (if c then join else meet)
                                $ map subterms vals
    up p (Mop c lab g val)
      | val1 == unit || any (p<<) ps = (Mop c lab g1 val, b)
      | True = if null ps then (Atom val2, True)

```

```

        else (Mop c lab g1 val2, b || val /= val2)
where q = p++[1]; (g1,b) = up q g
      val1 = fst $ getVal flow q
      f True = shares; f _ = flip subset
      val2 = mkList $ filter (f c (subterms val1) . h)
                           sig.states
      h st = map (sig.states!!) $ tr i
              where i = get $ search (== st) sig.states
      tr i = if lab == leaf "" then sig.trans!!i
              else sig.transL!!i!!j
              where j = get $ search (== lab) sig.labels
up p (Fix c g val)
| val1 == unit || any (p<<) ps = (Fix c g1 val, b)
| True = if f c subset valL val1L
          then if not b && p `elem` ps then (Atom val, True)
                  else (Fix c g1 val, b)
          else (Fix c g1 $ mkList $ h c valL val1L, True)
where f True = flip; f _ = id
      h True = join; h _ = meet
      q = p++[0]; (g1,b) = up q g
      val1 = fst $ getVal flow q
      valL = subterms val; val1L = subterms val1
up _ g = (g, False)

```

```

mkFlow :: Bool -> Sig -> Flow a -> (a -> TermS) -> TermS
mkFlow b sig flow mkVal = f flow
  where f (Atom val)           = mkVal val
        f (Neg g val)         = h "not" val [f g]
        f (Comb True gs val) = h "\\" val $ map f gs
        f (Comb _ gs val)    = h "/\" val $ map f gs
        f (Mop True (F "" []) g val) = h "EX" val [f g]
        f (Mop _ (F "" []) g val)   = h "AX" val [f g]
        f (Mop True lab g val)    = h "<>" val [lab,f g]
        f (Mop _ lab g val)      = h "#" val [lab,f g]
        f (Fix True g val)       = h "MU" val [f g]
        f (Fix _ g val)          = h "NU" val [f g]
        f (Pointer p)           = mkPos p
        h op val ts = if b then F (op ++ "::" ++ showTerm0 (mkVal val)) ts
                        else F op $ ts ++ [mkVal val]

```

## Examples

-- trans1

```

specs:          modal
preds:          two TWO X Y
copreds:        one ONE
constructs:     a b
hovars:         X Y

```

axioms:

```
states == [2] & labels == [a,b] &  
  
(2,b) -> 1<+>3 & (3,b) -> 3 & (3,a) -> 4 & (4,b) -> 3 &  
  
(one <==> NU X.(two/\(b#X))) &  
(two <==> MU Y.((a<>true)\/(b<>Y)))  
  
(ONE(st) ===> (TWO/\(b#ONE))(st)) &  
(TWO(st) <==> ((a<>true)\/(b<>TWO))(st))
```

conjects:

all(one)[1,2]	&	--> False	simplify depthfirst
all(two)[2,3,4]		--> True	simplify depthfirst
all(ONE)[3,4]		--> True	coinduction + narrow match

(1)

terms:

eval\$one <+>	--> [3,4]
eval\$two <+>	--> [2,3,4]

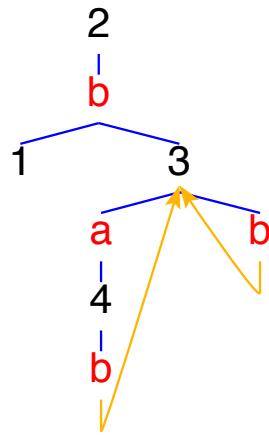


Fig. 3. *Transitions of trans1*

*Proof of (1):*

0. Derivation of  $\text{all}(\text{ONE})[3,4]$

All simplifications are admitted.

Equation removal is safe.

1. SIMPLIFYING the preceding trees (one step) leads to

$\text{ONE}(3)$  &  $\text{ONE}(4)$

2. Adding

(ONE0(z0) <==> z0 = 3 | z0 = 4)

to the axioms and applying COINDUCTION wrt

(ONE(st) ==> (TWO/\(b#ONE))\$st)

at positions [] of the preceding trees leads to

TWO(3) & all(ONE0)[3] & TWO(4)

The reducts have been simplified.

3. NARROWING the preceding trees (one step) leads to

ONE0(3) & TWO(4)

The axioms have been MATCHED against their redices.

The reducts have been simplified.

4. NARROWING the preceding trees (one step) leads to

TWO(4)

The axioms have been MATCHED against their redices.

The reducts have been simplified.

The formula coincides with no. 4

5. NARROWING the preceding trees (one step) leads to

TWO(3)

The axioms have been MATCHED against their redices.

The reducts have been simplified.

6. NARROWING the preceding trees (one step) leads to

True

The axioms have been MATCHED against their redices.

The reducts have been simplified.

For the simplification steps of

stateflow(one),

see [stateflow\\_one.html](#).

```

-- trans2

specs:           modal
preds:          X Y three six seven' Sat SatL
copreds:        NOTthree four five seven eight
constructs:    a A B
defuncts:      drawS
hovars:         X Y

axioms:

states == [1] & labels == [a] & atoms == [A,B] &
(1,a) -> 1<+>2 & (2,a) -> 3 & (3,a) -> 1<+>4 & (4,a) -> 4 &
A -> branch[2,3,4] & (B,a) -> 3 &
(Sat(at)$st <==> at `in` out$st) &
(SatL(at,lab)$st <==> at `in` outL(st)$lab) &

(three   <==> MU X.(a#(A\X))) &
(four    <==> NU X.(a#(A\X))) &
(five    <==> NU X.((a#X)/\six)) &
(six     <==> MU X.(A\/(a#X))) &
(seven   <==> NU X.(MU Y.(a#((A\X)\/Y)))) &

```

```

(eight      <==> NU X.(a#X)) &

(THREE(st)    <==> (a#(Sat(A)\/THREE))(st)) &
(NOTTHREE(st) ==> (a<>(not(Sat(A))\NOTTHREE))(st)) &
(four(st)     ==> (a#(Sat(A)/\four))(st)) &
(five(st)     ==> ((a#five)/\six)(st)) &
(six(st)      <==> (Sat(A)\/(a#six))(st)) &
(eight(st)    ==> (a#eight)(st)) &

(seven(st) ==> seven'(st)) &                                -- alternating fixpoints
(seven'(st) <==> (a#((A/\seven)\/seven'))(st)) &

drawS == wtree \$ fun(dark green$x,green$text$x,
                      red$x,text$x,
                      [] ,text[] ,
                      x,red$text$x)

```

conjects:

all(THREE)[2,4] &	--> True	match&narrow	
all(NOTTHREE)[1,3] &	--> True	match&narrow + coinduction	(1)
(THREE(st) & st `in` states ==> st = 2   st = 4) &			
	--> True	match&narrow + induction	(2)
(NOTTHREE(st) & st `in` states ==> st = 1   st = 3)			
	--> True	match&narrow	

```
map(eval)[three,not$three,four,five,six,seven,eight] & (3)
          --> [[2,4],[1,3],[4],[4],[2,3,4],[4],[1,2,3,4]]
```

```
-- stateflow$seven returns [2,4] instead of [4] because seven is alternating
terms: eval$not(three)/\six --> [3]
```

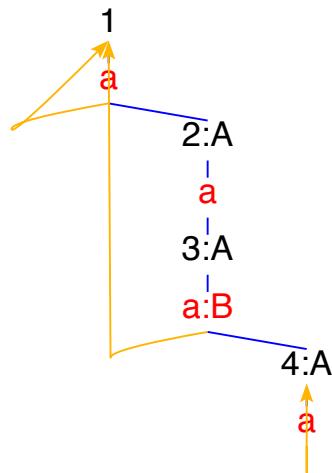


Fig. 4. Kripke model of `trans2`

*Proof of (1) :*

0. Derivation of `all(NOTTHREE) [1,3]`

All simplifications are admitted.

Equation removal is safe.

1. SIMPLIFYING the preceding trees (one step) leads to

NOTTHREE(1) & NOTTHREE(3)

2. Adding

(NOTTHREE0(z0) <==> z0 = 1 | z0 = 3)

to the axioms and applying COINDUCTION wrt

(NOTTHREE(st) ==> (a<>(not(Sat(A)))/\NOTTHREE))\$st)

at positions [] of the preceding trees leads to

A `NOTin` [A] & NOTTHREE0(4) & NOTTHREE0(2) | A `NOTin` [] & NOTTHREE0(1)

The reducts have been simplified.

3. NARROWING the preceding trees (one step) leads to

NOTTHREE0(1)

The axioms have been MATCHED against their redices.

The reducts have been simplified.

4. NARROWING the preceding trees (one step) leads to

True

The axioms have been MATCHED against their redices.

The reducts have been simplified.

*Proof of (2):*

0. Derivation of

THREE(st) & st `in` states ==> st = 2 | st = 4

All simplifications are admitted.

Equation removal is safe.

1. SHIFTING SUBFORMULAS at positions [0,1] of the preceding trees leads to

THREE(st) ==> (st `in` states ==> st = 2 | st = 4)

2. Adding

(THREE0(st) ==> (st `in` states ==> st = 2 | st = 4))

to the axioms and applying FIXPOINT INDUCTION wrt

(THREE(st) <==> (a#(Sat(A)\/THREE))\$st)

at positions [] of the preceding trees leads to

(A `in` [A] & A `in` [] ==> False) & (THREE0(2) & A `in` [] ==> False) &  
(A `in` [A] & THREE0(1) ==> False) & (THREE0(2) & THREE0(1) ==> False) &  
(THREE0(4) & A `in` [] ==> False) & (THREE0(4) & THREE0(1) ==> False)

The reducts have been simplified.

3. NARROWING the preceding trees (one step) leads to

THREE0(1) ==> False

The axioms have been MATCHED against their redices.

The reducts have been simplified.

4. NARROWING the preceding trees (one step) leads to

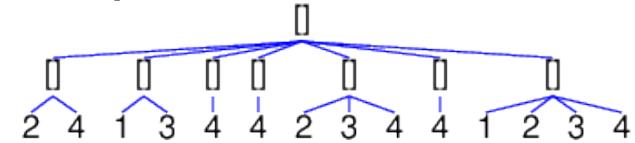
True

The axioms have been MATCHED against their redices.

The reducts have been simplified.

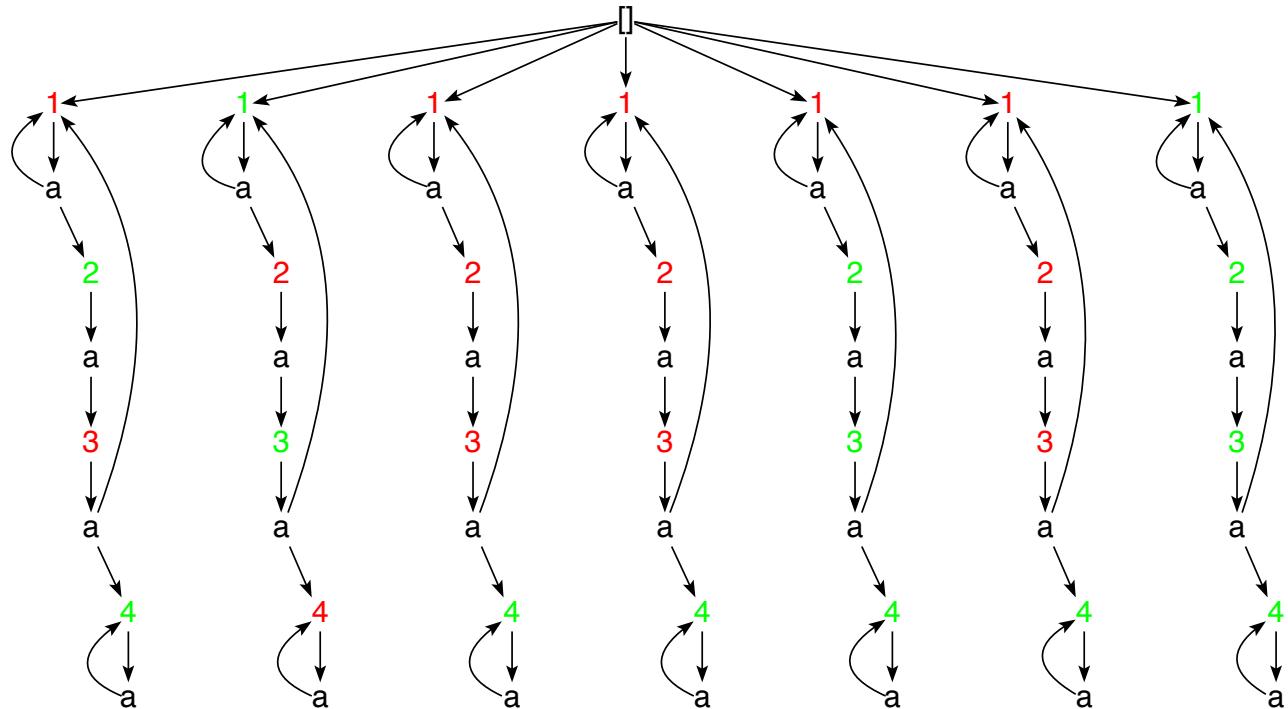
## Graphical representations of the set of states satisfying (3)

- enter `map(eval)[three,not$three,four,five,six,seven,eight]` into the text field



press *parse up*; press *simplify*

- enter `map(evalG)[three,not$three,four,five,six,seven,eight]` into the text field  
press *parse up*; press *simplify*; enter *drawS* into the entry field  
press *tree > tree*; press *paint*; adapt red support points and press *combis* to remove them



For the simplification steps of

```
map(stateflow)[three,not$three,four,five,six,seven,eight]
```

and

```
stateflow$not(three)/\six,
```

see [stateflow3.html](#) and [stateflow4.html](#), respectively.

## Acceptors and regular expressions

-- reglangs

constructs: a b c d e h aa gg pp vv final q qa qb qab q1 q2

defuncts: delta beta plus unfold unfoldND unfoldBro reg draw drawC

preds: Unfold

fovars: e st st' w k m

axioms:

states == [q] & labels == [a,b] & atoms == [final]

(q,a) -> qa & (q,b) -> qb &

(qa,a) -> q & (qa,b) -> qab &

(qb,a) -> qab & (qb,b) -> q &

(qab,a) -> qb & (qab,b) -> qa &

(q,a) -> q & (q,b) -> q1 &

(q1,a) -> q2 & (q1,b) -> q1 &

(q2,a) -> q1 & (q2,b) -> q1 &

final -> q & -- A1 acceptor of words with an even number of a's and an even  
-- number of b's if started in q

```

final -> qb & -- A2: acceptor of words with an even number of a's and an odd
-- number of b's if started in q

final -> q1 & -- A3: acceptor of reg$11

(Unfold(st) [] <== final -> st) &
(Unfold(st)(x:w) <== (st,x) -> st' & Unfold(st')$w) &

unfoldND$w == out <=< kfold(flip$transL)$w & -- nondeterministic unfold
-- (for kfold see base)
delta(st,x) == head$transL(st)$x &
beta$st == ite(null$out$st,0,1) &

unfold(st) == beta.foldl(delta)(st) & -- deterministic unfold

-- regular expressions

plus$e == e*star(e) &

reg$1 == star$plus(a)+plus(b*c) & --> star(a+b*c)

reg$2 == a+a*a*star(a)*b+a*b*star(b)+b & --> star(a)*b+a*star(b)

reg$3 == a+star(a)+eps+a & --> star(a)

```

```

reg$4 == (a+eps)*star(a)*(eps+a) &                                --> star(a)

reg$5 == (a+eps)*star(a+eps)*(a+eps)+a+eps &                      --> star(a)

reg$6 == star(a)*b*star(b+eps)*(b+eps)+star(a)*b & --> star(a)*b*star(b)

reg$7 == d*pp+c*h*e+a*b+c+a+a*b*c*d+star(a*a+b+a*b)+mt+c*aa+b*gg+eps+
         a+a*a+vv+a*a*8+d+b &
--> (a*a*8)+a+(c*aa)+c+d+(a*b*c*d)+(c*h*e)+(b*gg)+(d*pp) +
-- star((a*a)+(a*b)+b)+vv

reg$8 == a+((b+eps)*star(b+eps)*a)+ 
        ((a+((b+eps)*star(b+eps)*a))*star(a+b+eps+(mt*star(b+eps)*a))* 
         (a+b+eps+(mt*star(b+eps)*a))) &                                -- auto3
--> star(b)*a*star(a+b)

reg$9 == a+c+eps+((a+c+eps)*star(a+c+eps)*(a+c+eps))+ 
        ((b+((a+c+eps)*star(a+c+eps)*b))*star(a+b+eps+(c*star(a+c+eps)*b))* 
         (c+(c*star(a+c+eps)*(a+c+eps)))) &                                -- auto4
--> star(a+c)+(star(a+c)*b*star(a+b+(c*star(a+c)*b))*c*star(a+c))
-- autoToReg.minimize.regToAuto --> star(a+c+b*star(a+b)*c)

reg$10 == star(a)+star(a)*b*star(c*star(a)*b)*c*star(a) &

```

```

reg$11 == star(a)*b*star(a*(a+b)+b) &
-- widget interpreters

draw(m) == wtree(m)$fun((eps,k,n),text$eps,
                         (st,k,n),ite(Int$st,
                                      color(k,n)$circ$11,
                                      frame$blue$text$st)) &

drawC == wtree $ fun(eps,gif(cat,16,14),x,text$x)

```

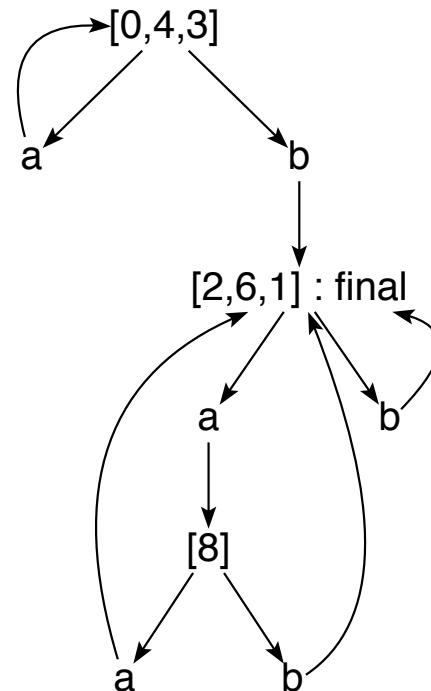
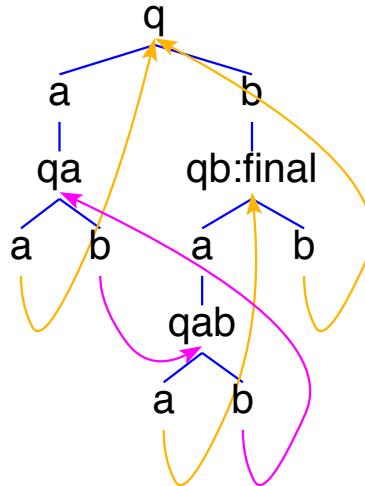
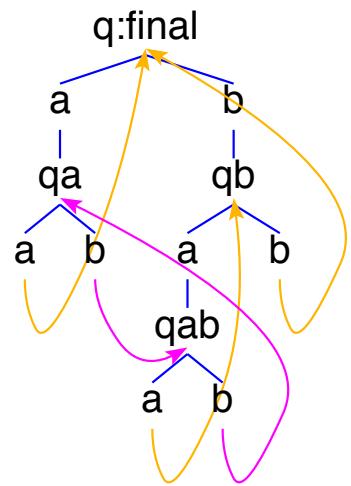
conjects:

Unfold(q) [] &	--> False	-- unify & narrow
Unfold(q) [b] &	--> True	-- derive & simplify & refute
Unfold(q) [b,b] &	--> False	
Unfold(q) [a,b,a,b,a,b,a] &	--> True	-- reglangs1
Unfold(q) [a,b,a,b,b,a,b,a] &	--> False	
Unfold(q) [a,b,a,a,b,a] &	--> False	
Unfold(q) [a,b,a,a,b,a,b]	--> True	

terms:

unfoldND[q] [] <+>	--> []	-- simplify
unfoldND[q] [b] <+>	--> [final]	

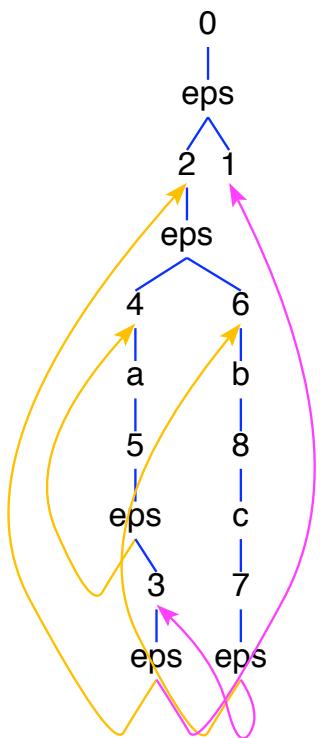
unfoldND[q] [b,b] <+>	--> []	
unfoldND[q] [a,b,a,b,a,b,a] <+>	--> [final]	
unfoldND[q] [a,b,a,b,b,a,b,a] <+>	--> []	
unfoldND[q] [a,b,a,a,b,a] <+>	--> []	-- 39 steps
unfoldND[q] [a,b,a,a,b,a,b] <+>	--> [final]	-- 45 steps
unfold(q) [a,b,a,a,b,a] <+>	--> 0	-- 23 steps
unfold(q) [a,b,a,a,b,a,b] <+>	--> 1	-- 26 steps
unfold(qgg) [a,b,a,a,b] <+>	--> 0	-- 22 steps
unfold(qgg) [a,b,a,a,b,a] <+>	--> 1	-- 25 steps
unfoldBro(reg\$2) [a,a,a,b] <+>	--> 1	
unfoldBro(reg\$2) [a,a,a,b,b] <+>	--> 0	
auto\$reg\$1 <+>	--> non-deterministic acceptor of reg\$1	
pauto\$reg\$1	--> deterministic acceptor of reg\$1	



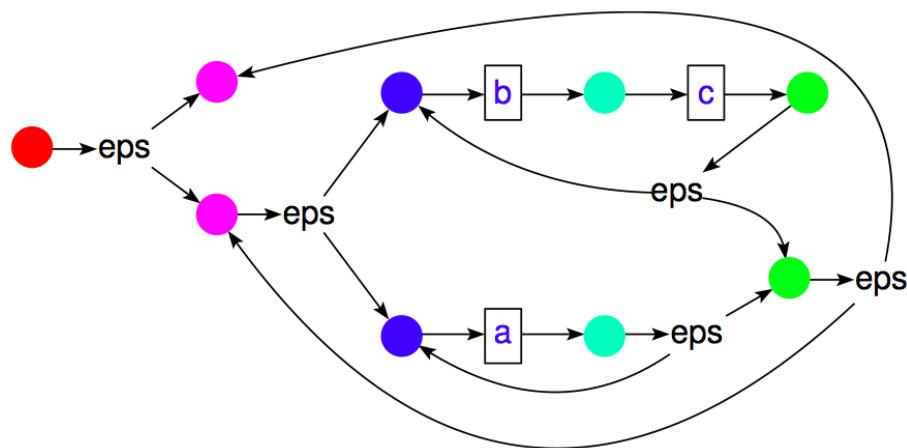
*A1, A2 and the minimal acceptor of `reg$11` that is isomorphic to A3 and whose state names were created by `powerAuto` (see below)*

The following nondeterministic automaton (with initial state 0, final state 1 and  $\epsilon$ -transitions) accepts the language of the regular expression  $reg1 = (a^+ + (bc)^+)^*$ . It results from simplifying `auto$reg$1`.

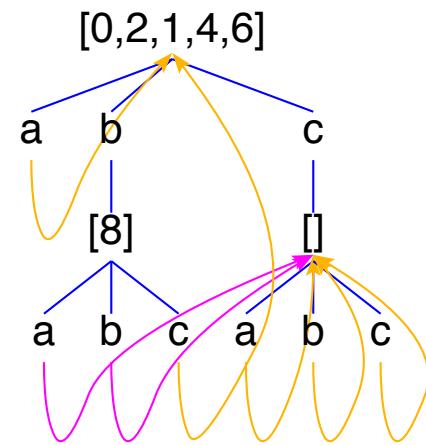
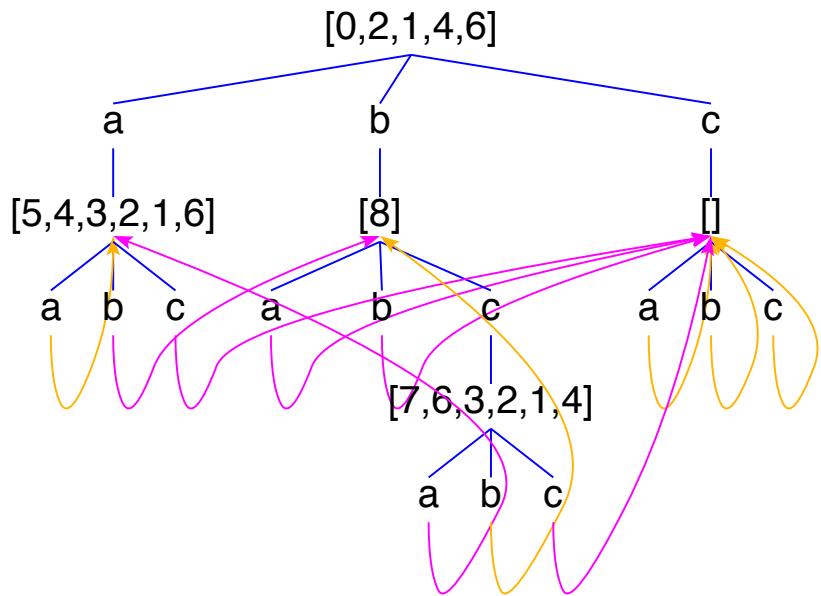
An application of `drawL` to the graph on the left leads to the graph on the right. The rotation by 90 degrees is achieved by drawing the graph in painter mode `t1-90`.



	eps	b	c	a
0	2 1			
2	4 6			
3	2 1			
4				5
5	4 3			
6		8		
7	6 3			
8			7	



The following equivalent deterministic automaton (with initial state  $[0,2,1,4,6]$  and final states  $[0,2,1,4,6]$ ,  $[5,4,3,2,1,6]$  and  $[7,6,3,2,1,4]$ ) results from simplifying `pauto$reg$1` or entering `reg$1` into the solver text field and pressing *specification > build Kripke model from regular expression*.



*Transitions of pauto\$reg\$1 - before and after minimization*

Again, the minimal automaton was obtained by pressing *specification > minimize* afterwards (see section 7).

## Data type, parser and unparser for regular expressions

```
data RegExp = Mt | Eps | Const String | Sum_ RegExp RegExp |  
Prod RegExp RegExp | Star RegExp deriving Eq  
  
parseRE :: Sig -> TermS -> Maybe (RegExp, [String])  
                                -- list of labels of acceptors  
parseRE _ (F "mt" [])      = Just (Mt, [])  
parseRE _ (F "eps" [])     = Just (Eps, ["eps"])  
parseRE sig (F a [])       = if sig.isDefunct a then Just (Var a, [a])  
                           else Just (Const a, [a])  
parseRE _ (F "+" [])       = Just (Mt, [])  
parseRE sig (F "+" [t])    = parseRE sig t  
parseRE sig (F "+" ts)    = do pairs <- mapM (parseRE sig) ts  
                            let (e:es,ass) = unzip pairs  
                            Just (foldl Sum_ e es,joinM ass)  
parseRE _ (F "*" [])       = Just (Eps, ["eps"])  
parseRE sig (F "*" [t])    = parseRE sig t  
parseRE sig (F "*" ts)    = do pairs <- mapM (parseRE sig) ts  
                            let (e:es,ass) = unzip pairs  
                            Just (foldl Prod e es,joinM ass)  
parseRE sig (F "plus" [t]) = do (e,as) <- parseRE sig t  
                            Just (Prod e $ Star e,as)  
parseRE sig (F "star" [t]) = do (e,as) <- parseRE sig t  
                            Just (Star e,as `join1` "eps")
```

```

parseRE sig (F "refl" [t]) = do (e,as) <- parseRE sig t
                                Just (Sum_ e Eps,as `join1` "eps")
parseRE _ (V x)               = Just (Var x,[x])
parseRE _ _                   = Nothing

showRE :: RegExp -> TermS
showRE Mt                      = leaf "mt"
showRE Eps                     = leaf "eps"
showRE (Const a)               = leaf a
showRE e@(Sum_ _ _) = case summands e of []  -> leaf "mt"
                           [e] -> showRE e
                           es  -> F "+" $ map showRE es
showRE e@(Prod _ _) = case factors e of []  -> leaf "eps"
                           [e] -> showRE e
                           es  -> F "*" $ map showRE es
showRE (Star e)                = F "star" [showRE e]
showRE (Var x)                 = V x

```

The following binary RegExp-relation are subrelations of regular language inclusion. In their defining equations,  $\leq$  means reverse implication.

```

instance Ord RegExp
  where Eps <= Star _                         -- used by summands
        e <= Star e' | e <= e'                  = True
        Prod e1 e2 <= Star e | e1 <= e && e2 <= Star e = True

```

```

Prod e1 e2 <= Star e | e1 <= Star e && e2 <= e = True
e <= Prod e' (Star _) | e == e' = True
e <= Prod (Star _) e' | e == e' = True
Prod e1 e2 <= Prod e3 e4           = e1 <= e3 && e2 <= e4
e <= Sum_ e1 e2                  = e <= e1 || e <= e2
e <= e'                           = e == e'

(<*) :: RegExp -> RegExp -> Bool                                -- used by factors
Sum_ Eps e <* Star e' | e == e' = True
e <* e' = False

(<+) :: RegExp -> RegExp -> Bool      -- Eps <+ e, e1*e2 <+ e3*e4 iff e1 <+ e3
Eps <+ e                           = True      -- products are ordered by left factors
Const a <+ Const b                 = a <= b
e@(Const _) <+ Prod e1 e2 = e <+ e1
e@(Star _) <+ Prod e1 e2 = e <+ e1
Prod e _ <+ e'@(Const _) = e <+ e'
Prod e _ <+ e'@(Star _) = e <+ e'
Prod e _ <+ Prod e' _           = e <+ e'
e <+ e'                           = e == e'

(+<) :: RegExp -> RegExp -> Bool      -- Eps +< e, e1*e2 <+ e3*e4 iff e2 <+ e4
Eps +< e                           = True      -- products are ordered by right factors
Const a +< Const b                 = a <= b
e@(Const _) +< Prod e1 e2 = e +< e2

```

```

e@(Star _) +< Prod e1 e2 = e +< e2
Prod _ e +< e'@(Const _) = e +< e'
Prod _ e +< e'@(Star _) = e +< e'
Prod _ e +< Prod _ e'      = e +< e'
e +< e'                      = e == e'

```

## *Flattening sums and products*

```

summands,factors :: RegExp -> [RegExp]
summands (Sum_ e e') = joinMapR (<=) summands [e,e']           -- e <= e' ==> e+e' = e'
summands Mt          = []                                         -- mt+e = e
summands e            = [e]                                         -- e+mt = e
factors (Prod e e')   = joinMapR (<*) factors [e,e']           -- e <* e' ==> e*e' = e'
factors Eps          = []                                         -- eps*e = e
factors e             = [e]                                         -- e*eps = e

joinMapR :: Ord b => (a -> [b]) -> [a] -> [b]
joinMapR f = foldl (foldl insertR) [] . map f

insertR :: Ord a => [a] -> a -> [a]
insertR s@(x:s') y | y <= x = s
                   | x <= y = insertR s' y
                   | True   = x:insertR s' y
insertR _ x = [x]

```

## *Sorting summands*

```

sortRE :: (RegExp -> RegExp -> Bool) -> ([RegExp] -> RegExp) -> RegExp
                                         -> RegExp
sortRE r prod e@(Sum_ _ _) = mkSumR $ map (sortRE r prod) $ qsort r
                                         $ summands e
sortRE r prod e@(Prod _ _) = prod $ map (sortRE r prod) $ factors e
sortRE r prod (Star e)      = Star $ sortRE r prod e
sortRE _ _ e                 = e

```

```

mkSumR,mkProdL,mkProdR :: [RegExp] -> RegExp
mkSumR []   = Mt
mkSumR es   = foldr1 Sum_ es
mkProdL []  = Eps
mkProdL es  = foldl1 Prod es
mkProdR []  = Eps
mkProdR es  = foldr1 Prod es

```

## Distributing products over sums

```

-- e*(eps+e') = e+e*e', e*(e1+e2) = e*e1+e*e2
-- (eps+e')*e = e+e'*e, (e1+e2)*e = e1*e+e2*e

```

```

distribute :: RegExp -> RegExp
distribute = fixpt (==) f
  where f e = case e of Sum_ _ _ -> mkSumR $ map f $ concatMap g $ summands e
        Prod e1 e2 -> case g e of [e1,e2] -> Sum_ (f e1) $ f e2
                                -> Prod (f e1) $ f e2

```

```

Star e      -> Star \$ distribute e
                  -> e
where g (Prod e (Sum_ Eps e')) = [e,Prod e e']
      g (Prod e (Sum_ e1 e2)) = [Prod e e1,Prod e e2]
      g (Prod (Sum_ Eps e') e) = [e,Prod e' e]
      g (Prod (Sum_ e1 e2) e) = [Prod e1 e,Prod e2 e]
      g e                      = [e]

```

## *Reducing regular expressions iteratively*

```

reduceLeft,reduceRight,reduceRE :: RegExp -> RegExp
reduceLeft  = fixpt (==) (reduceRE . sortRE (+<) mkProdL)
reduceRight = fixpt (==) (reduceRE . sortRE (<+) mkProdR)
reduceRE e  = case e of Sum_ e1 e2 -> if sizeRE e' < sizeRE e then e'
                                         else Sum_ (reduceRE e1) \$ reduceRE e2
                                             where es = summands e
                                                   e' = mkSumR \$ f es
                                                   f (Eps:es) = reduceS True es
                                                   f es        = reduceS False es
                                         Prod _ _ -> if Mt `elem` es then Mt else mkProdR es
                                                       where es = map reduceRE \$ factors e
Star Mt          -> Eps
Star Eps          -> Eps
Star (Star e)     -> Star e
Star (Sum_ Eps e) -> Star e
Star (Sum_ e Eps) -> Star e

```

```

-- star(star(e)+e') = star(e+e')
Star (Sum_ (Star e) e') -> Star $ Sum_ e e'
-- star(star(e)+e') = star(e+e')
Star (Sum_ e (Star e')) -> Star $ Sum_ e e'
-- star(e*star(e)+e') = star(e+e')
Star (Sum_ (Prod e1 (Star e2)) e3) | e1 == e2
                                         -> Star $ Sum_ e1 e3
-- star(star(e)*e+e') = star(e+e')
Star (Sum_ (Prod (Star e1) e2) e3) | e1 == e2
                                         -> Star $ Sum_ e1 e3
-- star(e'+e*star(e)) = star(e+e')
Star (Sum_ e1 (Prod e2 (Star e3))) | e2 == e3
                                         -> Star $ Sum_ e1 e2
-- star(e'+star(e)*e) = star(e+e')
Star (Sum_ e1 (Prod (Star e2) e3)) | e2 == e3
                                         -> Star $ Sum_ e1 e2
Star e -> Star $ reduceRE e
-           -> e

```

```

sizeRE :: RegExp -> Int
sizeRE Eps          = 0
sizeRE (Sum_ e e') = sizeRE e + sizeRE e'
sizeRE (Prod e e') = sizeRE e + sizeRE e' + 1
sizeRE (Star e)     = sizeRE e + 1
sizeRE _             = 1

```

```
reduceS :: Bool -> [RegExp] -> [RegExp]
```

```
-- eps+e*star(e) = star(e),   eps+star(e)*e = star(e)      (1)
-- eps+e*e*star(e) = e+star(e), eps+star(e)*e*e = e+star(e)    (2)
```

```
reduceS True (Prod e (Star e'):es) | e == e' = reduceS False $ Star e:es
reduceS True (Prod (Star e) e':es) | e == e' = reduceS False $ Star e:es
reduceS True (Prod e (Prod e1 (Star e2)):es)
                                         | e == e1 && e == e2 = reduceS False $ e:Star e:es
reduceS True (Prod (Prod e e1) (Star e2):es)
                                         | e == e1 && e == e2 = reduceS False $ e:Star e:es
reduceS True (Prod (Star e1) (Prod e2 e):es)
                                         | e == e1 && e == e2 = reduceS False $ e:Star e:es
reduceS True (Prod (Prod (Star e1) e2) e:es)
                                         | e == e1 && e == e2 = reduceS False $ e:Star e:es
```

```
-- e+e*e' = e*(eps+e'), e+e'*e = (eps+e')*e      prepares (1) and (2)
-- e*e1+e*e2 = e*(e1+e2), e1*e+e2*e = (e1+e2)*e
```

```
reduceS b (e:Prod e1 e2:es) | e == e1 = reduceS b $ Prod e (Sum_ Eps e2):es
reduceS b (Prod e1 e2:e:es) | e == e2 = reduceS b $ Prod (Sum_ Eps e1) e:es
reduceS b (Prod e1 e2:Prod e3 e4:es)
                                         | e1 == e3 = reduceS b $ Prod e1 (Sum_ e2 e4):es
reduceS b (Prod e1 e2:Prod e3 e4:es)
                                         | e2 == e4 = reduceS b $ Prod (Sum_ e1 e3) e2:es
```

```

reduceS b (e:es) = e:reduceS b es
reduceS True _    = [Eps]
reduceS _ _       = []

```

A regular expression **reg** entered into the text field is reduced by simplifying **reduce\$reg**.

A regular expression displayed on the canvas (resp. its highlighted subexpression) is reduced by pressing *specification > reduce regular expression*.

### *Nondeterministic acceptor with $\epsilon$ -transitions of a regular language*

```

type NDA = Int -> String -> [Int]

regToAuto :: RegExp -> ([Int],NDA)
regToAuto e = ([0..nextq-1],delta)
  where (delta,nextq) = eval e 0 1 (const2 []) 2
        eval :: RegExp -> Int -> Int -> NDA -> Int -> (NDA,Int)
        eval Mt _ _ delta nextq           = (delta,nextq)
        eval Eps q q' delta nextq        = (upd2L delta q "eps" q',nextq)
        eval (Const a) q q' delta nextq = (upd2L delta q a q',nextq)
        eval (Sum_ e e') q q' delta nextq = eval e' q q' delta' nextq'
          where (delta',nextq') = eval e q q' delta nextq
        eval (Prod e e') q q' delta nextq = eval e' nextq q' delta' nextq'
          where (delta',nextq') = eval e q nextq delta $ nextq+1

```

```

eval (Star e) q q' delta nextq      = (delta2,nextq')
    where q1 = nextq+1
          (delta1,nextq') = eval e nextq q1 delta $ q1+1
          delta2 = fold2 f delta1 [q,q1,q1,q] [nextq,nextq,q',q']
          f delta q = upd2L delta q "eps"

upd2L :: (Eq a,Eq b,Eq c) => (a -> b -> [c]) -> a -> b -> c -> a -> b -> [c]
upd2L f a b = upd f a . upd (f a) b . join1 (f a b)

```

*Deterministic acceptor constructed from a nondeterministic one with  $\epsilon$ -transitions*

```

type PDA = [Int] -> String -> [Int]

powerAuto :: NDA -> [String] -> ([[Int]],PDA)
powerAuto nda labs = (reachables,deltaH)
    where reachables = fixpt subset deltaM [epsHull [0]]
          deltaM qss = qss `join` [deltaH qs a | qs <- qss, a <- labs]
          deltaH qs = epsHull . delta qs
          delta qs a = joinMap (flip nda a) qs
          epsHull :: [Int] -> [Int]
          epsHull qs = if qs' `subset` qs then qs else epsHull $ qs `join` qs'
                           where qs' = delta qs "eps"

```

*regToAuto* and *powerAuto* are applied successively to the regular expression on the canvas by pressing *specification > build Kripke model from regular expression*.

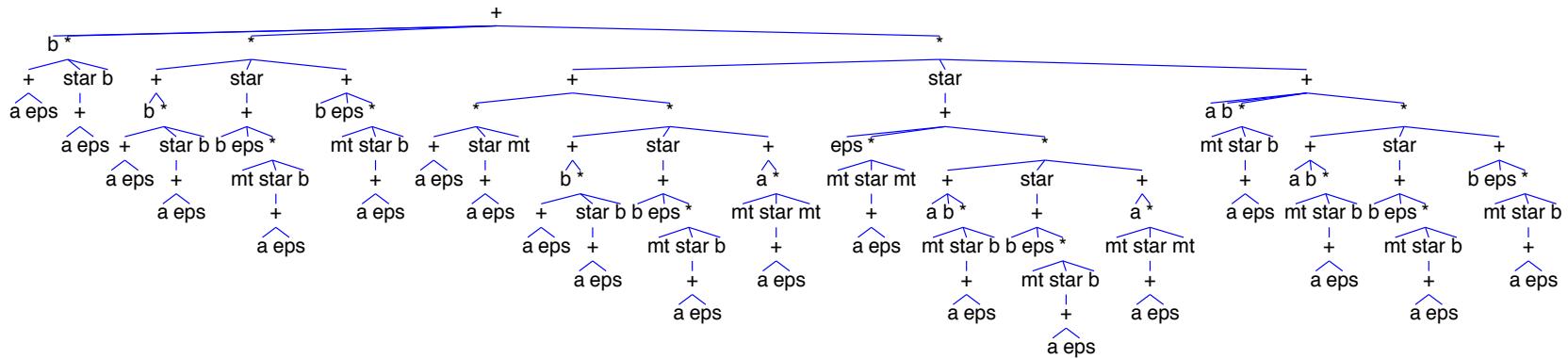
## Two ways of building a regular expression from an acceptor

### Kleene's algorithm

```
autoToReg :: Sig -> TermS -> RegExp
autoToReg sig start = if null finals then Const "no final states"
                      else foldl1 Sum_ $ map (f lg i) finals
    where finals = sig.value!!0
          lg = length sig.states-1
          Just i = search (== start) sig.states
          f (-1) i j = if i == j then Sum_ (delta i i) Eps else delta i j
          f k i j      = Sum_ (f' i j) $
                            Prod (f' i k) $ Prod (Star $ f' k k) $ f' k j
                            where f' = f $ k-1
          delta :: Int -> Int -> RegExp
          delta i j = if null labs then Mt else foldl1 Sum_ labs
                        where labs = [Const $ showTerm0 $ (sig.labels)!!k |
                                      k <- indices_ sig.labels,
                                      sig.transL!!i!!k == [j]]
```

If an initial state has been entered into the entry field and the current Kripke model is a deterministic acceptor, *autoToReg* is applied to it by pressing *specification > build regular expression*.

Regular expressions produced by Kleene's algorithm are often rather space-consuming. For instance, acceptor A3 of **reglangs** is transformed into:



Applying *reduceLeft* (see above; press *specification > reduce left-folded regular expression*) leads to:

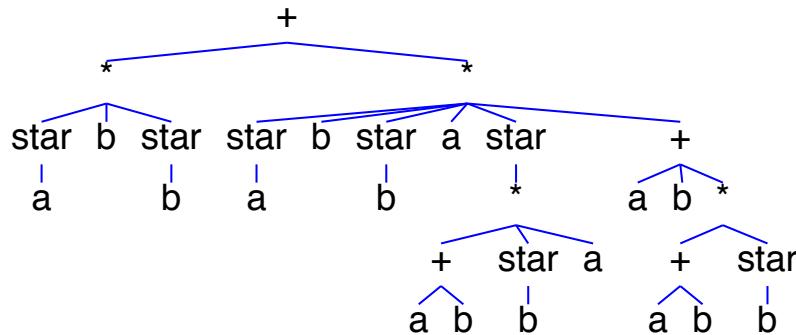
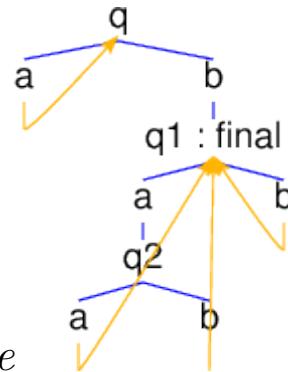


Fig. 5.

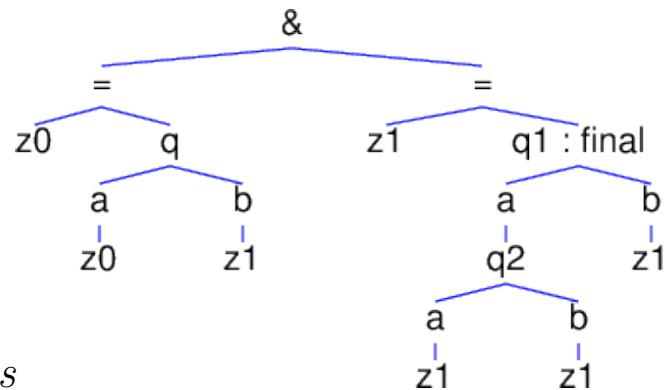
An alternative, more algebraic method to build regular expressions (REs) from acceptors has been described in [5], section 2.3, and recently been generalized to polynomial coalgebras

[2]. It resembles **Gaussian elimination** insofar as the acceptor (graph) is turned into a set of equations with variables (here for REs), which is then solved via a sequence of replacements of variables by solving REs.

For instance, the following sequence of Expander2 commands leads from the transition graph of A3 to a solution of variable  $z_0$  that is equivalent to the RE of Fig. 5. For proving the equivalence with Expander2, see chapter 9.

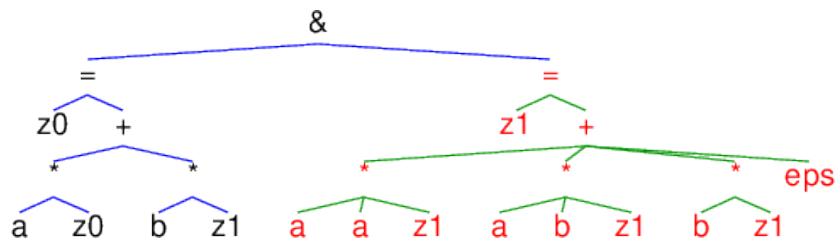
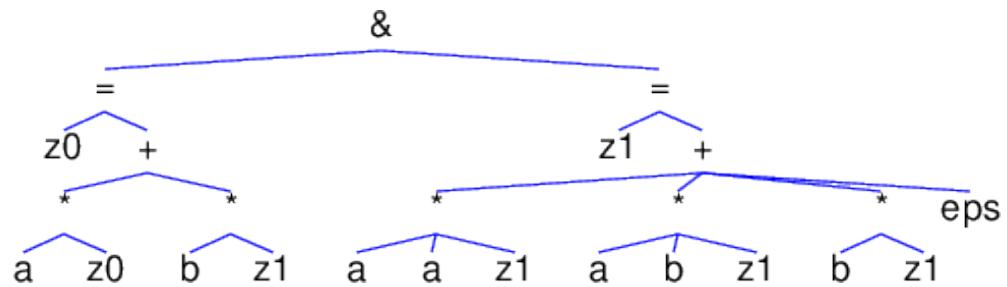


*graph > show graph of labelled Kripke model here*

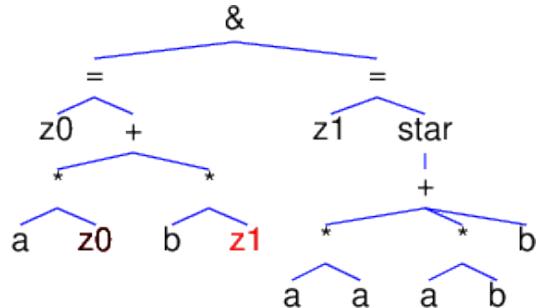
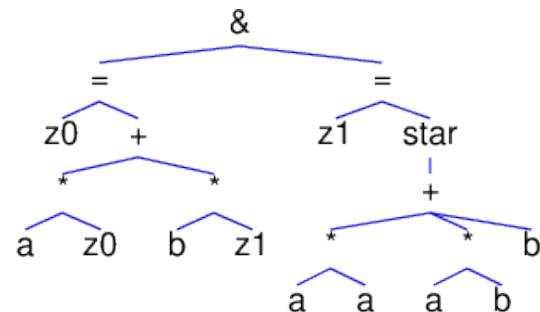


*graph > build iterative equations*

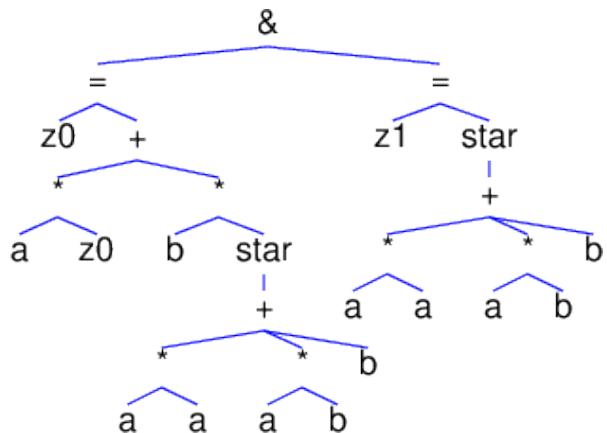
*specification > regular equations  
(see below)*

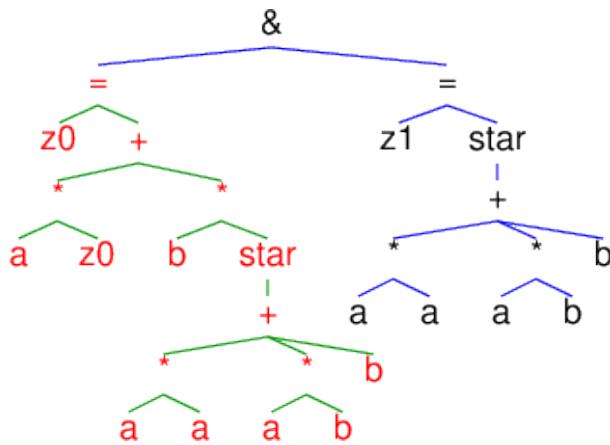


*specification >  
solve regular  
equation  
(see below)*

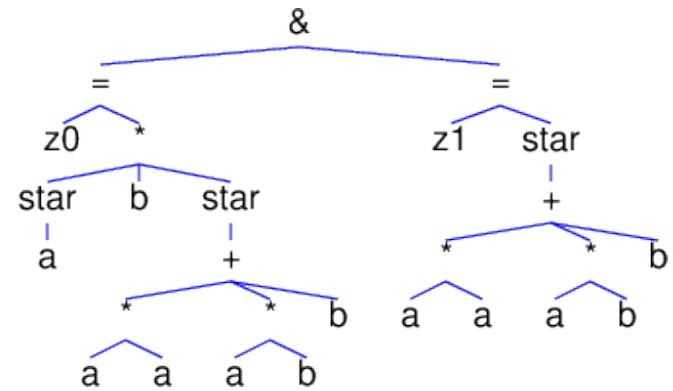


*specification >  
substitute variables  
(see below)*





*specification >  
solve regular  
equation*



```
autoEqsToRegEqs :: Sig -> [IterEq] -> [IterEq]
autoEqsToRegEqs sig = map f
```

```
where f (Equal x t) = case parseRE sig $ g t of
    Just (e,_) -> Equal x $ showRE $ distribute e
    _ -> Equal x $ leaf "OOPS"
    g (F q ts)     = F "+" $ if n >= 0 && drop n q == "final"
                    then leaf "eps":map h ts else map h ts
                    where n = length q-5
    g t             = t
    h (F lab [t]) = F "*" [F lab [],g t]
    h t             = t
```

The following function turns a *regular equation*  $x = t_1 \cdot x + \cdots + t_n \cdot x + t$  with into its least solution  $(t_1 + \cdots + t_n)^* \cdot t$ , which is unique if  $\epsilon \notin \{t_1, \dots, t_n\}$  (see [5], section 2.3).

```

solveRegEq :: Sig -> IterEq -> Maybe TermS
solveRegEq sig (Equal x t) = do (e,_) <- parseRE sig $ f $ case t of F "+" ts -> ts
                                _ -> [t]
                                         Just $ mkEq (V x) $ showRE e
where f ts = case us of [] -> leaf "mt"
                        [u] -> F "*" [star,u]
                        _ -> F "*" [star,F "+" us]
where us = ts `minus` recs
      recs = [t | t@(F "*" us) <- ts, last us == V x]
      star = case recs of
                  [] -> leaf "eps"
                  [rec] -> F "star" [g rec]
                  _ -> F "star" [F "+" $ map g recs]
g (F "*" [t,_]) = t
g (F "*" ts)     = F "*" $ init ts
g _               = leaf "OOPS"

```

```

substituteVars :: TermS -> [IterEq] -> [[Int]] -> Maybe TermS
substituteVars t eqs ps = do guard $ all isV ts
                                         Just $ fold2 replace1 t ps $ map (subst . root) ts
where ts = map (getSubterm1 t) ps
      subst x = if nothing t || x `isin` u then V x else u
                  where t = lookup x $ eqPairs eqs
                  u = get t

```

\*\*\*\*

-- eosum

constructs: esum osum elab olab final unfold  
defuncts: parity  
preds: even odd

axioms:

parity\$n == ite(even\$n,elab,olab) &  
(even\$0 <==> True) &  
(even\$suc\$x <==> odd\$x) &  
(odd\$0 <==> False) &  
(odd\$suc\$x <==> even\$x) &

-- acceptor of number sequences whose sum is even

states == [esum] & labels == [elab,olab] & atoms == [final] &  
(esum,elab) -> esum &  
(esum,olab) -> osum &  
(osum,olab) -> esum &  
(osum,elab) -> osum &

```
final -> esum &
```

```
unfold$s == out <=< (kfold(flip$transL)$s) . map(parity)
```

terms:

```
unfold[esum] [1..4] <+> --> [final]
```

```
unfold[esum] [1..5] <+> --> []
```

```
unfold[osum] [1..4] <+> --> []
```

```
unfold[osum] [1..5] <+> --> [final]
```

```
sum[1..4] <+> --> 10
```

```
sum[1..5] <+> --> 15
```

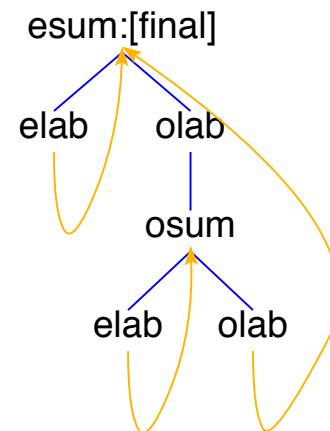


Fig. 6. Kripke model of eosum

## State equivalence and minimization

The command *specification > state equivalence* computes the equivalence of states,  $\sim \subseteq Q^2$ , of the actual Kripke model  $K = (Q, Lab, At, trans, transL, out, outL)$ .

$\sim$  is also called *K-bisimilarity* and defined as the greatest binary relation on  $Q$  such that for all  $q, q' \in Q$ ,

$$q \sim q' \Rightarrow \left\{ \begin{array}{l} out(q) = out(q') \wedge \\ \forall x \in Lab : outL(q)(x) = outL(q')(x) \wedge \\ (trans(q), trans(q')) \in lift(\sim) \wedge \\ \forall x \in Lab : (transL(q)(x), transL(q')(x)) \in lift(\sim) \end{array} \right.$$

where

$$lift(\sim) = \{(qs, qs') \in \mathcal{P}(Q)^2 \mid \forall q \in qs \exists q' \in qs' : q \sim q', \\ \forall q' \in qs' \exists q \in qs : q \sim q'\}.$$

Hence  $\sim$  can be constructed stepwise as follows:

$$\begin{aligned} \sim &= \bigcap_{n \in \mathbb{N}} \sim_n, \\ \sim_0 &= \{(q, q') \in Q^2 \mid out(q) = out(q'), \\ &\quad \forall x \in Lab : outL(q)(x) = outL(q')(x)\}, \end{aligned}$$

$$\begin{aligned}\sim_{n+1} = \{(q, q') \in Q^2 \mid & (trans(q), trans(q')) \in lift(\sim_n), \\ & \forall x \in Lab : (transL(q)(x), transL(q')(x)) \in lift(\sim_n)\}.\end{aligned}$$

Let  $A = \mathcal{P}(|Q|^2 \times \mathcal{P}(\mathcal{P}(|Q|^2)))$  and  $\chi(\supseteq)$  denote the characteristic function of inverse set inclusion.

Expander2 computes  $\sim$  with the Paull-Unger method:

$$\sim = \{(states!!i, states!!j) \mid \exists ps : (i, j, ps) \in \textcolor{red}{fixpt}(\chi(\supseteq))(bisimStep)(bisim_0)\}$$

where

$$\begin{aligned}bisim_0 = \{(i, j, \{(trans!!i, trans!!j)\}) \\ \cup \{(trans!!i!!k, trans!!j!!k) \mid 1 \leq k \leq |labels|\}) \\ \mid i < j, states!!i \sim_0 states!!j\}\end{aligned}$$

and for all  $R \in A$ ,

$$bisimStep(R) = \{(k, l, ps) \in R \mid \forall (is, js) \in ps \forall i \in is \exists j \in js : \\ i = j \vee \exists ps' : (i, j, ps') \in R\}.$$

For the fixpoint generator  $fixpt : (A \rightarrow A \rightarrow \text{Bool}) \rightarrow (A \rightarrow A) \rightarrow (A \rightarrow A)$ , see section Global Semantics.

## Haskell code for state equivalence

```
bisim :: Sig -> [(Int,Int)]
bisim = map (\(i,j,_) -> (i,j)) . fixpt supset bisimStep . bisim0

fixpt :: (a -> a -> Bool) -> (a -> a) -> a -> a
fixpt rel step = iter where
    iter a = if b `rel` a then a else iter b where b = step a

type Triples a b = [(a,a,[b])]
type TriplesL a  = Triples a ([a],[a])

bisim0 :: Sig -> TriplesL Int
bisim0 sig is ks = [(i,j,f i j) | i <- is, j <- is, i < j,
                           eqSet (==) (out sig!!i) $ out sig!!j,
                           and $ zipWith (eqSet (==)) (outL sig!!i) $ outL sig!!j]
    where is = indices_ sig.states; ks = indices_ sig.labels
          b = null sig.trans
          f i j = if b then s else s `join1` (sig.trans!!i,sig.trans!!j)
                    where s = mkSet $ map h ks
                          h k = (sig.transL!!i!!k,sig.transL!!j!!k)
```

```

bisimStep :: Eq a => TriplesL a -> TriplesL a
bisimStep trips = [trip | trip@(_,_,bs) <- trips, all r bs] where
    r (is,js) = forallThereis r' is js && forallThereis r' js is
    r' i j = i == j || just (lookupL i j trips) || just (lookupL j i trips)

lookupL :: (Eq a,Eq b) => a -> b -> [(a,b,c)] -> Maybe c
lookupL a b ((x,y,z):s) = if a == x && b == y then Just z else lookupL a b s
lookupL _ _ _ = Nothing

simplifyS (F " $\sim$ " [t,u]) sig | notnull sig.states
                                = do i <- search (== t) sig.states
                                    j <- search (== u) sig.states
                                    let cli = search (elem i) part
                                        clj = search (elem j) part
                                    Just $ mkConst $ cli == clj
                                where part = mkPartition (indices_ sig.states) $ bisim sig

mkPartition :: Eq a => [a] -> [(a,a)] -> [[a]]
mkPartition = foldl f . map single where
    f part (a,b) = if eqSet (==) cla clb
                    then part else (cla++clb):minus part s
                    where s@[cla,clb] = map g [a,b]
                        g a = head $ filter (elem a) part

```

Hence a goal of the form  $t \sim u$  is simplified by applying *bisim* to  $(t,u)$ .

The equivalence of  $t$  and  $u$  can also be checked by applying the axioms for  $eqState$  ( $eqStateL$  in the case of labelled system) or  $\sim\sim$  ( $\sim\sim L$  in the case of labelled system) of **modal** to  $(t, u)$  (see chapter 4). The value of  $eqState(t, u)$  ( $eqStateL(t, u)$ ) is obtained by simplification; the value of  $t \sim\sim u$  ( $t \sim\sim L u$ ) is obtained by a coinduction step, followed by simplification steps.

### *Stepwise construction of $\sim$*

```
simplifyS (F x@"equivStep" []) sig | notnull sig.states = Just $ F x [Hidden mat]
    where sts = map showTerm0 sig.states
          mat = ListMatL sts [(f i,f j,map (apply2 $ map f) ps)
                               | (i,j,ps) <- bisim0 sig]
          f = showTerm0 . (sig.states!!)

simplifyS (F x@"equivStep" [Hidden mat]) _ =
    do ListMatL sts s <- Just mat; let new = bisimStep s
       guard $ s /= new
       Just $ F x [Hidden $ ListMatL sts new]
```

If *matrices* has been selected in the *interpreter menu* below the *paint* button, a simplification step of **equivStep** returns a matrix representation of  $bisim_0$  in the painter. Further simplification steps in the painter return matrix representations of  $bisimStep(bisim_0)$ ,  $bisimStep^2(bisim_0)$ , etc.

The command *specification* > *minimize* minimizes the actual Kripke model  $K$  by building the quotient of  $K$  with respect to  $\sim$ :

```

mkQuotient :: Sig -> ([TermS],[[Int]],[[[Int]]],[[Int]],[[[[Int]]]])
mkQuotient sig = (states,tr,trL,va,vaL)
    where part = mkPartition (bisim sig) $ indices_ sig.states
          states = map ((sig.states!!) . minimum) part
          oldPos i = get $ search (== (states!!i)) sig.states
          newPos i = get $ search (elem i) part
          h = mkSet . map newPos
          [is,js,ks] = map indices_ [states,labels,atoms]
          tr = map f is where f i = h $ sig.trans!!oldPos i
          trL = map f is where f i = map (g i) js
                           g i x = h $ sig.transL!!oldPos i!!x
          va = map f ks where f i = h $ sig.value!!i
          vaL = map f ks where f i = map (g i) js
                           g i x = h $ sig.valueL!!i!!x

```

Two Kripke models  $\mathcal{K}$  and  $\mathcal{K}'$  with initial states  $st$  resp.  $st'$  can be checked for equivalence simply by proving  $st \sim st'$  with respect to the sum  $\mathcal{K} \uplus \mathcal{K}'$ .

## Examples

```
-- auto1
```

constructs: D E dot plus minus final

defuncts: states labels atoms

axioms:

```
states == [1]
```

```
& labels == [D,E,dot,plus,minus]
```

```
& atoms == [final]
```

```
& (1,D) -> 2 & (2,D) -> 3 & (2,dot) -> 4 & (3,D) -> 5 & (3,dot) -> 4
```

```
& (4,D) -> 6 & (4,E) -> 7 & (5,D) -> 5 & (5,dot) -> 4 & (6,D) -> 6 & (6,E) -> 7
```

```
& (7,D) -> 9 & (7,plus) -> 8 & (7,minus) -> 8 & (7,dot) -> 10 & (7,E) -> 10
```

```
& (8,D) -> 9 & (9,D) -> 9
```

```
& (x `in` [1,8,9] & y `in` labels-[D] |  
  x `in` [2,3,5] & y `in` labels-[D,dot] |  
  x `in` [4,6] & y `in` labels-[D,E] ==> (x,y) -> 10)
```

```
& final -> branch[2,3,4,5,6,9]
```

```
filter(5~)$states --> [2,3,5]
```

```
-- specification > state equivalence --> [(2,3),(2,5),(3,5),(4,6)]
```

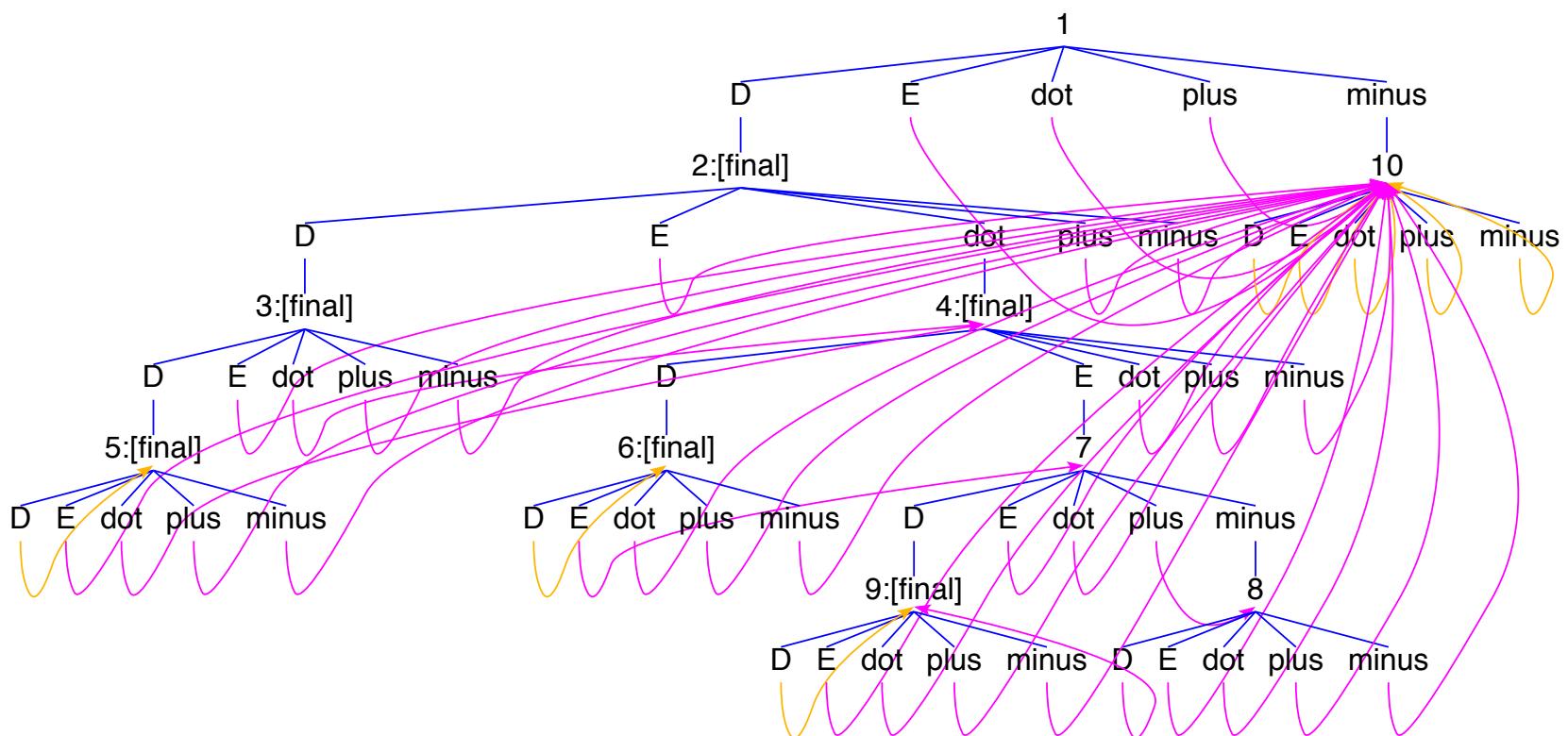


Fig. 7. Kripke model of auto1

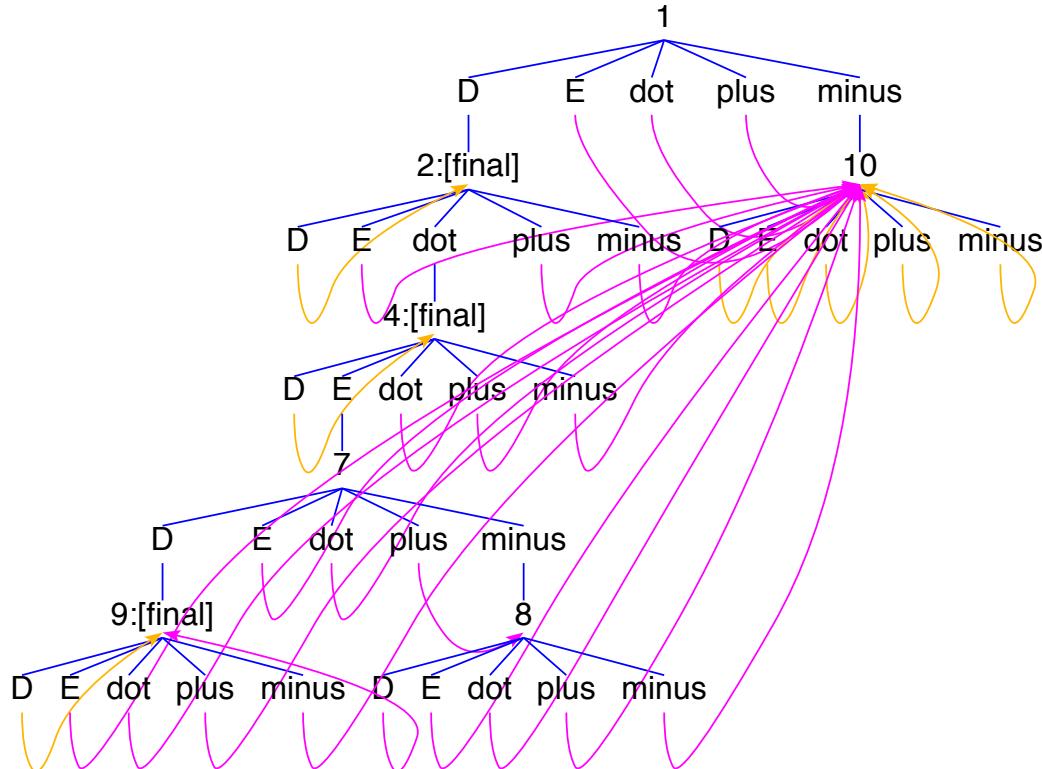


Fig. 8. Kripke model of `auto1` after minimization

The values of  $bisim_0$ ,  $bisimStep(bisim_0)$  and  $bisimStep^2(bisim_0)$  (obtained by stepwise simplifying `equivStep`; see above) are listed here: [autolequiv.html](#).

```
-- auto2
```

```
constructs: a b c final
```

```
axioms:
```

```
states == [1]
& labels == [a,b,c]
& atoms == [final]

& (1,a) -> 3 & (1,b) -> 4 & (1,c) -> 2 & (2,a) -> 5 & (2,b) -> 5 & (2,c) -> 4
& (3,a) -> 6 & (3,b) -> 2 & (3,c) -> 3 & (4,a) -> 3 & (4,b) -> 1 & (4,c) -> 6
& (5,a) -> 5 & (5,b) -> 2 & (5,c) -> 3 & (6,a) -> 5 & (6,b) -> 5 & (6,c) -> 1

& (final,a) -> branch[2,3,5,6] & (final,b) -> branch[1..6]
& (final,c) -> branch[1,4]

-- specification > state equivalence --> [(1,4),(2,6)]
```

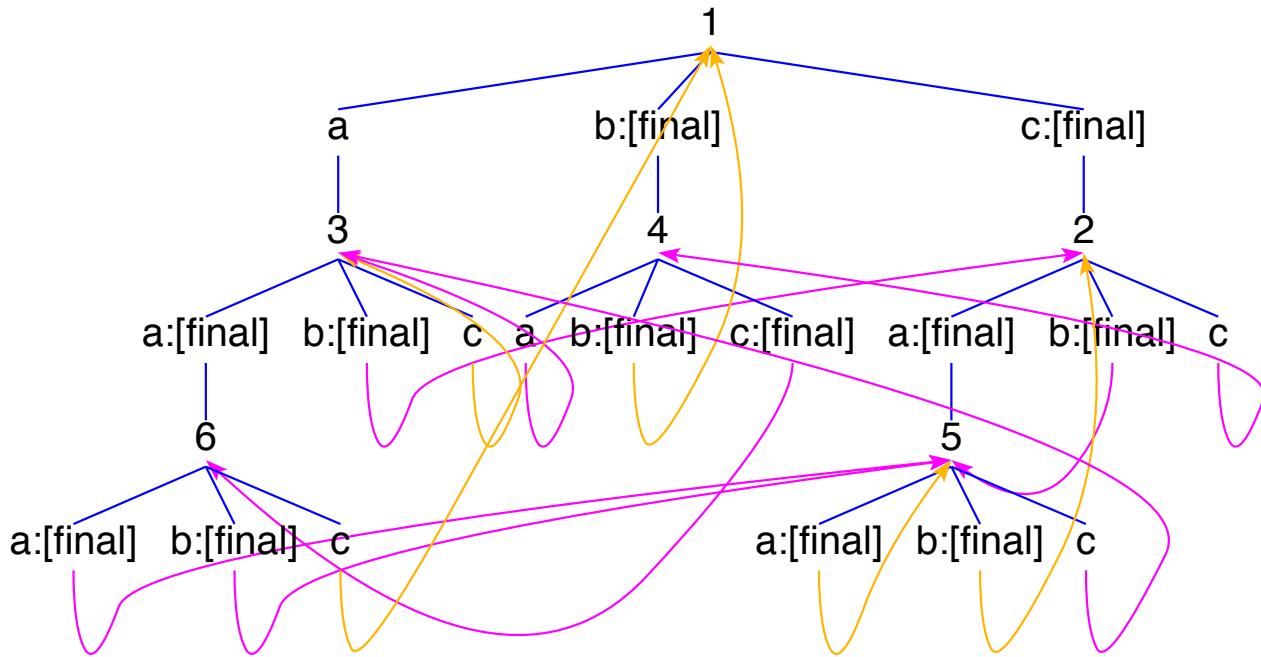


Fig. 9. *Kripke model of auto2*

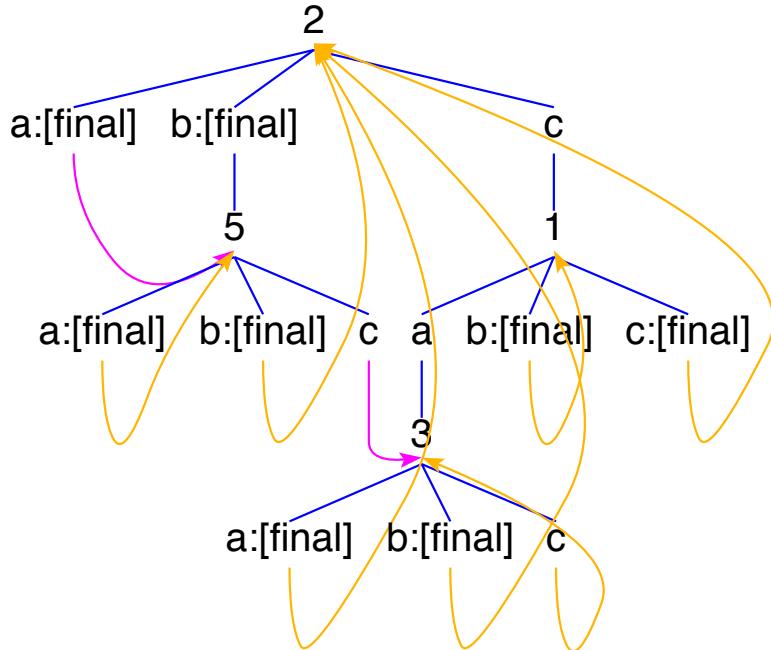


Fig. 10. Kripke model of auto2 after minimization

The values of  $bisim_0$ ,  $bisimStep(bisim_0)$  and  $bisimStep^2(bisim_0)$  (obtained by stepwise simplifying `equivStep`; see above) are listed here: [auto2equiv.html](#).

```
-- coin1
```

```
specs:      modal
constructs: start coin coffee tea odd is less
```

axioms:

```
states == [0] &
labels == [start,coin,coffee,tea] &
atoms == [less(5),is(6)] & -- (*)

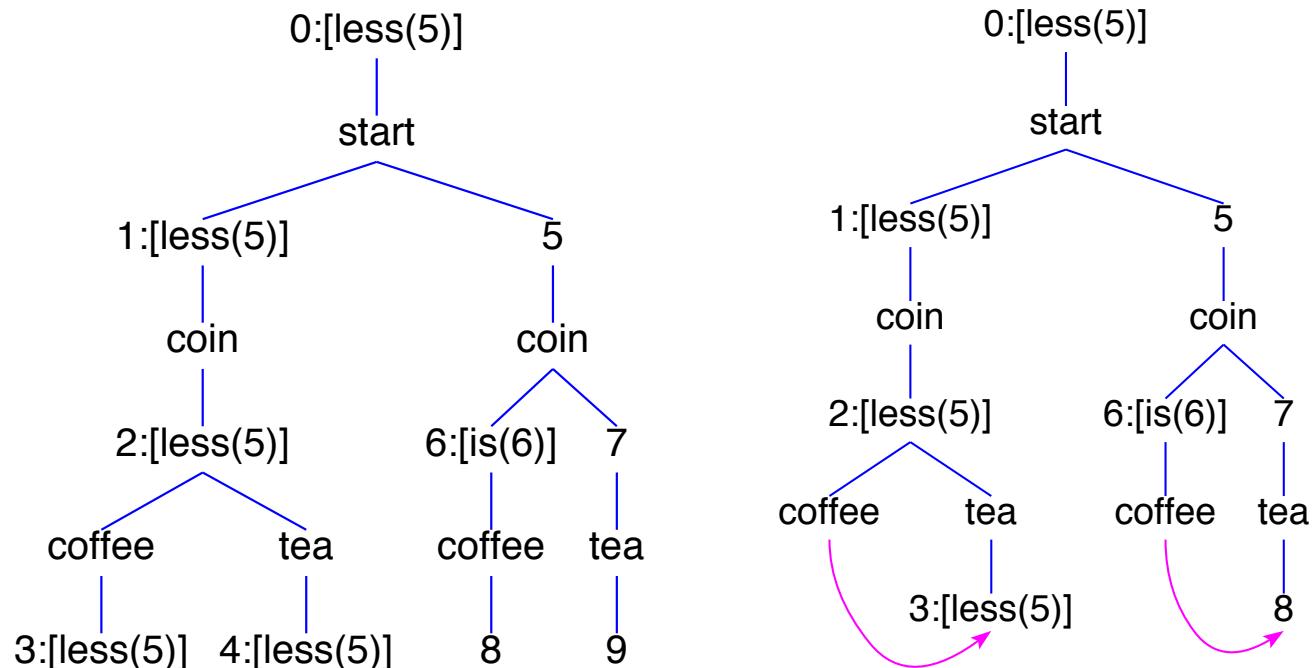
(0,start) -> branch[1,5] & (2,coffee) -> 3 &
(1,coin) -> 2 & (2,tea) -> 4 &
(5,coin) -> branch[6,7] & (6,coffee) -> 8 &
(7,tea) -> 9

is$st -> st &
less$st -> valid(<st)

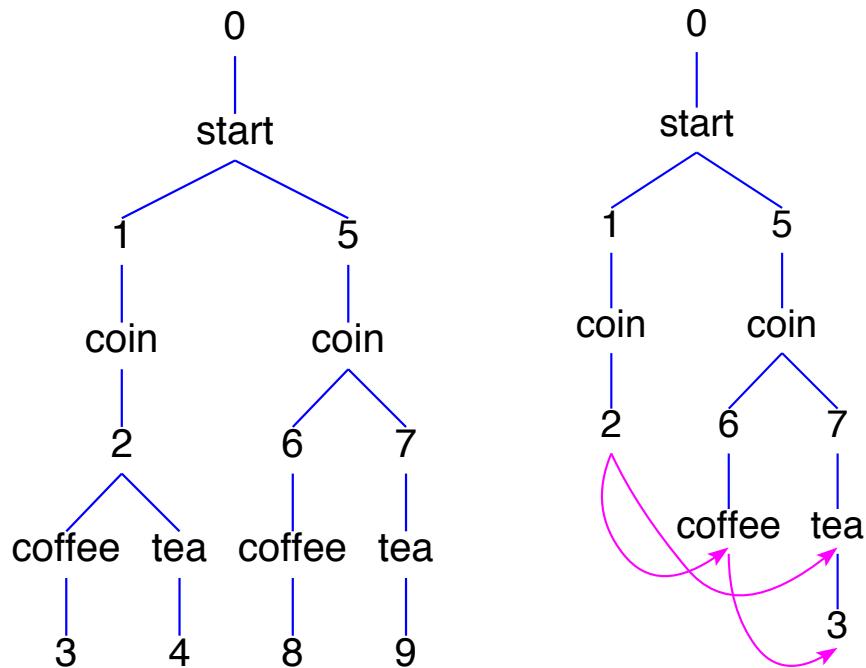
-- specification > state equivalence
    --> [(3,4),(3,8),(3,9),(4,8),(4,9),(8,9)]
    --> with (*): [(3,4),(8,9)]
```

terms:

```
eval(AF$less(5)\$/is(6)) --> [0,1,2,3,4,6]
<+> eval(FE$less(5)\$/is(6)) --> [0..9]
```



*Kripke model of coin1 with (\*) before and after minimization*



*Kripke model of coin1 without (\*) before and after minimization*

-- coin2

constructs: start coin coffee tea odd

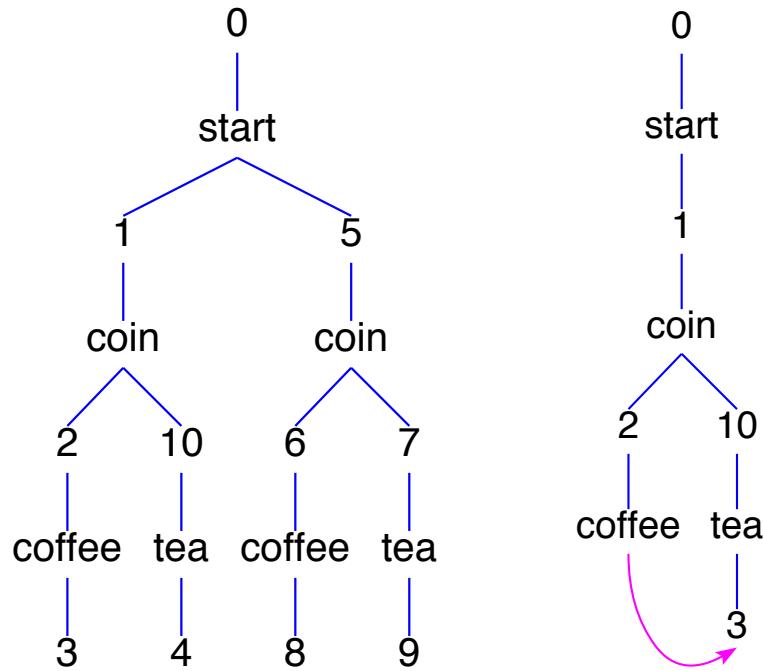
defuncts: states labels

axioms:

```

states == [0] & labels == [start,coin,coffee,tea] &
(0,start) -> branch[1,5] & (2,coffee) -> 3 &
(1,coin) -> branch[2,10] & (10,tea) -> 4 &
(5,coin) -> branch[6,7] & (6,coffee) -> 8 & (7,tea) -> 9
-- specification > state equivalence
--> [(1,5),(2,6),(10,7),(3,4),(3,8),(3,9),(4,8),(4,9),(8,9)]

```



*Kripke model of coin2 before and after minimization*

## Simplifications

(siehe auch [14], Kapitel 4 und 5, oder [15]) Simplifikationen führen zu **normalisierten Formeln**, die von folgender Grammatik erzeugt werden:

$$NF \rightarrow Impl \mid EC \mid AD$$

$$Impl \rightarrow \forall X(CA \Rightarrow DE)$$

$$CA \rightarrow \forall X NF \wedge \dots \wedge \forall X NF$$

$$DE \rightarrow \exists X NF \vee \dots \vee \exists X NF$$

$$EC \rightarrow \exists X(Ate \wedge \dots \wedge Ate)$$

$$AD \rightarrow \forall X(Ati \vee \dots \vee Ati)$$

$$Ate \rightarrow V = T \mid T = V \mid F(T) = F(T) \mid F(T) = C(T) \mid C(T) = F(T) \mid P(T)$$

$$Ati \rightarrow V \neq T \mid T \neq V \mid F(T) \neq F(T) \mid F(T) \neq C(T) \mid C(T) \neq F(T) \mid P(T)$$

$$C \rightarrow constructors$$

$$F \rightarrow defined\ functions$$

$$P \rightarrow relations$$

$$T \rightarrow terms\ and\ term\ tuples$$

$$X \rightarrow lists\ of\ bound\ variables$$

$$V \rightarrow variables$$

Jeder Simplifikationsschritt eines Termgraphen  $t$  besteht in der Tiefen- oder Breitensuche nach einem Redex, also einem Teilterm  $u$  von  $t$ , und dessen Ersetzung durch einen Term  $v$ , der sich wie folgt aus  $u$  ergibt:

- (1) Lässt sich  $u$  mit der u.g. Funktion *evaluate* zu einem von  $u$  verschiedenen Term  $v$  bzgl. eingebauter Algebren partiell auswerten, dann ist  $v$  das Redukt.
- (2) Scheitert (1) und ist eine der durch *simplifyGraph* definierten Graphersetzungsregeln auf  $u$  anwendbar, dann liefert deren Konklusion das Redukt.
- (3) Scheitert (2), dann wird versucht, ein Simplifikationsaxiom, also eine evtl. bedingte Gleichung

$$\varphi_1 \wedge \cdots \wedge \varphi_n \implies f(t_1, \dots, t_n) == t,$$

eine evtl. bedingte Äquivalenz

$$\varphi_1 \wedge \cdots \wedge \varphi_n \implies p(t_1, \dots, t_n) \iff \varphi$$

oder ein evtl. bedingtes Transitionsaxiom

$$\varphi_1 \wedge \cdots \wedge \varphi_n \implies f(t_1, \dots, t_n) \rightarrow t,$$

von links nach rechts auf  $u$  anzuwenden. Gelingt dies, dann liefert die Konklusion des Axioms das Redukt.

Ist das erste anwendbare Axiom eine Gleichung oder Äquivalenz, dann wird nur diese angewendet.

Ist es eine Transition, dann werden alle Transitionen unter den Axiomen der jeweiligen Spezifikation angewendet und deren Redukte mit  $\langle+\rangle$  zu einem Term verknüpft.

Transitionsaxiome werden auch beim *Narrowing* und *Rewriting* eingesetzt (siehe Kapitel 11). Als Simplifikationen verwendet sie Expander2 nur beim Aufbau einer Kripke-Struktur, die mit dem Befehl *specification > build Kripke model* erfolgt.

- (4) Scheitert (3), dann wird der Termgraph  $u$  zu einem Baum  $v$  expandiert. Befindet sich  $u$  an einer Position von  $t$  mit positiver Polarität und ist eine der durch *simplCoInd* definierten (Co)Induktionsregeln auf  $u$  anwendbar, dann liefert deren Konklusion das Redukt.
- (5) Scheitert (4), dann wird versucht, eine durch *simplifyF* definierte eingebaute Simplifikationsregel von links nach rechts auf  $v$  anzuwenden. Gelingt dies, dann liefert die Konklusion der Regel das Redukt. Die Menge der zugelassenen Signatursymbole kann mit entsprechenden Buttons im Menü *signature* verändert werden.
- (6) Scheitert auch (5), dann wird die Suche nach einem Redex abgebrochen und, wieder bei  $t$  beginnend, ein Teilterm gesucht, auf dessen Expansion zu einem Baum  $v$  eine durch *expandFix* definierte Expansionsregel für Fixpunkttausdrücke (s.u.) anwendbar ist. Gelingt dies, dann liefert die Konklusion der Regel das Redukt.

Bis auf die Schritte des Typs (4) transformieren Simplifikationen jede Formel  $\varphi$  in eine – bzgl. des jeweils zugrundeliegenden Modells – zu  $\varphi$  äquivalente Formel.

## Simplifikationen des Typs (1): **Die Funktion evaluate**

```
evaluate :: Sig -> Term String -> Term String
evaluate sig = eval
  where eval (F "<==>" [t,u]) | eqTerm t u = mkTrue
        eval (F "====>" [t,u])           = mkImpl (eval t) $ eval u
        eval (F "<====" [t,u])          = mkImpl (eval u) $ eval t
        eval (F "&" ts)                = mkConjunct $ map eval ts
        eval (F "|" ts)               = mkDisjunct $ map eval ts
        eval (F ('A':'n':'y':x) [t])   = evalQ mkAny x $ eval t
        eval (F ('A':'l':'l':x) [t])   = evalQ mkAll x $ eval t
        eval (F "Not" [t])            = mkNot sig $ eval t
        eval (F "id" [t])             = eval t
        eval (F "head" [F ":" [t,_]]) = eval t
        eval (F "tail" [F ":" [_ ,t]]) = eval t
        eval (F "head" [t])          = case eval t of F "[]" (t:_ ) -> t
                                         t -> F "head" [t]
        eval (F "tail" [t])          = case eval t of F "[]" (_:ts) -> mkList ts
                                         t -> F "tail" [t]
        eval (F "bool" [t])          = case eval t of F "True" [] -> mkConst 1
                                         F "False" [] -> mkConst 0
                                         t -> F "bool" [t]
        eval (F "ite" [t,u,v])       = case eval t of F "True" [] -> eval u
                                         F "False" [] -> eval v
                                         t -> F "ite" [t,u,v]
```

```

eval (F ":" [t,u])      = case eval u of F "[]" ts -> mkList $ t':ts
                                u -> F ":" [t',u]
                                where t' = eval t
eval (F "!!" [t,u])     = case eval t of
                                F "[]" ts | just i && k < length ts -> ts!!k
                                t -> F "!!" [t,eval u]
                                where i = foldArith intAlg u
                                      k = get i
eval (F "length" [t])   = case eval t of
                                F "[]" ts -> mkConst $ length ts
                                t -> F "length" [t]
eval (F "range" [t,u])  | just i && just k
                        = mkList $ map mkConst [get i..get k]
                        where i = foldArith intAlg t
                              k = foldArith intAlg u
eval (F "$" [F "lsec" [t,F x [],u]]) = F x [eval t,eval u]
eval (F "$" [F "rsec" [F x [],t],u]) = F x [eval u,eval t]
eval (F "$" [F x [],F "()" ts])       = F x $ map eval ts
eval (F "$" [F x [],t])               = F x [eval t]
eval (F "suc" [t]) | just i          = mkConst $ get i+1
                        where i = foldArith intAlg t
eval (F x p@[_,_]) | isOrd x && just ip = mkRel x $ get ip
                    | isOrd x && just rp = mkRel x $ get rp
                    | isOrd x && just fp = mkRel x $ get fp
                    | isOrd x && just sp = mkRel x $ get sp

```

```

where ip = mapM intAlg.parseA p
      rp = mapM realAlg.parseA p
      fp = mapM linAlg.parseA p
      sp = mapM (relAlg sig).parseA p

eval (F "==" [t,u])    = evalEq t u
eval (F "/=" [t,u])   = evalNeq t u
eval (F "<+>" ts)    = mkSum $ map eval ts
eval t | just i = mkConst $ get i
       | just r = mkConst $ get r
       | just f = mkLin $ get f
       | just s = mkList $ map (uncurry mkPair) $ sort (<) $ get s
                  where i = foldArith intAlg t
                        r = foldArith realAlg t
                        f = foldArith linAlg t
                        s = foldArith (relAlg sig) t
eval (F x ts)           = F x $ map eval ts
eval t                   = t

evalQ f x t = f (words x `meet` frees sig t) $ eval t

evalEq t u | eqTerm t u = mkTrue
evalEq (F x ts) (F y us)
            | x == y && f x && f y && all g ts && all g us
            = mkConst $ ts `eqset` us
            where f x = x `elem` ["{}","<+>"]

```

```

g = isValue sig
evalEq t (F "suc" [u]) | just n = evalEq (mkConst $ get n-1) u
                        where n = parsePnat t
evalEq (F "suc" [u]) t | just n = evalEq u (mkConst $ get n-1)
                        where n = parsePnat t
evalEq (F "[]" (t:ts)) (F ":" [u,v]) = mkConjunct
                                         [evalEq t u, evalEq (mkList ts) v]
evalEq (F ":" [u,v]) (F "[]" (t:ts)) = mkConjunct
                                         [evalEq u t, evalEq v $ mkList ts]
evalEq (F x ts) (F y us) | all sig.isConstruct [x,y] =
                           if x == y && length ts == length us
                           then mkConjunct $ zipWith (evalEq) ts us
                           else mkFalse
evalEq t u = mkEq (eval t) $ eval u

evalNeq t u | eqTerm t u = mkFalse
evalNeq (F "{}" ts) (F "{}" us)
          | just qs && just rs = mkConst $ not $ get qs `eqset` get rs
                                      where qs = mapM parseReal ts
                                            rs = mapM parseReal us
evalNeq t (F "suc" [u]) | just n = evalNeq (mkConst $ get n-1) u
                        where n = parsePnat t
evalNeq (F "suc" [u]) t | just n = evalNeq u (mkConst $ get n-1)
                        where n = parsePnat t
evalNeq (F "[]" (t:ts)) (F ":" [u,v]) =

```

```

mkDisjunct [evalNeq t u,evalNeq (mkList ts) v]
evalNeq (F ":" [u,v]) (F "[]" (t:ts)) =
    mkDisjunct [evalNeq u t,evalNeq v $ mkList ts]
evalNeq (F x ts) (F y us) | all sig.isConstruct [x,y] =
    if x == y && length ts == length us
    then mkDisjunct $ zipWith (evalNeq) ts us
    else mkTrue
evalNeq t u = mkNeq (eval t) $ eval u

mkAny xs (F ('A':'n':'y':y) [t]) = mkAny (xs++words y) t
mkAny xs (F "==" [t,u]) = F "==" [mkAll xs t,mkAny xs u]
mkAny xs (F "|" ts) = F "|" $ map (mkAny xs) ts
mkAny [] t = t
mkAny xs t = mkBinder "Any" xs [t]

mkAll xs (F ('A':'l':'l':y) [t]) = mkAll (xs++words y) t
mkAll xs (F "&" ts) = F "&" $ map (mkAll xs) ts
mkAll [] t = t
mkAll xs t = mkBinder "All" xs [t]

```

## Simplifikationen des Typs (4): **(Co)Induktionsregeln**

### Anwendung der Fixpunktinduktion auf eine Ungleichung

Sei  $A$  ein vollständiger Verband und  $t^A$  monoton oder  $A$  ein  $\omega$ -CPO und  $t^A$   $\omega$ -stetig (siehe [18], §3).

$$\frac{\mu x_1 \dots x_n. t \leq u}{t[\pi_i(u)/x_i \mid 1 \leq i \leq n] \leq u} \uparrow$$

```
simplifyF (F "<=" [F x [t], u]) _  
| leader x "mu" = Just $ F "<=" [t>>>forL us xs, u]  
    where _:xs = words x; us = mkGets xs u
```

$$\frac{(\mu x_1 \dots x_n. t)(\vec{x}) \leq u}{\forall \vec{x} (t[\pi_i(\lambda \vec{x}. u)/x_i \mid 1 \leq i \leq n](\vec{x}) \leq u)} \uparrow$$

```
simplifyF (F "<=" [F $" [F x [t], arg], u]) _  
| leader x "mu" && null (subterms arg) =  
    Just $ mkAll [root arg]  
    $ F "<=" [apply (t>>>forL us xs) arg, u]  
    where _:xs = words x  
          us = mkGets xs $ F "fun" [arg, u]
```

## Anwendung der Fixpunktinduktion auf eine Implikation

$$\frac{\mu x_1 \dots x_n. \varphi \Rightarrow \psi}{\varphi[\pi_i(\psi)/x_i \mid 1 \leq i \leq n] \Rightarrow \psi} \uparrow$$

```
simplifyF (F ``then`` [F x [t], u]) sig
| leader x "MU" && monotone sig xs t =
    Just $ F ``then`` [t>>>forL us xs, u]
    where _ : xs = words x; us = mkGets xs u
```

```
monotone :: Sig -> [String] -> Term String -> Bool
monotone sig xs = f True
where f pol (F ``then`` [t, u]) = f (not pol) t && f pol u
      f pol (F "not" [t]) = f (not pol) t
      f pol (F x ts) = if x `elem` xs then pol else all (f pol) ts
      f _ _ = True
```

$$\frac{(\mu x_1 \dots x_n. \varphi)(\vec{x}) \Rightarrow \psi}{\forall \vec{x} (\varphi[\pi_i(\lambda \vec{x}. \psi)/x_i \mid 1 \leq i \leq n](\vec{x}) \Rightarrow \psi)} \uparrow$$

```
simplifyF (F "==>" [F $" [F x [t], arg], u]) sig
| leader x "MU" && null (subterms arg) && monotone sig xs t =
    Just $ mkAll [root arg]
    $ mkImpl (apply (t>>>forL us xs) arg) u
    where _ : xs = words x; us = mkGets xs $ F "rel" [arg, u]
```

## Anwendung der Coinduktion auf eine Ungleichung

Sei  $A$  ein vollständiger Verband und  $t^A$  monoton oder  $A$  ein  $\omega$ -coCPO und  $t^A$   $\omega$ -costetig (siehe Fixpoints, Categories, and (Co)Algebraic Modeling, Kap. 3).

$$\frac{u \leq \nu x_1 \dots x_n. t}{u \leq t[\pi_i(u)/x_i \mid 1 \leq i \leq n]} \uparrow$$

```
simplifyF (F "<=" [u,F x [t]]) _  
| leader x "nu" = Just $ F "<=" [t>>>forL us xs,u]  
    where _:xs = words x; us = mkGets xs u
```

$$\frac{u \leq (\nu x_1 \dots x_n. t)(\vec{x})}{\forall \vec{x} (u \leq t[\pi_i(\lambda \vec{x}. u)/x_i \mid 1 \leq i \leq n](\vec{x}))} \uparrow$$

```
simplifyF (F "<=" [u,F $" [F x [t],arg]]) _  
| leader x "nu" && null (subterms arg) =  
    Just $ mkAll [root arg]  
    $ F "<=" [u,apply (t>>>forL us xs) arg]  
    where _:xs = words x  
          us = mkGets xs $ F "fun" [arg,u]
```

## Anwendung der Coinduktion auf eine Implikation

$$\frac{\psi \Rightarrow \nu x_1 \dots x_n. \varphi}{\psi \Rightarrow \varphi[\pi_i(\psi)/x_i \mid 1 \leq i \leq n]} \uparrow$$

```
simplifyF (F "`then`" [u,F x [t]]) sig
| leader x "NU" && monotone sig xs t =
    Just $ F "`then`" [u,t>>>forL us xs]
    where _:xs = words x; us = mkGets xs u
```

$$\frac{\psi \Rightarrow (\nu x_1 \dots x_n. \varphi)(\vec{x})}{\forall \vec{x} (\psi \Rightarrow \varphi[\pi_i(\lambda \vec{x}. \psi)/x_i \mid 1 \leq i \leq n](\vec{x}))} \uparrow$$

```
simplifyF (F "==>" [u,F $" [F x [t],arg]]) sig
| leader x "NU" && null (subterms arg) && monotone sig xs t =
    Just $ mkAll [root arg]
    $ mkImpl u $ apply (t>>>forL us xs) arg
    where _:xs = words x
        us = mkGets xs $ F "rel" [arg,u]
```

## Simplifikationen des Typs (5): **Die Spezifikation base**

```
-- base

defuncts: init last take drop <=< kfold
preds: P Q R
fovars: x y z s s' m n i j k
hovars: F G P Q R

axioms:

[x]++s == x:s &
[]++s == s &
s++[] == s &
(x:s)++s' == x:s++s' &

init[x] == [] &
init$x:y:s == x:init$y:s &
init(s)++[last$s] == s &

last[x] == x &
last$x:y:s == last$y:s &

take(0)$s == [] &
take(n)$x:s == x:take(n-1)$s &
```

```

drop(0)$s == s &
drop(n)$x:s == drop(n-1)$s &

length$x:s == length(s)+1 &
length$init$x:s == length$s &

map(F)$x:s == F(x):map(F)$s &

G<=<F == concat.map(G).F &                                -- Kleisli composition for [ ]

kfold(F) == foldl$fun((s,x),concat$map(F$x)$s) &          -- Kleisli fold for [ ]

x+0 == x & x+0.0 == x &
0+x == x & 0.0+x == x &

x-0 == x & x-0.0 == x &
0-x == -x & 0.0-x == -x &

x*0 == 0 & x*0.0 == 0 &
0*x == 0 & 0.0*x == 0 &
x*1 == x & x*1.0 == x &
1*x == x & 1.0*x == x &

0/x == 0 & 0.0/x == 0 &
x/1 == x & x/1.0 == x &

```

```
x**0 == 1 & x**0.0 == 1 &
0**x == 0 & 0.0**x == 0 &
x**1 == x & x**1.0 == x &
1**x == 1 & 1.0**x == 1 &

([] = x:s <==> False) &
(x:s = [] <==> False) &

([] =/= x:s <==> True) &
(x:s =/= [] <==> True) &

(x <= x <==> True) &
(x >= x <==> True) &

(x < x <==> False) &
(x < suc$x <==> True) &

(x > x <==> False) &
(suc$x > x <==> True) &
(suc(x)-y > x-y <==> True) &
(x-y > x-suc(y) <==> True) &

(x `in` [] <==> False) &
(x `NOTin` [] <==> True)
```

## Simplifikationen des Typs (5): **Subsumption**

Subsumption ist eine binäre, induktiv definierte Relation zwischen Formeln. In ihr sind alle **syntaktisch** – ohne Anwendung nichtlogischer Axiome – **überprüfbaren Implikationen** zusammengefasst.

$\sim$  bezeichne syntaktische Gleichheit modulo Umordnung von Argumenten permutativer Verknüpfungen sowie Umbenennung von Variablen.

$$\begin{aligned}
 \varphi \sim \psi &\implies \varphi \text{ subsumes } \psi \\
 \varphi \text{ subsumes } \psi &\implies \neg\psi \text{ subsumes } \neg\varphi \\
 \varphi' \text{ subsumes } \varphi \text{ and } \psi \text{ subsumes } \psi' &\implies \varphi \Rightarrow \psi \text{ subsumes } \varphi' \Rightarrow \psi' \\
 \exists 1 \leq i \leq n : \varphi \text{ subsumes } \psi_i &\implies \varphi \text{ subsumes } \psi_1 \vee \dots \vee \psi_n \\
 \forall 1 \leq i \leq n : \varphi_i \text{ subsumes } \psi &\implies \varphi_1 \vee \dots \vee \varphi_n \text{ subsumes } \psi \\
 \forall 1 \leq i \leq n : \varphi \text{ subsumes } \psi_i &\implies \varphi \text{ subsumes } \psi_1 \wedge \dots \wedge \psi_n \\
 \exists 1 \leq i \leq n : \varphi_i \text{ subsumes } \psi &\implies \varphi_1 \wedge \dots \wedge \varphi_n \text{ subsumes } \psi \\
 \varphi(\vec{x}) \text{ subsumes } \psi(\vec{x}) &\implies \exists \vec{x} \varphi(\vec{x}) \text{ subsumes } \exists \vec{y} \psi(\vec{y}) \\
 \varphi(\vec{x}) \text{ subsumes } \psi(\vec{x}) &\implies \forall \vec{x} \varphi(\vec{x}) \text{ subsumes } \forall \vec{y} \psi(\vec{y}) \\
 \varphi(\vec{x}) \text{ subsumes } \psi(\vec{x}) \text{ and } \vec{x} \cap \text{free } Vars(\psi) = \emptyset &\implies \exists \vec{x} \varphi(\vec{x}) \text{ subsumes } \psi \\
 \varphi(\vec{x}) \text{ subsumes } \psi(\vec{x}) \text{ and } \vec{x} \cap \text{free } Vars(\psi) = \emptyset &\implies \varphi \text{ subsumes } \forall \vec{y} \psi(\vec{y})
 \end{aligned}$$

$$\exists \vec{t} : \varphi \sim \psi(\vec{t}) \implies \varphi \text{ subsumes } \exists \vec{x} \psi(\vec{x})$$

$$\exists \vec{t} : \varphi(\vec{t}) \sim \psi \implies \forall \vec{x} \varphi(\vec{x}) \text{ subsumes } \psi$$

$$\exists \vec{t}, 1 \leq i_1, \dots, i_k \leq n : \varphi_{i_1} \wedge \dots \wedge \varphi_{i_k} \sim \psi(\vec{t}) \implies \varphi_1 \wedge \dots \wedge \varphi_n \text{ subsumes } \exists \vec{x} \psi(\vec{x})$$

$$\exists \vec{t}, 1 \leq i_1, \dots, i_k \leq n : \varphi(\vec{t}) \sim \psi_{i_1} \vee \dots \vee \psi_{i_k} \implies \forall \vec{x} \varphi(\vec{x}) \text{ subsumes } \psi_1 \vee \dots \vee \psi_n$$

**Subsumptionsregeln**  $\varphi$  subsumiere  $\psi$ .

$$\frac{\varphi \Rightarrow \psi}{True}$$

$$\frac{\varphi \Leftrightarrow \psi}{\psi \Rightarrow \varphi}$$

$$\frac{\varphi \wedge \psi}{\varphi}$$

$$\frac{\varphi \wedge \neg \psi}{False}$$

$$\frac{\varphi \vee \psi}{\psi}$$

$$\frac{\varphi \vee \neg \psi}{True}$$

$$\frac{\varphi \wedge (\psi \Rightarrow \delta)}{\varphi \wedge \delta} \quad \frac{\varphi \vee (\psi \Rightarrow \delta)}{True}$$

## Simplifikationen des Typs (5): **Verschiedenes**

### Move quantifiers upwards

Sei  $\vec{x} = \theta(\vec{x}_1 \cup \dots \cup \vec{x}_n)$  und  $\theta$  eine Umbenennung von Variablen derart, dass für alle  $1 \leq i \leq n$  keine Variable von  $\theta(\vec{x}_i)$  in einer Formel  $\theta(\varphi_j)$  mit  $1 \leq j \leq n$ ,  $j \neq i$ , frei vorkommt.

$$\frac{\exists \vec{x}_1 \varphi_1 \wedge \dots \wedge \exists \vec{x}_n \varphi_n}{\exists \vec{x} \theta(\varphi_1 \wedge \dots \wedge \varphi_n)} \quad \frac{\forall \vec{x}_1 \varphi_1 \vee \dots \vee \forall \vec{x}_n \varphi_n}{\forall \vec{x} \theta(\varphi_1 \vee \dots \vee \varphi_n)} \quad \frac{\exists \vec{x}_1 \varphi_1 \Rightarrow \forall \vec{x}_2 \varphi_2}{\forall \vec{x} \theta(\varphi_1 \Rightarrow \varphi_2)}$$

```
simplifyF c@(F "&" ts) sig = move ts [] [] False
  where move (F ('A':'n':'y':x) [t]:ts) us zs b =
        move ts (renameAll g t:us) (map g xs++zs) True
        where xs = words x
              g = renameAwayFrom sig xs (join ts us) c
        move (t:ts) us zs b = move ts (t:us) zs b
        move _ us zs b      = do guard b; Just $ mkAny (reverse zs)
                                  $ mkConjunct $ reverse us
```

## General term replacement

Sei  $\text{root}(t)$  ein Konstruktor,  $\text{root}(u)$  kein Konstruktor und  $u$  kein Teilterm von  $t$ .

$$\frac{t = u \wedge \varphi(t)}{t = u \wedge \varphi(u)} \quad \frac{t \neq u \vee \varphi(t)}{t \neq u \vee \varphi(u)}$$

$$\frac{\begin{array}{c} t = u \wedge \varphi(t) \\ \Rightarrow \psi(t) \end{array}}{\begin{array}{c} t = u \wedge \varphi(u) \\ \Rightarrow \psi(u) \end{array}} \quad \frac{\begin{array}{c} \varphi(t) \\ \Rightarrow t \neq u \vee \psi(t) \end{array}}{\begin{array}{c} \varphi(u) \\ \Rightarrow t \neq u \vee \psi(u) \end{array}}$$

## Zerlegung von Memberships, Gleichungen und Ungleichungen

$$\frac{t \in [t_1, \dots, t_n]}{t = t_1 \vee \dots \vee t = t_n} \quad \frac{t \in [t_1, \dots, t_n]}{t \neq t_1 \wedge \dots \wedge t \neq t_n} \quad \begin{array}{l} t \text{ contains a variable,} \\ t_1, \dots, t_n \text{ consist of constructors} \end{array}$$

Let  $c$  and  $d$  be different constructors.

$$\frac{c(t_1, \dots, t_n) = c(u_1, \dots, u_n)}{t_1 = u_1 \wedge \dots \wedge t_n = u_n} \quad \frac{c(t_1, \dots, t_n) = d(u_1, \dots, u_n)}{\text{False}}$$

$$\frac{c(t_1, \dots, t_n) \neq c(u_1, \dots, u_n)}{t_1 \neq u_1 \vee \dots \vee t_n \neq u_n} \quad \frac{c(t_1, \dots, t_n) \neq d(u_1, \dots, u_n)}{\text{True}}$$

## Remove equations and inequations

Let  $x \in \vec{x} \setminus var(t)$ .

$$\frac{\exists \vec{x}(x = t \wedge \varphi)}{\exists \vec{x}\varphi[t/x]}$$

$$\frac{\forall \vec{x}(x \neq t \vee \varphi)}{\forall \vec{x}\varphi[t/x]}$$

$$\frac{\forall \vec{x}((x = t \wedge \varphi) \Rightarrow \psi)}{\forall \vec{x}(\varphi \Rightarrow \psi)[t/x]}$$

$$\frac{\forall \vec{x}(\varphi \Rightarrow (x \neq t \vee \psi))}{\forall \vec{x}(\varphi \Rightarrow \psi)[t/x]}$$

The following rules also remove equations or inequations. However, in contrast to the preceding rules, they are—like induction and coinduction—proper expansion rules (see [18], chapter 10).

Let  $\sigma : V \rightarrow T_\Sigma(V)$  be a substitution of variables,  $dom(\sigma) = \{x \in V \mid \sigma(x) \neq x\}$  and  $t, u \in T_\Sigma(V) \setminus dom(\sigma)$  such that  $t\sigma = u\sigma$ .

$$\frac{\exists \text{ } dom(\sigma)(t = u \wedge \varphi)}{\varphi\sigma} \uparrow$$

$$\frac{\forall \text{ } dom(\sigma)(t \neq u \vee \varphi)}{\varphi\sigma} \uparrow$$

$$\frac{\forall \text{ } dom(\sigma)((t = u \wedge \varphi) \Rightarrow \psi)}{(\varphi \Rightarrow \psi)\sigma} \uparrow$$

$$\frac{\forall \text{ } dom(\sigma)(\varphi \Rightarrow (t \neq u \vee \psi))}{(\varphi \Rightarrow \psi)\sigma} \uparrow$$

Diese Regeln werden wie die obigen (Co)Induktionsregeln nur angewendet, wenn der erste Button im *signature*-Menü auf *simplification is unsafe* gesetzt ist.

## $\lambda$ -applications

Let  $x, y, x_1, \dots, x_n, xs$  be variables,  $p, p_1, \dots, p_n$  be patterns,  $b, b_1, \dots, b_n$  be formulas,  $t, u, t_1, \dots, t_n$  be arbitrary terms,  $\sigma, \sigma_1, \dots, \sigma_n$  be substitutions,  $es$  be a list of expressions of the form  $b \rightarrow t$  and  $ebs$  be a list of equations of the form  $p = t$ .

A **guarded pattern** is an expression of the form  $p|b$  ( $p||b$  in Expander2). An unguarded pattern  $p$  is identified with the guarded pattern  $p|True$ .

A **lazy pattern** is a pattern preceded by the symbol  $\sim$ .

$$\lambda(\sim p.t)(u) \rightarrow t\sigma \text{ if } p\sigma = u \quad (1)$$

$$\lambda(\sim p.t)(u) \rightarrow t[\lambda(p.x)(u)/x \mid x \in var(p)] \text{ if } p \not\leq u \quad (2)$$

$$\lambda(p.t)(u) \rightarrow t\sigma \text{ if } p\sigma = u \quad (3)$$

$$\begin{aligned} \lambda[p_1|b_1.t_1, \dots, p_n|b_n.t_n](t) &\rightarrow case[b_{i_1}\sigma_1 \rightarrow t_{i_1}\sigma_1, \dots, b_{i_k}\sigma_1 \rightarrow t_{i_k}\sigma_k] \\ &\quad \text{if for all } 1 \leq i \leq k, p_{i_j}\sigma_j = t_{i_j}, \text{ and } p_{i_1}, \dots, p_{i_k} \\ &\quad \text{are the patterns among } p_1, \dots, p_n \text{ that match } t \end{aligned} \quad (4)$$

$$case((True \rightarrow t) : es) \rightarrow t \quad (5)$$

$$case((False \rightarrow t) : es) \rightarrow case(es) \quad (6)$$

$$case[] \rightarrow undefined \quad (7)$$

$$\lambda((p|b.t \text{ where } ebs ++ [p' = u]) : es) \rightarrow \lambda((p|(\lambda p'.b)(u).(\lambda p'.t)(u) \text{ where } ebs) : es) \quad (8)$$

$$\lambda((t \text{ where } []) : es) \rightarrow \lambda(t : es) \quad (9)$$

Note that rule (2) reduces a  $\lambda$ -application  $\lambda(\sim p.t)(u)$  to (an instance of)  $t$  even if  $u$  does not match the pattern  $p$ .

If for all  $1 \leq j \leq k$ ,  $b_{ij}\sigma_j$  can be reduced to *True* or *False* via other rules, these rules together with (4)-(7) provide a complete rewrite system for evaluating  $\lambda$ -expressions with at least two non-lazy patterns.

```

simplifyA sig app@(F $" [F x [F "˜" [pat],body],arg])
| lambda x && just tsub = Just $ t>>>sub
  where tsub = bodyAndSub sig True app pat 1 arg
        (t,sub) = get tsub

simplifyA sig app@(F $" [F x [pat,body],arg])
| lambda x && just tsub = Just $ t>>>sub
  where tsub = bodyAndSub sig False app pat 1 arg
        (t,sub) = get tsub

simplifyA sig app@(F $" [F x ts,arg])
| lambda x && lg > 2 && even lg = Just $ F "CASE" $ pr1
  $ fold2 f ([] ,0,1) pats bodies where
    lg = length ts; pats = evens ts; bodies = odds ts
    f (cases,i,k) pat body = if just t then (cases++[get t],i+1,k+2)
                                else (cases,i,k+2)
    where t = concat [do F "||" [pat,b] <- Just pat; result pat b,
                      result pat mkTrue]

```

```

result pat b = do (u,sub) <- bodyAndSub sig False app pat k arg
                  Just $ F ">-" [b>>>sub,addToPoss [i,1] $ u>>>sub]

bodyAndSub sig lazy app pat k arg =
    if lazy then Just (u,forL (map f xs) xs)
              else do sub <- match sig xs arg' pat; Just (u,sub)
    where xs = sigVars sig pat
          arg' = dropFromPoss [1] arg
          f x = apply (F "fun" [pat,mkVar sig x]) arg'
          bodyPos = [0,k]
          body = getSubterm1 app bodyPos
          h = getSubAwayFrom body (bounded sig body) $ frees sig arg'
          t = collapseVars sig xs $ renameBound sig h body
          u = addChar 't' t $ targets t `minus` markPoss 't' t

simplifyT (F "CASE" (F ">-" [t,u]:_)) | isTrue t = Just u
simplifyT (F "CASE" (F ">-" [t,_]:ts)) | isFalse t = Just $ F "CASE" ts

```

## Simplifikationen des Typs (6): Expansion von Fixpunkt ausdrücken

### Expansion eines Fixpunktterms oder einer Fixpunktformel

Let  $fix \in \{\mu, \nu\}$ .

$$\frac{fix \ x.t}{t[(fix \ x.t)/x]}$$

```
expandFix :: Sig -> Bool -> [Int] -> Term String -> [Int] -> Maybe (Term String)
expandFix sig firstCall muPos t p =
    do guard $ not $ sig.blocked "fixes"
       F mu [body] <- Just redex
       guard $ isFix mu
       let [_,x] = words mu
           u = collapseVars sig [x] body
           v = if firstCall then redex else V $ 'e':mkPos0 muPos
           f = if x `elem` frees sig u then v `for` x else V
           reduct = instantiate (const id) u f []
       Just $ replace1 t p $ reduct
       where redex = addTargets (getSubterm t p) p
```

Let  $fix \in \{\mu, \nu\}$ .

$$\frac{\pi_i(fix\ x.(t_1, \dots, t_n))}{t_i} \quad 1 \leq i \leq n, \quad x \notin var(t_i)$$

```
simplifyT (F x [F mu [F "()" ts]])  
| just i && isFix mu && k < length ts && foldT f t = Just t  
    where i = parse (strNat "get") x  
          k = get i  
          t = ts!!k  
          [_,z] = words mu  
          f x bs = x /= z && and bs
```

For the auxiliary functions used here, see the modules *Eterm.hs* and *Esolve.hs* in the *Expander2* package.

## (Co)resolution and (co)induction on (co)predicates

(see also [18], chapters 10 and 11)

In [Expander2](#), (co)resolution steps are performed by pressing the *narrow* button. Unification of axioms with the actual goal is restricted to matching if the *match/unify* button left of the *narrow* button is set to *match*. The intervening *all/random* button admits to switch between the application of all applicable axioms in parallel (see the respective rule succedents) and the random selection of a single applicable rule.

### Resolution upon a least predicate $p$

$$\frac{p(t)}{\bigvee_{i=1}^k \exists Z_i : (\varphi_i \sigma_i \wedge \vec{x} = \vec{x} \sigma_i) \vee \bigvee_{i=k+1}^m \vec{x} = \vec{x} \sigma_i} \Updownarrow$$

where  $\gamma_1 \Rightarrow (p(t_1) \Leftarrow \varphi_1), \dots, \gamma_n \Rightarrow (p(t_n) \Leftarrow \varphi_n)$  are the axioms for  $p$  and

- (\*)  $\vec{x}$  is a list of the variables of  $t$ ,  
 for all  $1 \leq i \leq n$ ,  $t_i$  consists of  $\mathcal{C}$ -constructors and variables,  
 there are  $1 \leq k, m \leq n$  such that  
 for all  $1 \leq i \leq k$ ,  $t_i \sigma_i = t \sigma_i$ ,  $\gamma_i \sigma_i \vdash \text{True}$  and  $Z_i = \text{var}(t_i, \varphi_i)$ ,

$k < i \leq m$ ,  $t'_i\sigma_i = t\sigma_i$  for some  $t'_i \leq t_i$  with  $t'_i \notin V$  ( $\sigma_i$  partially unifies  $t_i$  and  $t$ ),  
for all  $m < i \leq n$ ,  $t$  is not partially unifiable with  $t_i$ .

## Coresolution upon a greatest predicate $p$

$$\frac{p(t)}{\bigwedge_{i=1}^k \forall Z_i : (\varphi_i\sigma_i \vee \vec{x} \neq \vec{x}\sigma_i) \wedge \bigwedge_{i=k+1}^m \vec{x} \neq \vec{x}\sigma_i}$$

where  $\gamma_1 \Rightarrow (p(t_1) \implies \varphi_1), \dots, \gamma_n \Rightarrow (p(t_n) \implies \varphi_n)$  are the axioms for  $p$  and  $(*)$  holds true.

## Narrowing upon a function $f$

$$\frac{r(\dots, f(t), \dots)}{\bigvee_{i=1}^k \exists Z_i : (r(\dots, u_i, \dots)\sigma_i \wedge \varphi_i\sigma_i \wedge \vec{x} = \vec{x}\sigma_i) \vee \bigvee_{i=k+1}^m (r(\dots, f(t), \dots)\sigma_i \wedge \vec{x} = \vec{x}\sigma_i)} \Updownarrow$$

where  $r$  is a predicate,  $\gamma_1 \Rightarrow (f(t_1) = u_1 \Leftarrow \varphi_1), \dots, \gamma_n \Rightarrow (f(t_n) = u_n \Leftarrow \varphi_n)$  are the axioms for  $f$  and  $(*)$  holds true.

As pointed out in [1, 9, 10], partial unification is needed for ensuring the completeness of equational narrowing if the redex  $f(t)$  is selected according to outermost (“lazy”) strategies, which — as in the case of rewriting — are the only ones that guarantee termination and optimality.

### Narrowing upon a transition relation $\rightarrow$

$$\frac{t \wedge v \rightarrow t'}{\bigvee_{i=1}^k \exists Z_i : ((u_i \wedge v) \sigma_i = t' \sigma_i \wedge \varphi_i \sigma_i \wedge \vec{x} = \vec{x} \sigma_i) \vee \bigvee_{i=k+1}^l ((t \wedge v) \sigma_i \rightarrow t' \sigma_i \wedge \vec{x} = \vec{x} \sigma_i)}$$

where  $\gamma_1 \Rightarrow (t_1 \rightarrow u_1 \Leftarrow \varphi_1), \dots, \gamma_n \Rightarrow (t_n \rightarrow u_n \Leftarrow \varphi_n)$  are the axioms for  $\rightarrow$ ,  $\sigma_i$  is a unifier modulo associativity and commutativity of  $\wedge$  and  $(*)$  holds true.

## Fixpoint coinduction over a least predicate $p$

$$(1) \quad \frac{p(x) \Rightarrow \psi(x)}{\bigwedge_{p(t) \Leftarrow \varphi \in AX} (\varphi[q/p] \Rightarrow \psi(t))} \uparrow$$

$$(2) \quad \frac{q(x) \Rightarrow \delta(x)}{\bigwedge_{p(t) \Leftarrow \varphi \in AX} (\varphi[q/p] \Rightarrow \delta(t))}$$

After applying (1), the predicate  $q$  and the axiom  $q(x) \Rightarrow \psi(x)$  are added to the current specification.

After applying (2), the axiom  $q(x) \Rightarrow \delta(x)$  is added to the current specification.

The correctness of (1) follows from the interpretation of  $p$  as the least relation satisfying  $AX_p$ . For the correctness of (2) see chapter 11 of [18].

## Fixpoint coinduction over a greatest predicate $p$

$$(1) \quad \frac{\psi(x) \Rightarrow p(x)}{\bigwedge_{p(t) \Rightarrow \varphi \in AX} (\psi(t) \Rightarrow \varphi[q/p])} \uparrow$$

$$(2) \quad \frac{\delta(x) \Rightarrow q(x)}{\bigwedge_{p(t) \Rightarrow \varphi \in AX} (\delta(t) \Rightarrow \varphi[q/p])} \quad p \notin \delta$$

After applying (1), the predicate  $q$  and the axiom  $q(x) \Leftarrow \psi(x)$  are added to the current specification. If (the interpretation of)  $p$  is the greatest bisimulation on the underlying model, then proof correctness is not violated by adding congruence axioms for  $q$ , which can be done in Expander2 as follows:

- Move the cursor on the canvas to an occurrence of  $q$ ;
- write words of the set  $\{refl, symm, tran\} \cup \{fn \mid f \text{ is a function symbol with arity } n\}$  into the entry field;
- press the *axioms > add > from entry field* button. The reflexivity, symmetry, transitivity and/or  $f$ -compatibility axioms for  $q$  are added to  $AX$ .

After applying (2), the axiom  $q(x) \Leftarrow \delta(x)$  is added to the current specification.

The correctness of (1) follows from the interpretation of  $p$  as the least relation satisfying  $AX_p$ . For the correctness of (2) see chapter 11 of [18].

(Co)Induction steps are performed by pressing the *selection > co/induction* button.

Expander2 kann (Co)Induktion auch auf Formeln anwenden, die – durch *Prämissen-* bzw. *Konklusionsdehnung* – in eine der Prämissen von (1) oder (2) transformierbar sind (siehe [14, 15]).

Außerdem lassen sich (1) und (2) auf mehrere – z.B. wechselseitig rekursiv definierte – (Co)Prädikate gleichzeitig anwenden.

Die in [15] und [16] erwähnte Möglichkeit, anstelle von  $AX_p$  die *n-fache Iteration*  $AX_p^n$  von  $AX_p$  mit

$$AX_p^n = \begin{cases} \{p(t) \Leftarrow \varphi[\text{result of } n \text{ resolution steps upon } p(u)/p(u)] \mid p(t) \Leftarrow \varphi \in AX_p\} & \text{bei Induktion} \\ \{p(t) \Rightarrow \varphi[\text{result of } n \text{ coresolution steps upon } p(u)/p(u)] \mid p(t) \Rightarrow \varphi \in AX_p\} & \text{bei Coinduktion} \end{cases}$$

in den obigen Regeln zu verwenden, wurde inzwischen aus Expander2 entfernt, weil es wohl kaum Anwendungen dafür gibt.

## Man-wolf-goat-cabbage problem

```
-- mwgc

specs:          modal
constructs:     m w g c final
defuncts:       goods draw1 draw2 draw3 pic river triple save'
preds:          Final

axioms:

goods == [w,g,c] &
labels == m:goods &
states == [labels] &
atoms == [final] &

final -> valid(Final) &
(Final$st <==> null$st | any(`in`[[m],[w,g],[g,c]])[st,labels-st]) &

(m `in` st & Not(Final$st) ==> (st,m) -> st-[m]) &

(m `NOTin` st & Not(Final$st) ==> (st,m) -> m:st) &

(x `in` goods & [m,x] `subset` st & Not(Final$st) ==> (st,x) -> st-[m,x]) &
```

```

(x `in` goods & [m,x] `NOTshares` st & Not(Final$st)
                         ==> (st,x) -> m:insertL(x)(st)(goods)) &

-- widget interpreters:

pic$m == gif(man,8,15) &
pic>w == gif(wolf,30,25) &
pic$g == gif(goat,28,25) &
pic$c == gif(cabbage,22,20) &
river == gif(wave,70,10) &
    -- blue$waveSF(5,20,45)

triple$s == center$shelf(1,3)[shelf(4)$map(pic)$s,river,
                           shelf(4)$map(pic)$labels-s] &

save'$s == save(triple$s,s) &

draw1 == wtree $ fun(red$x,text$x,x,triples$x) & -- paints states

draw2 == wtree $ fun(red$x,text$x,x,load$x) & -- loads and paints states

draw3 == wtree $ fun(red$m,pic$m,                  -- paints labels
                     red$x,center$shelf(2)[pic$m,pic$x],
                     x,text$x)

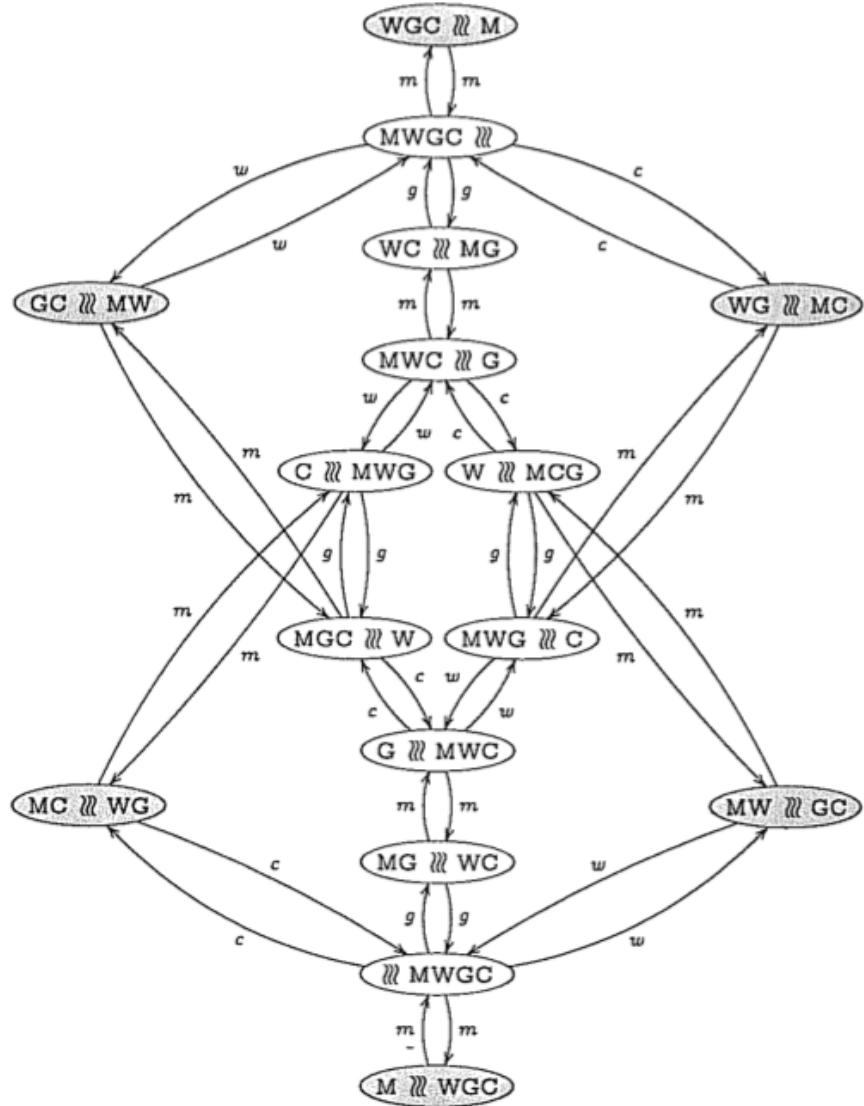
```

conjects:

```
(x `in` goods & x `in` [m,w,c]) & --> x = w | x = c  
EF(null)$labels --> True  
-- (189 depthfirst simplification steps;  
-- 79 parallel simplification steps + 3 subtree permutations)
```

terms:

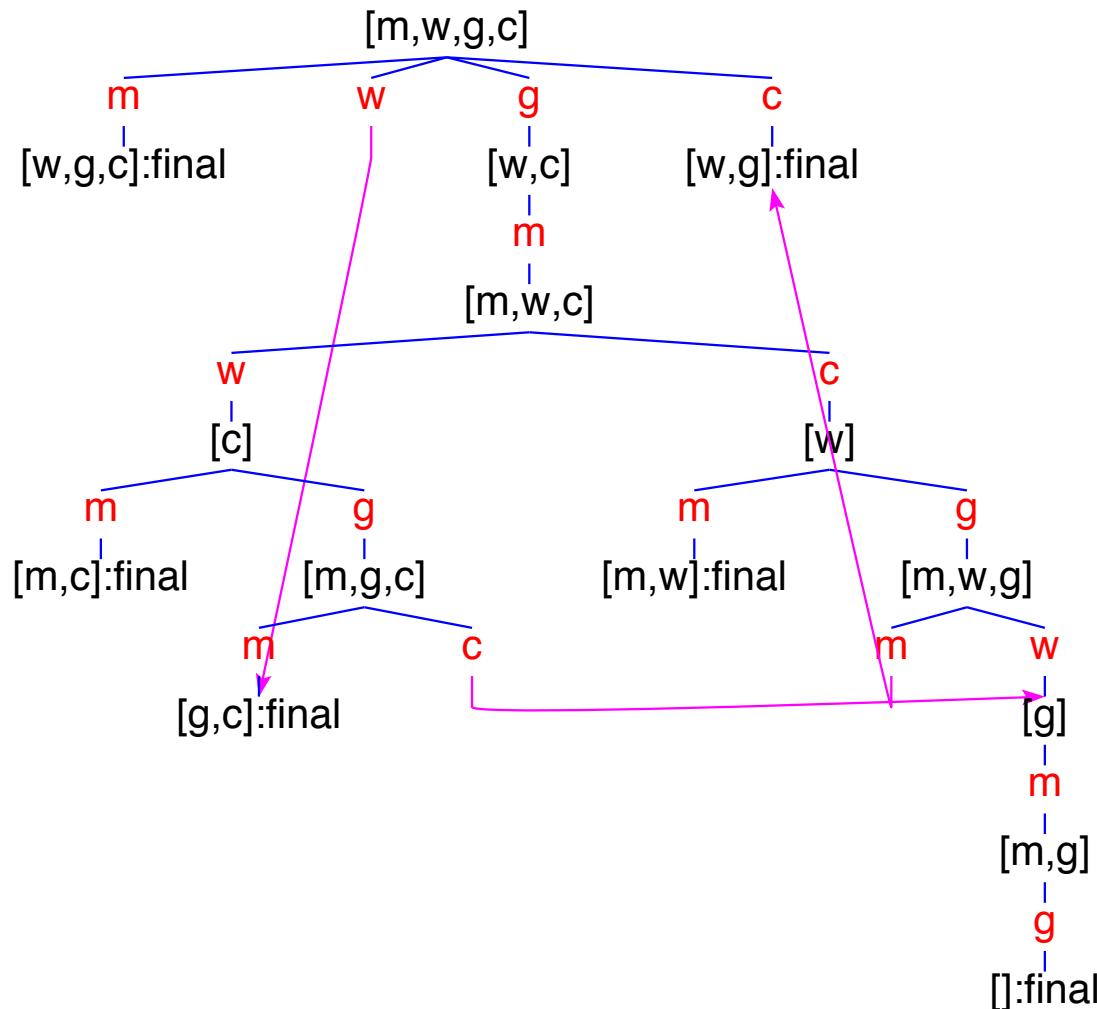
```
insertL(c)[w,g]$goods <+> --> [w,g,c]  
  
eval$EF$success <+>  
  
triple[m,g,c] <+>  
map(save')[[m,w,g,c],[w,c]] <+>  
map(load.string)[[m,w,g,c],[w,c]] <+>  
  
gggg(widg(tt(zz,widg(tt(pos 0 0,kk),load[w,c])),load[m,w,g,c]))
```



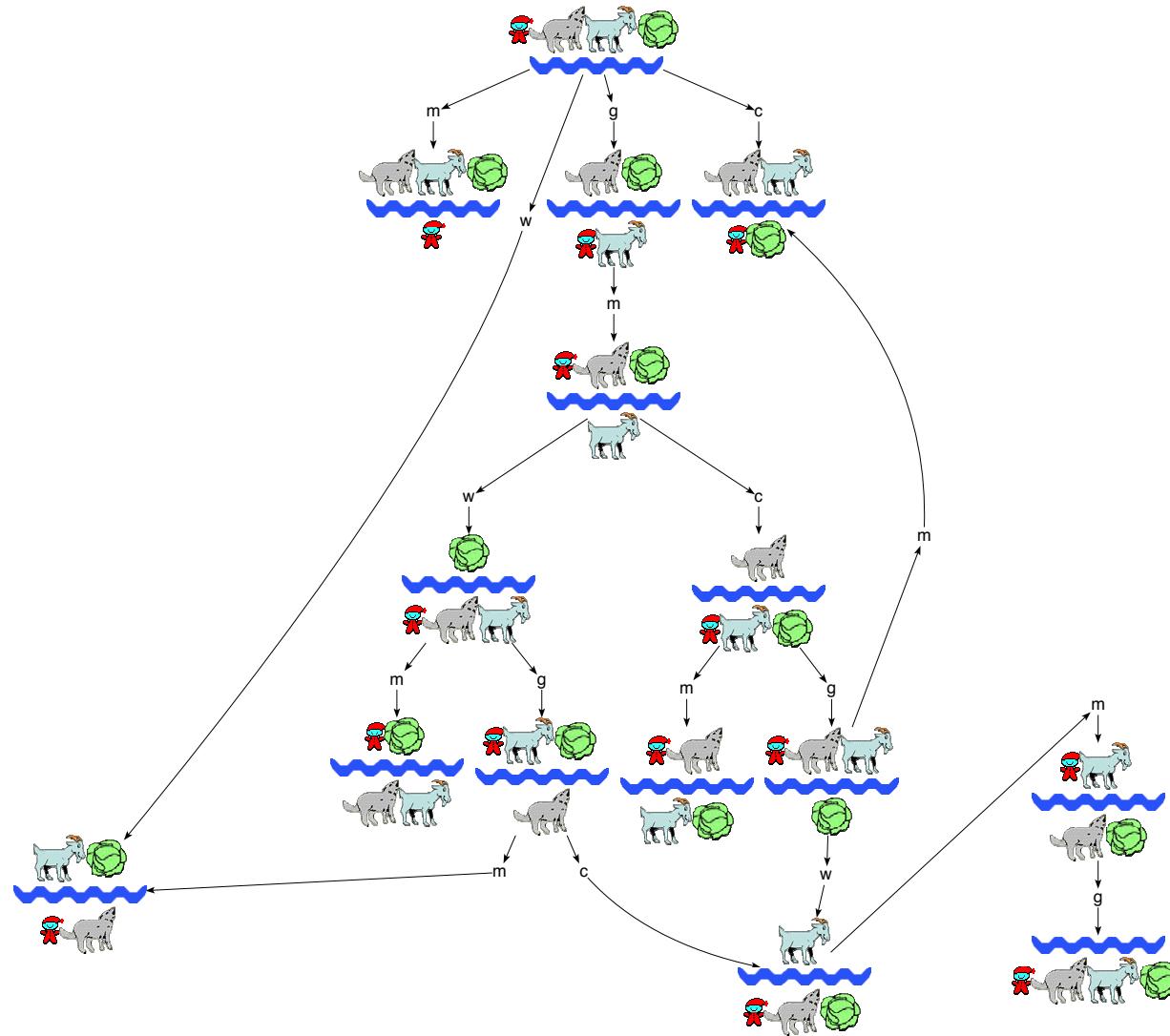
*Transitions of mwgc as presented in [6], Example 11.3*

press *specification > build Kripke model .. cycle-free*

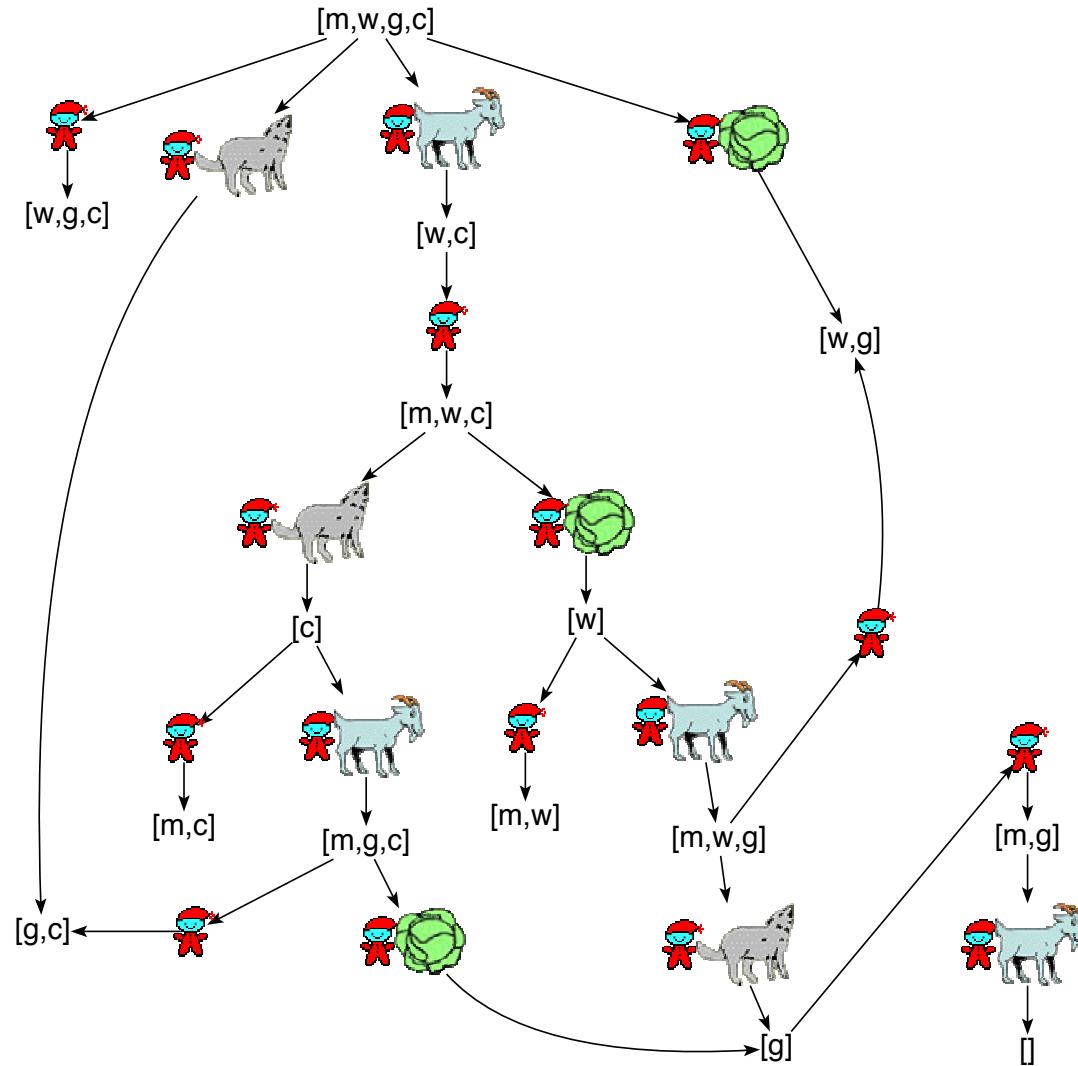
press *graph > show graph of Kripke model > here*



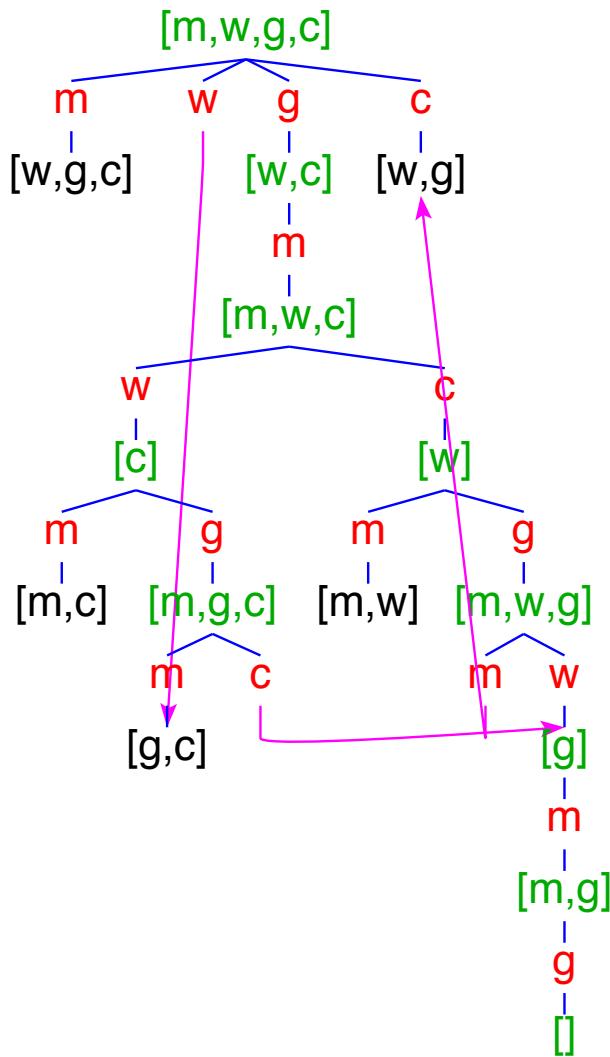
press *graph > show graph of labelled transitions* > here; enter *draw1* into the entry field  
press *tree > tree*; press *paint*; adapt red support points and press *combis* to remove them



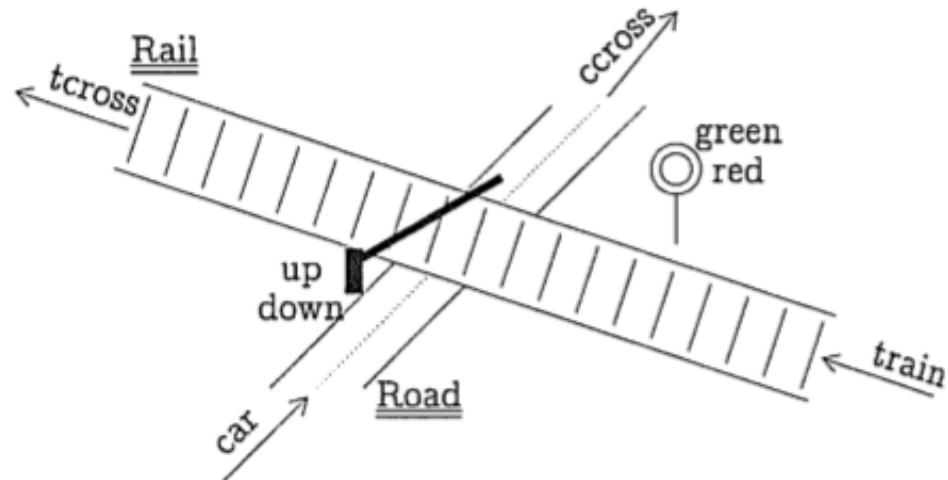
press *graph > show graph of labelled transitions* > here; enter *draw3* into the entry field  
press *tree > tree*; press *paint*; adapt red support points and press *combis* to remove them



enter *evalG\$EF\$success* into the text field; press *parse up*; press *simplify*



## Railway crossing (see [6], section 14.3)



A railway crossing ([6], Fig. 14.1)

*Process constructors* as defined in [6], sections 11.3 and 14.1

Name	Syntax	Semantics
PROCESS VARIABLE	$X$	If $E \xrightarrow{a} E'$ and $X \stackrel{\text{def}}{=} E$ then $X \xrightarrow{a} E'$
ACTION PREFIX	$a.E$	$a.E \xrightarrow{a} E$
CHOICE (1)	$E + F$	If $E \xrightarrow{a} E'$ then $E + F \xrightarrow{a} E'$ If $F \xrightarrow{a} F'$ then $E + F \xrightarrow{a} F'$
CHOICE (2)	$\sum\{E_i : i \in I\}$	If $E_j \xrightarrow{a} E'$ with $j \in I$ then $\sum\{E_i : i \in I\} \xrightarrow{a} E'$
NIL	$0$	no transitions ( $X \stackrel{\text{def}}{=} \sum \emptyset$ )
Parallel Composition	$E \parallel F$	If $E \xrightarrow{a} E'$ and $a \notin \text{Sort}(F)$ then $E \parallel F \xrightarrow{a} E' \parallel F$ . If $F \xrightarrow{a} F'$ and $a \notin \text{Sort}(E)$ then $E \parallel F \xrightarrow{a} E \parallel F'$ . If $E \xrightarrow{a} E'$ and $F \xrightarrow{a} F'$ and $a \in \text{Sort}(E) \cup \text{Sort}(F)$ then $E \parallel F \xrightarrow{a} E' \parallel F'$ .

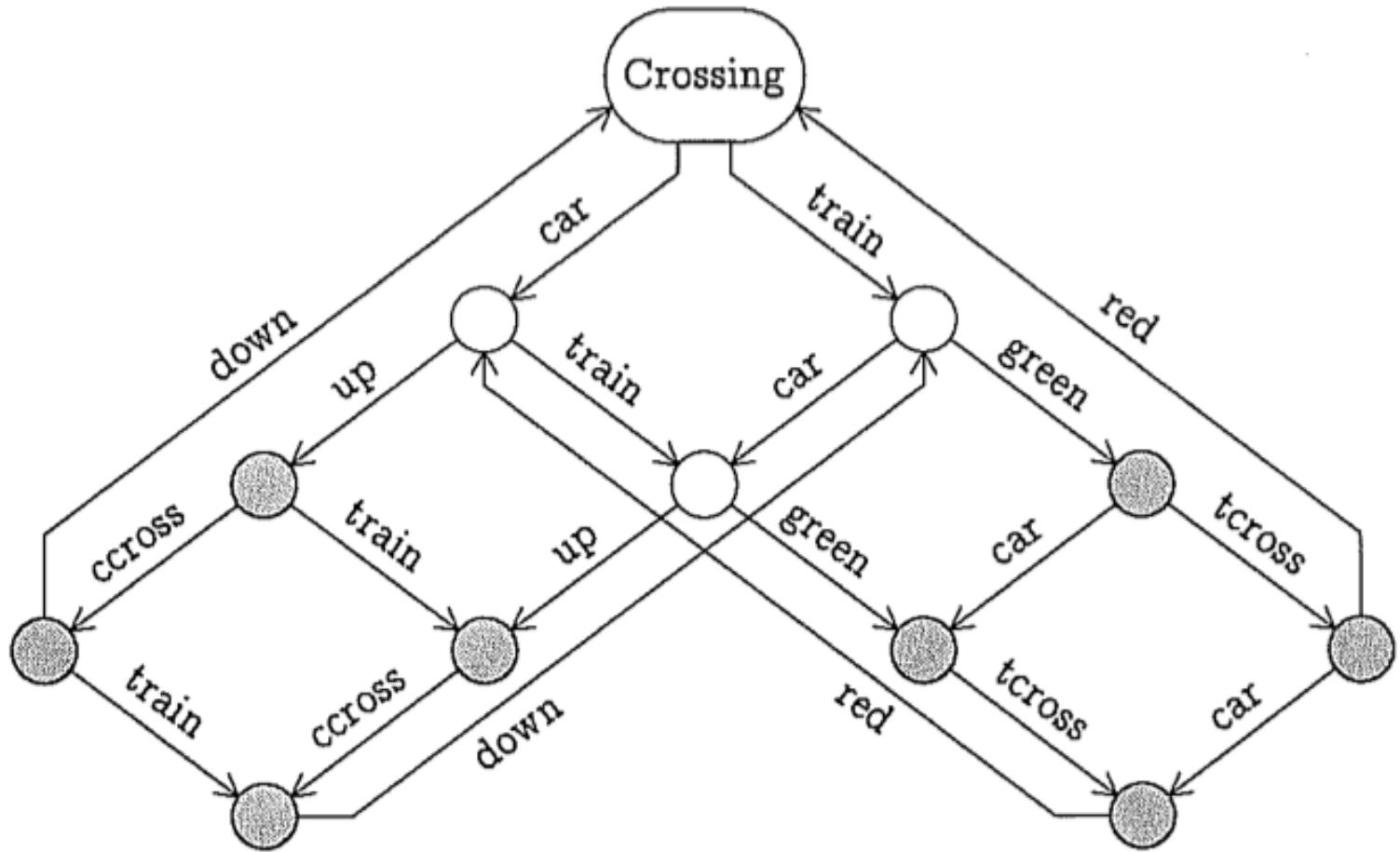
## Railway crossing processes

		actions to be synchronized
<i>Road</i>	$= car.up.ccross.down.Road$	$\{up, down\}$
<i>Rail</i>	$= train.green.tcross.red.Rail$	$\{green, red\}$
<i>Controller</i>	$= green.red.Controller + up.down.Controller$	$\{up, down, green, red\}$

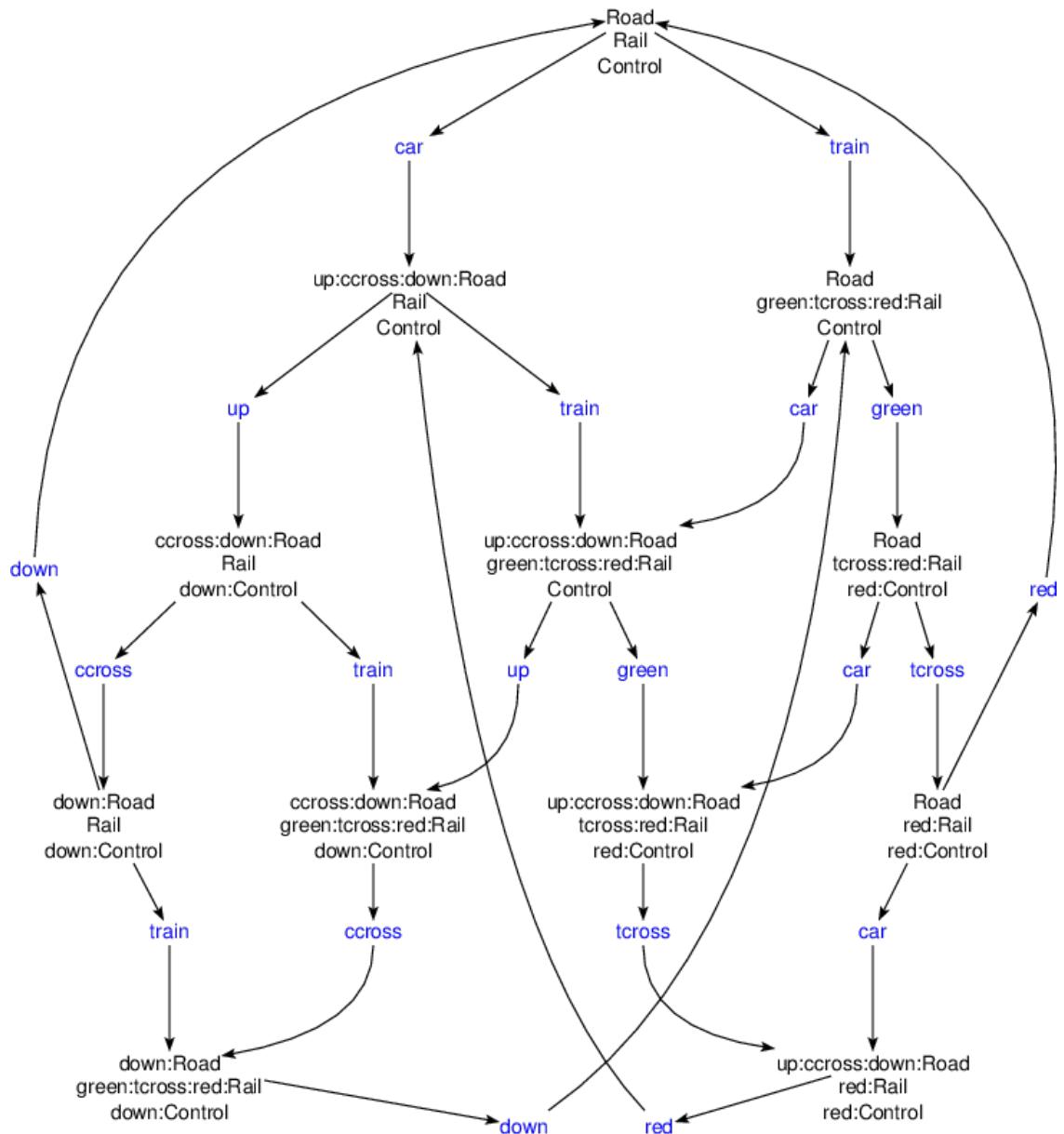
Equivalent processes must have the same sets of actions to be synchronized.

Process equivalence is the greatest relation  $\sim \subseteq Proc^2$  such that for all  $E, F \in Proc$ ,

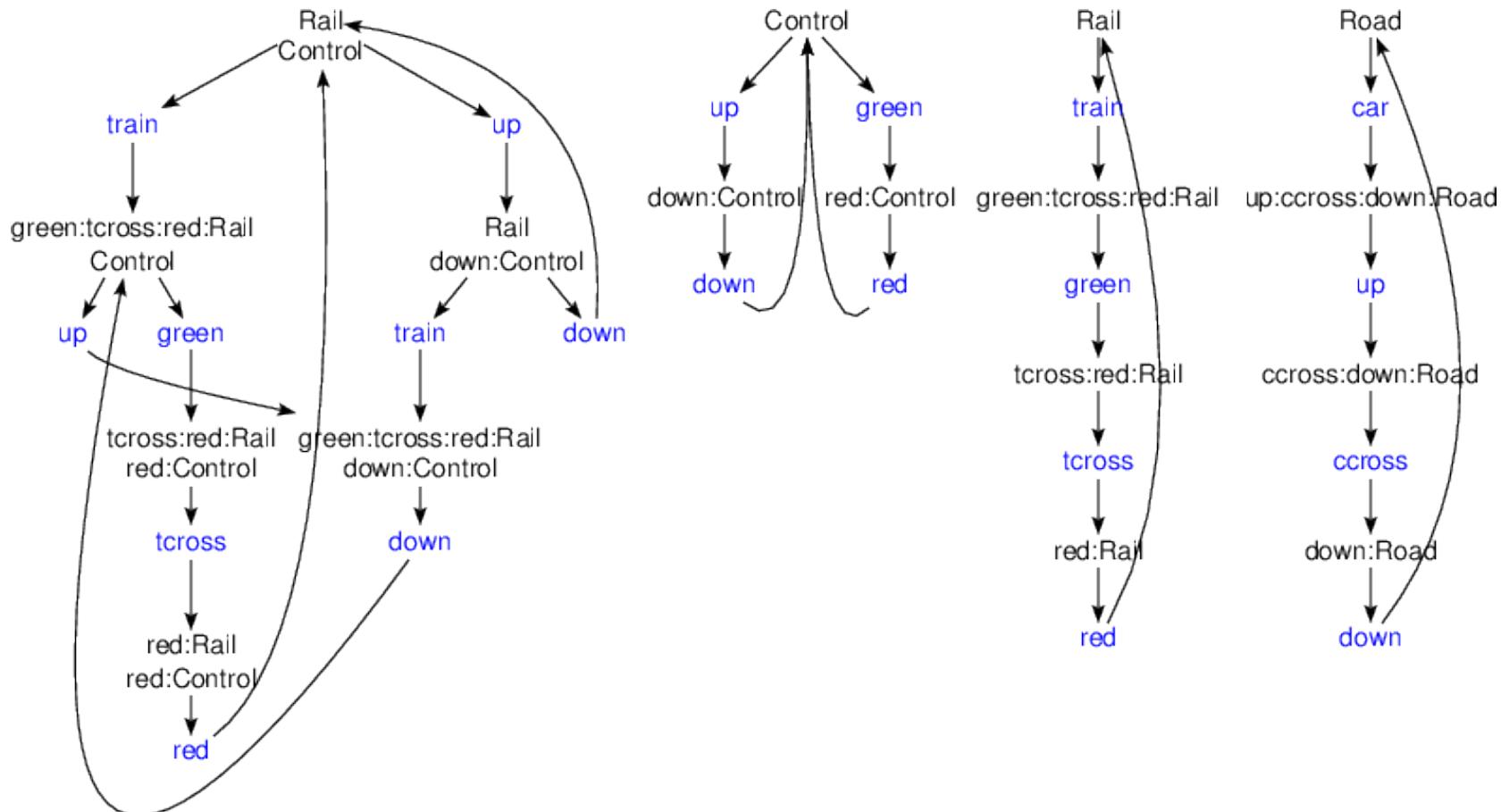
$$E \sim F \text{ implies } \forall a \in Act \left\{ \begin{array}{l} \forall E' \in \delta(E)(a) \exists F' \in \delta(F)(a) : E' \sim F' \wedge \\ \forall F' \in \delta(F)(a) \exists E' \in \delta(E)(a) : E' \sim F'. \end{array} \right.$$



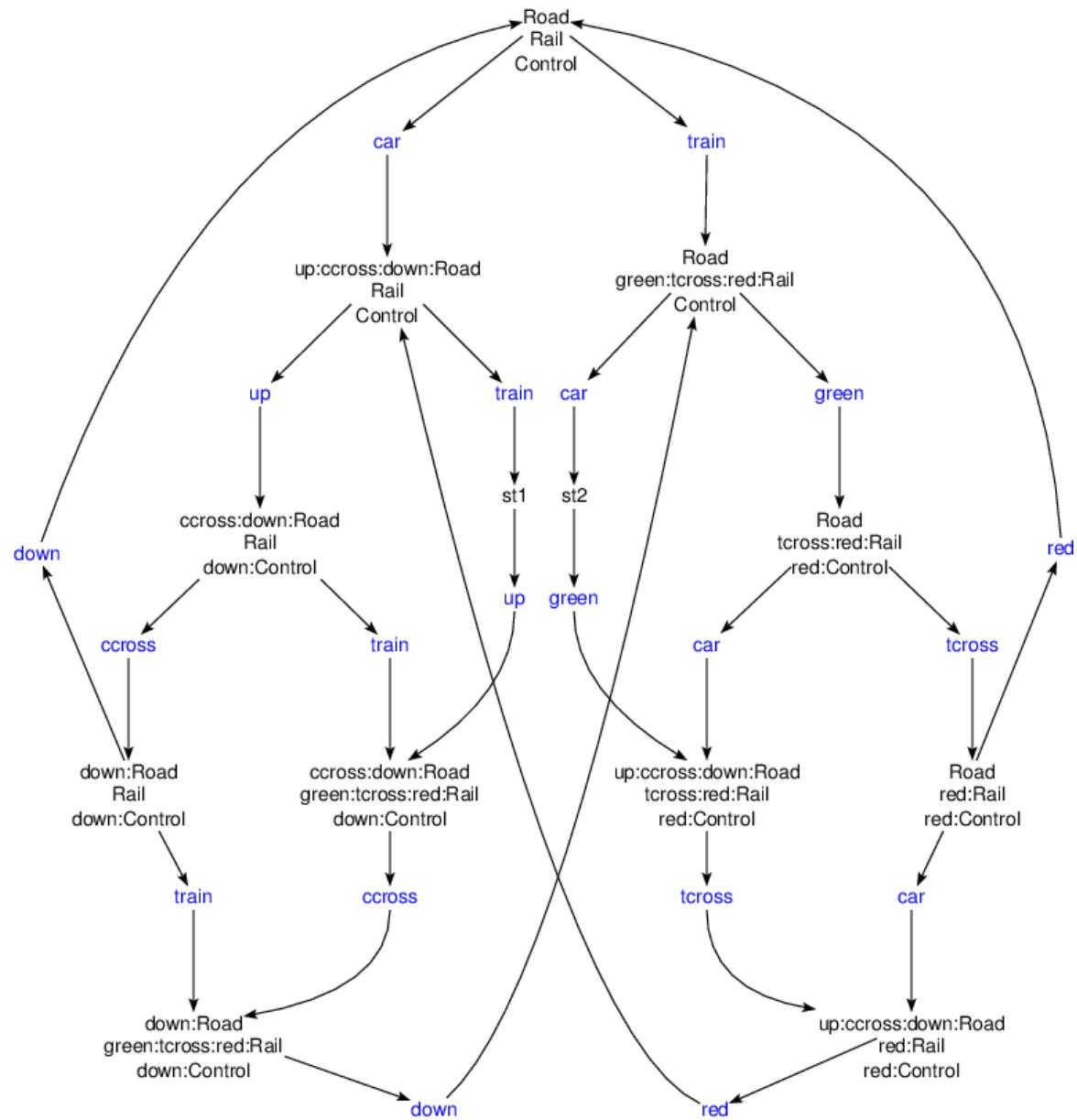
LTS of the Crossing process ([6], figure on page 365)



Expander2 version of the above LTS (generated by `build last Kripke model`) and `draw` has been applied (see below)



*Further states and transitions needed for computing the above LTS*



Refined version of the above LTS (generated by `build last Kripke model`) and `draw` has been applied (see below).

The middle state has been split into st1 and st2.

## *Properties*

- A car may not cross at the same time as a train. (1)
- If a car arrives, it crosses eventually. (2)
- If a train arrives, it crosses eventually. (3)

(1) its satisfied by both LTS, (2) and (3) only by the refined one.

-- CROSSING

```
constructs:      car train up down green red ccross tcross
                  Road Rail Control ||
defuncts:        Crossing draw sync
preds:           P X EF AF EG AG Car Train Green Up
fovars:          act act' E F G E' F'
hovars:          P X

axioms:
```

```

Crossing == Road||Rail||Control &
states == [Road,Rail,Control,Rail||Control,Crossing] &

labels == [car,train,up,down,green,red,ccross,tcross] &

draw == wtree $ fun(x||y||z,shelf(1)$map(text.noBrackets) [x,y,z] ,
                     x||y,shelf(1)$map(text.noBrackets) [x,y] ,
                     x,text$noBrackets$x) &

sync$Road      == [up,down] &
sync$Rail      == [green,red] &
sync$Control   == [up,down,green,red] &
sync$act:E     == sync$E &
sync$E||F      == sync(E)`join`sync(F) &

(Road,car)      -> up:ccross:down:Road &
(Rail,train)    -> green:tcross:red:Rail &
(Control,up)    -> down:Control &
(Control,green) -> red:Control &

```

```

-- ((up:ccross:down:Road) || Rail || Control, train) -> st1 &
-- (Road || (green:tcross:red:Rail) || Control, car) -> st2 &

-- (st1, up)
    -> (ccross:down:Road) || (green:tcross:red:Rail) || down:Control &
-- (st2, green)
    -> (up:ccross:down:Road) || (tcross:red:Rail) || red:Control &

(act:E, act) -> E &

(act `NOTin` sync(F)
-- & (E,F,act) =/= (Road, (green:tcross:red:Rail) || Control, car)
    ==> (E||F, act) -> branch$map(fun(E', E'||F))$transL(E)$act) &

(act `NOTin` sync(E)
-- & (E,F,act) =/= (up:ccross:down:Road, Rail || Control, train)
    ==> (E||F, act) -> branch$map(fun(F', E||F'))$transL(F)$act) &

(E||F, act) -> branch$map(fun((E', F'), E'||F'))
                           $prodL[transL(E)$act, transL(F)$act]

```

-- branching-time fixpoints

(EF\$P <==> MU X.(P\EX\$X))&  
(AF\$P <==> MU X.(P\/(AX(X)\EX\$true))) &  
(EG\$P <==> NU X.(P\/(EX(X)\AX>false))) &  
(AG\$P <==> NU X.(P\AX\$X))

-- finally on some path  
-- finally on each path  
-- generally on some path  
-- generally on each path

conjects:

Crossing `sats` AG\$(ccross#false)\tcross#false & -- (1)  
Crossing `sats` AG\$car#AF\$ccross<>true & -- (2)  
Crossing `sats` AG\$train#AF\$tcross<>true -- (3)

## Mutual exclusion

$idle(i)$  Process  $i$  is neither waiting nor in the critical section.

$wait(i)$  Process  $i$  asks for entering the critical section.

$crit(i)$  Process  $i$  is in the critical section.

**safety** There is at most one process in the critical section.

$$\neg(crit(1) \wedge crit(2))$$

**liveness** If a process asks for entering the critical section, it will enter it sometime.

$$wait(i) \Rightarrow AF(crit(i)), i = 1, 2$$

**non-blocking** Every process may ask for entering the critical section.

$$idle(i) \Rightarrow EX(wait(i)), i = 1, 2$$

**no strict sequencing** Every process that has left the critical section may enter it again before another process enters it.

$$EF(crit(i) \wedge (crit(i) EU (\neg crit(i) \wedge (\neg crit(j) EU crit(i))))), \\ i = 1, 2, j = 1, 2, i \neq j$$

In Fig. 11 below,  $idle(i)$ ,  $wait(i)$  and  $crit(i)$  are denoted by  $n_i$ ,  $t_i$  and  $c_i$ , respectively.

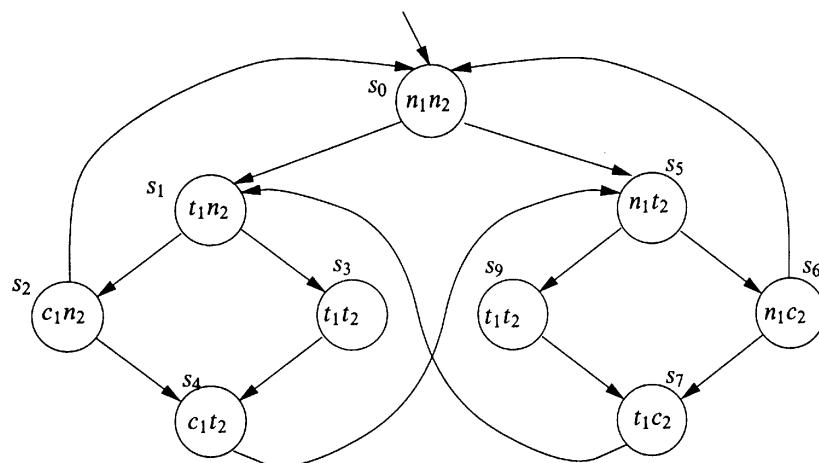
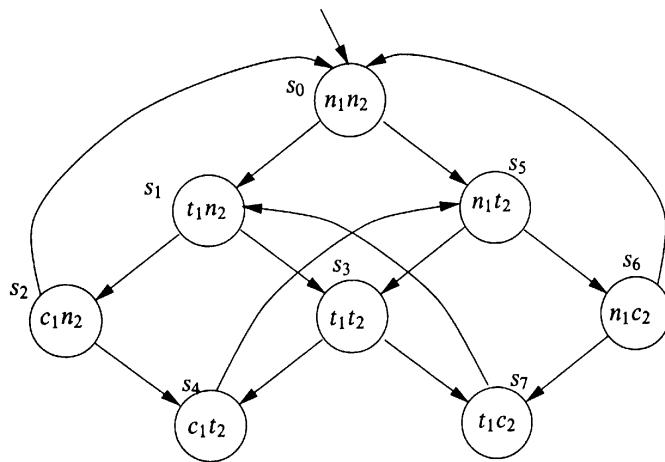


Fig. 11. Kripke models from [4], Example 3.3.1.

The first one violates the liveness condition because it does not consider the order in which the processes started waiting.

## Generalization to an arbitrary number of processes

The number must be entered into the entry field before calling *specification > build Kripke model*.

```
-- mutex

specs:          modal
constructs:     idle wait crit
preds:          live nonBlock noSeq crit' crits Idle Wait Crit Live NonBlock NoSeq
                  Crit' Crits  check
defuncts:       procs start pairs diffpairs pair drawI drawL drawK drawO drawS drawC
fovars:         st i c is ws cs k n

axioms:

states == [start] &
start == (procs, [], []) &
atoms == map($)$pairs &
pairs == prodL[[idle,wait,crit],procs] &

(is /= [] ==> (is,ws,cs) -> branch$map(fun(i,(is-[i],i:ws,cs)))$is) &
                                         -- i waits
(ws /= [] ==> (is,ws,[]) -> (is,init(ws),[last(ws)])) &           -- last(ws) enters
(is,ws,[c]) -> (insert(c)(is),ws,[]) &                                -- c leaves
```

```

-- 2 processes yield 9 states
-- 3 processes yield 31 states
-- 4 processes yield 129 states
-- 5 processes yield 651 states

idle$i -> valid$Idle$i &
wait$i -> valid$Wait$i &
crit$i -> valid$Crit$i &

(Idle(i)(is,ws,cs) <==> i `in` is) &
(Wait(i)(is,ws,cs) <==> i `in` ws) &
(Crit(i)(is,ws,cs) <==> i `in` cs) &

-- modal operator simplification

(live$i <==> (wait$i) `then` AF$crit$i) &

(nonBlock$i <==> (idle$i) `then` EX$wait$i) &

(noSeq$i <==> EF $ (crit$i) /\ ((crit$i) `EU`
                                ((not$crit$i) /\ ((crit'$i) `EU` crit$i)))) &

(crit'$i <==> all(not.crit)$procs-[i]) &

(crits <==> map(crit)$procs) &

```

```

-- modal operator simplification outside eval

(Live$i <==> (Wait$i) `then` AF$crit$i) &

(NonBlock$i <==> (Idle$i) `then` EX$Wait$i) &

(NoSeq$i <==> EF \$ (Crit$i) /\ ((Crit$i) `EU`  

                                ((not$Crit$i) /\ ((Crit'$i) `EU` Crit$i)))) &

(Crit'$i <==> all(not.Crit)$procs-[i]) &

(Crits <==> map(Crit)$procs) &

-- widget interpreters

drawI == wtree \$ fun(st,color(index(st,states),length$states)$circ$11) &

drawL == wtree(2,fun((st,k,n),color(k,n)$circ$11)) &

drawK == wtree \$ fun(st,matrix$filter(check$st)$pairs) &

(check(is,ws,cs)(idle,i) <==> i `in` is) &
(check(is,ws,cs)(wait,i) <==> i `in` ws) &
(check(is,ws,cs)(crit,i) <==> i `in` cs) &

```

```

draw0 == wtree $ fun(st,matrix$map(pair)$st) &
pair(idle$i) == (idle,i) &
pair(wait$i) == (wait,i) &
pair(crit$i) == (crit,i) &

drawS == wtree $ fun(sat$st,frame$text$st) &

drawC == wtree $ fun(sat$st,green$circ$11,st,red$circ$11) &

diffpairs == filter(=/=)$prodL$[procs,procs]

```

conjects:

(Crit\$1)\$start &	--> False
(EF\$Crit\$1)\$start &	--> True
(or\$Crits)\$start &	--> False
(Live\$1)\$start &	--> True
(NonBlock\$1)\$start &	--> True
(NoSeq\$1)\$start &	--> True ?
(EF\$or\$Crits)\$start &	--> True

```

(EF$and$Crits)$start &                                --> False ?
(and$map(EF.Crit)$procs)$start &                      --> True
(or$map(rel((x,y),EF$crit(x)/\crit$y))$diffpairs)$start &    --> False ?
(and$map(rel((x,y),EF$crit(x)\/crit$y))$diffpairs)$start &    --> True

(and$map(Live)$procs)$start &                            --> True
(and$map(NonBlock)$procs)$start &                        --> True
(and$map(NoSeq)$procs)$start &                           --> True ?
(and$map($)$prodL[[Live,NonBlock,NoSeq],procs])$start      --> True ?

```

terms:

```

eval(or$crits) <+>                                --> [([1],[], [0]), ([], [1], [0]), ([0],[], [1]),
--          ([], [0], [1])]

eval(live$1) <+>                                     --> states
eval(nonBlock$1) <+>                                 --> states
eval(noSeq$1) <+>                                 --> states

eval(EF$or$crits) <+>                               --> states
eval(EF$and$crits) <+>                             --> []
eval(and$map(EF.crit)$procs) <+>                  --> states
eval(or$map(rel((x,y),EF$crit(x)/\crit$y))$diffpairs) <+>    --> []
eval(and$map(rel((x,y),EF$crit(x)\/crit$y))$diffpairs) <+>    --> states

eval(and$map(live)$procs) <+>                         --> states

```

```

eval(and$map(nonBlock)$procs) <+> --> states
eval(and$map(noSeq)$procs) <+> --> states
eval(and$map($)$prodL[[live,nonBlock,noSeq],procs]) <+> --> states

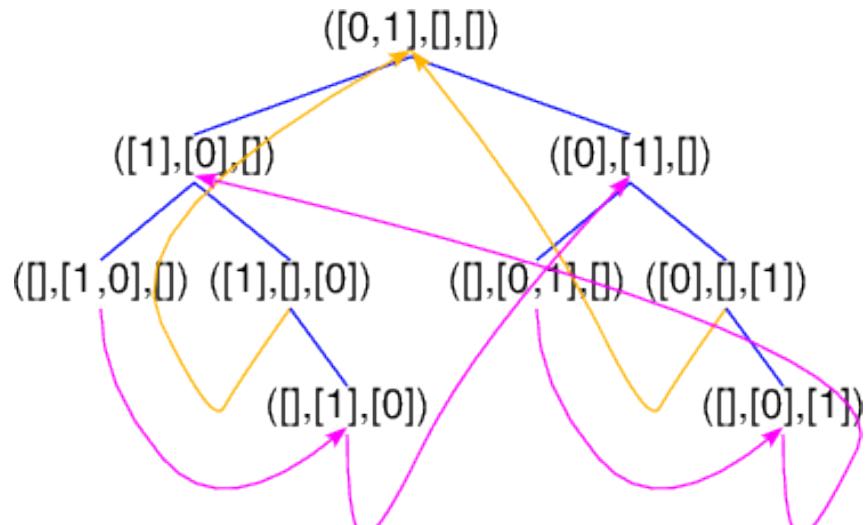
eval((idle$1)`EU`wait$1) <+> --> [([0],[1],[]),([],[],[1,0]),([],[],[0,1]),  

-- ([],[],[1,0]),([0,1],[],[]),([1],[0],[]),  

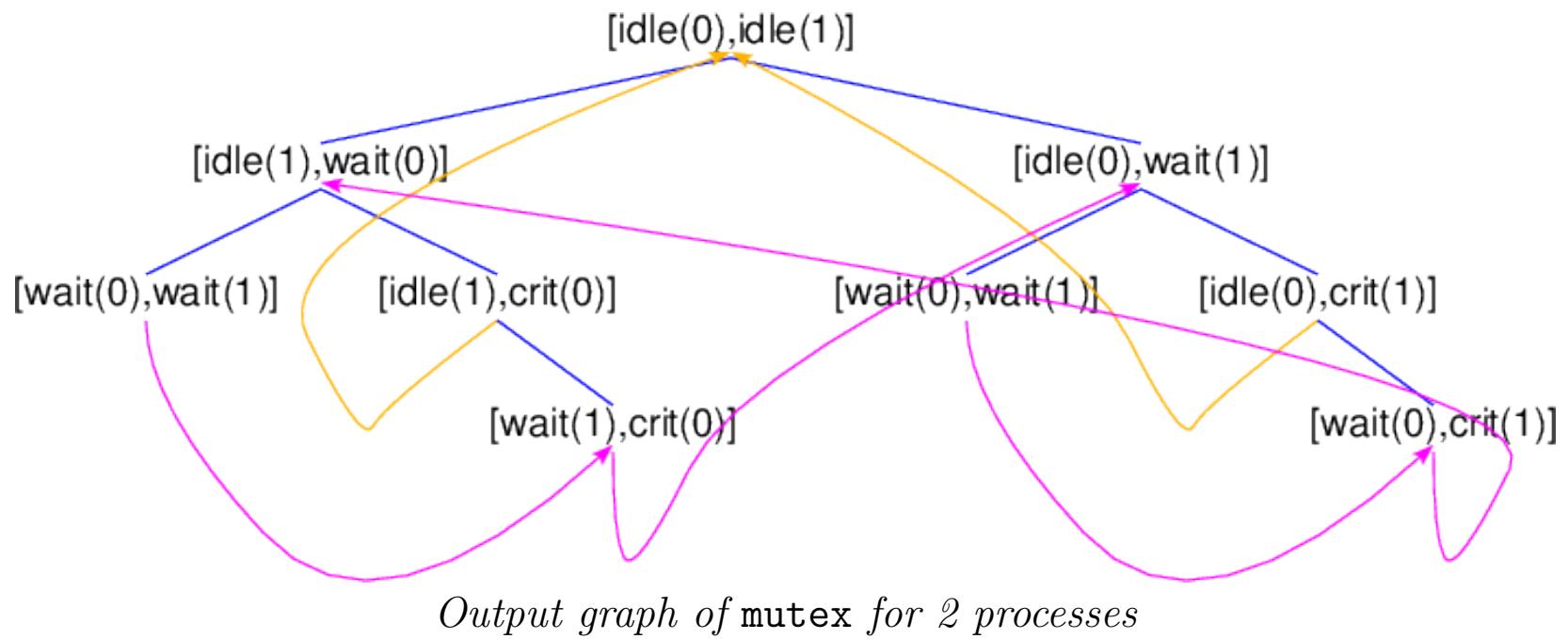
-- ([1],[],[0])]

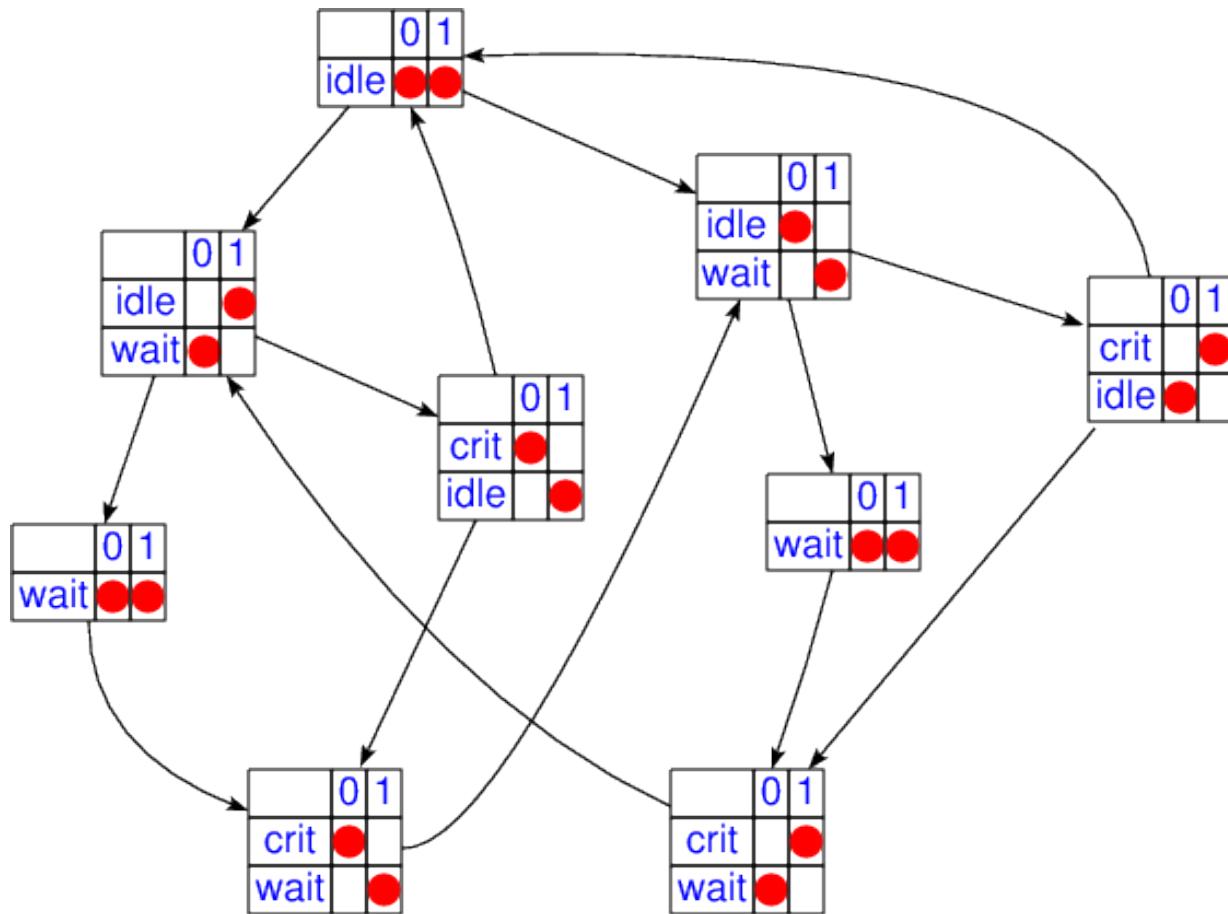
eval((idle$1)`EU`((wait$1)`EU`crit$1)) <+> --> states
eval(((idle$1)`EU`(wait$1))`EU`crit$1) --> states

```

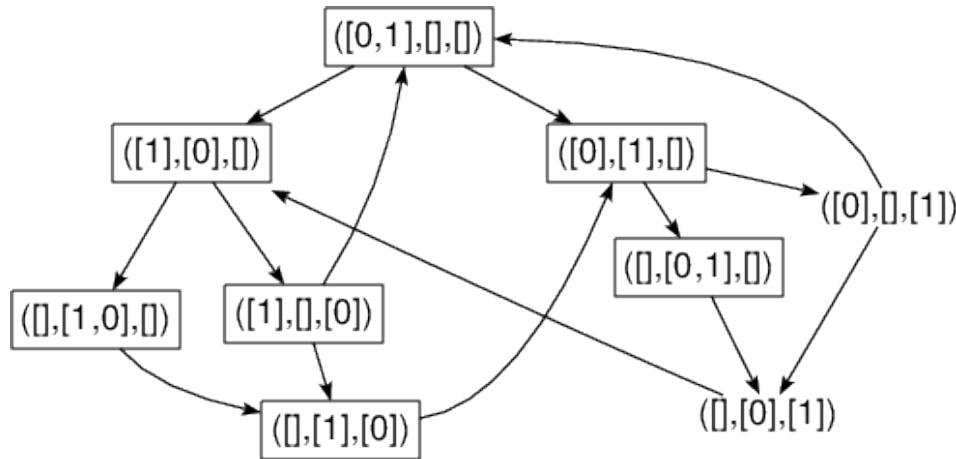
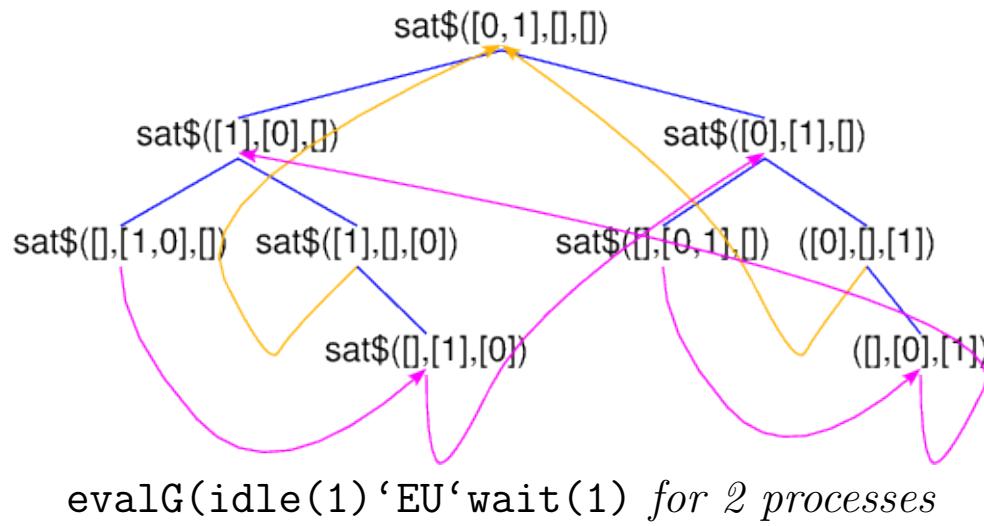


*Transition graph of mutex for 2 processes*

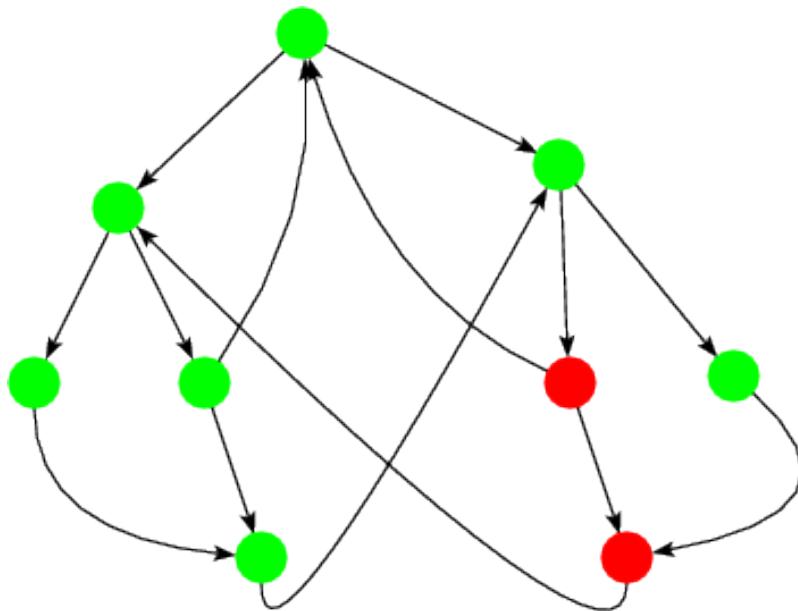




*Transition graph of mutex for 2 processes after application of drawK*



$\text{Result of evalG(idle(1) ' EU ' wait(1) for 2 processes after application of drawS}$



*Result of evalG(idle(1) 'EU' wait(1) for 2 processes after application of drawC*

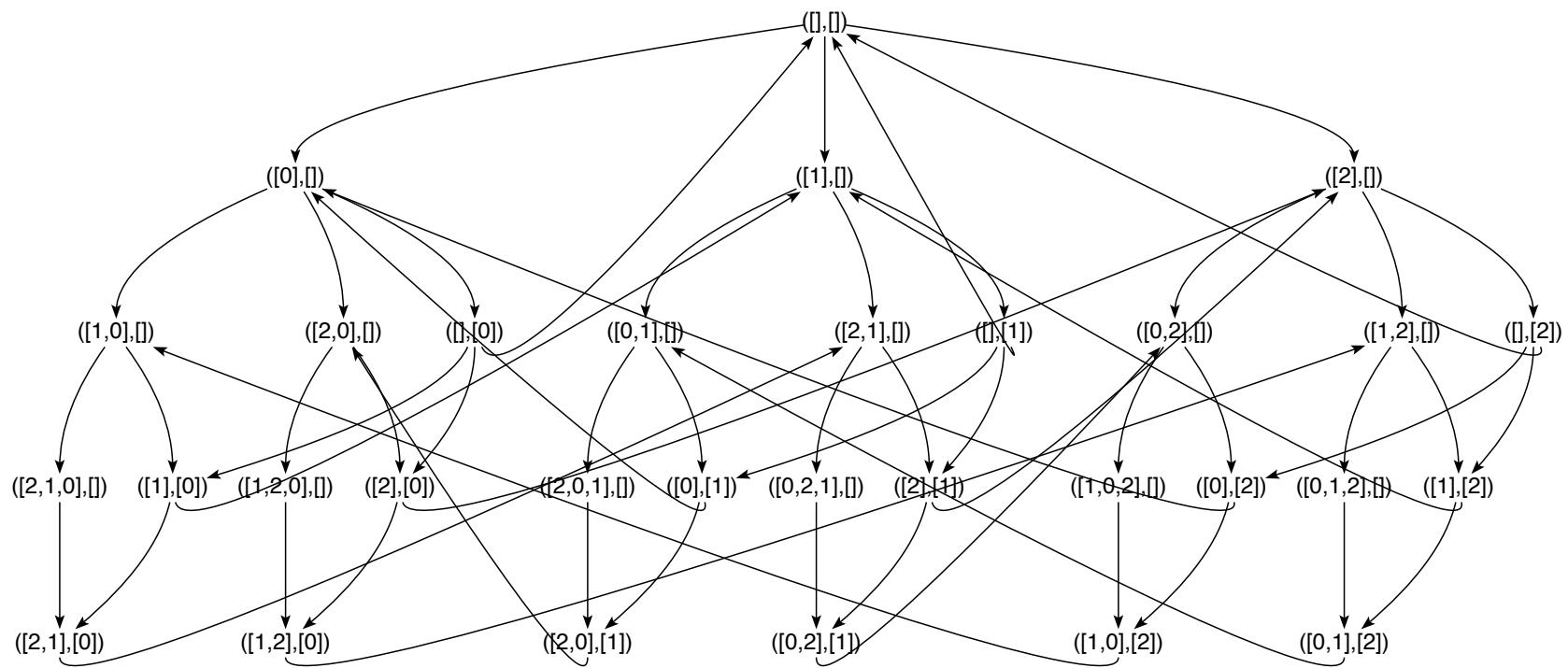


Fig. 12.  
*Transition graph of mutex for 3 processes*

-- MUTEXco

specs: modal

preds: Idle Wait Crit enabled safe noSeq

copreds: others

constructs: c

defuncts: request enter leave posi maxwait weight

fovars: xs ys xs' ys'

axioms:

(st >> st' <==> weight(st) > weight(st')) &  
weight(xs,ys) == (length(xs)-posi(xs++ys),maxwait-length(xs),length\$ys) &  
(st -> st' <==> st' = f\$st & enabled(f)\$st) &

(enabled(request\$x)(xs,ys) <==> Idle(x)(xs,ys) & maxwait > length\$xs) &  
enabled(enter)(x:xs,[]) &  
enabled(leave)(xs,[x]) &

request(x)(xs,ys) == (x:xs,ys) &  
enter(xs,ys) == (init\$xs,[last\$xs]) &  
leave(xs,ys) == (xs,[]) &

(Wait(x)(xs,ys) <==> x `in` xs) &  
(Crit(x)(xs,ys) <==> x `in` ys) &  
(Idle(x)(xs,ys) <==> x `NOTin` xs & x `NOTin` ys) &

```

safe(xs, []) & safe(xs, [x]) &
(others(P)(x)$st ==> (x /= y ==> P(y)$st)) &
(noSeq$x <==> EF $ Crit(x) /\ (Crit(x) `EU`  

                                (not(Crit$x) /\  

                                 (others(not.Crit)(x) `EU` Crit$x))))
```

theorems:

```

((xs,ys) >> st <==> c `in` xs & (xs,ys) -> st) &           -- transDesc
(x `in` xs' | x `in` ys'  

             <==> x `in` xs & (xs,ys) -> (xs',ys')) & -- transIn
(x `in` xs++ys <==> x `in` xs) &
(posi$x:s = suc$posi$s <==> c `in` s) &
(posi$s++s' = posi$s <==> c `in` s) &
(True ==> Any xs ys: st = (xs,ys)) &
(x `in` init$y:s | x = last$y:s <==> x = y | x `in` s)
```

conjects:

```

(safe$st ==> AG(safe)$st) &           -- AGsafe: proof by coinduction on AG
(Idle(x)(xs,ys) & length$xs < maxwait ==> EX(Wait$x)(xs,ys)) &  

                                         -- EXwait: proof by resolution on EX
(c `in` xs | c `in` ys ==> AF(Crit$c)(xs,ys))  

                                         -- AFcrit: proof by Noetherian induction wrt >>
```

Kann c ‘in’ ys weggelassen werden?

## Beweis von AGsafe:

safe(st) ==> AG(safe)\$st

Adding

(AGO(z0)\$st <== safe(st) & z0 = safe)  
& (notAGO(z0)\$st ==> Not(safe(st)) | z0 /= safe)

to the axioms and applying coinduction wrt

(AG(P)\$st ==> P(st) & AX(AG(P))\$st)

at position [] of the preceding formula leads to

All st:(safe(st) ==> AX(AGO(safe))\$st)

The reducts have been simplified.

Narrowing the preceding formula (4 steps) leads to 3 factors.

The current factor is given by

All xs x0:(x0 `NOTin` xs & maxwait > length(xs) ==> AG0(safe)(x0:xs, []))

The reducts have been simplified.

Narrowing the preceding factors (4 steps) leads to a single formula, which is given by

```
All x xs:(AX(AG0(safe))(xs,[x]))
```

The reducts have been simplified.

Narrowing the preceding formula (4 steps) leads to 2 factors.

The current factor is given by

```
All x xs x1:
```

```
(x1 `NOTin` xs & x1 =/= x & maxwait > length(xs) ==> safe(x1:xs,[x]))
```

The reducts have been simplified.

Narrowing the preceding factors (3 steps) leads to a single formula, which is given by

True

The reducts have been simplified.

Number of proof steps: 5

## Beweis von EXwait:

$\text{Idle}(x)(xs,ys) \ \& \ \text{length}(xs) < \text{maxwait} \implies \text{EX}(\text{Wait}(x))(xs,ys)$

Narrowing the preceding formula (1 step) leads to

$x \text{ `NOTin' } xs \ \& \ x \text{ `NOTin' } ys \ \& \ \text{length}(xs) < \text{maxwait} \implies$   
Any  $st':(xs,ys) \rightarrow st' \ \& \ \text{Wait}(x)\$st'$

The reducts have been simplified.

Narrowing the preceding formula (1 step) leads to

$x \text{ `NOTin' } xs \ \& \ x \text{ `NOTin' } ys \ \& \ \text{length}(xs) < \text{maxwait} \implies$   
Any  $f:(\text{enabled}(f)(xs,ys) \ \& \ \text{Wait}(x)\$f(xs,ys))$

The reducts have been simplified.

Narrowing the preceding formula (1 step) leads to

True

The reducts have been simplified.

Number of proof steps: 3

## Beweis von AFcrit:

$c \text{ `in' } xs \mid c \text{ `in' } ys \implies AF(Crit(c))(xs,ys)$

Selecting induction variables at positions [1,1,0], [1,1,1] of the preceding formula leads to

$c \text{ `in' } !xs \mid c \text{ `in' } !ys \implies AF(Crit(c))(!xs,!ys)$

Narrowing the preceding formula (2 steps) leads to

$c \text{ `in' } !xs \mid c \text{ `in' } !ys \implies$   
 $Crit(c)(!xs,!ys) \mid \text{All } st':(!xs,!ys) \rightarrow st' \implies AF(Crit(c))\$st'$

Applying the induction hypothesis

$AF(Crit(c))(xs,ys) \iff (c \text{ `in' } xs \mid c \text{ `in' } ys) \& (!xs,!ys) \gg (xs,ys)$

at position [1,1,0,1] of the preceding formula leads to

$c \text{ `in' } !xs \mid c \text{ `in' } !ys \implies$   
 $Crit(c)(!xs,!ys) \mid$   
 $\text{All } st':(!xs,!ys) \rightarrow st' \implies$   
Any  $xs\ ys:((c \text{ `in' } xs \mid c \text{ `in' } ys) \& (!xs,!ys) \gg (xs,ys) \&$   
 $st' = (xs,ys)))$

Applying the theorem

$(xs,ys) \gg st \iff c \text{ `in' } xs \ \& \ (xs,ys) \rightarrow st$

at position [1,1,0,1,0,1]

of the preceding formula leads to

$c \text{ `in' } !xs \mid c \text{ `in' } !ys \implies$   
 $\text{Crit}(c)(!xs,!ys) \mid$   
All  $st':(!xs,!ys) \rightarrow st' \implies$   
Any  $xs \ ys:((c \text{ `in' } xs \mid c \text{ `in' } ys) \ \&$   
 $(c \text{ `in' } !xs \ \& \ (!xs,!ys) \rightarrow (xs,ys)) \ \& \ st' = (xs,ys))$

Applying the theorem

$x \text{ `in' } xs' \mid x \text{ `in' } ys' \iff x \text{ `in' } xs \ \& \ (xs,ys) \rightarrow (xs',ys')$

at position [1,1,0,1,0,0]

of the preceding formula leads to

$c \text{ `in' } !xs \mid c \text{ `in' } !ys \implies$   
 $\text{Crit}(c)(!xs,!ys) \mid$   
All  $st':(!xs,!ys) \rightarrow st' \implies$   
Any  $xs \ ys:(\text{Any } xs1 \ ys1:$

$$(c \text{ `in' } xs_1 \& (xs_1, ys_1) \rightarrow (xs, ys)) \& \\ (c \text{ `in' } !xs \& (!xs, !ys) \rightarrow (xs, ys)) \& st' = (xs, ys)))$$

Substituting  $\neg xs$  for  $xs_1$  to the preceding formula leads to

$$c \text{ `in' } \neg xs \mid c \text{ `in' } \neg ys ==> \\ \text{Crit}(c)(\neg xs, \neg ys) \mid \\ \text{All } st':(\neg xs, \neg ys) \rightarrow st' ==> \\ \text{Any } xs \text{ } ys:(\text{Any } xs_1 \text{ } ys_1: \\ (c \text{ `in' } \neg xs \& (\neg xs, \neg ys_1) \rightarrow (xs, ys)) \& \\ (c \text{ `in' } \neg xs \& (\neg xs, \neg ys) \rightarrow (xs, ys)) \& st' = (xs, ys)))$$

Substituting  $\neg ys$  for  $ys_1$  to the preceding formula leads to

$$c \text{ `in' } \neg xs \mid c \text{ `in' } \neg ys ==> \\ \text{Crit}(c)(\neg xs, \neg ys) \mid \\ \text{All } st':(\neg xs, \neg ys) \rightarrow st' ==> \\ \text{Any } xs \text{ } ys:(\text{Any } xs_1 \text{ } ys_1: \\ (c \text{ `in' } \neg xs \& (\neg xs, \neg ys) \rightarrow (xs, ys)) \& \\ (c \text{ `in' } \neg xs \& (\neg xs, \neg ys) \rightarrow (xs, ys)) \& st' = (xs, ys)))$$

Simplifying the preceding formula (10 steps) leads to

$$\text{All } st':(c \text{ `in' } \neg xs \& (\neg xs, \neg ys) \rightarrow st' ==> \\ \text{Any } xs \text{ } ys:(c \text{ `in' } \neg xs \& (\neg xs, \neg ys) \rightarrow (xs, ys) \& st' = (xs, ys)) \mid$$

c `in` !ys)

Applying the theorem

Any xs ys:(st = (xs,ys))

at position [0,1,0] of the preceding formula leads to

All st':(c `in` !xs & (!xs,!ys) -> st' ==>  
(All st:Any xs2 ys2:  
         (st = (xs2,ys2)) ==>  
         Any xs ys:(c `in` !xs & (!xs,!ys) -> (xs,ys) & st' = (xs,ys))) |  
         c `in` !ys)

Substituting st' for st to the preceding formula leads to

All st':(c `in` !xs & (!xs,!ys) -> st' ==>  
(All st:Any xs2 ys2:  
         (st' = (xs2,ys2)) ==>  
         Any xs ys:(c `in` !xs & (!xs,!ys) -> (xs,ys) & st' = (xs,ys))) |  
         c `in` !ys)

Simplifying the preceding formula (15 steps) leads to

True

Number of proof steps: 11

### *Beweis von transDesc:*

$(xs,ys) \gg st \iff c \text{ `in' } xs \& (xs,ys) \rightarrow st$

The tree has been split.

The current formula is given by

$(xs,ys) \gg st \iff c \text{ `in' } xs \& (xs,ys) \rightarrow st$

Simplifying the preceding formula (2 steps) leads to

$c \text{ `in' } xs \& (xs,ys) \rightarrow st \implies$   
 $(\text{length}(xs) - \text{posi}(xs++ys), \text{maxwait} - \text{length}(xs), \text{length}(ys)) > \text{weight}(st)$

Narrowing the preceding formula (2 steps) leads to

$c \text{ `in' } xs \&$   
Any  $f: (st = f((xs,ys)) \&$   
    (Any  $x: (\text{Idle}(x)(xs,ys) \& \text{maxwait} > \text{length}(xs) \& f = \text{request}(x)) \mid$   
      Any  $x \ xs0: (f = \text{enter} \& xs = (x:xs0) \& ys = []) \mid$   
      Any  $x: (f = \text{leave} \& ys = [x])) \implies$   
 $(\text{length}(xs) - \text{posi}(xs++ys), \text{maxwait} - \text{length}(xs), \text{length}(ys)) > \text{weight}(st)$

Simplifying the preceding formula (11 steps) leads to 3 factors.  
The current factor is given by

```
All x:(c `in` xs & st = (request(x)(xs,ys)) & Idle(x)(xs,ys) &  
maxwait > length(xs) ==>  
(length(xs)-posi(xs++ys),maxwait-length(xs),length(ys)) > weight(st))
```

Simplifying the preceding factors (11 steps) leads to the factor

```
All x:(st = (x:xs,ys) & c `in` xs & x `NOTin` xs & x `NOTin` ys &  
maxwait > length(xs) ==>  
(length(xs)-posi(xs++ys)) > (suc(length(xs))-posi(x:(xs++ys))) |  
True & (length(xs)-posi(xs++ys)) = (suc(length(xs))-posi(x:(xs++ys))) |  
(maxwait-length(xs)) = (maxwait-suc(length(xs))) &  
length(ys) > length(ys) &  
(length(xs)-posi(xs++ys)) = (suc(length(xs))-posi(x:(xs++ys))))
```

Simplifying the preceding factors (2 steps) leads to the factor

```
All x:(st = (x:xs,ys) & c `in` xs & x `NOTin` xs & x `NOTin` ys &  
maxwait > length(xs) ==>  
(length(xs)-posi(xs++ys)) > (suc(length(xs))-posi(x:(xs++ys))) |  
(length(xs)-posi(xs++ys)) = (suc(length(xs))-posi(x:(xs++ys))))
```

Applying the theorem

$\text{posi}(x:s) = \text{suc}(\text{posi}(s)) \leqslant c \text{ `in' } s$

at position [0,1,1,1,1] of the preceding factors leads to the factor

All  $x:(st = (x:xs,ys) \ \& \ c \text{ `in' } xs \ \& \ x \text{ `NOTin' } xs \ \& \ x \text{ `NOTin' } ys \ \&$   
 $\text{maxwait} > \text{length}(xs) \ ==>$   
 $(\text{length}(xs)-\text{posi}(xs+ys)) > (\text{suc}(\text{length}(xs))-\text{posi}(x:(xs+ys))) \ |$   
 $(\text{length}(xs)-\text{posi}(xs+ys)) = (\text{suc}(\text{length}(xs))-\text{suc}(\text{posi}(xs+ys))) \ \&$   
 $c \text{ `in' } (xs+ys))$

Applying the theorem

$x \text{ `in' } (xs+ys) \leqslant x \text{ `in' } xs$

at position [0,1,1,1] of the preceding factors leads to the factor

All  $x:(st = (x:xs,ys) \ \& \ c \text{ `in' } xs \ \& \ x \text{ `NOTin' } xs \ \& \ x \text{ `NOTin' } ys \ \&$   
 $\text{maxwait} > \text{length}(xs) \ ==>$   
 $(\text{length}(xs)-\text{posi}(xs+ys)) > (\text{suc}(\text{length}(xs))-\text{posi}(x:(xs+ys))) \ |$   
 $(\text{length}(xs)-\text{posi}(xs+ys)) = (\text{suc}(\text{length}(xs))-\text{suc}(\text{posi}(xs+ys))) \ \&$   
 $c \text{ `in' } xs)$

Simplifying the preceding factors (4 steps) leads to 2 factors.

The current factor is given by

```
All f:(Any x xs0:(f = enter & xs = (x:xs0) & ys = []) & c `in` xs &
st = f(xs,ys) ==>
(length(xs)-posi(xs++ys),maxwait-length(xs),length(ys)) > weight(st))
```

Simplifying the preceding factors (11 steps) leads to the factor

```
All x xs0:(c = x & st = enter(x:xs0,[]) & ys = [] & xs = (x:xs0) |
c `in` xs0 & st = enter(x:xs0,[]) & ys = [] & xs = (x:xs0) ==>
(suc(length(xs0))-posi((x:xs0)++[]),maxwait-length(x:xs0),0) >
weight(st))
```

Simplifying the preceding factors (11 steps) leads to 3 factors.

The current factor is given by

```
All xs0:(st = (init(c:xs0),[last(c:xs0)]) & ys = [] & xs = (c:xs0) ==>
(suc(length(xs0))-posi(c:xs0)) > (length(xs0)-posi(c:xs0)) |
(suc(length(xs0))-posi(c:xs0)) = (length(xs0)-posi(c:xs0)) &
(maxwait-suc(length(xs0)),0) > (maxwait-length(xs0),1))
```

Simplifying the preceding factors (2 steps) leads to 2 factors.

The current factor is given by

```
All x xs0:(c `in` xs0 & st = enter(x:xs0,[]) & ys = [] & xs = (x:xs0) ==>
(suc(length(xs0))-posi(x:xs0),maxwait-suc(length(xs0)),0) >
```

`weight(st))`

Simplifying the preceding factors (7 steps) leads to a single formula, which is given by

All  $f$ :  
(Any  $x$ :  
( $f = \text{leave} \ \& \ ys = [x]$ )  $\&$   $c \text{'in'} xs \ \& \ st = f(xs,ys) \implies$   
 $(\text{length}(xs)-\text{posi}(xs+ys), \text{maxwait}-\text{length}(xs), \text{length}(ys)) > \text{weight}(st)$ )

Simplifying the preceding formula (11 steps) leads to

All  $x$ :  
( $st = (xs, [])$   $\&$   $c \text{'in'} xs \ \& \ ys = [x] \implies$   
 $(\text{length}(xs)-\text{posi}(xs+[x]), \text{maxwait}-\text{length}(xs), 1) >$   
 $(\text{length}(xs)-\text{posi}(xs), \text{maxwait}-\text{length}(xs), 0)$ )

Simplifying the preceding formula (3 steps) leads to

All  $x$ :  
( $st = (xs, [])$   $\&$   $c \text{'in'} xs \ \& \ ys = [x] \implies$   
 $(\text{length}(xs)-\text{posi}(xs+[x])) > (\text{length}(xs)-\text{posi}(xs)) \mid$   
 $(\text{length}(xs)-\text{posi}(xs+[x])) = (\text{length}(xs)-\text{posi}(xs))$ )

Applying the theorem

$\text{posi}(s+s') = \text{posi}(s) \leqslant c \text{'in'} s$

at position  $[0,1,1,0,1]$  of the preceding formula leads to

```
All x:(st = (xs,[]) & c `in` xs & ys = [x] ==>
  (length(xs)-posi(xs++[x])) > (length(xs)-posi(xs)) |
  (length(xs)-posi(xs)) = (length(xs)-posi(xs)) & c `in` xs)
```

Simplifying the preceding formula (3 steps) leads to

True

Number of proof steps: 17

*Beweis von transDesc:*

$$x \text{ `in` } xs' \mid x \text{ `in` } ys' \iff x \text{ `in` } xs \& (xs,ys) \rightarrow (xs',ys')$$

Simplifying the preceding formula (1 step) leads to

$$x \text{ `in` } xs \& (xs,ys) \rightarrow (xs',ys') \Rightarrow x \text{ `in` } xs' \mid x \text{ `in` } ys'$$

Narrowing the preceding formula (2 steps) leads to

```
x `in` xs &
Any f:((xs',ys')) = f((xs,ys)) &
  (Any x0:(Idle(x0)(xs,ys) & maxwait > length(xs) & f = request(x0)) |
   Any x0 xs0:
     (f = enter & xs = (x0:xs0) & ys = [])) |
```

```
Any x0:(f = leave & ys = [x0])))) ==>  
x `in` xs' | x `in` ys'
```

Simplifying the preceding formula (11 steps) leads to 3 factors.  
The current factor is given by

```
All x0:(x `in` xs & (xs',ys') = (request(x0)(xs,ys)) & Idle(x0)(xs,ys) &  
maxwait > length(xs) ==>  
x `in` xs' | x `in` ys')
```

Simplifying the preceding factors (10 steps) leads to 2 factors.  
The current factor is given by

```
All f:(Any x0 xs0:  
      (f = enter & xs = (x0:xs0) & ys = []) &  
      x `in` xs & (xs',ys') = f(xs,ys) ==>  
      x `in` xs' | x `in` ys')
```

Simplifying the preceding factors (11 steps) leads to 3 factors.  
The current factor is given by

```
All x0 xs0:  
(x = x0 & (xs',ys') = enter(x0:xs0,[]) & ys = [] & xs = (x0:xs0) ==>  
x `in` xs' | x `in` ys')
```

Simplifying the preceding factors (7 steps) leads to the factor

All  $xs0 : (ys' = [last(x:xs0)] \& xs' = init(x:xs0) \& ys = [] \& xs = (x:xs0) ==>$   
 $x \text{ `in` } xs' \mid x = last(x:xs0))$

Applying the theorem

$x \text{ `in` } init(y:s) \mid x = last(y:s) <==> x = y \mid x \text{ `in` } s$

at position [0,1] of the preceding factors leads to the factor

All  $xs0 : (ys' = [last(x:xs0)] \& xs' = init(x:xs0) \& ys = [] \& xs = (x:xs0) ==>$   
 $(x = x \mid x \text{ `in` } xs0) \& xs' = init(xs0))$

Simplifying the preceding factors (3 steps) leads to 2 factors.

The current factor is given by

All  $x0 xs0 :$   
 $(x \text{ `in` } xs0 \& (xs', ys') = enter(x0:xs0, []) \& ys = [] \& xs = (x0:xs0) ==>$   
 $x \text{ `in` } xs' \mid x \text{ `in` } ys')$

Simplifying the preceding factors (5 steps) leads to the factor

All  $x0 xs0 :$   
 $(ys' = [last(x0:xs0)] \& xs' = init(x0:xs0) \& x \text{ `in` } xs0 \& ys = [] \&$

```
xs = (x0:xs0) ==>
x `in` xs' | x = last(x0:xs0))
```

Replacing the subtrees at position [0,1,0,1]  
of the preceding factors leads to the factor

All x0 xs0:

```
(ys' = [last(x0:xs0)] & xs' = init(x0:xs0) & x `in` xs0 & ys = [] &
xs = (x0:xs0) ==>
x `in` init(x0:xs0) | x = last(x0:xs0))
```

Applying the theorem

```
x `in` init(y:s) | x = last(y:s) <==> x = y | x `in` s
```

at position [0,1] of the preceding factors leads to the factor

All x0 xs0:

```
(ys' = [last(x0:xs0)] & xs' = init(x0:xs0) & x `in` xs0 & ys = [] &
xs = (x0:xs0) ==>
x = x0 | x `in` xs0)
```

Simplifying the preceding factors (2 steps) leads to a single formula,  
which is given by

```
All f:(Any x0:(f = leave & ys = [x0]) & x `in` xs & (xs',ys') = f(xs,ys) ==>  
x `in` xs' | x `in` ys')
```

Simplifying the preceding formula (11 steps) leads to

```
All x0:(ys' = [] & xs' = xs & x `in` xs & ys = [x0] ==> x `in` xs' | False)
```

Simplifying the preceding formula (1 step) leads to

```
All x0:(ys' = [] & xs' = xs & x `in` xs & ys = [x0] ==> x `in` xs')
```

Replacing the subtrees at position [0,1,1] of the preceding formula leads to

```
All x0:(ys' = [] & xs' = xs & x `in` xs & ys = [x0] ==> x `in` xs)
```

Simplifying the preceding formula (2 steps) leads to

True

Number of proof steps: 16

## N-queens problem

```
-- queens

specs:          modal
constructs:     final
defuncts:       procs start add shift draw drawAll pic
preds:          Final safe queens loop rec select select'
fovars:         val

axioms:

  states == [start]
& start == (procs,[])
& atoms == [final]

& (s=/= [] ==> (s,val) -> branch$map(shift$s)$filter(safe$1)$map(add$val)$s)
& add(val)(x) == x:val
& (safe(n)[x] <==> True)
& (safe(n)(x:y:s) <==> x /= y+n & y /= x+n & safe(n+1)(x:s))
& shift(s)(x:val) == (s-[x],x:val)

& final -> valid$Final
& (Final <==> null.get0)
```

```

-- 4 queens yield 2 solutions and 15 further states
-- 5 queens yield 10 solutions and 44 further states
-- 6 queens yield 4 solutions and 149 further states
-- 7 queens yield 40 solutions and 512 further states
-- 8 queens yield 92 solutions and 1965 further states
-- 9 queens yield 352 solutions and 8042 further states
-- 10 queens yield 724 solutions and 34815 further states

& pic(s,val) == mat$zip(drop(length$s)$procs)$val

& draw == wtree$fun(st,ite(Final$st,pic$st,text$st))

& drawAll == wtree$pic

& drawT == wtree$fun(st,ite(Final$st,text$final,text$o))

-- logic programs for computing safe board valuations

& (queens(n,val) <== loop([1..n],[],([],val)))
& (loop(s,val,st) <== select(s,x) & safe(1)(x:val) & loop(s-[x],x:val,st))

& (queens(n,val) <== rec([1..n],val))
& rec([],[])
& (rec(x:s,y:val) <== select(x:s,y) & rec(x:s-[y],val) & safe(1)(y:val))

```

```
& select(x:s,x)
& (select(x:s,y) <==> select(s,y))
```

conjects:

```
-- 4 queens
EF(Final)$start      --> True          42 parallel simplification steps
& AF(Final)$start    --> False         219 depthfirst simplification steps
& EG(Final)$start    --> False         11 parallel simplification steps
& AG(Final)$start    --> False         10 parallel simplification steps
```

```
& queens(4,val) -- derive/simplify/refute --> val=[3,1,4,2] | val=[2,4,1,3]
```

```
& queens(5,val) {- derive/simplify/refute -->
                  val=[4,2,5,3,1] | val=[3,5,2,4,1] | val=[5,3,1,4,2] |
                  val=[4,1,3,5,2] | val=[5,2,4,1,3] | val=[1,4,2,5,3] |
                  val=[2,5,3,1,4] | val=[1,3,5,2,4] |
                  val=[3,1,4,2,5] | val=[2,4,1,3,5] -}
```

```
& select([1..4],x)
```

```
& (x=[(0,2),(1,0),(2,3),(3,1)] & [(0,1),(1,3),(2,0),(3,2)]=z)
```

terms:

```

eval$final

--> [([], [2, 0, 3, 1]), ([], [1, 3, 0, 2])]                                if procs = [0..3]

--> [([], [3, 1, 4, 2, 0]), ([], [2, 4, 1, 3, 0]), ([], [4, 2, 0, 3, 1]), ([], [3, 0, 2, 4, 1]),
--   ([], [4, 1, 3, 0, 2]), ([], [0, 3, 1, 4, 2]), ([], [1, 4, 2, 0, 3]), ([], [0, 2, 4, 1, 3]),
--   ([], [2, 0, 3, 1, 4]), ([], [1, 3, 0, 2, 4])]                            if procs = [0..4]
--> [([], [5, 3, 1, 6, 4, 2]), ([], [4, 1, 5, 2, 6, 3]), ([], [3, 6, 2, 5, 1, 4]),
--   ([], [2, 4, 6, 1, 3, 5])]                                              if procs = [1..6]

```

<+> eval\$EF\$final

```

--> [([], [2, 0, 3, 1]), ([], [1, 3, 0, 2]), ([0, 1, 2, 3], []), ([0, 2, 3], [1]), ([0, 1, 3], [2]),
--   ([0, 2], [3, 1]), ([1, 3], [0, 2]), ([2], [0, 3, 1]), ([1], [3, 0, 2])]           if procs = [0..3]

--> [([], [5, 3, 1, 6, 4, 2]), ([], [4, 1, 5, 2, 6, 3]), ([], [3, 6, 2, 5, 1, 4]),
--   ([], [2, 4, 6, 1, 3, 5]),
--   ([1, 2, 3, 4, 5, 6], []), ([1, 3, 4, 5, 6], [2]), ([1, 2, 4, 5, 6], [3]), ([1, 2, 3, 5, 6], [4]),
--   ([1, 2, 3, 4, 6], [5]), ([1, 3, 5, 6], [4, 2]), ([1, 2, 4, 5], [6, 3]), ([2, 3, 5, 6], [1, 4]),
--   ([1, 2, 4, 6], [3, 5]), ([1, 3, 5], [6, 4, 2]), ([1, 4, 5], [2, 6, 3]), ([2, 3, 6], [5, 1, 4]),
--   ([2, 4, 6], [1, 3, 5]), ([3, 5], [1, 6, 4, 2]), ([1, 4], [5, 2, 6, 3]), ([3, 6], [2, 5, 1, 4]),
--   ([2, 4], [6, 1, 3, 5]), ([5], [3, 1, 6, 4, 2]), ([4], [1, 5, 2, 6, 3]), ([3], [6, 2, 5, 1, 4]),
--   ([2], [4, 6, 1, 3, 5])]                                              if procs = [1..6]

```

<+> eval\$AF\$final

--> [([], [2, 0, 3, 1]), ([], [1, 3, 0, 2]), ([0, 2, 3], [1]), ([0, 1, 3], [2]), ([0, 2], [3, 1]),  
([1, 3], [0, 2]), ([2], [0, 3, 1]), ([1], [3, 0, 2])] if procs = [0..3]

<+> eval\$EG\$final

--> [([], [2, 0, 3, 1]), ([], [1, 3, 0, 2])] if procs = [0..3]

<+> eval\$AG\$final

--> [([], [2, 0, 3, 1]), ([], [1, 3, 0, 2])] if procs = [0..3]

enter a number of queens into the entry field; press *specification* > *build Kripke model*  
 press *graph* > *show graph of transitions* > *here*; enter *draw* into the entry field  
 press *tree* > *tree*; press *paint*

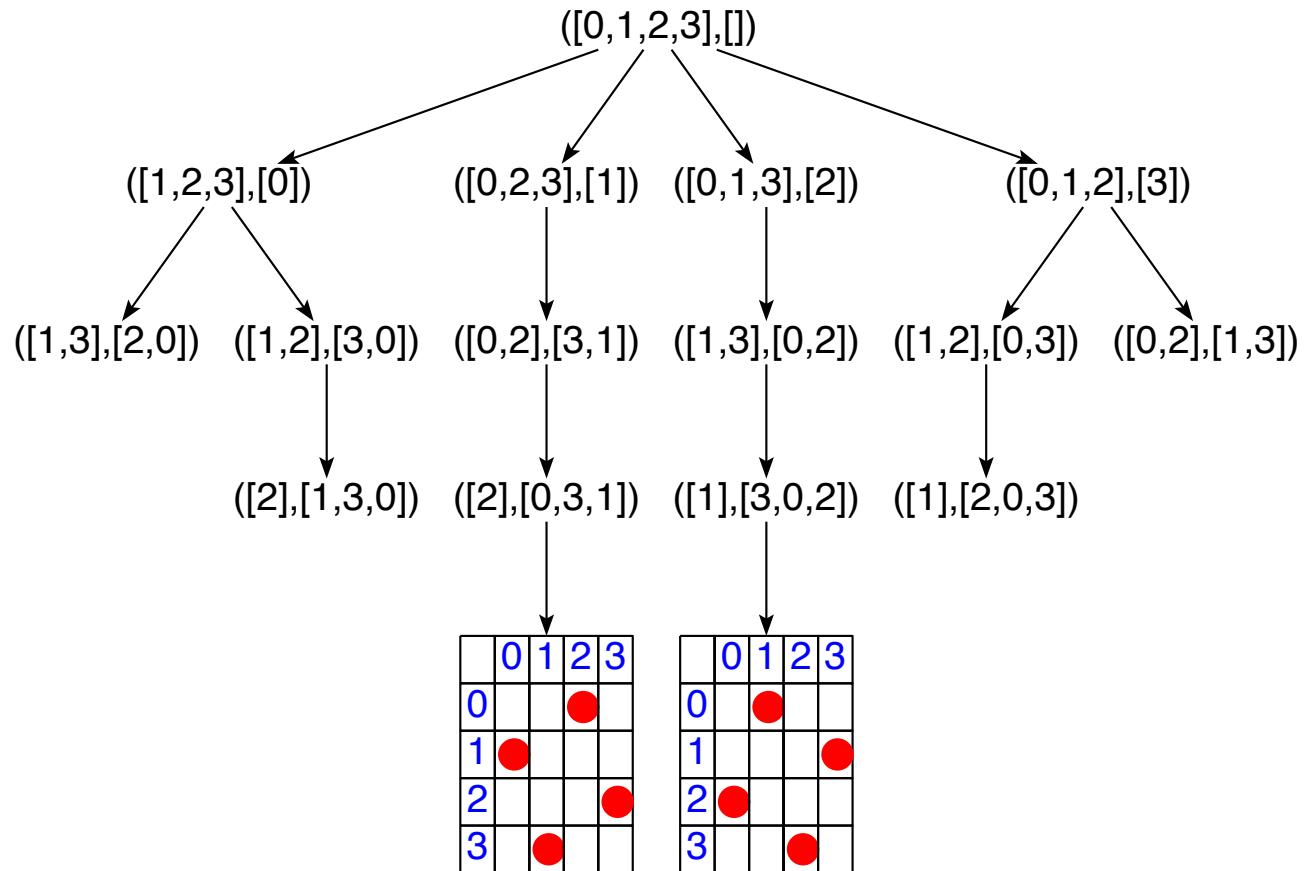
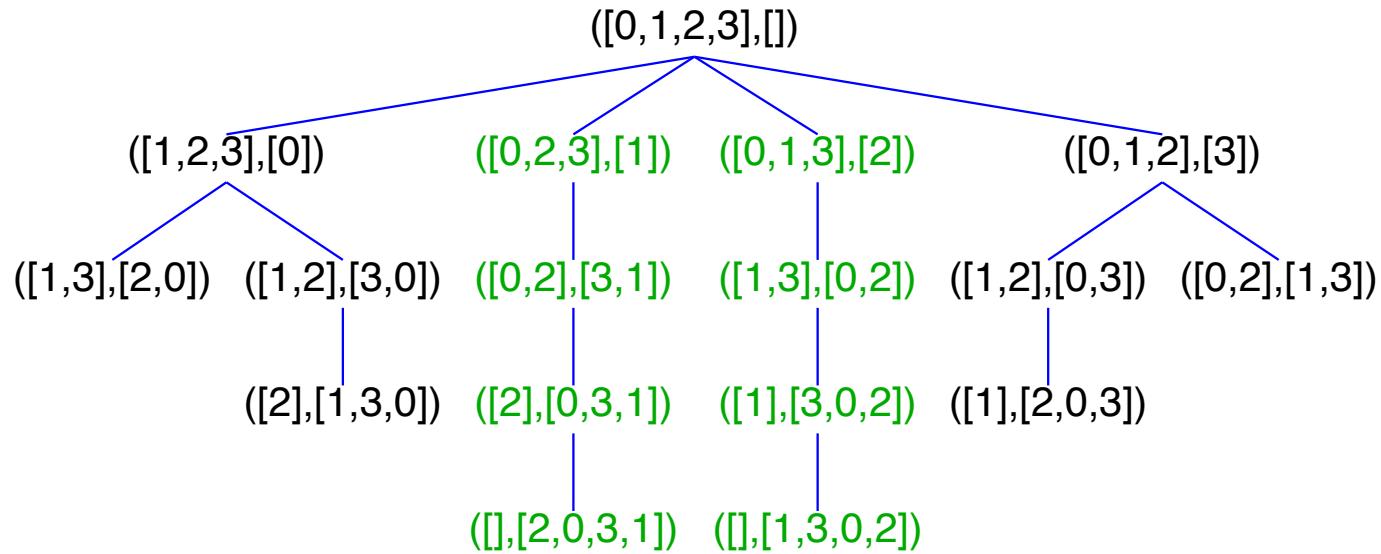


Fig. 13. Transition graph for 4 queens

enter *evalG\$AF\$final* into the text field

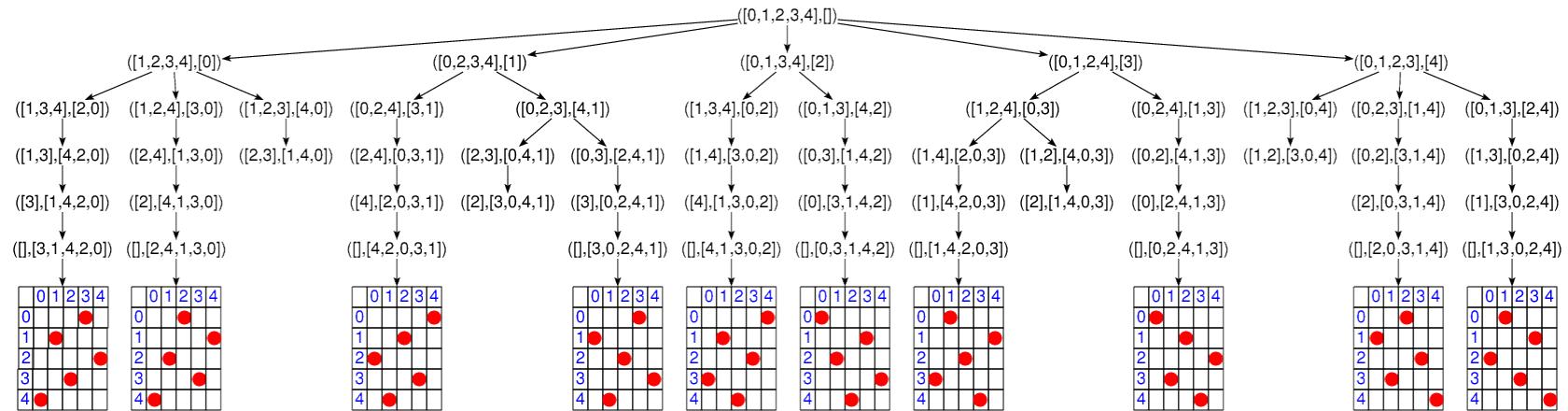
press *parse up*; press *simplify*



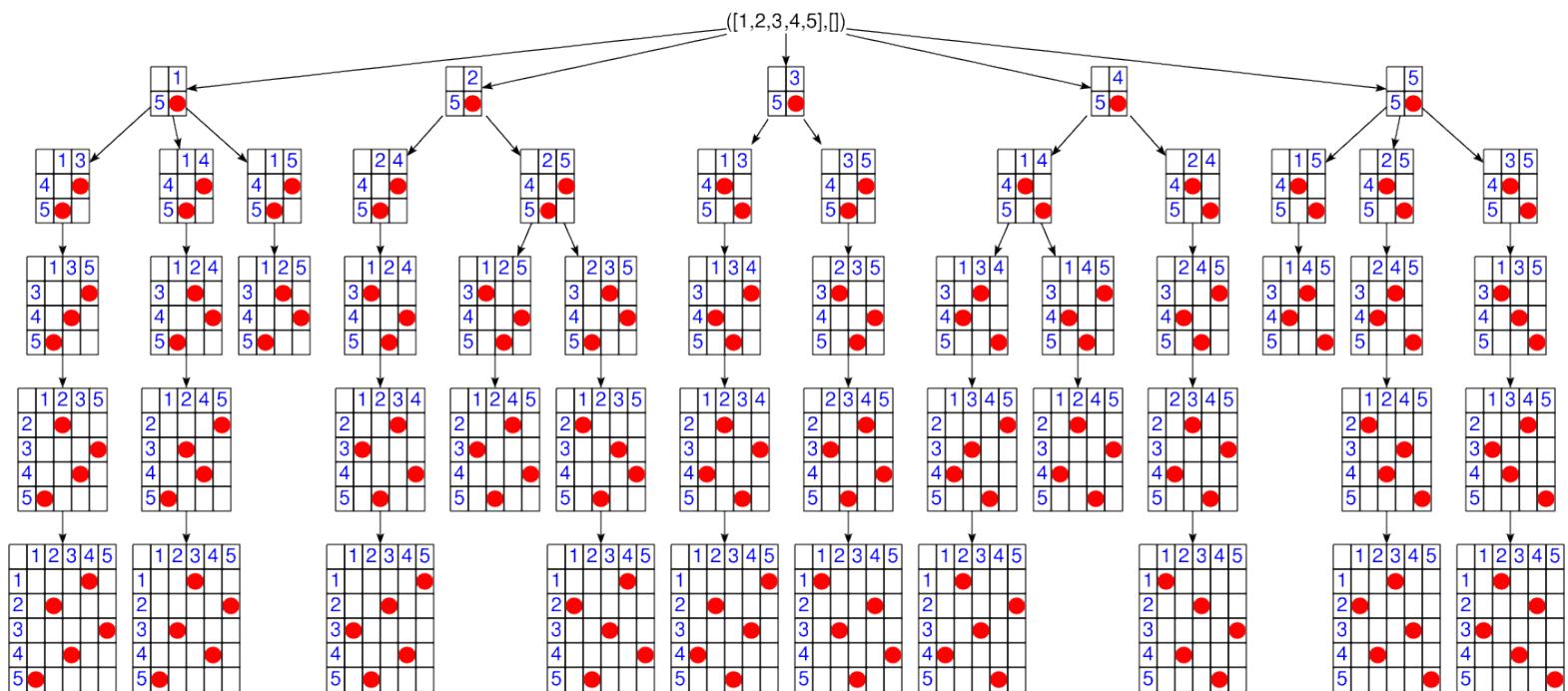
Expander2 computes Kripke models usually from a small number of axioms (here it is just a single one). This simplifies the specification, but may lead to time-consuming computations, especially if the numbers of processes (here: queens) is increased. In such cases, the transition system should be computed with the help of a faster language and then passed to Expander2, only for the purpose of reasoning about or displaying it.

For instance, the pictures below are based on the specifications *queensIO5* and *queensIO6*, respectively (see the *Examples* directory in your Expander2 package), which were generated by running a corresponding Haskell program.

## Transition graphs for 5 queens

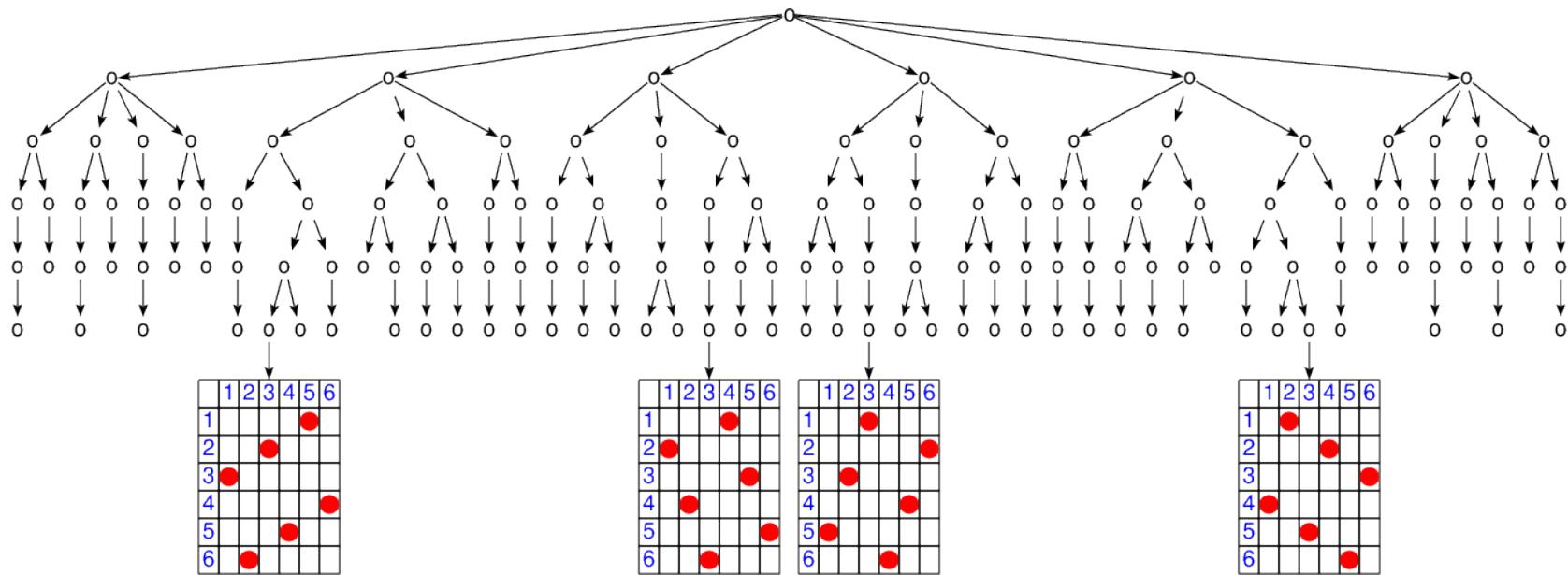


enter *drawAll* instead of *draw* into the entry field

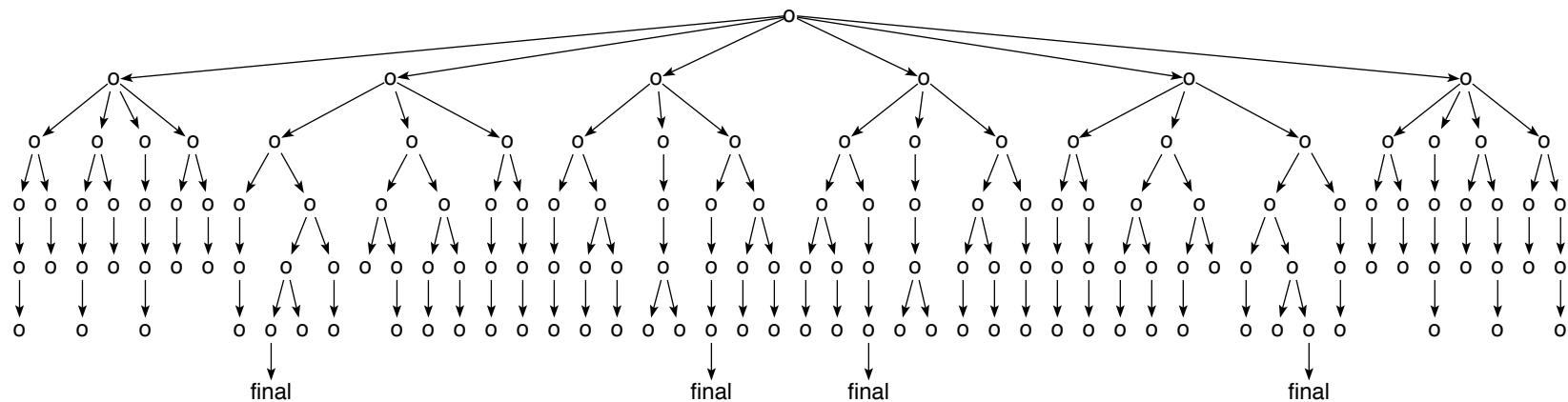


Transition graphs for 6 queens

enter *drawO* instead of *draw* into the entry field



enter *drawT* instead of *draw* into the entry field



## Robots

```
-- robot

specs:          modal
constructs:     final
defuncts:       start blocks blocksCode blocksPic pic
fovars:         pa

axioms:

states == [start] &
start == (0,0) &
atoms == [final] &

blocks == [(20,60),(60,20)] &

(x < 80 & (x+20,y) `NOTin` blocks ==> (x,y) -> (x+20,y)) &
(y < 120 & (x,y+20) `NOTin` blocks ==> (x,y) -> (x,y+20)) &

final -> (80,120) &

blocksCode == turt$map(fun(x,place(x)$red$circ$20))$blocks &
```

```
pic1$pa == turt[path(pa),blocksCode] &  
pic2$pa == turt[path(pa),load$blocksPic]
```

conjects:

```
EF(=(80,120))$start --> True (98 depth- or breadthfirst simplification steps;  
-- 83 simplification steps in parallel with  
-- selection > permute subtrees after 30 steps)  
& set(eval$EF$final) = set$states --> True  
& (80,120)`in`succs(0,0) --> True
```

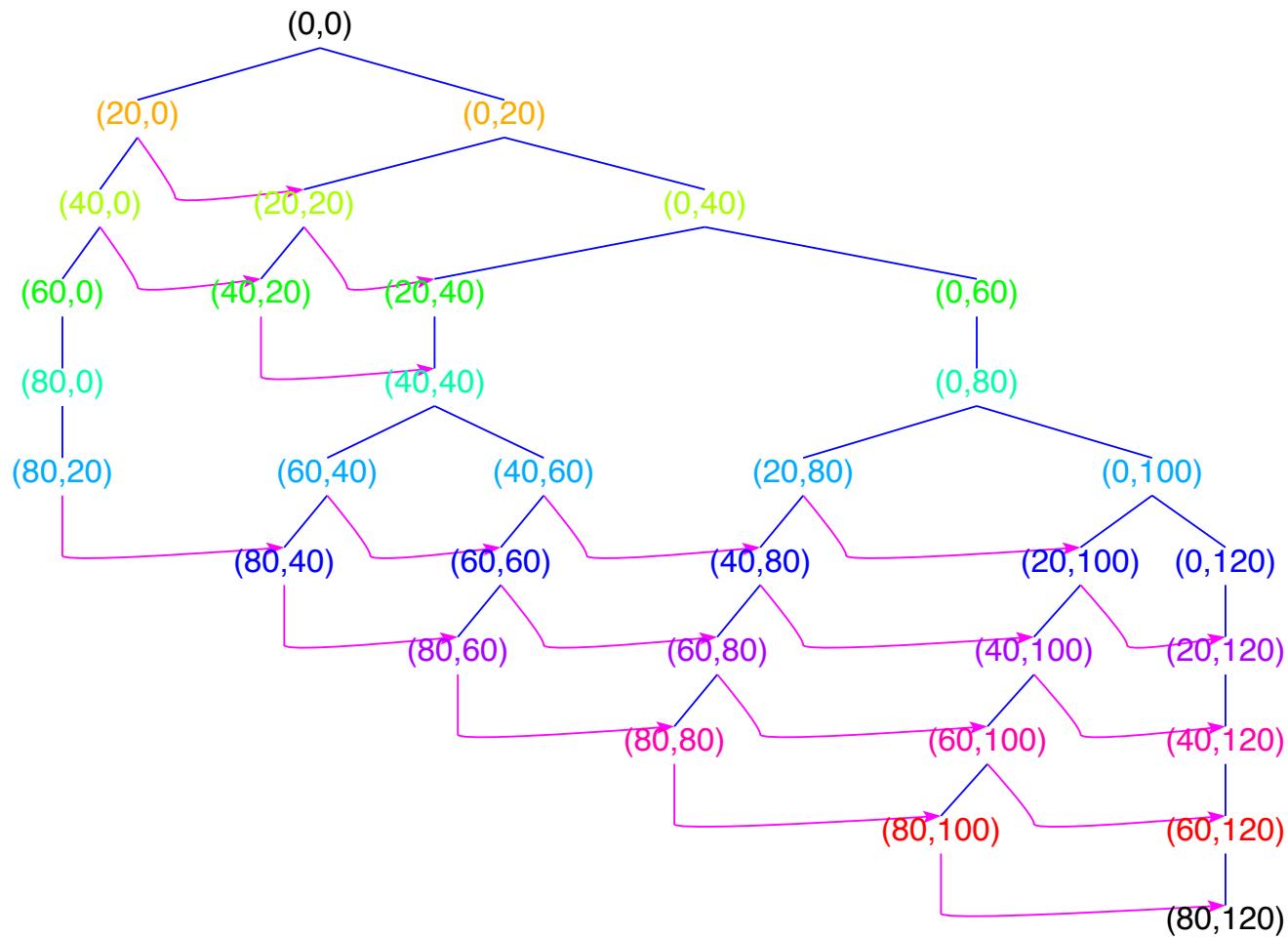
terms:

```
path[(0,0),(20,0),(40,0),(40,20),(40,40),(40,60),(40,80),(40,100),(40,120)] <+>  
--> path[(0,0),(40,0),(40,120)]  
  
map(pic1)$traces(0,0)(40,120) <+> -- 12 traces from (0,0) to (40,120)  
map(pic2)$traces(0,0)(40,120) <+>  
  
map(pic2)$traces(0,0)(80,120) <+> -- 106 traces from (0,0) to (80,120)  
  
save(blocksCode,blocksPic) <+> load$blocksPic
```

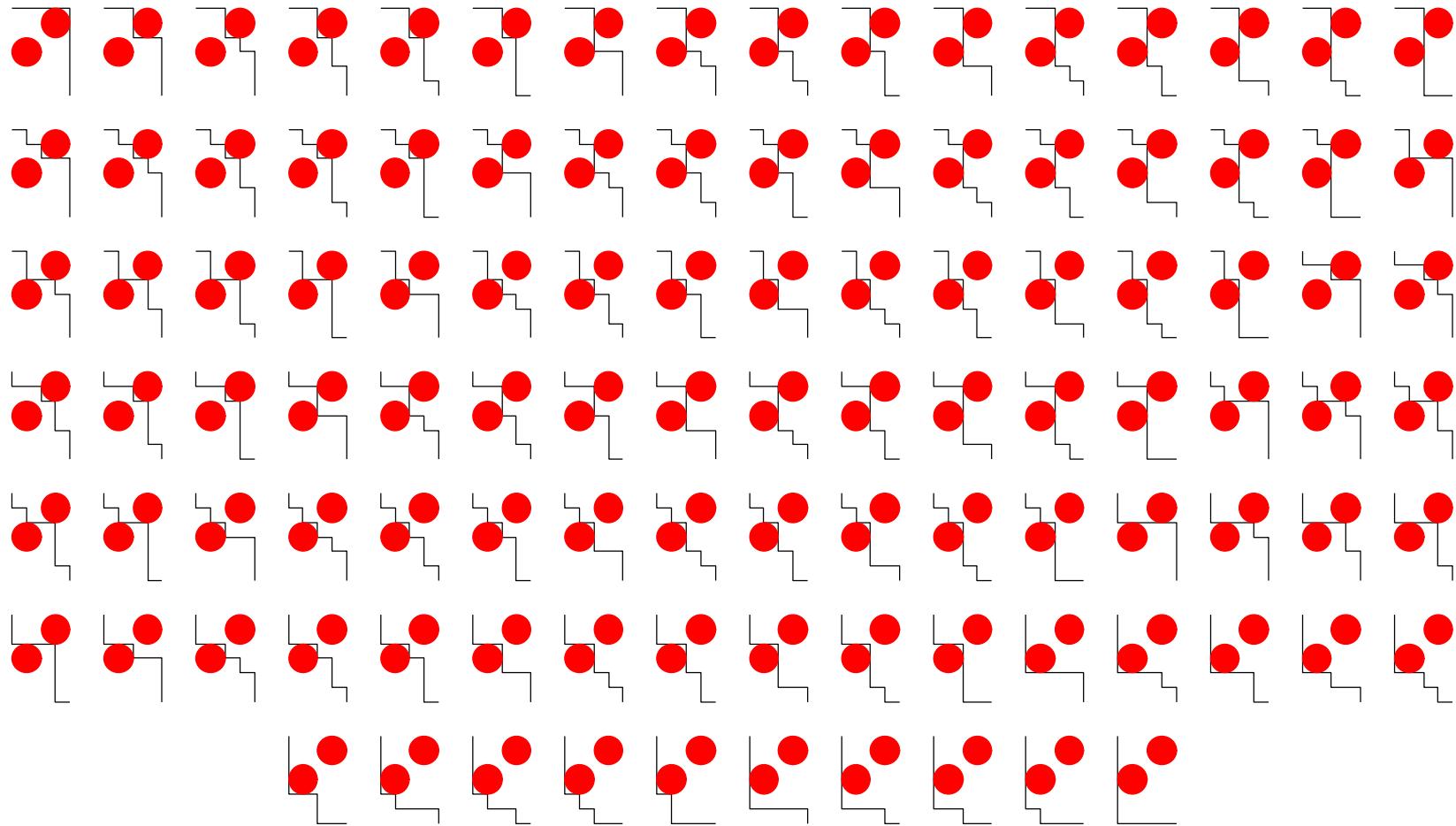
press *specification* > *build Kripke model*

press *graph* > *show graph of transitions* > *with state equivalence*

States in the same equivalence class are colored equally. Every black state is equivalent only to itself.



enter `map(pic1)$traces(0,0)(80,120)` into the text field; press *parse up*; press *simplify*  
press *tree > widgets*; press *paint*; enter `m116` into the mode field  
enter `3` into the space field; press *arrange*



## Filling problem

```
-- bottle

specs:          modal
constructs:     final
defuncts:       draw pic
preds:          Final

axioms:

states == [(0,0)] & atoms == [final]

& (x < 3 ==> (x,y) -> (3,y))                      -- fill x
& (y < 5 ==> (x,y) -> (x,5))                      -- fill y
& (x > 0 ==> (x,y) -> (0,y))                      -- clear x
& (y > 0 ==> (x,y) -> (x,0))                      -- clear y
& (x > 0 & x+y = z & z <= 5 ==> (x,y) -> (0,z)) -- add x to y
& (y > 0 & x+y = z & z <= 3 ==> (x,y) -> (z,0)) -- add y to x
& (x < 3 & x+y = z & z > 3 ==> (x,y) -> (3,z-3)) -- fill x from y
& (y < 5 & x+y = z & z > 5 ==> (x,y) -> (z-5,5)) -- fill y from x

& final -> valid$Final
& (Final(x,y) <==> x+y=4)
```

```

& draw == wtree$fun(st,ite(Final$st,pic$st,frame(3)$text$st))
& pic(x,y) == center$turt[pile(3,x),J$20,pile(5,y)]

conjects: EF(Final)(0,0) --> True (69 parallel simplification steps)

terms:

map(pic)[(0,0),(3,0),(0,3),(3,3),(1,5),(1,0),(0,1),(3,1)] <+> -- trace1
map(pic)[(0,0),(0,5),(3,2),(0,2),(2,0),(2,5),(3,4),(0,4)] <+> -- trace2

traces(0,0)(3,1) <+>
-- [[(0,0),(3,0),(0,3),(0,5),(3,2),(0,2),(2,0),(2,5),(3,4),(0,4),(3,1)],
-- [(0,0),(3,0),(0,3),(3,3),(1,5),(1,0),(0,1),(3,1)],
-- [(0,0),(0,5),(3,2),(3,0),(0,3),(3,3),(1,5),(1,0),(0,1),(3,1)],
-- [(0,0),(0,5),(3,2),(0,2),(2,0),(2,5),(3,4),(0,4),(3,1)]]

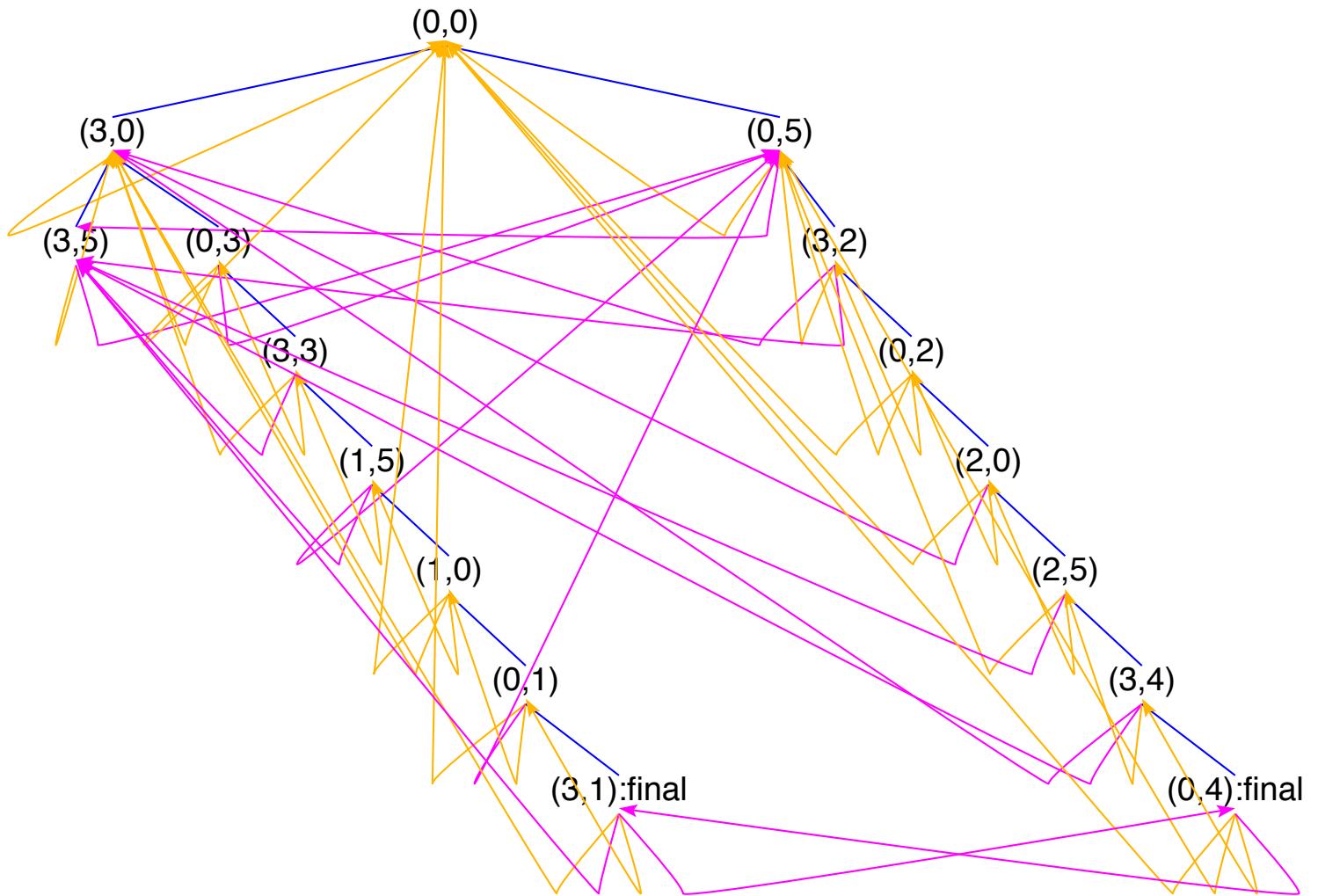
traces(0,0)(0,4) <+>
-- [[(0,0),(3,0),(0,3),(0,5),(3,2),(0,2),(2,0),(2,5),(3,4),(0,4)],
-- [(0,0),(3,0),(0,3),(3,3),(1,5),(1,0),(0,1),(3,1),(0,4)],
-- [(0,0),(0,5),(3,2),(3,0),(0,3),(3,3),(1,5),(1,0),(0,1),(3,1),(0,4)],
-- [(0,0),(0,5),(3,2),(0,2),(2,0),(2,5),(3,4),(0,4),(3,1)]]

eval$final <+> --> [(3,1),(0,4)]
evalG$EF$final

```

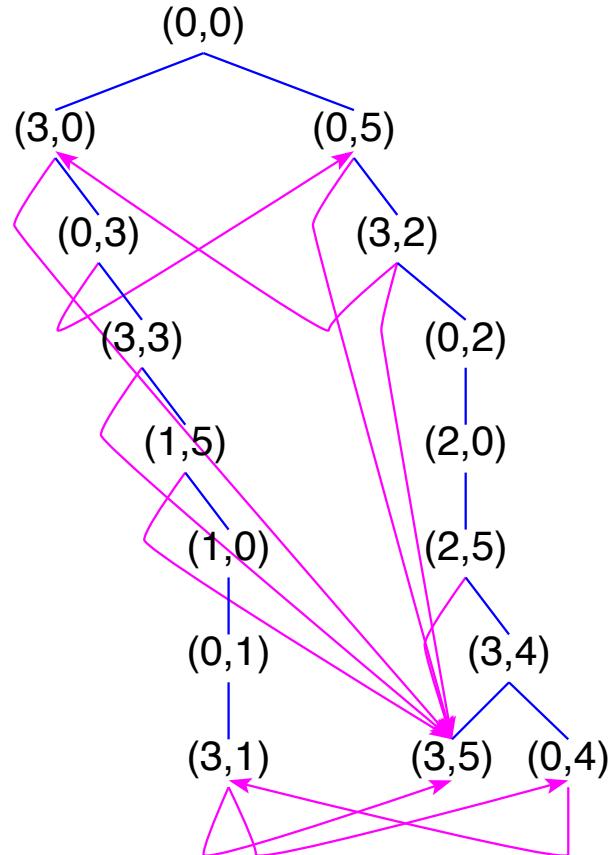
press *specification* > *build Kripke model*

press *graph* > *show graph of Kripke model* > *here*



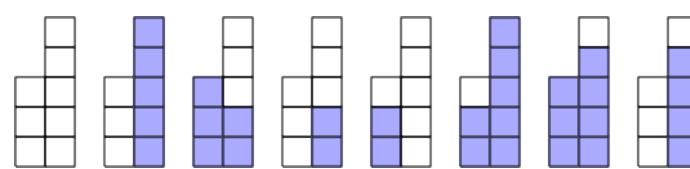
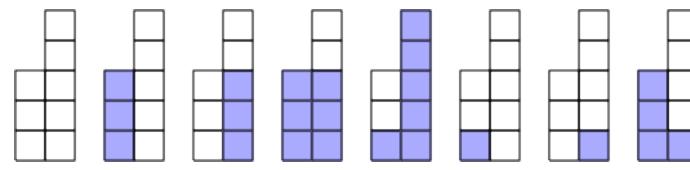
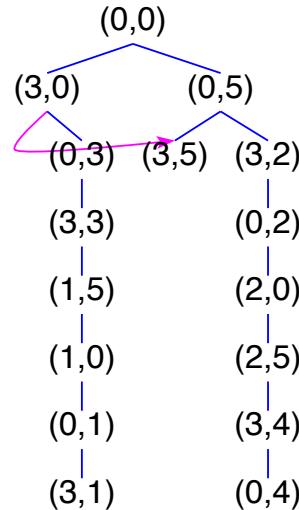
press *specification* > *build Kripke model .. cycle-free*

press *graph* > *show graph of transitions* > *here*



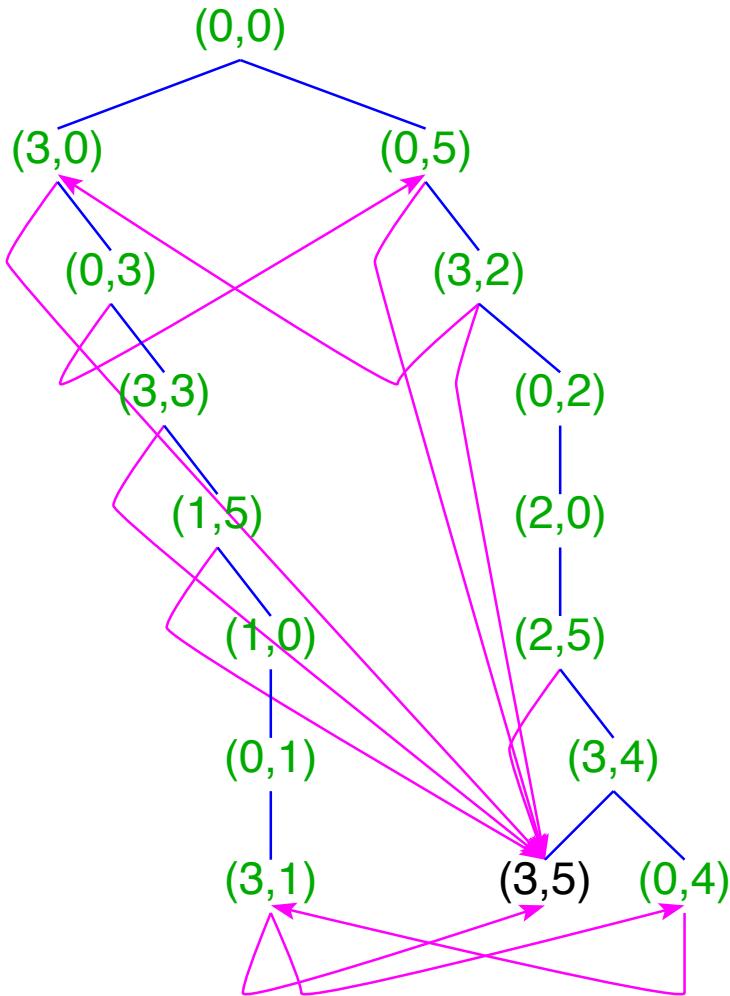
press *specification > build Kripke model .. pointer-free*

press *graph > show graph of transitions > here*

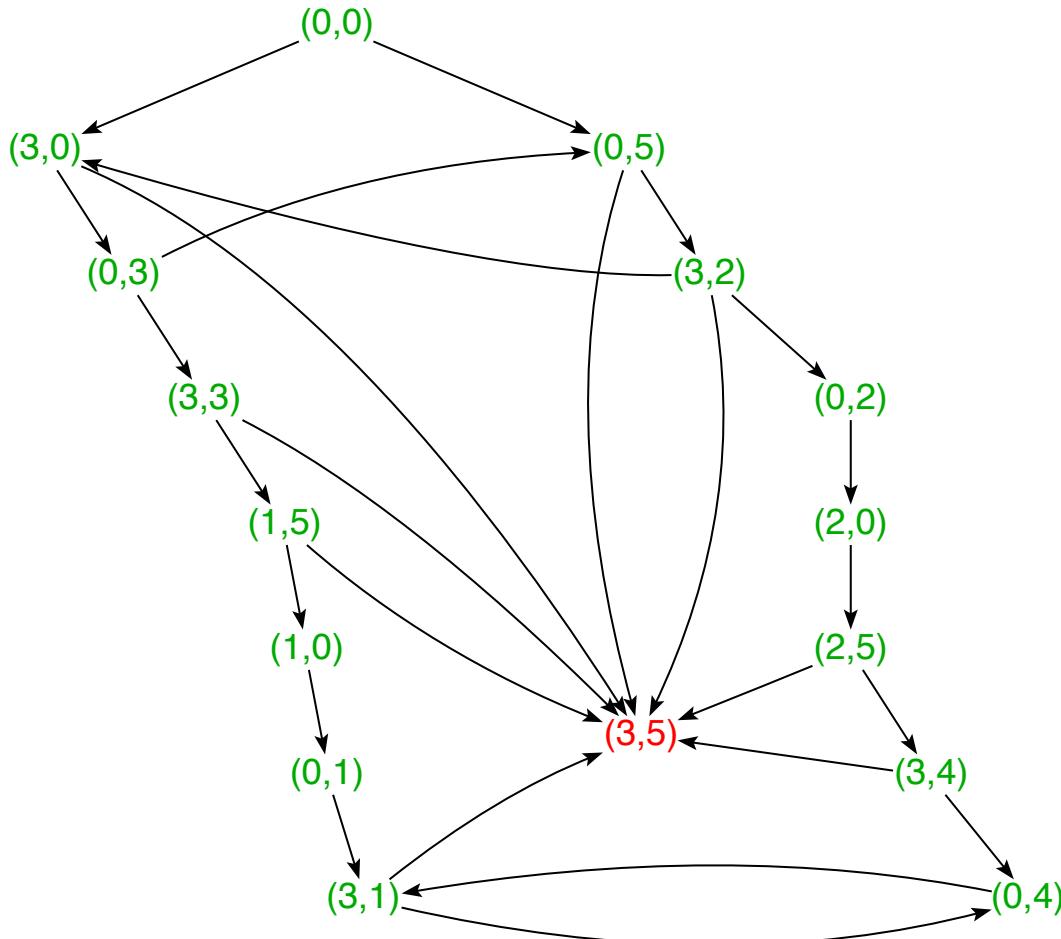


enter  $evalG\$EF\$final$  into the text field

press *parse up*; press *simplify*



enter *evalG\$EF\$final* into the text field; press *parse up*; press *simplify*  
enter *draw* into the entry field; press *tree > tree*; press *paint*  
adapt red support points and press *combis* to remove them



## Elevator

## Towers of Hanoi

```
-- HANOI

preds:          Initial Final EF
constructs:    A B C
defuncts:       procs lg start draw piles pic
fovars:         st xs ys zs

axioms:
lg == length(procs) &
start == A(procs)^B[]^C[] & states == [start] &

A(x:xs)^B[] -> A(xs)^B[x] &
A(x:xs)^C[] -> A(xs)^C[x] &
A[]^B(x:xs)^C(y:ys) -> A[x]^B(xs)^C(y:ys)<+>A[y]^B(x:xs)^C(ys) &

(x < y ==> A(x:xs)^B(y:ys) -> A(xs)^B(x:y:ys)) &
(y < x ==> A(x:xs)^B(y:ys) -> A(y:x:xs)^B(ys)) &
(x < y ==> A(x:xs)^C(y:ys) -> A(xs)^C(x:y:ys)) &
(y < x ==> A(x:xs)^C(y:ys) -> A(y:x:xs)^C(ys)) &
(x < y ==> B(x:xs)^C(y:ys) -> B(xs)^C(x:y:ys)) &
(y < x ==> B(x:xs)^C(y:ys) -> B(y:x:xs)^C(ys)) &
```

```

(Initial$st <==> st = start) &
(Final$A(xs)^B(ys)^C(zs) <==> ys = procs | zs = procs) &

draw == wtree $ fun(st,ite(Initial$st|Final$st)(piles$st,st)) &
piles$A(xs)^B(ys)^C(zs) == turt[tow$xs,J$16,tow$ys,J$16,tow$zs] &
tow$xs == scale(8)$pileR$lg:xs

conjects: start `sats` EF$Final          --> True

```

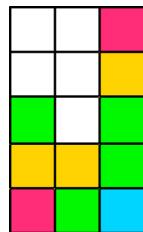


Fig. 14.  $\text{draw\$piles\$A}[0,1,2]^{\text{B}}[1,0]^{\text{C}}[2,1,0,0,4]$ , falls  $\text{lg} = 5$

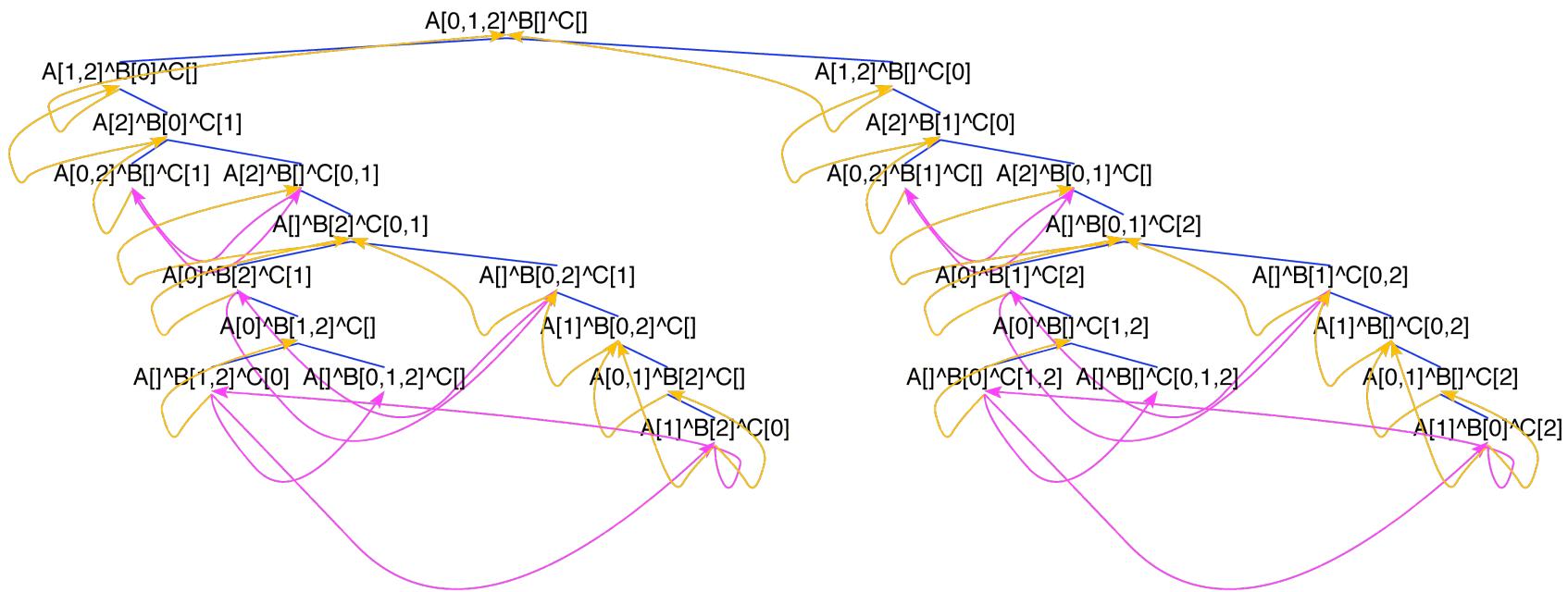


Fig. 15. *Transitionsgraph von HANOI für 3 Prozesse*

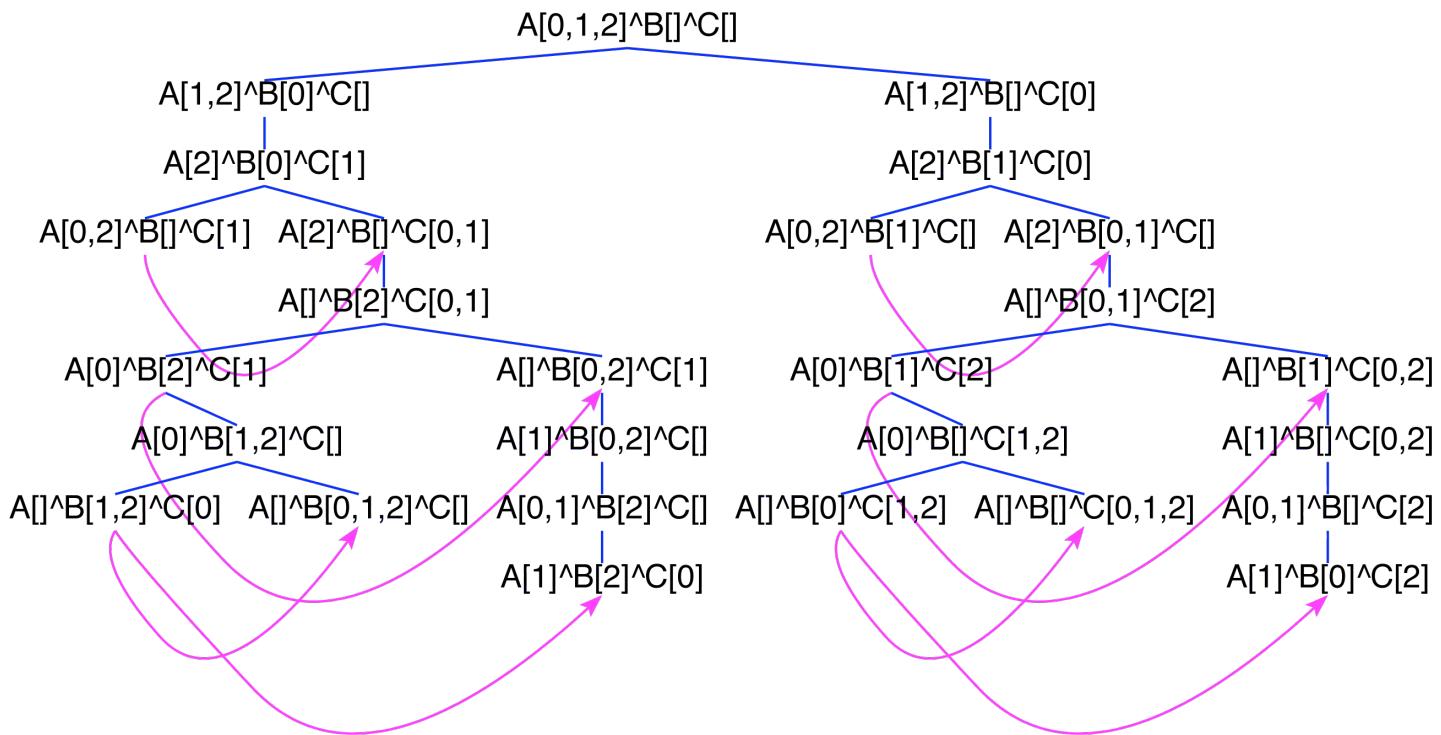


Fig. 16. ... bei Aufbau mit build cycle-free model

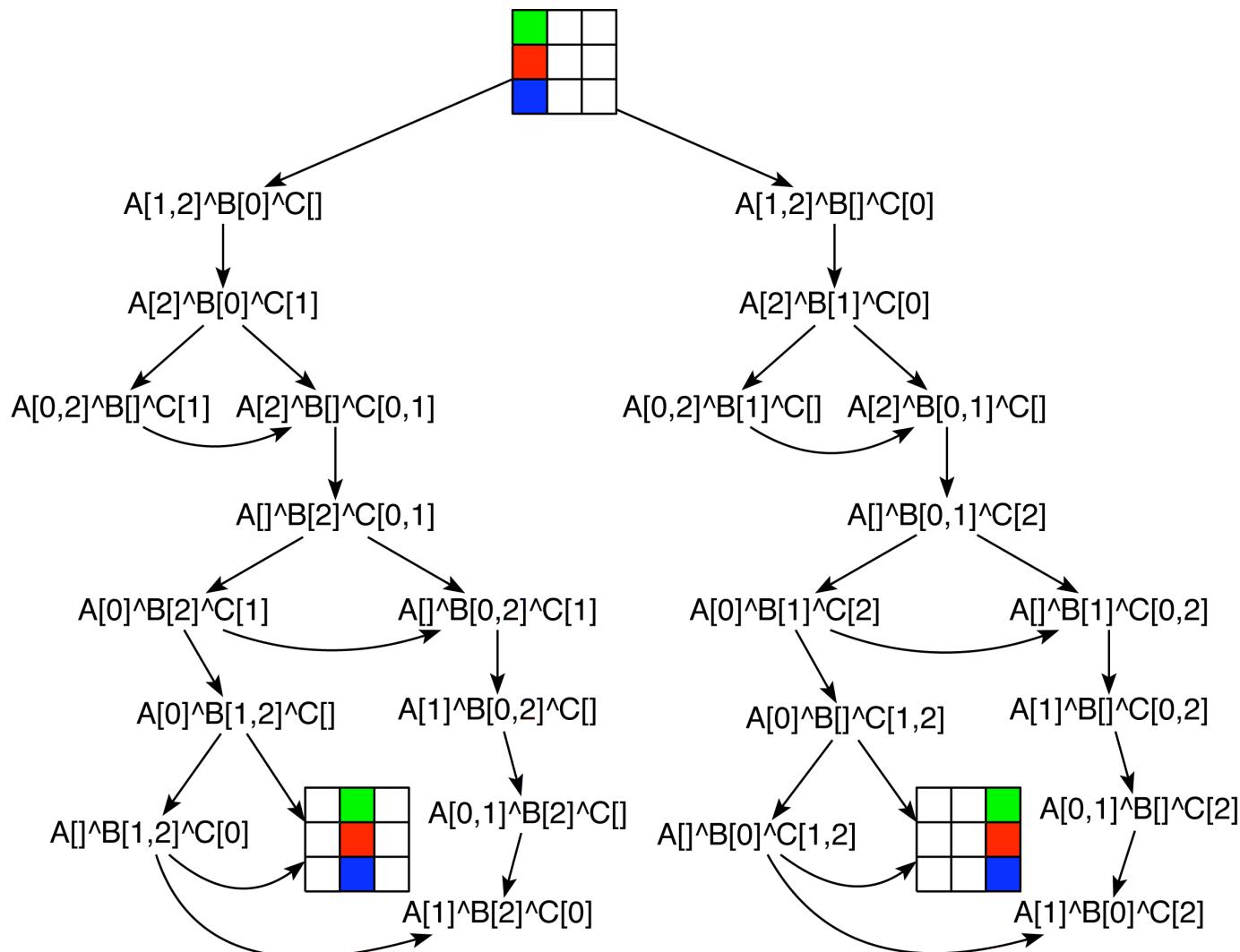


Fig. 17. ... bei Aufbau mit build cycle-free model und Aufruf des Painters mit draw

-- HANOIacts

preds:           Initial Final EF

constructs:      A B C AB BA AC CA BC CB

defuncts:        procs lg start draw piles pic

fovars:          xs ys zs

axioms:

lg == length(procs) &

start == A(procs)^B[]^C[] & states == [start] &

labels == [AB,BA,AC,CA,BC,CB] &

(A(x:xs)^B[],AB) -> A(xs)^B[x] &

(A(x:xs)^C[],AC) -> A(xs)^C[x] &

(A[]^B(x:xs),BA) -> A[x]^B(xs) &

(A[]^C(x:xs),CA) -> A[x]^C(xs) &

(B[]^C(x:xs),CB) -> B[x]^C(xs) &

(B(x:xs)^C[],BC) -> B(xs)^C[x] &

(x < y ==> (A(x:xs)^B(y:ys),AB) -> A(xs)^B(x:y:ys)) &

(y < x ==> (A(x:xs)^B(y:ys),BA) -> A(y:x:xs)^B(ys)) &

(x < y ==> (A(x:xs)^C(y:ys),AC) -> A(xs)^C(x:y:ys)) &

(y < x ==> (A(x:xs)^C(y:ys),CA) -> A(y:x:xs)^C(ys)) &

(x < y ==> (B(x:xs)^C(y:ys),BC) -> B(xs)^C(x:y:ys)) &

(y < x ==> (B(x:xs)^C(y:ys),CB) -> B(y:x:xs)^C(ys)) &

```

(Initial$st <==> st = start) &
(Final$A(xs)^B(ys)^C(zs) <==> ys = procs | zs = procs) &

draw == wtree $ fun(x,ite(x `in` labels)
                     (frameS$text$x,ite(Initial$x|Final$x)
                      (piles$x,x))) &
piles$A(xs)^B(ys)^C(zs) == center$turt[tow$xs,J$16,tow$ys,J$16,tow$zs] &
tow$xs == scale(8)$pileR$lg:xs

conjects: start `sats` EF$Final           --> True

```

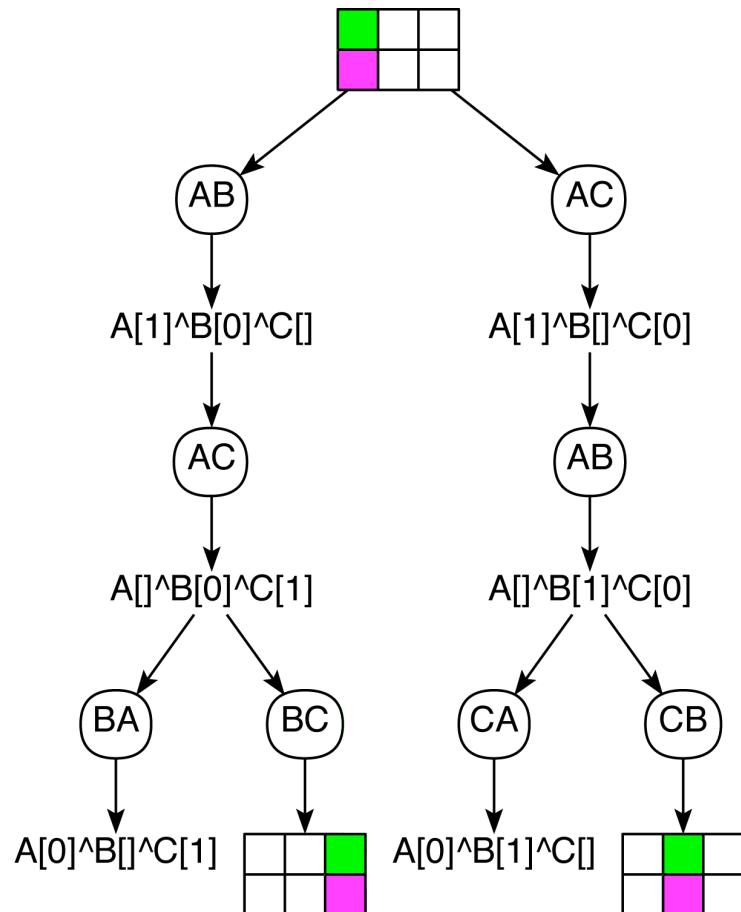


Fig. 18. *Transitionsgraph von Hanoiacts für 2 Prozesse bei Aufbau mit build pointer-free model und Aufruf des Painters mit draw*

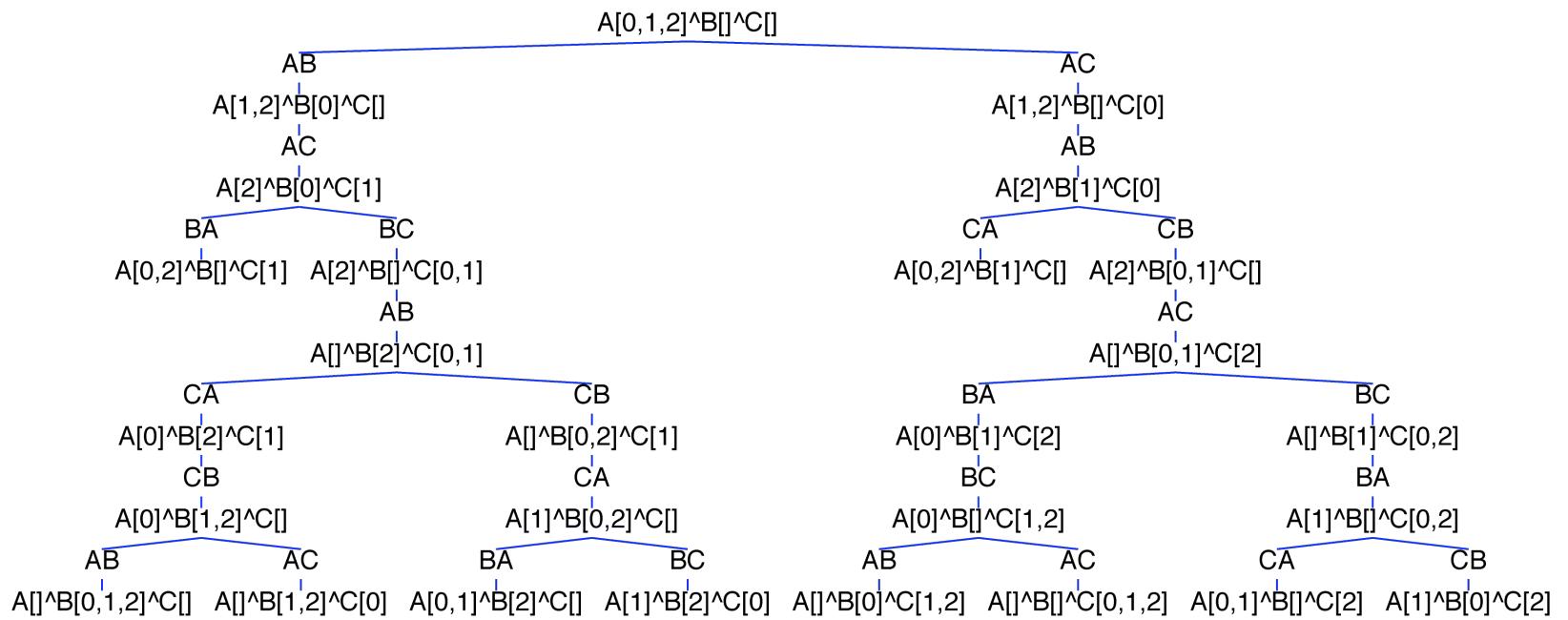


Fig. 19. *Transitionsgraph von HANOIacts für 3 Prozesse bei Aufbau mit build pointer-free model*

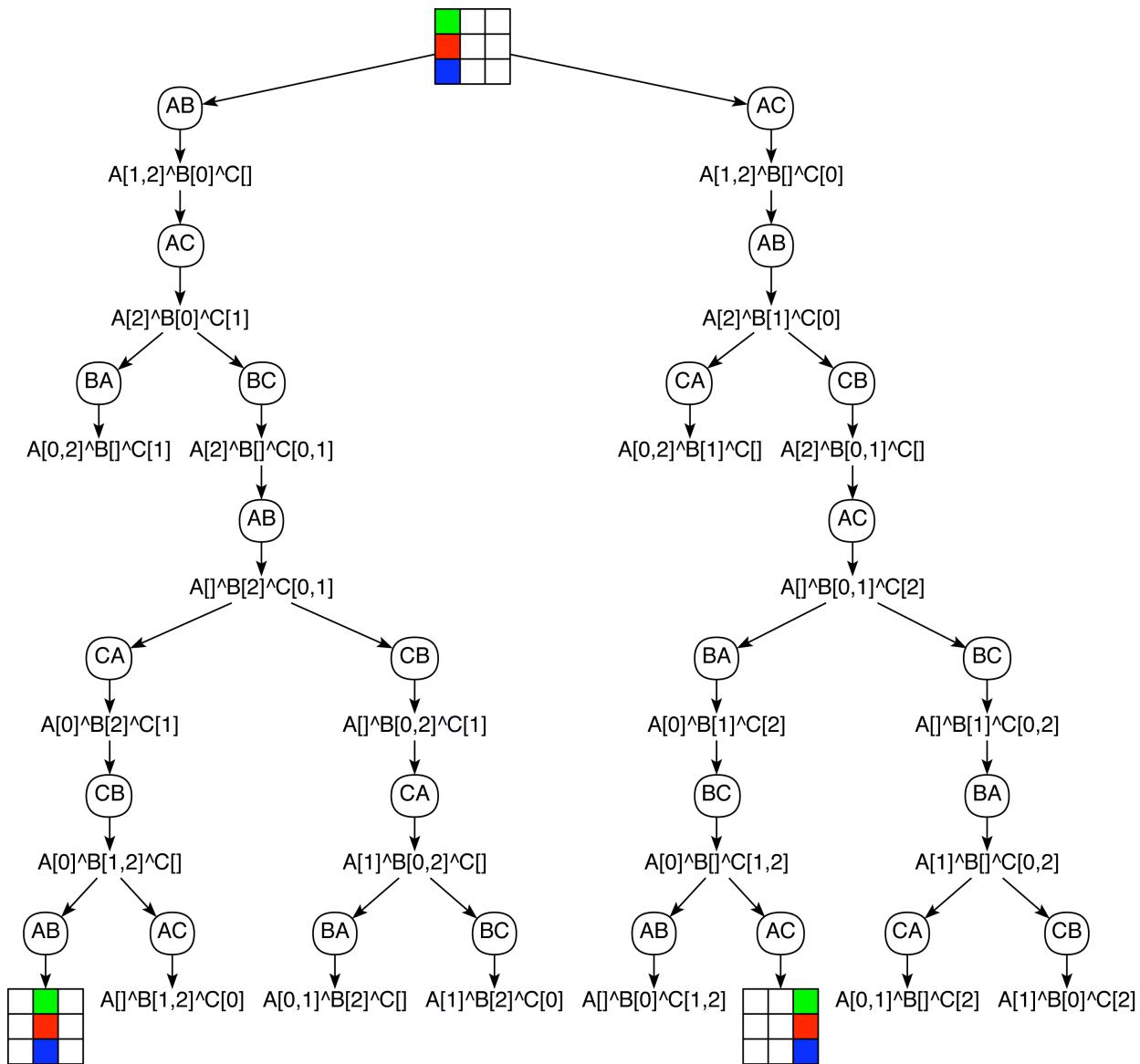


Fig. 20. ... bei Aufbau mit build pointer-free model und Aufruf des Painters mit draw

## Natural numbers

-- NAT

preds: X Nat even odd  
defuncts: div fib loop fibL loop1 loop2 sum  
fovars: q r n  
hovars: f X

axioms:

sum(0) = 0  
& sum(suc(x)) = sum(x)+x+1  
& (div(x,y) = (0,x) <==> x < y)  
& (div(x,y) = (q+1,r) <==> 0 < y & y <= x & div(x-y,y) = (q,r))  
& fib(0) == 0  
& fib(1) == 1  
& fib(suc(suc(n))) == fib(n)+fib(n+1)

& Nat(0)  
& (Nat(suc(x)) <==> Nat(x))

& even(0)  
& (even(suc(x)) <==> odd(x))

```

& (odd(suc(x)) <==> even(x))

-- & div(x,y) = loop(y,0,x)
& (r < y ==> loop(y,q,r) == (q,r))
& (r >= y ==> loop(y,q,r) == loop(y,q+1,r-y))
& (INV(x,y,q,r) <==> x = (y*q)+r)

& fibL(n) = loop1(n,0,1)
& loop1(0,x,y) = x
& loop1(suc(n),x,y) = loop1(n,y,x+y)

& loop2(f)(0)(x) == x
& loop2(f)(suc(n))(x) == f$loop2(f)(n)(x)

& suc(x) >> x

-- & (Nat(0) <==> True)
-- & (Nat(suc(x)) <==> Nat(x))

-- & (Nat <==> MU X.rel(x,x=0|Any y:x=suc(y)&X(y)))

-- & (INV(n,x,y,z) <==> n >= x & y = fib(n-x) & z = fib(n-x+1))

& (x >> y <==> x > y)

```

conjects:

```
(sum(x) = y ==> x*(x+1) = 2*y)                                -- sum1
& (div(x,y) = (q,r) ==> x = (y*q)+r & r < y)                  -- div
& (x = (y*q)+r ==> loop(y,q,r) = div(x,y))                    -- divloop
& fibL(x) = fib(x)                                              -- fib
& (Nat(x) ==> x+y = y+x)                                         -- comm
& (Nat(x) ==> x+(y+z) = (x+y)+z)                               -- assoc
& (Nat(x) ==> x < 2**x)                                           -- exp
& (Nat(x) ==> even(x) | odd(x))                                 -- evenodd
& (Nat(x) ==> suc(x)*x = x**2+x)                                -- pot
& (Nat(n) ==> loop2(f)(n)$f$x = f$loop2(f)(n)(x))            -- natloop
& (even(x) ==> x = 0 | Any y z:(x = y+z & odd(y) & odd(z)))  -- evenany

& div(5,4) = x
& div(5,x) = (1,1)
& Any x y:(x < y & div(5,3)=(x,y))
```

terms:

```
fun((suc(x),y),x+x+y)(6,10) <+>
fun((suc(x),y),fun(z,x+y+z)(5))(suc(z),10) <+>
filter(rel(x,x<5))[1,2,3,4,5,6] <+>
filter(rel(x,Int(x)))[1,2,3,6,4,5,6]
```

*Proof of evenodd by fixpoint induction:*

## 0. Derivation of

$\text{Nat}(x) \implies \text{even}(x) \mid \text{odd}(x)$

All simplifications are admitted.

Equation removal is safe.

### 1. Adding

$(\text{Nat}_0(x) \implies \text{even}(x) \mid \text{odd}(x))$

to the axioms and applying FIXPOINT INDUCTION wrt

$\text{Nat}(0)$

&  $(\text{Nat}(\text{suc}(x)) \leq \text{Nat}(x))$

at positions

[]

of the preceding formulas leads to

$\text{All } x : (\text{even}(0) \mid \text{odd}(0)) \ \& \ \text{All } x : (\text{even}(\text{suc}(x)) \mid \text{odd}(\text{suc}(x))) \leq \text{Nat}_0(x)$

2. NARROWING the preceding formulas (one step) leads to

$\text{All } x : (\text{True} \mid \text{odd}(0)) \ \& \ \text{All } x : (\text{even}(\text{suc}(x)) \mid \text{odd}(\text{suc}(x))) \leq \text{Nat}_0(x)$

The axioms have been MATCHED against their redices.

THIS GOAL COINCIDES WITH GOAL NO. 2

3. NARROWING the preceding formulas (one step) leads to

All x:(True | odd(0)) & All x:(odd(x) | odd(suc(x)) <==> Nat0(x))

The axioms have been MATCHED against their redices.

4. NARROWING the preceding formulas (one step) leads to

All x:(True | odd(0)) & All x:(odd(x) | even(x) <==> Nat0(x))

The axioms have been MATCHED against their redices.

5. NARROWING the preceding formulas (one step) leads to

All x:(True | odd(0)) & All x:(odd(x) | even(x) <==> even(x) | odd(x))

The axioms have been MATCHED against their redices.

6. SIMPLIFYING the preceding formulas (one step) leads to

True

*Proof of evenodd by Noetherian induction:*

0. Derivation of

$$\text{Nat}(x) \implies \text{even}(x) \mid \text{odd}(x)$$

All simplifications are admitted.

Equation removal is safe.

1. SELECTING INDUCTION VARIABLES at positions

[0,0]

of the preceding formulas leads to

$$\text{Nat}(!x) \implies \text{even}(!x) \mid \text{odd}(!x)$$

2. Applying the axioms

$$\text{even}(0)$$

$$\& (\text{even}(\text{suc}(x)) \leqslant \text{odd}(x))$$

at positions

[1,0]

of the preceding formulas leads to

$$\text{Nat}(!x) \implies (!x = 0 \mid \text{Any } x : (\text{odd}(x) \& !x = \text{suc}(x))) \mid \text{odd}(!x)$$

### 3. Applying the theorem

$(\text{odd}(\text{suc}(x)) \leq \text{even}(x))$

at positions

[1,1]

of the preceding formulas leads to

$\text{Nat}(!x) \implies$

$(!x = 0 \mid \text{Any } x : (\text{odd}(x) \ \& \ !x = \text{suc}(x))) \mid \text{Any } x_0 : (\text{even}(x_0) \ \& \ !x = \text{suc}(x_0))$

### 4. SIMPLIFYING the preceding formulas (one step) leads to

$\text{Nat}(!x) \implies$

$!x = 0 \mid \text{Any } x : (\text{odd}(x) \ \& \ !x = \text{suc}(x)) \mid \text{Any } x_0 : (\text{even}(x_0) \ \& \ !x = \text{suc}(x_0))$

### 5. Applying the axioms

$\text{Nat}(0)$

$\& (\text{Nat}(\text{suc}(x)) \leq \text{Nat}(x))$

at positions

[0]

of the preceding formulas leads to

```
!x = 0 | Any x1:(Nat(x1) & !x = suc(x1)) ==>  
!x = 0 | Any x:(odd(x) & !x = suc(x)) | Any x0:(even(x0) & !x = suc(x0))
```

## 6. Applying the induction hypothesis

$\text{Nat}(x) \implies (\exists x >> x \implies \text{even}(x) \mid \text{odd}(x))$

at positions

[0,1,0,0]

of the preceding formulas leads to

```
!x = 0 | Any x1:((!x >> x1 ==> even(x1) | odd(x1)) & !x = suc(x1)) ==>  
!x = 0 | Any x:(odd(x) & !x = suc(x)) | Any x0:(even(x0) & !x = suc(x0))
```

## 7. NARROWING at positions

[0,1,0,0,0]

of the preceding formulas (one step) leads to

```
!x = 0 |  
Any x1:((!x = suc(x1) | !x > x1 ==> even(x1) | odd(x1)) & !x = suc(x1)) ==>  
!x = 0 | Any x:(odd(x) & !x = suc(x)) | Any x0:(even(x0) & !x = suc(x0))
```

## 8. SIMPLIFYING the preceding formulas (19 steps) leads to

True

## Lists

```
-- LIST

specs:      nat
preds:      P any zipAny sorted part NOTsorted >>
copreds:    all zipAll ~
defuncts:   F bag map foldl flatten ext scan zip zipWith
            evens odds mergesort split merge isort insert splitAt
fovarks:    x y z xs ys s s' s1 s2 z1 z2 p
hovarks:    F P

axioms:

x:s >> s
& (s >> s' <==> s >> s1 & s1 >> s')

& bag(x:s) = x^bag(s)
& bag(s++s') = bag(s)^bag(s')

& map(F) [] = []
& map(F)(x:s) = F(x):map(F)(s)

& foldl(F)(x) [] = x
```

```

& foldl(F)(x)(y:s) = foldl(F)(F(x,y))(s)
--& sum(s) = foldl(+)(0)(s)
--& product(s) = foldl(*)(1)(s)

& flatten[] == []
& flatten(s:p) == s++flatten(p)

& ext(F)(s) = flatten(map(F)(s))

& scan(F)(x)[] = [x]
& scan(F)(x)(y:s) = x:scan(F)(F(x,y))(s)

& zip[] [] = []
& zip(x:s)(y:s') = (x,y):zip(s)(s')
& zipWith(F)[] [] = []
& zipWith(F)(x:s)(y:s') = F(x,y):zipWith(F)(s)(s')

& (any(P)(x:s) <==> P(x) | any(P)(s))
& (all(P)(x:s) ==> P(x) & all(P)(s))
& (zipAny(P)(x:s)(y:s') <==> P(x,y) | zipAny(P)(s)(s'))
& (zipAll(P)(x:s)(y:s') ==> P(x,y) & zipAll(P)(s)(s'))

& part([x],[[x]])
& (part(x:y:s,[x]:p) <==> part(y:s,p))
& (part(x:y:s,(x:s'):p) <==> part(y:s,s':p))

```

```

& evens[] = []
& evens(x:s) = x:odds(s)
& odds[] = []
& odds(x:s) = evens(s)

& mergesort[] = []
& mergesort[x] = [x]
& (mergesort(x:y:s) = merge(mergesort(x:s1),mergesort(y:s2)))
  <==== split(s) = (s1,s2)

& split[] = ([] ,[])
& split[x] = ([x] ,[])
& (split(x:(y:s)) = (x:s1,y:s2) <==== split(s) = (s1,s2))

& (merge(x:s,y:s') = x:merge(s,y:s') <==== x <= y)
& (merge(x:s,y:s') = y:merge(x:s,s') <==== x > y)
& merge([],s) = s
& merge(s,[]) = s

& isort[] = []
& isort[x] = [x]
& isort(x:s) = insert(x,isort(s))

& insert(x,[]) = [x]

```

```

& (insert(x,y:s) = x:y:s <==> x <= y)
& (insert(x,y:s) = y:insert(x,s) <==> x > y)

& sorted([])
& sorted([x])
& (sorted(x:y:s) <==> x <= y & sorted(y:s))

& (s ~ s' ==> bag(s) = bag(s'))

& splitAt(0)(s) = ([] ,s)
& splitAt(suc(n))[] = ([] ,[])
& (splitAt(suc(n))(x:s) = (x:s1,s2) <==> splitAt(n)(s) = (s1,s2))

& x:s>>s
& (s >> s' <==> s >> s1 & s1 >> s')

```

theorems:

```

NOTsorted(s) <==> Not(sorted(s))
& (sorted(s) & sorted(s') ==> sorted(merge(s,s')))
& (sorted(s) ==> sorted(insert(x,s)))
& (split(s) = (s1,s2) ==> s ~ s1++s2)
& (s ~ merge(s1,s2) <==> s ~ s1++s2)
& (s ~ insert(x,s') <==> s ~ x:s')
& (sorted(x:s) ==> sorted(s))

```

```

& (sorted(x:s) & sorted(y:s') & x <= y & sorted(s1) & s1~(s++y:s')
      ===> sorted(x:s1))
& (x > y ===> y <= x)
& y:x:s++s' ~ x:s++y:s'
& s'++x:s ~ x:s++s'

```

conjects:

```

(part(s,p) ==> flatten(p) = s)                                & -- partflatten
(mergesort(s) = s' ==> sorted(s'))                               &
(mergesort(s) = s' ==> s ~ s')                                 &
(isort(s) = s' ==> sorted(s'))                                 &
(isort(s) = s' ==> s ~ s')                                 &
(merge(s1,s2) = s & sorted(s1) & sorted(s2)
   ==> sorted(s) & s ~ s1++s2)                                &
(map(F)(s) = s' ==> lg(s) = lg(s'))                           &
zip(evens(s),odds(s)) = s                                     &
-- prem subsumes conc:
All x s z:
  (sorted(x:s) & All s': (NOTsorted(s') | x:s = s')
   ==> NOTsorted(z++[x]) | x:s = z++[x])                  &
splitAt(3)[5..12] = s

```

terms:

```
flatten([[1,2,3],[5,6,7,8],[1,2,3]])      <+>
merge([1,3,5],[2,4,6,8])
```

*Proof of partflatten by fixpoint induction:*

0. Derivation of

```
(part(s,p)==>(flatten(p)=s))
```

All simplifications are admitted.

Equation removal is safe.

1. Adding

```
(part0(s,p) ==> flatten(p) = s)
```

to the axioms and applying FIXPOINT INDUCTION wrt

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

at positions

```
[]
```

of the preceding formulas leads to

(All x p y s s':((flatten([[x]])=[x]))&All x p y s s':(((flatten(([x]:p))=(x:(y:s)))<=

2. SIMPLIFYING the preceding formulas (12 steps) leads to

(All p y s:((part0((y:s),p)==>(flatten(p)=(y:s))))&All p y s s':((part0((y:s),(s':p))==

3. Applying the theorem

(part0(s,p) ==> flatten(p) = s)

at positions

[1,0,0]

[0,0,0]

of the preceding formulas leads to

(All p y s:(((flatten(p)=(y:s))==>(flatten(p)=(y:s))))&All p y s s':(((flatten((s':p))=

The axioms have been MATCHED against their redices.

4. SIMPLIFYING the preceding formulas (3 steps) leads to

True

*Proof of partflatten by Noetherian induction:*

0. Derivation of

```
part(s,p) ==> flatten(p) = s
```

All simplifications are admitted.

Equation removal is safe.

1. SELECTING INDUCTION VARIABLES at positions

[0,0]

of the preceding formulas leads to

```
All p:(part(!s,p) ==> flatten(p) = !s)
```

2. NARROWING the preceding formulas (one step) leads to

```
All p:(Any x:(!s = [x] & p = [[x]])) |  
  Any x y s p0:  
    (part(y:s,p0) & !s = (x:(y:s)) & p = ([x]:p0)) |  
  Any x y s s' p0:  
    (part(y:s,s':p0) & !s = (x:(y:s)) & p = ((x:s'):p0)) ==>  
      flatten(p) = !s)
```

3. SIMPLIFYING the preceding formulas (28 steps) leads to

```
All x y s p0:  
  (!s = (x:(y:s)) & part(y:s,p0) ==> flatten(p0) = (y:s)) &
```

All  $x\ y\ s\ s'\ p0$ :

$(!s = (x:(y:s)) \ \& \ part(y:s, s':p0) ==> (s'++flatten(p0)) = (y:s))$

#### 4. Applying the induction hypothesis

$part(s,p) ==> (!s >> s ==> flatten(p) = s)$

at positions

[1,0,0,1]

[0,0,0,1]

of the preceding formulas leads to

All  $x\ y\ s\ p0$ :

$(!s = (x:(y:s)) \ \& \ (!s >> (y:s) ==> flatten(p0) = (y:s)) ==>$   
 $flatten(p0) = (y:s)) \ \&$

All  $x\ y\ s\ s'\ p0$ :

$(!s = (x:(y:s)) \ \& \ (!s >> (y:s) ==> flatten(s':p0) = (y:s)) ==>$   
 $(s'++flatten(p0)) = (y:s))$

#### 5. Applying the theorem

$(x:s) >> s$

at positions

[1,0,0,1,0]

[0,0,0,1,0]

of the preceding formulas leads to

All x y s p0:

(!s = (x:(y:s)) & (Any x0:(!s = (x0:(y:s)))) ==> flatten(p0) = (y:s)) ==>  
flatten(p0) = (y:s)) &

All x y s s' p0:

(!s = (x:(y:s)) & (Any x1:(!s = (x1:(y:s)))) ==> flatten(s':p0) = (y:s)) ==>  
(s'++flatten(p0)) = (y:s))

6. SIMPLIFYING the preceding formulas (15 steps) leads to

True

Video for both proofs: <http://fdit-www.cs.tu-dortmund.de/~peter/Expander2/PAFL.mp4>

## Streams

```
-- stream

defuncts: zeros ones blink blink' evens odds map zipWith zip bzip
          fact facts iter1 iter2 fib fib0 fib1 addTail nats loop
          inv switch morse morse1 morse2 morsef
preds:    Nat X P true false not \/ /\ `then` `leadsto` `ble`
          Fmu Gnu Fpath GF FG `U` `Umu` `Wnu` `Rnu` eqstream
copreds:  NatStream Gpath `le` `W` `Rpath` ~
fovars:   n z z' s s'
hovars:   X

axioms:

  head(zeros) == 0
& tail(zeros) == zeros

  & head(ones) == 1
& tail(ones) == ones

  & head(blink) == 0
& tail(blink) == 1:blink
```

```

-- & (blink = 1:blink <==> False)           -- used in fairblink2
                                              -- and notfairblink2
& blink' == mu s.(0:1:s)

& head(evens(s)) == head(s)
& tail(evens(s)) == evens(tail(tail(s)))
& odds == evens.tail

-- & tail(evens(s)) == odds(tail(s))
-- & head(odds(s)) == head(tail(s))
-- & tail(odds(s)) == odds(tail(tail(s)))

& head(map(F)(s)) == F(head(s))
& tail(map(F)(s)) == map(F)(tail(s))

& head(zipWith(F)(s)(s')) == F(head(s),head(s'))
& tail(zipWith(F)(s)(s')) == zipWith(F)(tail(s),tail(s'))

& head(zip(s,s')) == head(s)
& tail(zip(s,s')) == zip(s',tail(s))

& head(bzip(s,s',0)) == head(s)
& head(bzip(s,s',1)) == head(s')
& tail(bzip(s,s',0)) == bzip(tail(s),s',1)
& tail(bzip(s,s',1)) == bzip(s,tail(s'),0)

```

```

& fact(0) == 1
& fact(suc(n)) == n*fact(n)

& head(facts(n)) == fact(n)
& tail(facts(n)) == facts(suc(n))

& head(iter1(f,x)) == x
& tail(iter1(f,x)) == iter1(f,f(x))

& head(iter2(f,x)) == x
& tail(iter2(f,x)) == map(f)(iter2(f,x))

& nats(n) == iter1(suc,n)
& nats'(n) == iter2(suc,n)

& loop(f)(0)(x) == x
& loop(f)(suc(n))(x) == f(loop(f)(n)(x))

& head$inv$s == switch$head$s
& tail$inv$s == inv$tail$s

& switch(0) == 1
& switch(1) == 0
& switch$switch$x == x                                -- used by invinv

```

```
& head$morse == 0
& head$tail$morse == 1
& tail$tail$morse == zip(tail$morse,inv$tail$morse)

& head$morsef$s == head$s
& head$tail$morsef$s == switch$head$s
& tail$tail$morsef$s == morsef$tail$s

& head$morse1$s == head$s
& tail$morse1$s == morse2$s
& head$morse2$s == switch$head$s
& tail$morse2$s == morse1$tail$s

& fib(0) == 1
& fib(1) == 1
& fib(suc(suc(n))) == fib(n)+fib(n+1)

& head(fib0(n)) == fib(n)
& tail(fib0(n)) == fib0(n+1)

& head(fib1(n)) == fib(n)
& head(tail(fib1(n))) == fib(n+1)
& tail(tail(fib1(n))) == addTail(zip(fib1(n),tail(fib1(n))))
```

```

& head(addTail(s)) == head(s)+head(tail(s))
& tail(addTail(s)) == addTail(tail(tail(s)))

& head$s+s' == head(s)+head(s')
& tail$s+s' == tail(s)+tail(s')

-- state equivalence

& (s ~ s' ==> head$s = head$s' & tail$s ~ tail$s')

& (eqstream <==> NU X.rel((s,s'),head$s=head$s' & X(tail$s,tail$s')))

& (Fpath(P)$s <==> P$s | Fpath(P)$tail$s) -- finally
& (Gpath(P)$s ==> P$s & Gpath(P)$tail$s) -- generally
& ((P`U`Q)$s <==> Q$s | P$s & (P`U`Q)$tail$s) -- until
& ((P`W`Q)$s ==> Q$s | P$s & (P`W`Q)$tail$s) -- weak until
& ((P`Rpath`Q)$s ==> Q$s & (P$s | (P`Rpath`Q)$tail$s)) -- release

& (GF <==> Gpath.Fpath)
& (FG <==> Fpath.Gpath)

& (Fmu$P <==> MU X.(P\X.tail)) -- finally
& (Gnu$P <==> NU X.(P\X.tail)) -- generally
& (P`Umu`Q <==> MU X.(Q\/(P\X.tail))) -- until
& (P`Wnu`Q <==> NU X.(Q\/(P\X.tail))) -- weak until

```

```

& (P`Rnu`Q <==> NU X.(Q/\(P\/X.tail))) -- release
& (P`leadsto`Q <==> Gmu$P`then`Fmu$Q)

& ((P/\Q)$s <==> P$s & Q$s)
& ((P\Q)$s <==> P$s | Q$s)

& (Nat$x <==> Int(x) & x >= 0)

& (NatStream(x:s) ==> Nat(x) & NatStream$s)

& (s `le` s' ==> head$s `ble` head(s'))
& (s `le` s' ==> (head$s = head$s' ==> tail$s `le` tail$s'))

```

theorems:

iter1(f,loop(f)(n)(x)) = map(loop(f)(n))(iter2(f,x))	-- used in iter1iter1
& map(loop(f)(0))(s) = s	-- used in iter1iter1
& map(loop(f)(suc(n)))(s) = map(loop(f)(n))(map(f)(s))	-- used in iterloop
& loop(f)(n)(f(x)) = f(loop(f)(n)(x))	-- used in maploop -- see NAT (natloop)
& tail(fib1(n)) = fib1(n+1)	-- used in fib01
& fib((n+1)+1) = (fib(n)+fib(n+1))	-- used in fibtail
& evens(zip(s,s')) = s	-- used in evodmorse
& s ~ s	-- used in evodmorse
& zip(s,inv(s)) = morsef(s)	-- used in morsef

```

& (x `ble` z <==> x `ble` y & y `ble` z)
& (x `ble` y & y `ble` z & x = z ==> y = z & x = y)
& (x+z `ble` y+z' <==> x `ble` y & z `ble` z')

& (Fpath(Q)$s <==> (true`U`Q)$s)
& (Gpath(P)$s <==> (P`W`false)$s)
& ((P`U`Q)$s <==> (P`W`Q)$s & Fpath(Q)$s)
& ((P`W`Q)$s <==> (P`U`Q)$s | Gpath(P)$s)
& (not(Fpath$P)$s <==> (Gpath$not$P)$s)
& (not(Gpath$P)$s <==> (Fpath$not$P)$s)
& (not(P`Rpath`Q)$s <==> (not(P)`U`not(Q))$s)

```

conjects:

blink ~ zip(zeros,ones)	-- blinkzip
& evens\$zip(s,s') ~ s	-- evenszip
& odds\$zip(s,s') ~ s'	-- oddszip
& zip(evens\$s,odds\$s) ~ s	-- zipEvensOdds
& evens\$x:s ~ x:odds\$s	-- evensodds
& bzip(s,s',0) ~ zip(s,s')	-- bzip
& bzip(s,s',1) ~ zip(s',s)	
& inv\$inv\$s ~ s	-- invinv
& iter1(f,n) ~ iter2(f,n)	-- iter1iter2
& iter1(f,loop(f)(n)(x)) ~ map(loop(f)(n))(iter2(f,x))	-- iterloop

```

& map(loop(f)(0))(s) ~ s                                -- maploop0
& map(loop(f)(suc(n)))(s) ~ map(loop(f)(n))(map(f)(s)) -- maploop
& map(f)(iter1(f,x)) ~ iter1(f,f(x))                  -- mapiter1
& map(fact)(nats(n)) ~ facts(n)                        -- mapfact
& fib0(n) ~ fib1(n)                                    -- fib01
& fib1(n+1) ~ tail(fib1(n))                           -- fibtail
& tail(fib1(n+1)) ~ addTail(zip(fib1(n),tail(fib1(n)))) -- tailfib
& evens$morse ~ Morse & odds$tail$morse ~ tail$morse   -- evodmorse
& Morse$inv$s ~ inv$Morse$S                           -- morseinv
& Morse1$inv$s ~ inv$Morse1$S                         -- morseinv2
& Morse$Morse ~ Morse                                 -- Morsef
& zip(s,inv$s) ~ Morse$S                            -- zipmorse
& Morse ~ 0:zip(inv$Morse,tail$Morse)

& Fmu((=0).head)$1:2:3:blink                         --> True

& GF((=0).head)$blink                                --> True      -- fairblink
& Not(GF((=0).head)$blink)                           --> False     -- notfairblink
& GF((=2).head)$blink                                --> False     -- fairblink2
& Not(GF((=2).head)$blink)                           --> True      -- notfairblink2

& GF((=0).head)$mu s.(0:1:s)                         --> True      -- fairblinkmu
& NatStream$mu s.(1:2:3:s)                           --> True      -- natstream

& eqstream(blink,0:1:blink)

```

```

& (s = x & s' = zip(evens$x,odds$x) |
  s = tail$x & s' = zip(odds$x,evens$x) ==> eqstream(s,s')) 

& (x `le` y & y `le` z ==> x `le` z)                                -- kozen1
& (x `le` y & z `le` z' ==> x+z `le` y+z')                          -- kozen2

-- & GF((=!x).head)$blink      --> !x=0 | !x=1                      -- fairblinkx
-- & NatStream$1:2:3:!s          --> !s = 1:2:3:!s |                  -- natstreamSol
--   !s = 2:3:!s | !s = 3:!s

```

### *Proof of blinkzip by coinduction:*

#### 0. Derivation of

blink ~ zip(zero,one)

All simplifications are admitted.

Equation removal is safe.

#### 1. Adding

(~0(z0,z1) <==> z0 = blink & z1 = zip(zero,one))

to the axioms and applying COINDUCTION wrt

(s ~ s' ==> head(s) = head(s') & tail(s) ~ tail(s'))

at positions

[]

of the preceding formulas leads to

All  $s s' : (s = \text{blink} \& s' = \text{zip(zero,one)} \implies \text{head}(s) = \text{head}(s') \& \sim 0(\text{tail}(s), \text{tail}(s')))$

2. SIMPLIFYING the preceding formulas (12 steps) leads to

$\sim 0(1:\text{blink}, \text{zip(one,zero)})$

3. Adding

$(\sim 0(z2, z3) \iff z2 = (1:\text{blink}) \& z3 = \text{zip(one,zero)})$

to the axioms and applying COINDUCTION wrt

$(s \sim s' \implies \text{head}(s) = \text{head}(s') \& \text{tail}(s) \sim \text{tail}(s'))$

at positions

[]

of the preceding formulas leads to

All  $s s' : (s = (1:\text{blink}) \& s' = \text{zip(one,zero)} \implies$

```
head(s) = head(s') & ~0(tail(s),tail(s')))
```

4. SIMPLIFYING the preceding formulas (12 steps) leads to

```
~0(blink,zip(zero,one))
```

5. NARROWING the preceding formulas (one step) leads to

```
blink = blink & zip(zero,one) = zip(zero,one) |
blink = (1:blink) & zip(zero,one) = zip(one,zero)
```

The axioms have been MATCHED against their redices.

6. SIMPLIFYING the preceding formulas (one step) leads to

True

*Proof of bzip by coinduction:*

0. Derivation of

```
bzip(s,s',0) ~ zip(s,s')
```

All simplifications are admitted.

Equation removal is safe.

## 1. Adding

$(\sim_0(z_0, z_1) \iff z_0 = \text{bzip}(s, s', 0) \ \& \ z_1 = \text{zip}(s, s'))$

to the axioms and applying COINDUCTION wrt

$(s \sim s' \implies \text{head}(s) = \text{head}(s') \ \& \ \text{tail}(s) \sim \text{tail}(s'))$

at positions

[]

of the preceding formulas leads to

All  $s \ s' : (\text{Any } s_0 \ s'_0 :$

$(s = \text{bzip}(s_0, s'_0, 0) \ \& \ s' = \text{zip}(s_0, s'_0)) \implies$   
 $\text{head}(s) = \text{head}(s') \ \& \ \sim_0(\text{tail}(s), \text{tail}(s')))$

THIS GOAL COINCIDES WITH GOAL NO. 1

2. SIMPLIFYING the preceding formulas (12 steps) leads to

All  $s_0 \ s'_0 :$

$\sim_0(\text{bzip}(\text{tail}(s_0), s'_0, 1), \text{zip}(s'_0, \text{tail}(s_0)))$

3. Adding

$(\sim_0(z_2, z_3) \iff z_2 = \text{bzip}(\text{tail}(s_0), s'_0, 1) \ \& \ z_3 = \text{zip}(s'_0, \text{tail}(s_0)))$

to the axioms and applying COINDUCTION wrt

$$(s \sim s' \implies \text{head}(s) = \text{head}(s') \ \& \ \text{tail}(s) \sim \text{tail}(s'))$$

at positions

[0]

of the preceding formulas leads to

All  $s_0 s'_0$ :

All  $s s'$ : (Any  $s_0 s'_0$ :

$$(s = \text{bzip}(\text{tail}(s_0), s'_0, 1) \ \& \ s' = \text{zip}(s'_0, \text{tail}(s_0))) \implies \\ \text{head}(s) = \text{head}(s') \ \& \ \sim^0(\text{tail}(s), \text{tail}(s')))$$

4. SIMPLIFYING the preceding formulas (12 steps) leads to

All  $s_0 s'_0$ :

$$\sim^0(\text{bzip}(\text{tail}(s_0), \text{tail}(s'_0), 0), \text{zip}(\text{tail}(s_0), \text{tail}(s'_0)))$$

5. Applying the theorem

$$(\sim^0(z_0, z_1) \iff z_0 = \text{bzip}(s, s', 0) \ \& \ z_1 = \text{zip}(s, s'))$$

at positions

[0]

of the preceding formulas leads to

All  $s0\ s'0$ :

Any  $s\ s' : (bzip(tail(s0), tail(s'0), 0) = bzip(s, s', 0) \ \&\ zip(tail(s0), tail(s'0)) = zip(s, s'))$

6. SUBSTITUTING  $tail(s'0)$  FOR  $s'$  to the preceding formulas leads to

All  $s0\ s'0$ :

Any  $s\ s' : (bzip(tail(s0), tail(s'0), 0) = bzip(tail(s0), tail(s'0), 0) \ \&\ zip(tail(s0), tail(s'0)) = zip(tail(s0), tail(s'0)))$

7. SIMPLIFYING the preceding formulas (one step) leads to

True

*Proof of fairblink by coinduction:*

0. Derivation of

$(G(F(((=0).head)))\$blink)$

All simplifications are admitted.

Equation removal is safe.

1. Adding

$(G_0(z_0) \& z_1 <==> z_0 = F((=0).head) \& z_1 = \text{blink})$

to the axioms and applying COINDUCTION wrt

$(G(P)s ==> P(s) \& G(P)\$tail(s))$

at positions

[]

of the preceding formulas leads to

All  $P\ s: (((P=F((=0).head)) \& (s=\text{blink})) ==> (P(s) \& (G_0(P)\$tail(s))))$

2. SIMPLIFYING the preceding formulas (6 steps) leads to

$((F((=0).head)\$blink) \& (G_0(F((=0).head))\$1:\text{blink}))$

3. Applying the theorem

$(F(P)s <==> P(s) \mid F(P)\$tail(s))$

at positions

[0]

of the preceding formulas leads to

((((=0).head)\$blink) | (F(((=0).head))\$tail(blink)))&(G0(F(((=0).head)))\$(1:blink)))

The axioms have been MATCHED against their redices.

4. SIMPLIFYING the preceding formulas (6 steps) leads to

(G0(F(((=0).head)))\$(1:blink))

5. Adding

(G0(z2)\$z3 <== z2 = F((=0).head) & z3 = (1:blink))

to the axioms and applying COINDUCTION wrt

(G(P)\$s ==> P(s) & G(P)\$tail(s))

at positions

[]

of the preceding formulas leads to

All P s:(((P=F(((=0).head)))&(s=(1:blink)))==>(P(s)&(G0(P)\$tail(s))))

6. SIMPLIFYING the preceding formulas (6 steps) leads to

((F(((=0).head))\$(1:blink))&(G0(F(((=0).head)))\$blink))

## 7. Applying the axioms

```
(G0(z0)$z1 <== z0 = F((=0).head) & z1 = blink)
& (G0(z2)$z3 <== z2 = F((=0).head) & z3 = (1:blink))
```

at positions

[1]

of the preceding formulas leads to

```
((F(((=0).head))$(1:blink))&(((F(((=0).head))=F(((=0).head)))&(blink=blink)) |
 ((F(((=0).head))=F(((=0).head)))&(blink=(1:blink))))
```

The axioms have been MATCHED against their redices.

## 8. SIMPLIFYING the preceding formulas (one step) leads to

```
(F(((=0).head))$(1:blink))
```

THIS GOAL COINCIDES WITH GOAL NO. 8

## 9. NARROWING the preceding formulas (one step) leads to

```
((((=0).head))$(1:blink)) | (F(((=0).head))$tail((1:blink))))
```

The axioms have been MATCHED against their redices.

10. SIMPLIFYING the preceding formulas (5 steps) leads to

$(F(((=0).head))\$blink)$

11. NARROWING the preceding formulas (one step) leads to

$((((=0).head)\$blink) | (F(((=0).head))\$tail(blink)))$

The axioms have been MATCHED against their redices.

12. SIMPLIFYING the preceding formulas (4 steps) leads to

True

*Proof of fairblinkS by simplification:*

0. Derivation of

$s = \text{blink} \mid s = (1:\text{blink}) \implies GS(FS((=0).head))\$s$

All simplifications are admitted.

Equation removal is safe.

1. SIMPLIFYING the preceding formulas (one step) leads to

```
s = blink | s = (1:blink) ==> NU X.(FS((=0).head)/\ (X.tail))$s
```

2. SIMPLIFYING the preceding formulas (one step) leads to

```
All s:(s = blink | s = (1:blink) ==>  
      (FS((=0).head)/\ (rel(s,(s = blink | s = (1:blink))).tail))$s)
```

3. SIMPLIFYING the preceding formulas (one step) leads to

```
All s:((s = blink ==>  
      (FS((=0).head)/\ (rel(s,(s = blink | s = (1:blink))).tail))$s) &  
      (s = (1:blink) ==>  
      (FS((=0).head)/\ (rel(s,(s = blink | s = (1:blink))).tail))$s))
```

4. SIMPLIFYING the preceding formulas (one step) leads to

```
All s:(s = blink ==> (FS((=0).head)/\ (rel(s,(s = blink | s = (1:blink))).tail))$s) &  
All s:(s = (1:blink) ==>  
      (FS((=0).head)/\ (rel(s,(s = blink | s = (1:blink))).tail))$s)
```

5. SIMPLIFYING the preceding formulas (one step) leads to

```
All s:((FS((=0).head)/\ (rel(s,(s = blink | s = (1:blink))).tail))$blink) &  
All s:(s = (1:blink) ==>  
      (FS((=0).head)/\ (rel(s,(s = blink | s = (1:blink))).tail))$s)
```

6. SIMPLIFYING the preceding formulas (one step) leads to

```
(FS((=0).head) \ (rel(s, (s = blink | s = (1:blink))).tail))$blink &
All s:(s = (1:blink) ==>
    (FS((=0).head) \ (rel(s, (s = blink | s = (1:blink))).tail))$s)
```

7. SIMPLIFYING the preceding formulas (one step) leads to

```
(FS((=0).head)$blink & (rel(s, (s = blink | s = (1:blink))).tail)$blink) &
All s:(s = (1:blink) ==>
    (FS((=0).head) \ (rel(s, (s = blink | s = (1:blink))).tail))$s)
```

8. SIMPLIFYING the preceding formulas (11 steps) leads to

```
(1:blink) = blink & MU X.(((=0).head) \ /(X.tail))$(1:blink) &
(rel(s, (s = blink | s = (1:blink))).tail)$(1:blink) &
MU X.(((=0).head) \ /(X.tail))$blink |
tail(blink) = (1:blink) &
All s:(s = (1:blink) ==>
    (FS((=0).head) \ (rel(s, (s = blink | s = (1:blink))).tail))$s) &
MU X.(((=0).head) \ /(X.tail))$blink
```

9. SIMPLIFYING the preceding formulas (11 steps) leads to

```
(FS((=0).head) /\ (rel(s, (s = blink | s = (1:blink))).tail))$(1:blink) &  
MU X.(((=0).head) \/(X.tail))$blink |  
MU X.(((=0).head) \/(X.tail))$blink & (1:blink) = blink &  
MU X.(((=0).head) \/(X.tail))$(1:blink)
```

10. SIMPLIFYING the preceding formulas (11 steps) leads to

```
MU X.(((=0).head) \/(X.tail))$blink & MU X.(((=0).head) \/(X.tail))$(1:blink)
```

11. SIMPLIFYING the preceding formulas (11 steps) leads to

```
(=0)$head(1:blink) | (MU X.(((=0).head) \/(X.tail)).tail)$(1:blink)
```

12. SIMPLIFYING the preceding formulas (11 steps) leads to

True

## Binary trees

```
-- infbintree

defuncts:  left root right mirror
fovars:    t t'

axioms:

  left(mirror(t)) == right(t)
& root(mirror(t)) == root(t)
& right(mirror(t)) == left(t)

& (t ~ t' ==> left(t) ~ left(t') & root(t) = root(t') & right(t) ~ right(t'))

conjects:  mirror(mirror(t)) ~ t
```

*Beweis:*

Adding

```
(x ~0 y <==> x = y)
& (x ~0 y <==> y ~0 x)
& (x ~0 z <==> x ~0 y & y ~0 z)
```

```
& (z0 ~0 z1 <==> z0 = mirror(mirror(t)) & z1 = t)
& (x ~/~0 y ==> x /= y)
& (x ~/~0 y ==> Not(y ~0 x))
& (x ~/~0 z ==> Not(x ~0 y) | Not(y ~0 z))
& (z0 ~/~0 z1 ==> z0 /= mirror(mirror(t)) | z1 /= t)
```

to the axioms and applying coinduction w.r.t.

```
(t ~ t' ==> left(t) ~ left(t') & root(t) = root(t') & right(t) ~ right(t'))
```

at position [] of the preceding formula leads to

```
All t0:(left(t0) ~0 left(t0)) & All t0:(right(t0) ~0 right(t0))
```

The reducts have been simplified.

Narrowing the preceding formula (1 step) leads to

```
All t0:(right(t0) ~0 right(t0))
```

The axioms were MATCHED against their redices.

The reducts have been simplified.

Narrowing the preceding formula (1 step) leads to

True

The axioms were MATCHED against their redices.

The reducts have been simplified.

Number of proof steps: 3

-- cobintree

constructs: undef

defuncts: lrr mt subtree mirror

preds: finite exists P

copreds: infinite forall ~pr

fovarts: t u t' u'

hovars: P

axioms:

(lrr(mirror(t)) = undef <==> lrr(t) = undef)

& (lrr(mirror(t)) = (mirror(u'),x,mirror(u)) <==> lrr(t) = (u,x,u'))

& (t ~ t' ==> lrr(t) ~pr lrr(t'))

& ((t,x,u) ~pr (t',y,u') ==> t ~ t' & x = y & u ~ u')

& (undef ~pr (t,x,u) ==> False)

& ((t,x,u) ~pr undef ==> False)

```

& lrr(mt) = undef

& subtree(t,[]) = t
& (subtree(t,x:s) = undef <==> lrr(t) = undef)
& (subtree(t,0:s) = subtree(u,s) <==> lrr(t) = (u,x,u'))
& (subtree(t,1:s) = subtree(u',s) <==> lrr(t) = (u,x,u'))

& (finite(t) <==> lrr(t) = undef | lrr(t) = (u,x,u') & finite(u) & finite(u'))
& (infinite(t) ==> Any u x u':(lrr(T) = (u,x,u') & (infinite(u) | infinite(u'))))

& (exists(P)(t) <==> lrr(t) = (u,x,u') & (P(x) | exists(P)(u) | exists(P)(u')))
& (forall(P)(t) ==> (lrr(t) = (u,x,u') ==> forall(P)(u) & P(x) & forall(P)(u')))

conjects: mirror(mirror(t)) ~ t

```

*Beweis:*

Adding

```

(x ~0 y <==> x = y)
& (x ~0 y <==> y ~0 x)
& (x ~0 z <==> x ~0 y & y ~0 z)
& (z0 ~0 z1 <==> z0 = mirror(mirror(t)) & z1 = t)
& (x ~/~0 y ==> x /= y)
& (x ~/~0 y ==> Not(y ~0 x))

```

```

& (x ~/~/0 z ==> Not(x ~0 y) | Not(y ~0 z))
& (z0 ~/~/0 z1 ==> z0 =/= mirror(mirror(t)) | z1 =/= t)

```

to the axioms and applying coinduction w.r.t.

$$(t \sim t' ==> lrr(t) \sim_{pr} lrr(t'))$$

at position [] of the preceding formula leads to

$$\text{All } t \ t': (\text{Any } t_0: (t = \text{mirror}(\text{mirror}(t_0)) \ \& \ t' = t_0) ==> lrr(t) \sim_{pr} lrr(t'))$$

Simplifying the preceding formula (7 steps) leads to

$$\text{All } t_0: (lrr(\text{mirror}(\text{mirror}(t_0))) \sim_{pr} lrr(t_0))$$

Adding

```

(x ~pr0 y <==> x = y)
& (x ~pr0 y <==> y ~pr0 x)
& (x ~pr0 z <==> x ~pr0 y & y ~pr0 z)
& (z2 ~pr0 z3 <==> z2 = lrr(mirror(mirror(t0))) & z3 = lrr(t0))
& (x ~/~pr0 y ==> x =/= y)
& (x ~/~pr0 y ==> Not(y ~pr0 x))
& (x ~/~pr0 z ==> Not(x ~pr0 y) | Not(y ~pr0 z))
& (z2 ~/~pr0 z3 ==> z2 =/= lrr(mirror(mirror(t0))) | z3 =/= lrr(t0))

```

to the axioms and applying coinduction w.r.t.

```
((t,x,u) ~pr (t',y,u') ==> t ~ t' & x = y & u ~ u')  
& (undef ~pr (t,x,u) ==> False)  
& ((t,x,u) ~pr undef ==> False)
```

at position [0] of the preceding formula leads to

```
All t0:(All t x u t' y u':(Any t0:((t,x,u) = lrr(mirror(mirror(t0))) &  
                                (t',y,u') = lrr(t0)) ==>  
                                t ~0 t' & x = y & u ~0 u') &  
All t x u t' y u':(Any t0:(undef = lrr(mirror(mirror(t0))) &  
                                (t,x,u) = lrr(t0)) ==>  
                                False) &  
All t x u t' y u':(Any t0:((t,x,u) = lrr(mirror(mirror(t0))) &  
                                undef = lrr(t0)) ==>  
                                False))
```

Simplifying the preceding formula (1 step) leads to

```
All t x u t' y u':(Any t0:((t,x,u) = lrr(mirror(mirror(t0))) &  
                                (t',y,u') = lrr(t0)) ==>  
                                t ~0 t' & x = y & u ~0 u') &  
All t x u t0:(Not(undef = lrr(mirror(mirror(t0)))) | Not((t,x,u) = lrr(t0))) &
```

All t x u t0:(Not((t,x,u) = lrr(mirror(mirror(t0)))) | Not(undef = lrr(t0)))

The tree has been split.

The current formula is given by

All t x u t' y u':(Any t0:((t,x,u) = lrr(mirror(mirror(t0))) &  
(t',y,u') = lrr(t0)) ==>  
t ~0 t' & x = y & u ~0 u')

Applying the axioms

(lrr(mirror(t)) = undef <== lrr(t) = undef)  
& (lrr(mirror(t)) = (mirror(u'),x,mirror(u)) <== lrr(t) = (u,x,u'))

at position [0,0,0,0,1] of the preceding factors leads to the factor

All t x u t' y u':(Any t0:(((t,x,u) = undef & lrr(mirror(t0)) = undef |  
Any u'0 x0 u0:((t,x,u) = (mirror(u'0),x0,mirror(u0)) &  
lrr(mirror(t0)) = (u0,x0,u'0))) &  
(t',y,u') = lrr(t0)) ==>  
t ~0 t' & x = y & u ~0 u')

The axioms were MATCHED against their redices.

Simplifying at position [0,0,0,0]

of the preceding factors (5 steps) leads to the factor

All t x u t' y u':(Any t0:(Any u'0 x0 u0:(u = mirror(u0) &  
lrr(mirror(t0)) = (u0,x,u'0) &  
t = mirror(u'0)) &  
(t',y,u') = lrr(t0)) ==>  
t ~0 t' & x = y & u ~0 u')

Applying the axioms

(lrr(mirror(t)) = undef <==> lrr(t) = undef)  
& (lrr(mirror(t)) = (mirror(u'),x,mirror(u)) <==> lrr(t) = (u,x,u'))

at position [0,0,0,0,0,1,0] of the preceding factors leads to the factor

All t x u t' y u':(Any t0:(Any u'0 x0 u0:(u = mirror(u0) &  
(undef = (u0,x,u'0) & lrr(t0) = undef |  
Any u'1 x1 u1:((mirror(u'1),x1,  
mirror(u1)) = (u0,x,  
u'0) &  
lrr(t0) = (u1,x1,u'1))) &  
t = mirror(u'0)) &  
(t',y,u') = lrr(t0)) ==>  
t ~0 t' & x = y & u ~0 u')

The axioms were MATCHED against their redices.

Simplifying at position [0,0,0,0,0,1]  
of the preceding factors (5 steps) leads to the factor

All t x u t' y u':(Any t0:(Any u'0 x0 u0:(u = mirror(u0) &  
                          Any u'1 x1 u1:(mirror(u1) = u'0 &  
                          lrr(t0) = (u1,x,u'1) &  
                          mirror(u'1) = u0) &  
                          t = mirror(u'0)) &  
                          (t',y,u') = lrr(t0)) ==>  
                          t ~0 t' & x = y & u ~0 u')

Simplifying the preceding factors (29 steps) leads to the factor

All x t0 u'1 u1:(lrr(t0) = (u1,x,u'1) ==> mirror(mirror(u'1)) ~0 u'1)

Applying the axioms

(x ~0 y <==> x = y)  
& (x ~0 y <==> y ~0 x)  
& (x ~0 z <==> x ~0 y & y ~0 z)  
& (z0 ~0 z1 <==> z0 = mirror(mirror(z1)) & z1 = t)

at position [0,1] of the preceding factors leads to the factor

```
All x t0 u'1 u1:(lrr(t0) = (u1,x,u'1) ==>
    mirror(mirror(u'1)) = u'1 | u'1 ~0 mirror(mirror(u'1)) |
    Any y:(mirror(mirror(u'1)) ~0 y & y ~0 u'1) |
    Any t:(mirror(mirror(u'1)) = mirror(mirror(t)) & u'1 = t))
```

The axioms were MATCHED against their redices.

Substituting u'1 for t to the preceding factors leads to the factor

```
All x t0 u'1 u1:(lrr(t0) = (u1,x,u'1) ==>
    mirror(mirror(u'1)) = u'1 | u'1 ~0 mirror(mirror(u'1)) |
    Any y:(mirror(mirror(u'1)) ~0 y & y ~0 u'1) |
    Any t u'1:(mirror(mirror(u'1)) = mirror(mirror(u'1)) &
        u'1 = u'1))
```

Simplifying the preceding factors (1 step) leads to 2 factors.

The current factor is given by

```
All t x u t0:(Not(undefined = lrr(mirror(mirror(t0)))) | Not((t,x,u) = lrr(t0)))
```

Applying the axioms

$$\begin{aligned} & (\text{lrr}(\text{mirror}(t)) = \text{undefined} \iff \text{lrr}(t) = \text{undefined}) \\ & \& (\text{lrr}(\text{mirror}(t)) = (\text{mirror}(u'), x, \text{mirror}(u)) \iff \text{lrr}(t) = (u, x, u')) \end{aligned}$$

at position [0,0,0,1] of the preceding factors leads to the factor

```
All t x u t0:(Not(undefined = undefined & lrr(mirror(t0)) = undefined |  
          Any u' x3 u2:(undefined = (mirror(u'),x3,mirror(u2)) &  
                      lrr(mirror(t0)) = (u2,x3,u')))) |  
          Not((t,x,u) = lrr(t0)))
```

The axioms were MATCHED against their redices.

Simplifying the preceding factors (4 steps) leads to the factor

```
All t x u t0:(lrr(mirror(t0)) =/= undefined | (t,x,u) =/= lrr(t0))
```

Applying the axioms

```
(lrr(mirror(t)) = undefined <==> lrr(t) = undefined)  
& (lrr(mirror(t)) = (mirror(u'),x,mirror(u)) <==> lrr(t) = (u,x,u'))
```

at position [0,0,0] of the preceding factors leads to the factor

```
All t x u t0:((undefined =/= undefined & lrr(t0) = undefined |  
          Any u' x4 u3:((mirror(u'),x4,mirror(u3)) =/= undefined &  
                      lrr(t0) = (u3,x4,u')))) |  
          (t,x,u) =/= lrr(t0))
```

The axioms were MATCHED against their redices.

Simplifying the preceding factors (11 steps) leads to a single formula, which is given by

All t x u t0:(Not((t,x,u) = lrr(mirror(mirror(t0)))) | Not(undefined = lrr(t0)))

Applying the axioms

(lrr(mirror(t)) = undefined <==> lrr(t) = undefined)  
& (lrr(mirror(t)) = (mirror(u'),x,mirror(u)) <==> lrr(t) = (u,x,u'))

at position [0,0,0,1] of the preceding formula leads to

All t x u t0:(Not((t,x,u) = undefined & lrr(mirror(t0)) = undefined |  
Any u' x5 u4:((t,x,u) = (mirror(u'),x5,mirror(u4)) &  
lrr(mirror(t0)) = (u4,x5,u'))) |  
Not(undefined = lrr(t0)))

The axioms were MATCHED against their redices.

Simplifying the preceding formula (31 steps) leads to

All t0 u4 x5 u':(undefined /= lrr(t0) | lrr(mirror(t0)) /= (u4,x5,u'))

Applying the axioms

```
(lrr(mirror(t)) = undef <==> lrr(t) = undef)
& (lrr(mirror(t)) = (mirror(u'),x,mirror(u)) <==> lrr(t) = (u,x,u'))
```

at position [0,1,0] of the preceding formula leads to

```
All t0 u4 x5 u':(undef /= lrr(t0) |
    (undef /= (u4,x5,u') & lrr(t0) = undef |
        Any u'2 x u:((mirror(u'2),x,mirror(u)) /= (u4,x5,u') &
            lrr(t0) = (u,x,u'2))))
```

The axioms were MATCHED against their redices.

Simplifying the preceding formula (5 steps) leads to

True

Number of proof steps: 21

## Lazy evaluation

```
-- lazy

specs:      bool
constructs: leaf #
defuncts:   is_even mergesortL splitL mergeL mergesort split merge
            mergesortF splitF mergeF replaceL replace foldRepL foldRep
            pall palF reveqL reveq reveqI reveqF sortTL sortT sortTI
            leavesRepL leavesRep leavesRepI foo goo nats nats' fibs fibs'
            RequestsL ClientL Requests Client CSI
            || tree1 tree2 tree3
preds:      even odd
fovars:     t u r a b requests acc t1 t2 u1 u2 s1 s2 ls ls1 ls2
hovars:    f g client

axioms:

(even$0 <==> True) &
(even$suc$x <==> odd$x) &
(odd$x <==> Not(even$x)) &

is_even == fun(x||even$x,true,x||odd$x,false) &
```

```

-- MERGESORT

(mergesortL(x:y:s) = mergeL(mergesortL$x:s1)$mergesortL$y:s2
<==== splitL$s = (s1,s2)) &
mergesortL[x] = [x] &
mergesortL[] = [] &

(splitL$x:y:s = (x:s1,y:s2) <==== splitL$s = (s1,s2)) &
splitL[x] = ([x],[]) &
splitL[] = ([] ,[]) &

(mergeL(x:s,y:s') = x:mergeL(s,y:s') <==== x <= y) &
(mergeL(x:s,y:s') = y:mergeL(x:s,s') <==== x > y) &
mergeL([],s) = s &
mergeL(s,[]) = s &

(split$s = (s1,s2)
     ==> mergesort$x:y:s == merge(mergesort$x:s1,mergesort$y:s2)) &
mergesort(s) == s &

(split$s = (s1,s2) ==> split$x:y:s == (x:s1,y:s2)) &
split$s == (s,[]) &

(x <= y ==> merge(x:s,y:s') == x:merge(s,y:s')) &
merge(x:s,y:s') == y:merge(x:s,s') &

```

```

merge([],s)      == s &
merge(s,[])     == s &

mergesortF(x:y:s) == fun((s1,s2),mergeF(mergesortF$x:s1)$mergesortF$y:s2)
                     $splitF$s &
mergesortF[x]   == [x] &
mergesortF[]    == [] &

splitF(x:y:s) == fun((s1,s2),(x:s1,y:s2))$splitF$s &
(length(s) <= 1 ==> splitF(s) == (s,[])) &

mergeF(x:s)$y:s' == ite(x<=y,x:mergeF(s)$y:s',y:mergeF(x:s)$s') &
mergeF[]$s        == s &
mergeF(s)[]       == s &

-- PALINDROMES

(pall$s = b <==> reveqL(s)$r = (r,b)) &

reveqL[]$s        = ([] ,true) &
(reveqL(x:s)$y:s' = (r++[x],eq(x,y)`and`b) <==> reveqL(s)$s' = (r,b)) &

pal$s == get1$mu r b.reveq(s)(r) &                                -- reveq(s)$s' = (reverse$s,s=s')
reveq[]$s == ([] ,true) &

```

```

(reveq(s)$tail$s' = (r,b) ==> reveq(x:s)$s' == (r++[x],eq(x,head$s')`and`b)) &
palI$s == get1$mu r b.reveqI(s)(r)[] &
-- revEqI(s)(s')$acc = (reverse(s)++acc,s=s')
reveqI[](s)$acc == (acc,true) &
(reveqI(s)(tail$s')$x:acc = (r,b)
 ==> reveqI(x:s)(s')$acc == (r,eq(x,head$s')`and`b)) &

(reveqF$s = (r,f) ==> palF$s == f$r) &

reveqF[] == ([] ,fun(s,true)) &
(reveqF$s = (r,f) ==> reveqF$x:s == (r++[x] ,fun(y:s',eq(x,y)`and`f$s')))) &

-- BINARY TREES

(replaceL(f)$t = u <==> foldRepL(f)(t)$x = (x,u)) &

foldRepL(f)(leaf$x)$y = (x,leaf$y) &
(foldRepL(f)(t1#t2)$x = (f(y,z),u1#u2) <==> foldRepL(f)(t1)$x = (y,u1) &
foldRepL(f)(t2)$x = (z,u2)) &

replace(f)$t == get1$mu x u.foldRep(f)(t)(x) &

foldRep(f)(leaf$x)$y == (x,leaf(y)) &
(foldRep(f)(t1)$x = (y,u1) & foldRep(f)(t2)$x = (z,u2))

```

```

==> foldRep(f)(t1#t2)$x == (f(y,z),u1#u2)) &

tree1 == leaf(3)#{leaf(2)#{leaf(6)) &

tree2 == (leaf(9)#{leaf(3))#{leaf(2) &

tree3 == (leaf(9)#{leaf(3))#{(leaf(2)#{leaf(6)) &

-- TREE SORTING

(sortTL$t = u <== leavesRepL(t)(sort$ls) = (ls,u,s)) &

(leavesRepL(leaf$x)$ls = ([x],leaf$y,s) <== ls = y:s) &
(leavesRepL(t1#t2)$ls = (ls1++ls2,u1#u2,s2)
                         <== leavesRepL(t1)$ls = (ls1,u1,s1) &
                         leavesRepL(t2)$s1 = (ls2,u2,s2)) &

sortT$t == get1$mu ls u s.leavesRep(t)(sort$ls) &

leavesRep(leaf$x)$ls == ([x],leaf$head$ls,tail$ls) &
(leavesRepL(t1)$ls = (ls1,u1,s1) & leavesRepL(t2)$s1 = (ls2,u2,s2)
                         ==> leavesRep(t1#t2)$ls == (ls1++ls2,u1#u2,s2)) &

sortTI$t == get1$mu ls u s.leavesRepI(t)(sort$ls)[] &

```

```

leavesRepI(leaf$x)(ls)$acc == (x:acc,leaf$head$ls,tail$ls) &
(leavesRepI(t1)(ls)$acc = (ls1,u1,s1) & leavesRepI(t2)(s1)$ls1 = (ls2,u2,s2)
 ==> leavesRepI(t1#t2)(ls)$acc == (ls2,u1#u2,s2)) &

-- INFINITE OBJECTS

nats == 0:map(+1)$nats &

nats' == mu s.(0:map(+1)$s) &

natsf == mu f.fun(n,n:f$n+1) & -- nats = natsf$0

fib = 1:1:zipWith(+) (fib)$tail$fib & -- simplify breadthfirst/parallel
fib' == mu s.(1:1:zipWith(+) (s)$tail$s) & -- simplify breadthfirst/parallel

-- CLIENT-SERVER INTERACTION

RequestsL(f)$g = ClientL(g)(0)$map(f)$RequestsL(f)$g &

(ClientL(g)(a)$s = a:ClientL(g)(g$b)$s' <==> s = b:s') &

Requests(f)(g)$a == Client(g)(a)$map(f)$Requests(f)(g)$a &

Client(g)(a)$s == a:Client(g)(g$head$s)$tail$s &

```

```
CSI(f)(g)$a == get0$mu requests client.  
          (client(a)$map(f)$requests,  
           fun(a,fun(s,a:client(g$head$s)$tail$s))) &
```

-- LAZY MULTIPLICATION

```
foo(x) == ite(x=0,0,foo(x-1)*foo(x+1)) &  
goo(x) == ite(x=0,0,goo(x+1)*goo(x-1))
```

conjects:

mergesortL[3,2,1,4] = s	& -- match and narrow
replaceL(min)\$tree1 = t	& -- match and narrow
replaceL(+)\$tree1 = t	& -- match and narrow
palL[2,3,2] = b	& -- unify and narrow, simpl
palL[2,3,1,2] = b	& -- unify and narrow, simpl
sortTL\$tree1 = t	-- match and narrow, simpl
take(3)\$RequestsL(+1)(*2) = s	-- match and narrow, simpl -- 9 steps (10.2.20)

terms:

mergesort[3,2,1,4]	<+>	
mergesortF[3,2,1,4]	<+>	
replace(min)\$tree1	<+>	
replace(+\$tree1	<+>	
pal[2,3,2]	<+>	
	--> bool(3 = head[]) `and` bool(2 = head(tail(tail[])))	(4.2.20)
palI[2,3,2]	<+> -- 9 steps	(10.2.20)
palI[2,3,1,2]	<+> -- 5 steps	(10.2.20)
palF[2,3,2]	<+> -- 10 steps	(10.2.20)
palF[2,3,1,2]	<+> -- 9 steps	(10.2.20)
sortT\$tree1	<+> -- wrong result	(4.2.20)
sortTI\$tree1/2	<+> -- 11 steps	(4.2.20)
sortTI\$tree3	<+> -- 12 steps	(5.2.20)
take(3)\$nats	<+> -- 21 steps	(1.11.17)
take(3)\$nats'	<+> -- 22 steps	(1.11.17)
take(3)\$Requests(+1)(*2)\$0	<+> --> [0,2,6] -- 30 steps	(10.2.20)

```
take(4)$Requests(+1)(*2)$0 <+> --> [0,2,6,14] (10.2.20)
                                -- 47 steps
take(5)$Requests(+1)(*2)$0 <+> --> [0,2,6,14,30] (10.2.20)
                                -- 66 steps
take(6)$Requests(+1)(*2)$0 <+> --> [0,2,6,14,30,62] (10.2.20)
                                -- 89 steps
take(3)$CSI(+1)(*2)$0 <+> --> [0,2,6] (9.2.20)
                                -- 42 steps
foo(3) <+> --> 0 (4.2.20)
        -- 15 depthfirst, 19 parallel, 27 breadthfirst steps
goo(3) --> 0 (4.2.20)
        -- depthfirst is non-terminating, 19 parallel, 48 breadthfirst steps
```

## Further examples

[14], Examples 1 to 5

phil, echo, election, alternating bit protocol, knight, puzzle, lift, message delivery, peterson

## Beispiele aus [13]

### [13], §4.2: Logische Programme

```
append([], L, L)
append(X:L, L1, X:L2) <== append(L, L1, L2)

sorted([])
sorted(X:[])
sorted(X:Y:L) <== X <= Y /\ sorted(Y:L)

quicksort([], [])
quicksort(Z:L, L3) <== filter(Z, L, Low, High) /\ quicksort(Low, L1) /\ 
                     quicksort(High, L2) /\ append(L1, Z:L2, L3)

filter(Z, [], [], []).
filter(Z, X:L, X:Low, High) <== X <= Z /\ filter(Z, L, Low, High)
filter(Z, X:L, Low, X:High) <== X > Z /\ filter(Z, L, Low, High)

perm([], [])
perm(X:L, P) <== perm(L, Q) /\ insert(X, Q, P)

insert(X, Q, X:Q)
insert(X, Y:Q, Y:P) <== insert(X, Q, P)
```

```
part([],[])
part(X:L,[X]:Q) <== part(L,Q)
part(X:L,P) <== part(L,Q) /\ glue(X,Q,P)
```

```
glue(X,[],[X]:[])
glue(X,L:Q,(X:L):Q)
glue(X,L:Q,L:P) <== glue(X,Q,P)
```

Any q:(perm([1,2,3],p) & q++[2] = p)

Narrowing the preceding formula leads to

Any q:(Any q0:(perm([2,3],q0) & insert(1,q0,p)) & q++[2] = p)

Simplifying the preceding formula (2 steps) leads to

Any q0:(perm([2,3],q0) & insert(1,q0,p)) & Any q:(q++[2] = p)

Narrowing the preceding formula leads to

Any q0:(Any q1:(perm([3],q1) & insert(2,q1,q0)) & insert(1,q0,p)) &  
Any q:(q++[2] = p)

Narrowing the preceding formula leads to

```
Any q0:(Any q1:(Any q2:(perm([],q2) & insert(3,q2,q1)) & insert(2,q1,q0)) &  
      insert(1,q0,p)) &  
Any q:(q++[2] = p)
```

Narrowing the preceding formula leads to

```
Any q0:(Any q1:(Any q2:(q2 = [] & insert(3,q2,q1)) & insert(2,q1,q0)) &  
      insert(1,q0,p)) &  
Any q:(q++[2] = p)
```

Simplifying the preceding formula (2 steps) leads to

```
Any q0:(Any q1:(insert(3,[],q1) & insert(2,q1,q0)) & insert(1,q0,p)) &  
Any q:(q++[2] = p)
```

Narrowing the preceding formula leads to

```
Any q0:(Any q1:(q1 = 3:[] & insert(2,q1,q0)) & insert(1,q0,p)) &  
Any q:(q++[2] = p)
```

Simplifying the preceding formula (3 steps) leads to

```
Any q0:(insert(2,[3],q0) & insert(1,q0,p)) & Any q:(q++[2] = p)
```

Narrowing the preceding formula leads to

```
Any q0:((q0 = 2:[3] | Any p5:(insert(2,[],p5) & q0 = 3:p5)) & insert(1,q0,p)) &  
Any q:(q++[2] = p)
```

Simplifying the preceding formula (6 steps) leads to

```
insert(1,[2,3],p) & Any q:(q++[2] = p) |  
Any q0:(Any p5:(insert(2,[],p5) & q0 = 3:p5) & insert(1,q0,p)) &  
Any q:(q++[2] = p)
```

Narrowing the preceding formula leads to

```
(p = 1:[2,3] | Any p6:(insert(1,[3],p6) & p = 2:p6)) & Any q:(q++[2] = p) |  
Any q0:(Any p5:(insert(2,[],p5) & q0 = 3:p5) & insert(1,q0,p)) &  
Any q:(q++[2] = p)
```

Simplifying the preceding formula (9 steps) leads to

```
Any p6:(insert(1,[3],p6) & p = 2:p6) & Any q:(q++[2] = p) |  
Any q0:(Any p5:(insert(2,[],p5) & q0 = 3:p5) & insert(1,q0,p)) &  
Any q:(q++[2] = p)
```

Narrowing the preceding formula leads to

```
Any p6:((p6 = 1:[3] | Any p7:(insert(1,[],p7) & p6 = 3:p7)) & p = 2:p6) &
Any q:(q++[2] = p) |
Any q0:(Any p5:(insert(2,[],p5) & q0 = 3:p5) & insert(1,q0,p)) &
Any q:(q++[2] = p)
```

Simplifying the preceding formula (14 steps) leads to

```
Any p6:(Any p7:(insert(1,[],p7) & p6 = 3:p7) & p = 2:p6) & Any q:(q++[2] = p) |
Any q0:(Any p5:(insert(2,[],p5) & q0 = 3:p5) & insert(1,q0,p)) &
Any q:(q++[2] = p)
```

Narrowing the preceding formula leads to

```
Any p6:(Any p7:(p7 = 1:[] & p6 = 3:p7) & p = 2:p6) & Any q:(q++[2] = p) |
Any q0:(Any p5:(insert(2,[],p5) & q0 = 3:p5) & insert(1,q0,p)) &
Any q:(q++[2] = p)
```

Simplifying the preceding formula (17 steps) leads to

```
Any q0:(Any p5:(insert(2,[],p5) & q0 = 3:p5) & insert(1,q0,p)) &
Any q:(q++[2] = p)
```

Narrowing the preceding formula leads to

```
Any q0:(Any p5:(p5 = 2:[] & q0 = 3:p5) & insert(1,q0,p)) & Any q:(q++[2] = p)
```

Simplifying the preceding formula (9 steps) leads to

$\text{insert}(1, [3, 2], p) \ \& \ \text{Any } q: (q++[2] = p)$

Narrowing the preceding formula leads to

$(p = 1:[3, 2] \mid \text{Any } p10: (\text{insert}(1, [2], p10) \ \& \ p = 3:p10)) \ \& \ \text{Any } q: (q++[2] = p)$

Simplifying the preceding formula (7 steps) leads to

$p = [1, 3, 2] \mid \text{Any } p10: (\text{insert}(1, [2], p10) \ \& \ p = 3:p10) \ \& \ \text{Any } q: (q++[2] = p)$

Narrowing the preceding formula leads to

$p = [1, 3, 2] \mid$   
 $\text{Any } p10: ((p10 = 1:[2] \mid \text{Any } p11: (\text{insert}(1, [], p11) \ \& \ p10 = 2:p11)) \ \& \ p = 3:p10) \ \&$   
 $\text{Any } q: (q++[2] = p)$

Simplifying the preceding formula (13 steps) leads to

$p = [1, 3, 2] \mid p = [3, 1, 2] \mid$   
 $\text{Any } p10: (\text{Any } p11: (\text{insert}(1, [], p11) \ \& \ p10 = 2:p11) \ \& \ p = 3:p10) \ \&$   
 $\text{Any } q: (q++[2] = p)$

Narrowing the preceding formula leads to

$p = [1,3,2] \mid p = [3,1,2] \mid$   
Any  $p_{10}:(\text{Any } p_{11}:(p_{11} = 1:\[] \& p_{10} = 2:p_{11}) \& p = 3:p_{10}) \& \text{Any } q:(q++[2] = p)$

Simplifying the preceding formula (17 steps) leads to

$p = [1,3,2] \mid p = [3,1,2]$

Number of proof steps: 24

### [13], §5.1: Partielle Unifikation

$f(f(x,y),z) = 0$

Narrowing the preceding formula leads to

$f(f(x,y),0) = 0 \& z = 0 \mid \text{Any } y_0:(2 = 0 \& z = \text{suc}(y_0))$

Narrowing the preceding formula leads to

$(f(0,0) = 0 \& x = 0 \& y = 0 \mid \text{Any } x_1:(f(1,0) = 0 \& x = \text{suc}(x_1) \& y = 0) \mid$   
 $\text{Any } y_1:(f(2,0) = 0 \& y = \text{suc}(y_1))) \&$   
 $z = 0 \mid$   
Any  $y_0:(2 = 0 \& z = \text{suc}(y_0))$

Narrowing the preceding formula leads to

$$(0 = 0 \ \& \ x = 0 \ \& \ y = 0 \mid \text{Any } x1:(f(1,0) = 0 \ \& \ x = \text{suc}(x1) \ \& \ y = 0) \mid \\ \text{Any } y1:(f(2,0) = 0 \ \& \ y = \text{suc}(y1))) \ \& \\ z = 0 \mid \\ \text{Any } y0:(2 = 0 \ \& \ z = \text{suc}(y0))$$

Narrowing the preceding formula leads to

$$(0 = 0 \ \& \ x = 0 \ \& \ y = 0 \mid \text{Any } x1:(1 = 0 \ \& \ x = \text{suc}(x1) \ \& \ y = 0) \mid \\ \text{Any } y1:(f(2,0) = 0 \ \& \ y = \text{suc}(y1))) \ \& \\ z = 0 \mid \\ \text{Any } y0:(2 = 0 \ \& \ z = \text{suc}(y0))$$

Narrowing the preceding formula leads to

$$(0 = 0 \ \& \ x = 0 \ \& \ y = 0 \mid \text{Any } x1:(1 = 0 \ \& \ x = \text{suc}(x1) \ \& \ y = 0) \mid \\ \text{Any } y1:(1 = 0 \ \& \ y = \text{suc}(y1))) \ \& \\ z = 0 \mid \\ \text{Any } y0:(2 = 0 \ \& \ z = \text{suc}(y0))$$

Simplifying the preceding formula (21 steps) leads to

$$x = 0 \ \& \ y = 0 \ \& \ z = 0$$

Number of proof steps: 6

Solutions:

$$x = 0 \ \& \ y = 0 \ \& \ z = 0$$

### [13], Beispiel 5.1.3

NAT

sorts	<i>nat</i>
constructs	$0 : \rightarrow \text{nat}$
	$\text{suc} : \text{nat} \rightarrow \text{nat}$
defuncts	$1, 2, 3, \dots : \rightarrow \text{nat}$
	$+, -, *, \min, \max : \text{nat} \times \text{nat} \rightarrow \text{nat}$
preds	$\leq, \geq, <, > : \text{nat} \times \text{nat}$
vars	$m, n : \text{nat}$
axioms	$1 \equiv \text{suc}(0)$
	$2 \equiv \text{suc}(1)$
	$3 \equiv \text{suc}(2)$
	$\dots$
	$n + 0 \equiv n$
	$m + \text{suc}(n) \equiv \text{suc}(m + n)$
	$n - 0 \equiv n$
	$0 - n \equiv 0$
	$\text{suc}(m) - \text{suc}(n) \equiv m - n$

$$\begin{aligned}
n * 0 &\equiv 0 \\
m * \text{suc}(n) &\equiv (m * n) + m \\
\text{min}(m, n) \equiv m &\Leftarrow m \leq n \\
\text{min}(m, n) \equiv n &\Leftarrow n \leq m \\
\text{max}(m, n) \equiv n &\Leftarrow m \leq n \\
\text{max}(m, n) \equiv m &\Leftarrow n \leq m \\
0 \leq n & \\
\text{suc}(m) \leq \text{suc}(n) &\Leftarrow m \leq n \\
m \geq n &\Leftarrow n \leq m \\
0 < \text{suc}(n) & \\
\text{suc}(m) < \text{suc}(n) &\Leftarrow m < n \\
m > n &\Leftarrow n < m
\end{aligned}$$

STACK = NAT and

<b>sorts</b>	<i>stack</i>
<b>constructs</b>	$() \rightarrowtail 1 + \text{nat}$
	<i>just:nat</i> $\rightarrow 1 + \text{nat}$
	<i>empty</i> $\rightarrowtail \text{stack}$
	<i>push:nat</i> $\times \text{stack} \rightarrow \text{stack}$
<b>defuncts</b>	<i>top:stack</i> $\rightarrow 1 + \text{nat}$
	<i>pop:stack</i> $\rightarrow \text{stack}$
<b>vars</b>	<i>x : nat s : stack</i>
<b>Horn axioms</b>	$\text{top}(\text{empty}) \equiv ()$
	$\text{pop}(\text{empty}) \equiv \text{empty}$
	$\text{top}(\text{push}(x, s)) \equiv \text{just}(x)$

$$pop(push(x, s)) \equiv s$$

DIVREP = NAT then

sorts	<i>tree</i>
constructs	$() \rightarrow 1 + (nat \times nat)$
	$(-, -) : nat \times nat \rightarrow 1 + (nat \times nat)$
	$leaf : nat \rightarrow tree$
	$-\#- : tree \times tree \rightarrow tree$
defuncts	$div : nat \times nat \rightarrow 1 + (nat \times nat)$
	$rep\&min : tree \times nat \rightarrow tree \times nat$
	$repByMin : tree \rightarrow tree$
vars	$m, n, n', q, r : nat \quad T, T', U, U' : tree$
Horn axioms	$div(m, n) \equiv (0, m) \Leftarrow m < n$
	$div(m, n) \equiv (suc(q), r) \Leftarrow 0 < n \wedge n \leq m \wedge div(m - n, n) \equiv (q, r)$
	$div(m, 0) \equiv ()$
	$rep\&min(leaf(n), m) \equiv (leaf(m), n)$
	$rep\&min(T \# T', m) \equiv (U \# U', min(n, n'))$
	$\Leftarrow rep\&min(T, m) \equiv (U, n) \wedge rep\&min(T', m) \equiv (U', n')$
	$repByMin(T) \equiv T' \Leftarrow rep\&min(T, m) \equiv (T', m)$

$div(m, n)$  berechnet den ganzzahligen Quotienten von  $m$  und  $n$  und gleichzeitig den verbleibenden Rest.

$rep\&min(T, m)$  liefert den minimalen Eintrag von  $T$  sowie einen neuen Baum, der aus  $T$  entsteht, wenn man alle Einträge durch  $m$  ersetzt.  $repByMin(T)$  liefert den Baum, der aus  $T$  entsteht, wenn man alle Einträge durch den minimalen Eintrag von  $T$  ersetzt.

`repByMin(leaf(3) #(leaf(2) #leaf(6))) = T`

Narrowing the preceding formula leads to

`Any T' m: (T' = T & repAndMin(leaf(3) #(leaf(2) #leaf(6)), m) = (T', m))`

Narrowing the preceding formula leads to

`Any T' m: (T' = T &`

`Any U U' n n': ((U#U', min(n, n')) = (T', m) &  
repAndMin(leaf(3), m) = (U, n) &  
repAndMin(leaf(2) #leaf(6), m) = (U', n')))`

Narrowing the preceding formula leads to

`Any T' m: (T' = T &`

`Any U U' n n': ((U#U', min(n, n')) = (T', m) & (leaf(m), 3) = (U, n) &  
repAndMin(leaf(2) #leaf(6), m) = (U', n')))`

Narrowing the preceding formula leads to

`Any T' m: (T' = T &`

`Any U U' n n': ((U#U', min(n, n')) = (T', m) & (leaf(m), 3) = (U, n) &  
Any U1 U'1 n1 n'1: ((U1#U'1, min(n1, n'1)) = (U', n') &  
repAndMin(leaf(2), m) = (U1, n1) &`

```
repAndMin(leaf(6),m) = (U'1,n'1))))
```

Narrowing the preceding formula leads to

Any  $T'$   $m$ : ( $T' = T$  &

    Any  $U$   $U'$   $n$   $n'$ : ( $(U \# U', \min(n, n')) = (T', m)$  &  $(\text{leaf}(m), 3) = (U, n)$  &

        Any  $U_1$   $U'_1$   $n_1$   $n'_1$ : ( $(U_1 \# U'_1, \min(n_1, n'_1)) = (U', n')$  &

$(\text{leaf}(m), 2) = (U_1, n_1)$  &

$\text{repAndMin}(\text{leaf}(6), m) = (U'_1, n'_1))))$

Narrowing the preceding formula leads to

Any  $T'$   $m$ : ( $T' = T$  &

    Any  $U$   $U'$   $n$   $n'$ : ( $(U \# U', \min(n, n')) = (T', m)$  &  $(\text{leaf}(m), 3) = (U, n)$  &

        Any  $U_1$   $U'_1$   $n_1$   $n'_1$ : ( $(U_1 \# U'_1, \min(n_1, n'_1)) = (U', n')$  &

$(\text{leaf}(m), 2) = (U_1, n_1)$  &

$(\text{leaf}(m), 6) = (U'_1, n'_1))))$

Simplifying the preceding formula (38 steps) leads to

$\text{leaf}(2) \# (\text{leaf}(2) \# \text{leaf}(2)) = T$

### [13], Beispiel 6.4.3: MAP implementiert STACK

Keller werden implementiert als Paare, bestehend aus einem Feld (= Funktion mit endlichem Definitionsbereich) und einem Zeiger auf das zuletzt eingetragene Element. Die Kelleroperationen werden entsprechend verfeinert. Wir beginnen mit einer parametrisierten Version der Spezifikation STACK aus [13], Beispiel 5.1.4, die das Komplement der *stack*-Gleichheit enthält (siehe [13], Beispiel 6.3.2):

*SP = STACK[NEQ(entry)] where STACK =*

<b>sorts</b>	<i>stack(entry)</i>
<b>constructs</b>	$() \rightarrow 1 + entry$
	$just : entry \rightarrow 1 + entry$
	$empty \rightarrow stack(entry)$
	$push : entry \times stack(entry) \rightarrow stack(entry)$
<b>defuncts</b>	$top : stack(entry) \rightarrow 1 + entry$
	$pop : stack(entry) \rightarrow stack(entry)$
<b>preds</b>	$\neq : stack(entry) \times stack(entry)$
<b>vars</b>	$x : entry \ s : stack(entry)$
<b>Horn axioms</b>	$top(empty) \equiv ()$
	$pop(empty) \equiv empty$
	$top(push(x, s)) \equiv just(x)$
	$pop(push(x, s)) \equiv s$
	Axiome für $\neq$ gemäß [13], Satz 6.2.7

*MAP[NEQ(index),NEQ(entry)] where MAP =*

<b>sorts</b>	<i>map(index, entry)</i>
<b>constructs</b>	$index : index \rightarrow index + entry$

	$entry : entry \rightarrow index + entry$	
	$new \mapsto map(index, entry)$	
	$upd : index \times entry \times map(index, entry) \rightarrow map(index, entry)$	“update”
defuncts	$get : map(index, entry) \times index \rightarrow index + entry$	
preds	$\equiv : map(index, entry) \times map(index, entry)$	
	$\not\equiv : (index + entry) \times (index + entry)$	
vars	$i, j : index \quad x : entry \quad f : map(index, entry)$	
axioms	$get(new, i) \equiv index(i)$	
	$get(upd(i, x, f), i) \equiv entry(x)$	
	$get(upd(i, x, f), j) \equiv get(f, j) \Leftarrow i \not\equiv j$	
	Axiome für $\not\equiv$ gemäß [13], Satz 6.2.7	

Die Herbrandmodelle der Aktualisierungen von  $MAP[NEQ(index), NEQ(entry)]$  liefern leider keine eindeutigen Repräsentationen von Feldern. Das ist für die hier behandelte Verwendung von MAP als Verfeinerung aber auch irrelevant. Zu einer adäquateren Modellierung von Feldern vgl. [13], §6.4.

Der Repräsentationsmorphismus  $rep$  bildet alle  $SP$ -Symbole bis auf die folgenden auf sich selbst ab:

$$\begin{aligned}
 rep(1) &= nat, \\
 rep(stack(entry)) &= map(nat, entry) \times nat, \\
 rep(\equiv : stack(entry) \times stack(entry)) &= \sim : (map(nat, entry) \times nat) \times (map(nat, entry) \times nat), \\
 rep(\not\equiv : stack(entry) \times stack(entry)) &= \not\sim : (map(nat, entry) \times nat) \times (map(nat, entry) \times nat).
 \end{aligned}$$

Es folgt die Verfeinerung als Extension einer Aktualisierung des ersten Parameters von  $MAP[NEQ(index), NEQ(entry)]$  durch NAT (siehe [13], Def. 6.3.4):

$SP' = \text{MAP}_{nat/index}[\text{NAT}][\text{NEQ}(entry)]$  and

defuncts	$() : \rightarrow nat + entry$
	$just : entry \rightarrow nat + entry$
	$empty : \rightarrow map(nat, entry) \times nat$
	$push : entry \times (map(nat, entry) \times nat) \rightarrow map(nat, entry) \times nat$
	$top : map(nat, entry) \times nat \rightarrow nat + entry$
	$pop : map(nat, entry) \times nat \rightarrow map(nat, entry) \times nat$
preds	$\not\sim : (map(nat, entry) \times nat) \times (map(nat, entry) \times nat)$
copreds	$\sim : (map(nat, entry) \times nat) \times (map(nat, entry) \times nat)$
vars	$i, j : nat \quad x : entry \quad f : map(nat, entry) \quad s, s' : map(nat, entry) \times nat$
axioms	$() \equiv index(0)$
	$just(x) \equiv entry(x)$
	$empty \equiv (new, 0)$
	$push(x, s) \equiv (upd(suc(i), x, f), suc(i)) \Leftarrow s \equiv (f, i)$
	$top(f, i) \equiv get(f, i)$
	$pop(f, i) \equiv (f, i - 1)$
	$s \sim s' \Rightarrow top(s) \equiv top(s') \wedge pop(s) \sim pop(s')$
	$s \not\sim s' \Leftarrow top(s) \not\equiv top(s')$
	$s \not\sim s' \Leftarrow pop(s) \not\sim pop(s')$

$SP$  und  $SP'$  sind orthogonal und terminierend, also nach [13], Kor. 5.2.7 konfluent. Da beide Spezifikationen vollständig sind, sind sie nach [13], Kor. 5.1.15 auch konsistent, also funktional. Demzufolge sind nach [13], Satz 6.2.7  $\not\equiv^{Her(SP)}$  und  $\not\equiv^{Her(SP')}$  die Komplemente von  $\equiv^{Her(SP)}$  bzw.  $\equiv^{Her(SP')}$ . Die duale Version von [13], Satze 6.2.6, mit deren Hilfe man Komplemente von Coprädiakten als Prädikate spezifizieren kann, liefert die Komplementeigenschaft von  $\not\sim^{Her(SP')}$  bzgl.  $\sim^{Her(SP')}$ .

Nach [13], Lemma 6.1.2(2) und Korollar 6.4.2 ist  $SP'$  eine Verfeinerung von  $SP$  entlang  $rep$ , wenn die folgenden  $rep$ -Bilder der Axiome von  $SP$  induktive Theoreme von  $SP'$  sind:

$$top(empty) \equiv () \tag{1}$$

$$top(push(x, s)) \equiv just(x) \tag{2}$$

$$pop(empty) \sim empty \tag{3}$$

$$pop(push(x, s)) \sim s \tag{4}$$

$$empty \not\sim push(x, s) \tag{5}$$

$$push(x, s) \not\sim empty \tag{6}$$

$$push(x, s) \not\sim push(y, s') \iff x \not\equiv y \tag{7}$$

$$push(x, s) \not\sim push(y, s') \iff s \not\sim s' \tag{8}$$

$$s \sim s \tag{9}$$

$$s \sim s' \iff s' \sim s \tag{10}$$

$$s \sim s'' \iff s \sim s' \wedge s' \sim s'' \tag{11}$$

$$push(x, s) \sim push(x, s') \iff s \sim s' \tag{12}$$

$$top(s) \equiv top(s') \iff s \sim s' \tag{13}$$

$$pop(s) \sim pop(s') \iff s \sim s' \tag{14}$$

Da die Relation  $\sim$  im Herbrandmodell von  $SP'$  als größte Lösung ihres Axioms interpretiert wird und der Äquivalenzabschluss von  $\sim^{Her(SP')}$  dieses Axiom ebenfalls erfüllt, gelten die Formeln (9)-(11) in  $Her(SP')$ .

☞ Führen Sie dieses Argument näher aus!

Die Konjunktion von (13) und (14) ist zum Axiom für  $\sim$  äquivalent. Es bleiben also (1)-(8) und (12) zu zeigen. Die folgenden Narrowing-Ableitungen von (1)-(4) und (12) wurden mit Expander2 erstellt.

Die Ableitung von (4) verwendet das Lemma

$$upd(i, x, f), j) \sim (f, j) \Leftarrow i > j, \quad (15)$$

dessen Beweis wir in [13], Beispiel 7.3.1 führen werden.

$$\text{top}(\text{empty}) = \text{nothing} \quad (1)$$

Narrowing the preceding formula leads to

$$\text{top}((\text{new}, 0)) = \text{nothing}$$

Narrowing the preceding formula leads to

$$\text{top}((\text{new}, 0)) = \text{index}(0)$$

Simplifying the preceding formula leads to

$$\text{top}(\text{new}, 0) = \text{index}(0)$$

Narrowing the preceding formula leads to

$$\text{get}(\text{new}, 0) = \text{index}(0)$$

Narrowing the preceding formula leads to

$$\text{index}(0) = \text{index}(0)$$

Simplifying the preceding formula leads to

True

---

$$\text{top}(\text{push}(x, s)) = \text{just}(x) \quad (2)$$

Narrowing the preceding formula leads to

$$\text{Any } f \ i : (\text{top}((\text{upd}(\text{suc}(i), x, f), \text{suc}(i))) = \text{just}(x) \ \& \ s = (f, i))$$

Narrowing the preceding formula leads to

$$\text{Any } f \ i : (\text{top}((\text{upd}(\text{suc}(i), x, f), \text{suc}(i))) = \text{entry}(x) \ \& \ s = (f, i))$$

Simplifying the preceding formula leads to

$$\text{Any } f \ i : (\text{top}(\text{upd}(\text{suc}(i), x, f), \text{suc}(i))) = \text{entry}(x) \ \& \ s = (f, i))$$

Narrowing the preceding formula leads to

$$\text{Any } f \ i : (\text{get}(\text{upd}(\text{suc}(i), x, f), \text{suc}(i))) = \text{entry}(x) \ \& \ s = (f, i))$$

Narrowing the preceding formula leads to

Any  $f\ i:(entry(x) = entry(x) \mid get(f, suc(i)) = entry(x) \& suc(i) =/ suc(i)) \& s = (f, i)$

Simplifying the preceding formula (7 steps) leads to

Any  $f\ i:(s = (f, i))$

Applying the theorem

True  $\implies$  Any  $f\ i:(s = (f, i))$

at position [] of the preceding formula leads to

All  $s0:($ Any  $f2\ i2:(s0 = (f2, i2))) \implies$  Any  $f\ i:(s = (f, i))$

Simplifying the preceding formula leads to

True

---

pop(empty)  $\sim$  empty (3)

Applying the axiom resp. theorem

$\text{empty} = (\text{new}, 0)$

at positions [1], [0,0] of the preceding formula leads to

$\text{pop}((\text{new}, 0)) \sim (\text{new}, 0)$

Simplifying the preceding formula leads to

$\text{pop}(\text{new}, 0) \sim (\text{new}, 0)$

Applying the axiom resp. theorem

$\text{pop}(f, i) = (f, \text{pred}(i))$

at position [0] of the preceding formula leads to

$(\text{new}, \text{pred}(0)) \sim (\text{new}, 0)$

Applying the axiom resp. theorem

$\text{pred}(0) = 0$

at position [0,1] of the preceding formula leads to

$(\text{new}, 0) \sim (\text{new}, 0)$

Simplifying the preceding formula leads to  
-- Anwendung von (9)

True

---

$$\text{pop}(\text{push}(x,s)) \sim s \quad (4)$$

Applying the axiom resp. theorem

$$\text{push}(x, (f, i)) = (\text{upd}(\text{suc}(i), x, f), \text{suc}(i))$$

at position [0,0] of the preceding formula leads to

$$\text{Any } f \ i : (\text{pop}((\text{upd}(\text{suc}(i), x, f), \text{suc}(i))) \sim (f, i) \ \& \ s = (f, i))$$

Simplifying the preceding formula leads to

$$\text{Any } f \ i : (\text{pop}(\text{upd}(\text{suc}(i), x, f), \text{suc}(i)) \sim (f, i) \ \& \ s = (f, i))$$

Applying the axiom resp. theorem

$$\text{pop}(f, i) = (f, \text{pred}(i))$$

at position [0,0,0] of the preceding formula leads to

Any  $f\ i:((upd(suc(i),x,f),pred(suc(i))) \sim (f,i) \ \& \ s = (f,i))$

Applying the axiom resp. theorem

$pred(suc(i)) = i$

at position  $[0,0,0,1]$  of the preceding formula leads to

Any  $f\ i:((upd(suc(i),x,f),i) \sim (f,i) \ \& \ s = (f,i))$

Applying the theorem

$(upd(i,x,f),j) \sim (f,j) \iff i > j$

at position  $[0,0]$  of the preceding formula leads to

Any  $f\ i:(suc(i) > i \ \& \ s = (f,i))$

Applying the axiom resp. theorem

$suc(i) > i$

at position  $[0,0]$  of the preceding formula leads to

Any  $f\ i:(\text{True} \ \& \ s = (f, i))$

Simplifying the preceding formula leads to

Any  $f\ i:(s = (f, i))$

Applying the theorem

True  $\implies$  Any  $f\ i:(s = (f, i))$

at position [] of the preceding formula leads to

All  $s_0:(\text{Any } f_2\ i_4:(s_0 = (f_2, i_4))) \implies \text{Any } f\ i:(s = (f, i))$

Simplifying the preceding formula leads to

True

---

$$s \sim s' \implies \text{push}(x, s) \sim \text{push}(x, s') \quad (12)$$

Applying the axiom resp. theorem

$$s \sim s' \implies \text{top}(s) = \text{top}(s') \ \& \ \text{pop}(s) \sim \text{pop}(s')$$

at position [1] of the preceding formula leads to

$$\begin{aligned}(s \sim s' \implies \text{top}(\text{push}(x, s)) &= \text{top}(\text{push}(x, s')) \ \& \\(s \sim s' \implies \text{pop}(\text{push}(x, s)) &\sim \text{pop}(\text{push}(x, s')))\end{aligned}$$

The reducts have been simplified.

Applying the axiom resp. theorem

$$\text{top}(\text{push}(x, s)) = \text{entry}(x) \tag{2}$$

at position [0,1,0] of the preceding formula leads to

$$\begin{aligned}(s \sim s' \implies \text{entry}(x) &= \text{top}(\text{push}(x, s')) \ \& \\(s \sim s' \implies \text{pop}(\text{push}(x, s)) &\sim \text{pop}(\text{push}(x, s')))\end{aligned}$$

The reducts have been simplified.

Applying the axiom resp. theorem

$$\text{top}(\text{push}(x, s)) = \text{entry}(x)$$

at position [0,1,1] of the preceding formula leads to

$$s \sim s' \implies \text{pop}(\text{push}(x, s)) \sim \text{pop}(\text{push}(x, s'))$$

The reducts have been simplified.

A transitivity axiom at position [1] of the preceding formula leads to  
-- Anwendung von (11)

$$s \sim s' \implies \text{Any } z_0: (\text{pop}(\text{push}(x, s)) \sim z_0 \ \& \ z_0 \sim \text{pop}(\text{push}(x, s')))$$

Applying the theorem

$$\text{pop}(\text{push}(x, s)) \sim s \tag{4}$$

at position [1,0,0] of the preceding formula leads to

$$s \sim s' \implies s \sim \text{pop}(\text{push}(x, s'))$$

The reducts have been simplified.

A transitivity axiom at position [1] of the preceding formula leads to  
-- Anwendung von (11)

$$s \sim s' \implies \text{Any } z_1: (s \sim z_1 \ \& \ z_1 \sim \text{pop}(\text{push}(x, s')))$$

Applying the theorem

$$\text{pop}(\text{push}(x, s)) \sim s \tag{4}$$

at position [1,0,1] of the preceding formula leads to

True

☞ Zeigen Sie (5)-(11)!

☞ Wie ist der Abstraktionshomomorphismus dieser Verfeinerung definiert (siehe [13], Def. 6.4.1)?

### [13], Beispiel 6.4.4: STACK implementiert SYMTAB

Hier geht es um die Implementierung von Symboltabellen, oder allgemeiner: blockstrukturierten Abbildungen, durch Keller, deren Elemente Felder im Sinne von MAP (s.o.) sind. Im Unterschied zu Bsp. ?? besteht hier die Abstraktion von den implementierenden hin zur implementierten Spezifikation nicht in der Gleichsetzung von Daten, sondern in einer echten Restriktion des implementierenden Datenbereichs. Zumindest eins seiner Elemente, nämlich der leere Keller, implementiert nichts. Formal ausgedrückt,  $\text{Her}(SP')_{rep}$  ist hier eine echte Unterstruktur von  $\text{Her}(SP')|_{rep}$  (siehe Def. ??, ?? und ??). Zunächst die (parametrisierte) Symboltabellspezifikation:

$SP = \text{SYMTAB}[\text{NEQ}(index), \text{NEQ}(entry)]$  where SYMTAB =

sorts	$\text{symtab}(index, entry)$
constructs	$() \rightarrow 1 + entry$
	$just: entry \rightarrow 1 + entry$
	$init: \rightarrow \text{symtab}(index, entry)$
	$enter: \text{symtab}(index, entry) \rightarrow \text{symtab}(index, entry)$
	$add: index \times entry \times \text{symtab}(index, entry) \rightarrow \text{symtab}(index, entry)$
defuncts	$leave: \text{symtab}(index, entry) \rightarrow \text{symtab}(index, entry)$

	$retrieve : \text{syntab}(index, entry) \times index \rightarrow 1 + entry$	
preds	$\not\equiv : \text{syntab}(index, entry) \times \text{syntab}(index, entry)$	(A)
vars	$i, j : index \quad x : entry \quad s : \text{syntab}(index, entry)$	
Horn axioms	$leave(init) \equiv init$	(B)
	$leave(\text{enter}(s)) \equiv s$	
	$leave(\text{add}(i, x, s)) \equiv leave(s)$	
	$retrieve(init, i) \equiv ()$	
	$retrieve(\text{enter}(s), i) \equiv retrieve(s, i)$	
	$retrieve(\text{add}(i, x, s), i) \equiv just(x)$	
	$retrieve(\text{add}(i, x, s), j) \equiv retrieve(s, j) \Leftarrow i \not\equiv j$	
	Axiome für $\not\equiv$ gemäß [13], Satz 6.2.7	

Es folgt die Verfeinerung als Extension der Aktualisierung (des Parameters  $\text{NEQ}(entry)$ ) von  $\text{STACK}[\text{NEQ}(entry)]$  durch  $\text{MAP}(\text{NEQ}(index), \text{NEQ}(entry))$  (siehe [13], Beispiel 6.4.3):

	$SP' = \text{STACK}_{map(index, entry)/entry}[\text{MAP}(\text{NEQ}(index), \text{NEQ}(entry))] \text{ and}$
defuncts	$init : \rightarrow \text{stack}(\text{map}(index, entry))$
	$\text{enter} : \text{stack}(\text{map}(index, entry)) \rightarrow \text{stack}(\text{map}(index, entry))$
	$\text{add} : index \times entry \times \text{stack} \rightarrow \text{stack}(\text{map}(index, entry))$
	$\text{leave} : \text{stack}(\text{map}(index, entry)) \rightarrow \text{stack}(\text{map}(index, entry))$
	$\text{retrieve} : \text{stack}(\text{map}(index, entry)) \times index \rightarrow 1 + entry$
preds	$\not\equiv : \text{stack}(\text{map}(index, entry)) \times \text{stack}(\text{map}(index, entry))$
vars	$i, j : index \quad x : entry \quad f : \text{map}(index, entry) \quad s : \text{stack}(\text{map}(index, entry))$
axioms	$init \equiv \text{push}(\text{new}, \text{empty})$
	$\text{enter}(s) \equiv \text{push}(\text{new}, s)$
	$\text{add}(i, x, \text{empty}) \equiv \text{empty}$

$$\begin{aligned}
& add(i, x, push(f, s)) \equiv push(upd(i, x, f), s) \\
& leave(empty) \equiv empty \\
& leave(push(f, empty)) \equiv push(f, empty) \\
& leave(push(f, push(g, s))) \equiv push(g, s) \\
& retrieve(empty, i) \equiv () \\
& retrieve(push(new, s), i) \equiv retrieve(s, i) \\
& retrieve(push(upd(i, x, f), s), i) \equiv just(x) \\
& retrieve(push(upd(i, x, f), s), j) \equiv retrieve(push(f, s), j) \Leftarrow i \neq j \\
& \text{Axiome für } \neq \text{ gemäß [13], Satz 6.2.7}
\end{aligned} \tag{C}$$

Wie im Beispiel [13], 6.4.3 sind  $SP$  und  $SP'$  sind orthogonal und terminierend, also nach Kor. 5.2.7 konfluent. Da beide Spezifikationen vollständig sind, sind sie nach Kor. 5.1.15 auch konsistent, also funktional. Demzufolge sind nach Satz 6.2.7  $\not\equiv^{Her(SP)}$  und  $\not\equiv^{Her(SP')}$  die Komplemente von  $\equiv^{Her(SP)}$  bzw.  $\equiv^{Her(SP')}$ .

Der Repräsentationsmorphismus  $rep$  bildet  $syntab$  auf  $stack$  und alle anderen  $SP$ -Symbole auf sich selbst ab. Im Gegensatz zu [13], Beispiel 6.4.3 ist hier das Refinement-Modell  $Her(SP')_{rep}$  eine echte Restriktion des  $rep$ -Reduktes von  $Her(SP')$ : Jeder nur aus Funktionen von  $SP$  zusammengesetzte  $stack$ -Term ist  $SP'$ -äquivalent zu einer Normalform der Gestalt  $push(f, s)$ , repräsentiert also einen nichtleeren Keller. Damit ist nach [13], Lemma 6.1.2(2) und Korollar 6.4.2  $SP'$  eine Verfeinerung von  $SP$  entlang  $rep$ , wenn alle mit  $rep$  übersetzten Instanzen von SYMTAB-Axiomen, die  $stack$ -Variablen durch Terme der Form  $push(f, s)$  substituieren, induktive Theoreme von  $SP'$  sind.

☞ Zeigen Sie, dass  $Her(SP')$  Axiom (A) erfüllt! Zeigen Sie, dass  $Her(SP')$  Axiom (A) nicht erfüllt, wenn man Axiom (C) durch

$$leave(push(f, empty)) \equiv empty$$

ersetzt!

- ☞ Welche Terme von  $Her(SP')$  enthält  $Her(SP')_{rep}$  nicht?
- ☞ Zeigen Sie mithilfe von [13], Lemma 6.1.2(2), dass  $Her(SP')_{rep}$ , aber nicht  $Her(SP')$ , Axiom (B) erfüllt!
- ☞ Zeigen Sie mithilfe von [13], Lemma 6.1.2(2), dass  $Her(SP')_{rep}$  alle Axiome von  $SP$  erfüllt!

### [13], Beispiel 6.4.5: MAP implementiert HEAP

Ein Heap ist ein binärer Baum, der sich so als Feld darstellen lässt, dass alle Knoteneinträge auf benachbarte Feldelemente abgebildet werden. Die folgende Heapspezifikation stellt neben den Binärbaumkonstruktionen die Funktion *heapify* zur Verfügung, die einen Heap in einen strukturgleichen, aber partiell geordneten Heap umwandelt (vgl. [13], Beispiel 7.4.4):

$SP = \text{HEAP}[\text{ORD}(entry)]$  where  $\text{HEAP} =$

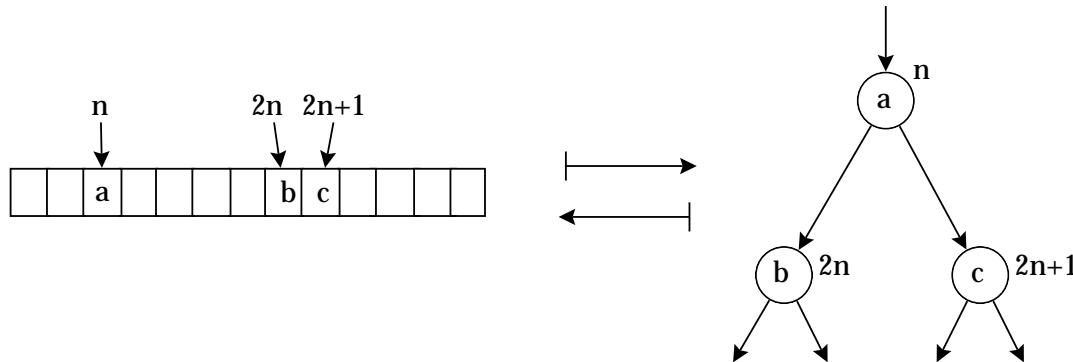
<b>sorts</b>	$bintree(entry)$
<b>constructs</b>	$mt : \rightarrow bintree(entry)$ $\# - \# - : bintree(entry) \times entry \times bintree(entry) \rightarrow bintree(entry)$
<b>defuncts</b>	$heapify, sift : bintree(entry) \rightarrow bintree(entry)$
<b>vars</b>	$x, y, z : entry$ $T1, T2, T3, T4 : bintree(entry)$
<b>Horn axioms</b>	$heapify(mt) \equiv mt$ $heapify(T1 \# x \# T2) \equiv sift(heapify(T1) \# x \# heapify(T2))$ $sift(mt) \equiv mt$ $sift(mt \# x \# mt) \equiv mt \# x \# mt$ $sift((mt \# y \# mt) \# x \# mt) \equiv (mt \# y \# mt) \# x \# mt \iff x \leq y$

$$\begin{aligned}
sift((m\#y\#mt)\#x\#mt) &\equiv (mt\#x\#mt)\#y\#mt \iff x > y \\
sift((T1\#y\#T2)\#x\#(T3\#z\#T4)) &\equiv (T1\#y\#T2)\#x\#(T3\#z\#T4) \\
&\iff x \leq y \wedge x \leq z \\
sift((T1\#y\#T2)\#x\#(T3\#z\#T4)) &\equiv sift(T1\#x\#T2)\#y\#(T3\#z\#T4) \\
&\iff x > y \wedge y \leq z \\
sift((T1\#y\#T2)\#x\#(T3\#z\#T4)) &\equiv (T1\#y\#T2)\#z\#sift(T3\#x\#T4) \\
&\iff x > z \wedge y > z
\end{aligned}$$

In den restlichen Fällen ist das Argument von *sift* kein Heap:

$$\begin{aligned}
sift(mt\#x\#(T1\#y\#T2)) &\equiv mt \\
sift(((T1\#x\#T2)\#y\#T3)\#z\#mt) &\equiv mt \\
sift((T1\#x\#(T2\#y\#T3))\#z\#mt) &\equiv mt
\end{aligned}$$

*sift* lässt den Eintrag  $x$  der Wurzel eines Heaps so lange in diesen “einsinken”, bis er auf einen Knoten trifft, dessen direkter Nachfolger größer als  $x$  sind. *heapify(T)* wendet *sift* auf alle Teilbäume von  $T$  an und macht  $T$  so zum partiell geordneten Baum.



Vom Feld zum Heap und zurück.

Die Refinementspezifikation implementiert zunächst die Heapkonstruktoren durch Feldfunktionen. Die definierten Funktionen *heapify* und *sift* werden dann – modulo dieser Datenstrukturtransformation – durch die gleichen Algorithmen implementiert, die schon in HEAP zur Spezifikation von *heapify* bzw. *sift* verwendet wurden. Dazu benötigen wir noch die Rückübersetzung *heap2map* von Heaps in Felder.

$SP' = \text{MAP}_{\text{nat}/\text{index}}[\text{NAT}][\text{ORD}(\text{entry})]$  and

**defuncts**

- $mt : (\rightarrow \text{nat}, \text{entry}) \text{map}(\text{nat}, \text{entry})$
- $\#_-\#_- : \text{map}(\text{nat}, \text{entry}) \times \text{entry} \times \text{map}(\text{nat}, \text{entry}) \rightarrow \text{map}(\text{nat}, \text{entry})$
- $\text{heapify}, \text{sift} : \text{map}(\text{nat}, \text{entry}) \rightarrow \text{map}(\text{nat}, \text{entry})$
- $\text{shift} : (\text{nat} \rightarrow \text{nat}) \times \text{map}(\text{nat}, \text{entry}) \rightarrow \text{map}(\text{nat}, \text{entry})$
- $- + - : \text{map}(\text{nat}, \text{entry}) \times \text{map}(\text{nat}, \text{entry}) \rightarrow \text{map}(\text{nat}, \text{entry})$
- $g_1, g_2, h_1, h_2 : \text{nat} \rightarrow \text{nat}$

**vars**

- $i, j, k : \text{nat}$
- $x, y, z : \text{entry}$
- $f, f_1, f_2, g : \text{map}(\text{nat}, \text{entry})$
- $h : \text{nat} \rightarrow \text{nat}$

**axioms**

- $mt \equiv \text{new}$
- $f_1 \# x \# f_2 \equiv \text{upd}(1, x, \text{shift}(g_1, f_1) + \text{shift}(g_2, f_2))$
- $\text{shift}(h, \text{new}) \equiv \text{new}$
- $\text{shift}(h, \text{upd}(i, x, f)) \equiv \text{upd}(h(i), x, \text{shift}(h, f))$
- $\text{new} + f \equiv f$
- $\text{upd}(i, x, f) + g \equiv \text{upd}(i, x, f + g)$
- $g_1(k) \equiv k + 2^i \Leftarrow k \equiv 2^i + j \wedge \forall r, s : (k \equiv 2^r + s \Rightarrow r \leq i)^{\textcolor{red}{1}}$
- $g_2(k) \equiv k + 2^{i+1} \Leftarrow k \equiv 2^i + j \wedge \forall r, s : (k \equiv 2^r + s \Rightarrow r \leq i)$
- $\text{heapify}(\text{new}) \equiv \text{new}$
- $\text{heapify}(\text{upd}(i, x, f)) \equiv \text{sift}(\text{heapify}(\text{shift}(h_1, f)) \# x \# \text{heapify}(\text{shift}(h_2, f)))$
- $h_1(k) \equiv k - 2^i \Leftarrow k \equiv 2^{i+1} + j \wedge \forall r, s : (k \equiv 2^{r+1} + s \Rightarrow r \leq i)$
- $h_2(k) \equiv k - 2^{i+1} \Leftarrow k \equiv 2^{i+1} + 2^i + j \wedge \forall r, s : (k \equiv 2^{r+1} + 2^r + s \Rightarrow r \leq i)$
- $\text{sift}(\text{new}) \equiv \text{new}$

$$\begin{aligned}
sift(upd(i, x, new)) &\equiv upd(i, x, new) \\
sift(upd(i, x, f)) \equiv upd(i, x, f) &\Leftarrow f \equiv upd(2 * i, y, new) \wedge x \leq y \\
sift(upd(i, x, f)) \equiv upd(i, y, upd(2 * i, x, new)) &\Leftarrow f \equiv upd(2 * i, y, new) \wedge x > y \\
sift(upd(i, x, f)) \equiv upd(i, x, f) &\Leftarrow f \equiv upd(2 * i, y, upd(2 * i + 1, z, g)) \wedge x \leq y \wedge x \leq z \\
sift(upd(i, x, f)) \equiv upd(i, y, upd(2 * i + 1, z, sift(upd(2 * i, x, f)))) \\
&\Leftarrow f \equiv upd(2 * i, y, upd(2 * i + 1, z, g)) \wedge x > y \wedge y \leq z \\
sift(upd(i, x, f)) \equiv upd(i, z, upd(2 * i, y, sift(upd(2 * i + 1, x, f)))) \\
&\Leftarrow f \equiv upd(2 * i, y, upd(2 * i + 1, z, g)) \wedge x > z \wedge y > z
\end{aligned}$$

”Übersetzung der 3 Fälle, in denen das Argument von *sift* kein Heap ist:

$$\begin{aligned}
sift(upd(i, x, f)) \equiv new &\Leftarrow get(f, 2 * i) \equiv index(j) \wedge f \equiv upd(2 * i + 1, y, g) \\
sift(upd(i, z, f)) \equiv new \\
&\Leftarrow get(f, 2 * i + 1) \equiv index(j) \wedge f \equiv upd(2 * i, x, upd(4 * i, y, g)) \\
sift(upd(i, z, f)) \equiv new \\
&\Leftarrow get(f, 2 * i + 1) \equiv index(j) \wedge f \equiv upd(2 * i, x, upd(4 * i + 1, y, g))
\end{aligned}$$

Der Repräsentationsmorphismus *rep* bildet *bintree* auf *map* und alle anderen *SP*-Symbole auf sich selbst ab. Die Implementierungsaxiome wurden ähnlich den in Kapitel 8 abgeleiteten Programmen aus dem Beweis von Anforderungen an sie hergeleitet. Die Anforderungen sind hier die Axiome von HEAP. Sie müssen ja im Refinementmodell  $Her(SP')_{rep}$  gelten, damit *SP'* eine Verfeinerung von HEAP ist.

Wie in [13], Beispiel 6.4.4 gehören auch hier nicht alle Elemente des *SP'*-Herbrandmodells zum Refinementmodell  $Her(SP')_{rep}$ : Jeder nur aus definierten Funktionen von *SP'* zusammengesetzte Grundterm der Sorte *map* ist zu einer Normalform der Gestalt

$$upd(1, x_1, upd(2, x_2, \dots, upd(n, x_n, new) \dots ))$$

---

<sup>1</sup>Keine Hornformel, ließe sich aber unter Verwendung einer Hilfssuchfunktion in eine solche umwandeln.

$SP'$ -äquivalent. Damit ist nach [13], Lemma 6.1.2(2) und Korollar 6.4.2  $SP'$  eine Verfeinerung von  $SP$  entlang  $rep$ , wenn alle mit  $rep$  übersetzten Instanzen von HEAP-Axiomen, die  $map$ -Variablen solche Normalformen zuordnen, induktive Theoreme von  $SP'$  sind.

Der **Heapsort**-Algorithmus überführt zunächst ein Feld mithilfe von *heapify* in die Felddarstellung eines partiell geordneten Baums  $T$ . Da dessen Wurzel das kleinste Element unter allen Knoteneinträgen von  $T$  enthält, wird dieser Knoten aus  $T$  entfernt und zum ersten Element des sortierten Feldes. Das – bzgl. seiner Feldposition – größte Blatt von  $T$  wird zur neuen Wurzel von  $T$  und mithilfe von *sift* so lange “nach unten geschoben”, bis sein Eintrag von Nachfolgereinträgen nicht mehr überschritten wird. Der Baum  $T$  hat jetzt einen Knoten weniger und seine Wurzel enthält das zweitkleinste Element des Ausgangsfeldes. Hat  $T n$  Knoten, dann terminiert der Algorithmus nach maximal  $n$  Iterationen des eben beschriebenen Schrittes mit einer sortierten Permutation des Ausgangsfeldes. Als Erweiterung von  $SP'$  lautet der Heapsort-Algorithmus wie folgt:

HEAPSORT =  $SP'$  and

<b>constructs</b>	$() : \rightarrow 1 + nat$
	$just : nat \rightarrow 1 + nat$
<b>defuncts</b>	$heapsort, loop : map(nat, entry) \rightarrow map(nat, entry)$
	$remove : map(nat, entry) \times nat \rightarrow map(nat, entry)$
	$maxindex : map(nat, entry) \rightarrow 1 + nat$
<b>axioms</b>	$heapsort(f) \equiv loop(heapify(f))$
	$loop(new) \equiv new$
	$loop(upd(i, x, new)) \equiv upd(i, x, new)$
	$loop(upd(i, x, f)) \equiv upd(i, x, loop(sift(upd(suc(i), y, remove(f, k))))))$
	$\Leftarrow maxindex(f) \equiv just(k) \wedge get(f, k) \equiv entry(y)$
	$remove(new, i) \equiv new$

$$\begin{aligned}
remove(upd(i, x, f), i) &\equiv f \\
remove(upd(i, x, f), j) &\equiv upd(i, x, remove(f, j)) \quad \Leftarrow \quad i \neq j \\
maxindex(new) &\equiv () \\
maxindex(upd(i, x, f)) &\equiv just(i) \quad \Leftarrow \quad maxindex(f) \equiv () \\
maxindex(upd(i, x, f)) &\equiv just(max(i, k)) \quad \Leftarrow \quad maxindex(f) \equiv just(k)
\end{aligned}$$

Obwohl *heapsort* hier vollständig spezifiziert ist, genügt es, die Korrektheit des Algorithmus', d.h. die Gültigkeit der Formel

$$sorted(heapSort(f)) \wedge heapSort(f) \text{ 'ist eine Permutation von' } f$$

nur für die o.g. im Refinementmodell vorkommenden Normalforminstanzen von  $f$  zu zeigen. Spezifikationen des Sortiertheitsprädikates und der Permutationsrelation (auf Listen!) findet man in den Beispielen [13], 6.3.2 bzw. 7.2.6.

### [13], §6.5: Ausnahmen und Nichtdeterminismus

Die Spezifikation MAP von Feldern (siehe Beispiel [13], 6.4.3) liefert ein Herbrand-Modell, in dem mehrere inäquivalente Grundnormalformen dasselbe Feld repräsentieren. So sind  $upd(i, x, f)$  und  $upd(i, x, upd(i, y, f))$  nicht MAP-äquivalent, obwohl beide Terme für dasselbe Feld stehen, weil die letzte Änderung von  $f$  an der Stelle  $i$  alle vorangegangenen Updates dieser Stelle überschreibt. In §6.1 haben wir Coprädiakte eingeführt, zunächst um Komplementprädikate zu spezifizieren. Eine Axiomatisierung der MAP-Äquivalenz als Coprädikat löst auch das eben beschriebene Problem. Wir erweitern MAP um das Coprädikat  $\sim: map \times map$  und das co-Hornaxiom

$$f \sim g \quad \Rightarrow \quad get(f, i) \equiv get(g, i).$$

Die Interpretation von  $\sim$  im Herbrandmodell als größte Relation, die dieses Axiom erfüllt (siehe [13],

Def. 6.2.2), führt dazu, dass nicht nur die beiden o.g. Terme äquivalent sind, sondern auch alle anderen Grundnormalformen, die dasselbe Feld repräsentieren. Genauer gesagt,  $\text{Her}(\text{MAP})$  erfüllt folgende Äquivalenzen:

$$\begin{aligned} \text{upd}(i, x, \text{upd}(i, y, f)) &\sim \text{upd}(i, x, f) \\ \text{upd}(i, x, \text{upd}(j, y, f)) &\sim \text{upd}(j, y, \text{upd}(i, x, f)) \quad \Leftarrow \quad i \neq j \end{aligned}$$

Ein anderes Beispiel für eine Klasse von Äquivalenzrelationen, die sich einfach als Coprädiakte spezifizieren lässt, sind *starke Äquivalenzen*, die die Identität der Elemente eines Summanden einer zweistelligen Summensorte erhalten, jedoch alle Elemente des anderen Summanden gleichsetzen soll (siehe [13], Def. 6.5.1). Die *index + entry*-Komponente des Abstraktionshomomorphismus in Beispiel [13], 6.4.3 hat z.B. eine starke Äquivalenz als Äquivalenzkern: Er bildet nämlich alle Grundnormalformen der Gestalt  $\text{index}(t)$  auf  $\text{index}(0)$  ab, diejenigen der Gestalt  $\text{entry}(t)$  bleiben jedoch erhalten.

Gleichheitsprädiakte werden verwendet, um Funktionen zu axiomatisieren. Umgekehrt werden in den co-Hornformeln, die eine Äquivalenzrelation axiomatisieren, Funktionen verwendet: *top* und *pop* in [13], Beispiel 6.4.3, *get* in MAP und *def* in den Axiomen für  $\approx$ . Dies ist neben der Komplementspezifizierbarkeit eine weitere Dualität zwischen Prädikaten und Coprädiaken.

### [13], Beispiel 6.7.3

REPAL = NAT and LIST[TRIV( $s$ )] and (siehe [13], 5.1.5 und 6.3.2)

```

sorts      tree
constructs leaf:nat → tree
           #-:tree × tree → tree
defuncts   rev:list( $s$ ) → list( $s$ )
           pal:list( $s$ ) → bool
           rep:tree × nat → tree

```

```

min:tree → nat
repByMin:tree → tree
vars      x, y : entry  L, L' : list(s)  m, n : nat  T, T' : tree
axioms   rev([]) ≡ []
          rev(x : L) ≡ rev(L) ++[x]
(A)      pal(L) ≡ eq(L, rev(L))
          min(leaf(n)) ≡ n
          min(T#T') ≡ min(min(T), min(T'))
          rep(leaf(m), n) ≡ leaf(n)
          rep(T#T', n) ≡ rep(T, n)#rep(T', n)
(B)      repByMin(T) ≡ rep(T, min(T))

```

$rev(L)$  revertiert die Liste  $L$ . Das Herbrandmodell von REPAL erfüllt  $pal(L) \equiv true$  genau dann, wenn  $L$  ein **Palindrom** ist.  $rep(T, n)$  ersetzt alle Blatteinträge von  $T$  durch  $n$ .  $min(T)$  berechnet den minimalen Blatteintrag des Baums  $T$ .  $repByMin(T)$  ersetzt alle Blatteinträge von  $T$  durch das Minimum der Blatteinträge von  $T$ . Die Axiome für  $pal$  bzw.  $repByMin$  liefern recht ineffiziente Implementierungen der beiden Funktionen, weil jede Liste bzw. jeder Baum zweimal durchlaufen werden müssen, um das jeweilige Ergebnis zu berechnen. Mithilfe von  $\lambda$ -Abstraktionen lassen sie sich effizienter implementieren:

REPALA = NAT and LIST[TRIV( $s$ )] and

```

sorts      tree
constructs leaf:nat → tree
           -#:tree × tree → tree
defuncts  eq&rev:list(s) → ((list(s) → bool) × list(s))
           pal:list → bool

```

	$rep\&min : tree \rightarrow ((nat \rightarrow tree) \times nat)$
	$repByMin : tree \rightarrow tree$
vars	$x, y : s \ L, L', L_1 : list(s) \ f : list(s) \rightarrow bool \ k, m, n : nat \ T, T' : tree \ g, h : nat \rightarrow tree$
axioms	$eq\&rev([]) \equiv (\lambda[]. true \mathbin{\parallel} \lambda x : L. false, [])$
	$eq\&rev(x : L) \equiv (\lambda[]. false \mathbin{\parallel} \lambda y : L'. (eq(x, y) \text{ and } f(L')), L_1 ++ [x])$
	$\Leftarrow eq\&rev(L) \equiv (f, L_1)$
(A')	$pal(L) \equiv f(L') \Leftarrow eq\&rev(L) \equiv (f, L')$
	$rep\&min(leaf(n)) \equiv (leaf, n)$
	$rep\&min(T \# T') \equiv (\lambda k. (g(k) \# h(k)), min(m, n))$
	$\Leftarrow rep\&min(T) \equiv (g, m) \wedge rep\&min(T') \equiv (h, n)$
(B')	$repByMin(T) \equiv g(n) \Leftarrow rep\&min(T) \equiv (g, n)$

## [13], §7.2: Induktion

Mit Expander2 kann man auch Noethersche Induktionsbeweise wie folgt führen. Nach Auswahl der beweisenden Formel, i.a. eine Implikation  $prem \Rightarrow conc$ , werden Variablen  $x_1, \dots, x_n$  angeklickt, die damit als Induktionsvariablen deklariert sind. Außerdem erzeugt das System zwei Lemmas, die Induktionshypthesen entsprechen:

$$conc' \Leftarrow (x_1, \dots, x_n) \gg (x'_1, \dots, x'_n) \wedge prem' \quad (4)$$

$$prem' \Rightarrow ((x_1, \dots, x_n) \gg (x'_1, \dots, x'_n) \Rightarrow conc') \quad (5)$$

$conc'$  und  $prem'$  werden aus  $conc$  bzw.  $prem$  gebildet, indem jedes Vorkommen einer Induktionsvariablen  $x$  durch  $x'$  ersetzt wird. Die **Abstiegsbedingung**  $(x_1, \dots, x_n) \gg (x'_1, \dots, x'_n)$  ist ein Atom, wobei  $\gg$  die – ggf. noch unspezifizierte – Induktionsordnung bezeichnet,  $x_1, \dots, x_n$  die ursprünglichen

Induktionsvariablen sind und  $x'_1, \dots, x'_n$  eine Kopie davon, die bei Anwendung des Lemmas auf eine Instanz von *prem* bzw. *conc* instantiiert wird. Die Anwendung von (4) bzw. (5) kann erst dann als Induktionsschritt bezeichnet werden, wenn das Redukt der Anwendung, d.h. die entsprechende Instanz der Prämisse von (4) bzw. Konklusion von (5) bewiesen, also auf *True* reduziert worden ist. Dazu muss die Induktionsordnung  $\gg$  axiomatisiert werden, natürlich so, dass ihre Interpretation im Herbrandmodell der jeweiligen Spezifikation wohlfundiert ist.

```

PARTITION = LIST[TRIV(entry)] and
            LIST{list(entry)/entry}[LIST[TRIV(entry)]] then
defuncts flatten:list(list(entry))->list(entry)
preds part:list*list(list(entry))
vars x,y:s s,s':list(entry) p:list(list(entry))
axioms flatten([]) = []
            flatten(s:p) = s++flatten(p)
            part([x], [[x]])
            part(x:y:s, [x]:p) <== part(y:s,p)
            part(x:y:s, (x:s') :p) <== part(y:s,s' :p)

```

Die Korrektheit (der Axiome von) *part* wird durch folgende Hornformel beschrieben:

$$part(s, p) \Rightarrow s \equiv flatten(p). \quad (6)$$

Wir wählen  $s$  als Induktionsvariable (und markieren sie mit einem Ausrufungszeichen), werden also für einen Noetherschen Induktionsbeweis ein Prädikat  $\gg$ :  $list(entry) \times list(entry)$  brauchen, dessen Interpretation im initialen Modell von PARTITION wohlfundiert ist. Die Induktionshypotesen (4) und (5) lauten hier demnach wie folgt:

```

s' = flatten(p') <==> part(s',p') & !s >> s'
part(s',p') ==> (!s >> s' ==> s' = flatten(p'))

```

Das zweite wird an den Stellen (A) und (B) angewendet. Im jeweils folgenden Schritt wird die eben erzeugte Abstiegsbedingung durch Resolution mit dem (einzigsten) Axiom  $x : s \gg s$  für die Induktionsordnung sofort wieder eliminiert.

$\text{part}(s,p) ==> s = \text{flatten}(p)$

Selecting induction variables at position [0,0] of the preceding formula leads to

All  $p : (\text{part}(!s,p) ==> !s = \text{flatten}(p))$

Applying the axioms

```

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

```

at positions [0,1,1], [0,0] of the preceding formula leads to

```

All x:(!s = [x] ==> [] = flatten[]) &
All x y s p0:(!s = x:y:s & part(y:s, p0) ==> y:s = flatten(p0)) &
All x y s s' p0:(!s = x:y:s & part(y:s, s':p0) ==> y:s = s'++flatten(p0))

```

The reducts have been simplified.

Applying the axiom resp. theorem

flatten[] = []

at position [0,0,1,1] of the preceding formula leads to

All x y s p0:(!s = x:y:s & part(y:s,p0) ==> y:s = flatten(p0)) &  
All x y s s' p0:(!s = x:y:s & part(y:s,s':p0) ==> y:s = s'++flatten(p0))

The reducts have been simplified.

Shifting subformulas at position [0,0,0,1] of the preceding formula leads to

All x y s p0:(!s = x:y:s ==> (part(y:s,p0) ==> y:s = flatten(p0))) &  
All x y s s' p0:(!s = x:y:s & part(y:s,s':p0) ==> y:s = s'++flatten(p0))

Applying the induction hypothesis

part(s,p) ==> (!s >> s ==> s = flatten(p)) (A)

at position [0,0,1,0] of the preceding formula leads to

```

All x y s p0:(!s = x:y:s & (x:y:s >> y:s ==> y:s = flatten(p0)) ==>
          y:s = flatten(p0)) &
All x y s s' p0:(!s = x:y:s & part(y:s,s':p0) ==> y:s = s'++flatten(p0))

```

The reducts have been simplified.

Applying the axioms

```

x:s >> s
& (x >> y <==> x > y)

```

at position [0,0,0,1,0] of the preceding formula leads to

```
All x y s s' p0:(!s = x:y:s & part(y:s,s':p0) ==> y:s = s'++flatten(p0))
```

The reducts have been simplified.

Shifting subformulas at position [0,0,1] of the preceding formula leads to

```
All x y s s' p0:(!s = x:y:s ==> (part(y:s,s':p0) ==> y:s = s'++flatten(p0)))
```

Applying the induction hypothesis

$$\text{part}(s,p) ==> (!s >> s ==> s = \text{flatten}(p)) \quad (\text{B})$$

at position [0,1,0] of the preceding formula leads to

```
All x y s s' p0:(!s = x:y:s & (x:y:s >> y:s ==> y:s = flatten(s':p0)) ==>  
y:s = s'++flatten(p0))
```

The reducts have been simplified.

Applying the axioms

```
x:s >> s  
& (x >> y <==> x > y)
```

at position [0,0,1,0] of the preceding formula leads to

```
All x y s s' p0:(!s = x:y:s & y:s = flatten(s':p0) ==> y:s = s'++flatten(p0))
```

The reducts have been simplified.

Applying the axiom resp. theorem

```
s++flatten(p) = flatten(s:p)
```

at position [0,1,1] of the preceding formula leads to

True

Selbst um einfache Aussagen, deren induktive Gültigkeit schon beim ersten Hinsehen klar ist, formal zu beweisen, sind manchmal nichttriviale Induktionsordnungen erforderlich – wie das folgende Beispiel zeigt (siehe C.-P. Wirth, *Descente Infinie + Deduction*, Logic Journal of the IGPL 12, No. 1 (2004) 1-96, §3.2.2).

```

PQ = NAT and
      preds      P:nat
                  Q:nat*nat
                  `gt`:(nat*nat)*(nat*nat)          -- lexikographische Ordnung
                  >>:(nat*nat*nat)*(nat*nat*nat)    -- Induktionsordnung
                  Nat:nat
      axioms      P(0)
                  P(suc(x)) <== P(x) /\ Q(x,suc(x))
                  Q(x,0)
                  Q(x,suc(y)) <== P(x) /\ Q(x,y)
                  (x,y) `gt` (x',y') <== x > x'
                  (x,y) `gt` (x,y') <== y > y'
                  (x,y,z) >> (x',y',z') <== x > x' /\ x > y'
                  (x,y,z) >> (x',y',z') <== (y,z) `gt` (x',0) /\ 
                                         (y,z) `gt` (y',z')
                  Nat(0)
                  Nat(suc(x)) <== Nat(x)

```

Wir beweisen die Formel

$$Nat(x) \wedge Nat(y) \wedge Nat(z) \Rightarrow p(x) \wedge q(y, z)$$

durch Noethersche Induktion, wobei alle drei Variablen  $x, y, z$  als Induktionsvariablen deklariert werden. Das Prädikat  $Nat$  schränkt die Instanzen von  $x, y, z$  auf Grundterme ein, die zu Normalformen der Sorte  $nat$  reduziert werden können. Ohne diese Einschränkung wäre die Formel nicht beweisbar.

Die Induktionshypotesen (4) und (5) lauten hier demnach wie folgt:

$$\begin{aligned} P(x') \wedge Q(y', z') &\quad <==> Nat(x') \wedge Nat(y') \wedge Nat(z') \wedge (!x, !y, !z) \gg (x', y', z') \\ Nat(x') \wedge Nat(y') \wedge Nat(z') &\quad ==> (!x, !y, !z) \gg (x', y', z') ==> (P(x') \wedge Q(y', z')) \end{aligned}$$

Die erste wird an den Stellen (A) und (B) angewendet.

$$Nat(x) \wedge Nat(y) \wedge Nat(z) ==> P(x) \wedge Q(y, z)$$

Selecting induction variables at positions [0,0,0], [0,1,0], [0,2,0] of the preceding for

$$Nat(!x) \wedge Nat(!y) \wedge Nat(!z) ==> P(!x) \wedge Q(!y, !z)$$

Applying the axioms

$$\begin{aligned} &Q(x, 0) \\ &\wedge (Q(x, suc(y)) <==> P(x) \wedge Q(x, y)) \\ &\wedge P(0) \\ &\wedge (P(suc(x)) <==> P(x) \wedge Q(x, suc(x))) \end{aligned}$$

at positions [1,1],[1,0] of the preceding formula leads to

$$\begin{aligned} & (\text{Nat}(!x) \ \& \ \text{Nat}(!y) \ \& \ \text{Nat}(!z) \ ==> \\ & \quad !x = 0 \mid \text{Any } x:(P(x) \ \& \ Q(x, \text{suc}(x)) \ \& \ !x = \text{suc}(x))) \ \& \\ & (\text{Nat}(!x) \ \& \ \text{Nat}(!y) \ \& \ \text{Nat}(!z) \ ==> !z = 0 \mid P(!y) \ \& \ \text{Any } y:(Q(!y, y) \ \& \ !z = \text{suc}(y))) \end{aligned}$$

The reducts have been simplified.

Applying the induction hypothesis

$$P(x) \ \& \ Q(y, z) <==> (\text{Nat}(x) \ \& \ \text{Nat}(y) \ \& \ \text{Nat}(z)) \ \& \ (!x, !y, !z) >> (x, y, z) \tag{A}$$

at positions [0,1,1,0,0],[0,1,1,0,1] of the preceding formula leads to

$$\begin{aligned} & (\text{Nat}(!x) \ \& \ \text{Nat}(!y) \ \& \ \text{Nat}(!z) \ ==> \\ & \quad !x = 0 \mid \\ & \quad \text{Any } x:(!x = \text{suc}(x) \ \& \ \text{Nat}(x) \ \& \ \text{Nat}(\text{suc}(x)) \ \& \ (\text{suc}(x), !y, !z) >> (x, x, \text{suc}(x))) \ \& \\ & (\text{Nat}(!x) \ \& \ \text{Nat}(!y) \ \& \ \text{Nat}(!z) \ ==> !z = 0 \mid P(!y) \ \& \ \text{Any } y:(Q(!y, y) \ \& \ !z = \text{suc}(y))) \end{aligned}$$

The reducts have been simplified.

Moving up quantifiers at position [1,1,1,1] of the preceding formula leads to

$$(\text{Nat}(!x) \ \& \ \text{Nat}(!y) \ \& \ \text{Nat}(!z) \ ==>$$

$\neg x = 0 \mid$   
 Any  $x:(\neg x = \text{suc}(x) \ \& \ \text{Nat}(x) \ \& \ \text{Nat}(\text{suc}(x)) \ \& \ (\text{suc}(x), \neg y, \neg z) \gg (x, x, \text{suc}(x))) \ \&$   
 $(\text{Nat}(\neg x) \ \& \ \text{Nat}(\neg y) \ \& \ \text{Nat}(\neg z) \implies \neg z = 0 \mid \text{Any } y:(P(\neg y) \ \& \ Q(\neg y, y) \ \& \ \neg z = \text{suc}(y)))$

Applying the induction hypothesis

$P(x) \ \& \ Q(y, z) \iff (\text{Nat}(x) \ \& \ \text{Nat}(y) \ \& \ \text{Nat}(z)) \ \& \ (\neg x, \neg y, \neg z) \gg (x, y, z)$  (B)

at positions  $[1, 1, 1, 0, 0], [1, 1, 1, 0, 1]$  of the preceding formula leads to

$(\text{Nat}(\neg x) \ \& \ \text{Nat}(\neg y) \ \& \ \text{Nat}(\neg z) \implies$   
 $\neg x = 0 \mid$   
 Any  $x:(\neg x = \text{suc}(x) \ \& \ \text{Nat}(x) \ \& \ \text{Nat}(\text{suc}(x)) \ \& \ (\text{suc}(x), \neg y, \neg z) \gg (x, x, \text{suc}(x))) \ \&$   
 $(\text{Nat}(\neg x) \ \& \ \text{Nat}(\neg y) \ \& \ \text{Nat}(\neg z) \implies$   
 $\neg z = 0 \mid \text{Nat}(\neg y) \ \& \ \text{Any } y:(\neg z = \text{suc}(y) \ \& \ \text{Nat}(y) \ \& \ (\neg x, \neg y, \text{suc}(y)) \gg (\neg y, \neg y, y)))$

The reducts have been simplified.

Applying the axiom resp. theorem

$(x, y, z) \gg (x', y', z') \iff x > x' \ \& \ x > y'$

at position  $[0, 1, 1, 0, 3]$  of the preceding formula leads to

$(\text{Nat}(\neg x) \ \& \ \text{Nat}(\neg y) \ \& \ \text{Nat}(\neg z) \implies$

$$\begin{aligned} !x = 0 \mid \text{Any } x: (!x = \text{suc}(x) \ \& \ \text{Nat}(x) \ \& \ \text{Nat}(\text{suc}(x)) \ \& \ \text{suc}(x) > x)) \ \& \\ (\text{Nat}(!x) \ \& \ \text{Nat}(!y) \ \& \ \text{Nat}(!z) ==> \\ !z = 0 \mid \text{Nat}(!y) \ \& \ \text{Any } y: (!z = \text{suc}(y) \ \& \ \text{Nat}(y) \ \& \ (!x, !y, \text{suc}(y)) \gg (!y, !y, y))) \end{aligned}$$

The reducts have been simplified.

Applying the axiom resp. theorem

$$(x, y, z) \gg (x', y', z') <==> (y, z) \text{ `gt' } (x', 0) \ \& \ (y, z) \text{ `gt' } (y', z')$$

at position [1,1,1,1,0,2] of the preceding formula leads to

$$\begin{aligned} (\text{Nat}(!x) \ \& \ \text{Nat}(!y) \ \& \ \text{Nat}(!z) ==> \\ !x = 0 \mid \text{Any } x: (!x = \text{suc}(x) \ \& \ \text{Nat}(x) \ \& \ \text{Nat}(\text{suc}(x)) \ \& \ \text{suc}(x) > x)) \ \& \\ (\text{Nat}(!x) \ \& \ \text{Nat}(!y) \ \& \ \text{Nat}(!z) ==> \\ !z = 0 \mid \\ \text{Nat}(!y) \ \& \\ \text{Any } y: (!z = \text{suc}(y) \ \& \ \text{Nat}(y) \ \& \ (!y, \text{suc}(y)) \text{ `gt' } (!y, 0) \ \& \\ (!y, \text{suc}(y)) \text{ `gt' } (!y, y))) \end{aligned}$$

The reducts have been simplified.

Applying the axioms

$$((x, y) \text{ `gt' } (x', y') <==> x > x')$$

& ((x,y) `gt` (x,y')) <==> y > y')

at positions [1,1,1,1,0,3], [1,1,1,1,0,2] of the preceding formula leads to

(Nat(!x) & Nat(!y) & Nat(!z) ==>  
!x = 0 | Any x:(!x = suc(x) & Nat(x) & Nat(suc(x)) & suc(x) > x)) &  
(Nat(!x) & Nat(!y) & Nat(!z) ==>  
!z = 0 | Nat(!y) & Any y:(!z = suc(y) & Nat(y) & suc(y) > y))

The reducts have been simplified.

Applying the axiom resp. theorem

suc(x) > x

at position [0,1,1,0,3] of the preceding formula leads to

(Nat(!x) & Nat(!y) & Nat(!z) ==>  
!x = 0 | Any x:(!x = suc(x) & Nat(x) & Nat(suc(x)))) &  
(Nat(!x) & Nat(!y) & Nat(!z) ==>  
!z = 0 | Nat(!y) & Any y:(!z = suc(y) & Nat(y) & suc(y) > y))

The reducts have been simplified.

Applying the axioms

$(\text{Nat}(\text{suc}(x)) \leq \text{Nat}(x))$   
 $\& \text{Nat}(0)$

at positions  $[0,1,1,0,2], [0,0,0]$  of the preceding formula leads to

$\text{Nat}(!x) \& \text{Nat}(!y) \& \text{Nat}(!z) \Rightarrow$   
 $!z = 0 \mid \text{Nat}(!y) \& \text{Any } y: (!z = \text{suc}(y) \& \text{Nat}(y) \& \text{suc}(y) > y)$

The reducts have been simplified.

Applying the axiom resp. theorem

$\text{suc}(x) > x$

at position  $[1,1,1,0,2]$  of the preceding formula leads to

$\text{Nat}(!x) \& \text{Nat}(!y) \& \text{Nat}(!z) \Rightarrow !z = 0 \mid \text{Nat}(!y) \& \text{Any } y: (!z = \text{suc}(y) \& \text{Nat}(y))$

The reducts have been simplified.

Applying the axioms

$\text{Nat}(0)$   
 $\& (\text{Nat}(\text{suc}(x)) \leq \text{Nat}(x))$

at position [0,2] of the preceding formula leads to

True

### [13], Beispiel 7.2.6: Mergesort

Wir zeigen die Korrektheit eines funktionalen Sortierprogramms (*Sortieren durch Mischen*). Es wird als Extension von LIST'[ORD(entry)] spezifiziert. Die Beweise setzen voraus, dass die Relation  $\sim$  (“ist Permutation von”) eine Kongruenzrelation ist. Axiome für  $\sim$  werden nicht verwendet, aber mehrere Lemmas.

```
MERGESORT = LIST' [ORD(entry)] and
  defuncts  sort:list(nat)->list(entry)
            merge:list(entry)*list(entry)->list(entry)
            split:list(entry)->(list(entry)*list(entry))
  preds      ~:list(entry)*list(entry)
  vars       x,y:s  s,s1,s2,s',s1',s2':list(entry)
  axioms    sort([]) = []
            sort([x]) = [x]
            sort(x:y:s) = merge(s1',s2')
            <==  split(s) = (s1,s2) & sort(x:s1) = s1'
                           & sort(y:s2) = s2'
            merge([],s) = s
            merge(s,[]) = s
            merge(x:s1,y:s2) = x:merge(s1,y:s2)  <==  x <= y
```

	$\text{merge}(x:s_1, y:s_2) = y:\text{merge}(x:s_1, s_2) \quad \Leftarrow\Rightarrow x > y$	
	$\text{split}([]) = ([] , [])$	
	$\text{split}([x]) = ([x] , [])$	
	$\text{split}(x:y:s) = (x:s_1, y:s_2) \quad \Leftarrow\Rightarrow \text{split}(s) = (s_1, s_2)$	
lemmas	$\text{sorted}(s_1) \ \& \ \text{sorted}(s_2) \implies \text{sorted}(\text{merge}(s_1, s_2))$	(1)
	$\text{split}(s) = (s_1, s_2) \implies s \sim s_1++s_2$	(2)
	$s \sim \text{merge}(s_1, s_2) \implies s \sim s_1++s_2$	(3)
	$x:y:(s++s') \sim (x:s)++(y:s')$	(4)
	$s'++x:s \sim x:s++s'$	(5)
	$x > y \implies y \leq x$	(6)

Die Korrektheitsanforderungen an (die Axiome für) *sort* lauten:

$$\begin{aligned} \text{sort}(s) \equiv s' &\Rightarrow \text{sorted}(s'), \\ \text{sort}(s) \equiv s' &\Rightarrow s \sim s'. \end{aligned}$$

In beiden Teilen führt die Anwendung der Fixpunktinduktion zur Einführung eines Prädikats, das die jeweilige Konklusion  $\varphi(x)$  (s.o.) wiedergibt. Die automatisch erzeugten (co-Horn-)Axiome (siehe §7.3) für diese Prädikate lauten demnach wie folgt:

$$\text{sort0}(s, s') \implies \text{sorted}(s') \quad \text{bzw.} \quad \text{sort1}(s, s') \implies s \sim s'$$

Hier sind die Beweise:

$$\text{sort}(s) = s' \implies \text{sorted}(s')$$

Applying fixpoint induction w.r.t.

```

sort[] = []
& sort[x] = [x]
& (sort(x:(y:s)) = merge(z0,z1)
  <== split(s) = (s1,s2) & sort(x:s1) = z0 & sort(y:s2) = z1)

```

to the preceding tree leads to the formula

```

All x y s7 z0 z1 s1 s2:
( sort0([],[])
  & sort0([x],[x])
  & (sort0(x:(y:s7),merge(z0,z1))
    <== split(s7) = (s1,s2) & sort0(x:s1,z0) & sort0(y:s2,z1)))

```

Simplifying (4 steps) the preceding tree leads to new ones.

The current factor is given by

```
sort0([],[])
```

Narrowing the preceding factor leads to the factor

```
sorted[]
```

Narrowing the preceding factor leads to new ones. The current factor is given by

All x: (sort0([x], [x]))

Narrowing the preceding factor leads to the factor

All x2: (sorted[x2])

Narrowing the preceding factor leads to a new formula. The current formula is given by

All x y s7 z0 z1 s1 s2:  
(split(s7) = (s1,s2) & sort0(x:s1,z0) & sort0(y:s2,z1)  
==> sort0(x:(y:s7),merge(z0,z1)))

Applying the axiom

sort0(s,s') ==> sorted(s')

at positions [0,1],[0,0,2],[0,0,1] of the preceding tree leads to the formula

All x y s7 z0 z1 s1 s2:  
(split(s7) = (s1,s2) & sorted(z0) & sorted(z1) ==> sorted(merge(z0,z1)))

Applying the theorem

sorted(s) & sorted(s') ==> sorted(merge(s,s'))

at position [0,0] of the preceding tree leads to the formula

All x y s7 z0 z1 s1 s2:

(sorted(merge(z0,z1)) & split(s7) = (s1,s2) ==> sorted(merge(z0,z1)))

Simplifying (3 steps) the preceding tree leads to the formula

True

\*\*\*\*\*

sort(s) = s' ==> s ~ s'

Applying fixpoint induction w.r.t.

```
sort[] = []
& sort[x] = [x]
& (sort(x:(y:s)) = merge(z3,z4)
  <==> split(s) = (s1,s2) & sort(x:s1) = z3 & sort(y:s2) = z4)
```

to the preceding tree leads to the formula

All x y s7 z3 z4 s1 s2:

```
( sort1([],[])
  & sort1([x],[x]) )
```

```
& (sort1(x:(y:s7),merge(z3,z4))
  <==> split(s7) = (s1,s2) & sort1(x:s1,z3) & sort1(y:s2,z4)))
```

Applying the axiom

```
sort1(s,s') ==> s ~ s'
```

at positions [0,2,1,2], [0,2,1,1], [0,2,0], [0,1], [0,0]  
of the preceding tree leads to the formula

All x y s7 z3 z4 s1 s2:

```
( [] ~ []
 & [x] ~ [x]
 & (x:(y:s7) ~ merge(z3,z4) <==> split(s7) = (s1,s2) & x:s1 ~ z3 & y:s2 ~ z4))
```

Simplifying (13 steps) the preceding tree leads to the formula

```
All x y s7 s1 s2: (split(s7) = (s1,s2) ==> x:(y:s7) ~ merge(x:s1,y:s2))
```

Applying the theorem

```
split(s) = (s1,s2) ==> s ~ s1++s2
```

at position [0,0] of the preceding tree leads to the formula

All  $x \ y \ s7 \ s8 \ s9$ :  $(s7 \sim s8++s9 \Rightarrow x:(y:s7) \sim \text{merge}(x:s8, y:s9))$

Applying the theorem

$$s \sim \text{merge}(s1, s2) \Leftrightarrow s \sim s1++s2$$

at position [0,1] of the preceding tree leads to the formula

All  $x \ y \ s7 \ s8 \ s9$ :  $(s7 \sim s8++s9 \Rightarrow x:(y:s7) \sim (x:s8)++(y:s9))$

Simplifying the preceding tree leads to the formula

All  $x \ y \ s7 \ s8 \ s9$ :  $(s7 \sim s8++s9 \Rightarrow x:(y:s7) \sim x:(s8++(y:s9)))$

Applying the theorem

$$x:y:(s1++s2) \sim x:(s1++(y:s2))$$

at position [0,1] of the preceding tree leads to the formula

All  $x3 \ y2 \ s7 \ s8 \ s9$ :  $(s7 \sim s8++s9 \Rightarrow \text{True})$

Simplifying (3 steps) the preceding tree leads to the formula

True

☞ *sorted* spezifiziert aufsteigende Sortierung. Geben Sie Hornaxiome für ein Prädikat *sortedB* an, das im Herbrandmodell genau dann gilt, wenn die Argumentliste absteigend sortiert ist. Zeigen Sie

$$\text{reverse}(L) \equiv L' \Rightarrow (\text{sorted}(L) \Rightarrow \text{sortedB}(L'))$$

mit Fixpunktinduktion über *reverse* (siehe Bsp. [13], 6.3.2)!

[13], Beispiel 7.2.7: Quick- und Bubblesort Zeigen Sie die Korrektheit von *quicksort* und *bubblesort* bzgl. folgender Spezifikationen:

```

QUICKSORT = LIST' [ORD(entry)] and
  defuncts  sort:list(entry)->list(entry)
            smaller,greater:nat->(entry->bool)
  vars      x,y:nat  s,s1,s2:list(entry)
  axioms    sort([]) = []
            sort(x:s) = sort(s1)++x:sort(s2)
                        <== filter(smaller(x),s) = s1 /\ filter(greater(x),s)
            smaller(x)(y) = true  <== x <= y
            smaller(x)(y) = false <== x > y
            greater(x)(y) = not(smaller(x)(y))

BUBBLESORT = LIST' [ORD(entry)] then
  defuncts  sort:list(entry)->list(entry)
            bubble:list(entry)->list(entry)
  vars      x,y:entry  s,s':list(entry)
  axioms    sort(s) = s'  <== bubble(s) = s' /\ s = s'
            sort(s) = sort(s') <== bubble(s) = s' /\ s =/= s'
```

```
bubble([]) = []
bubble([x]) = [x]
bubble(x:y:s) = x:bubble(y:s) <== x <= y
bubble(x:y:s) = y:bubble(x:s) <== x > y
```

- ☞ Optimieren Sie BUBBLESORT durch Erweiterung des Wertebereiches von *bubble* um einen Booleschen Wert (*flag*), der angibt, ob der jeweilige Aufruf von *bubble* die Argumentliste verändert hat! Anstatt die Bedingung  $s = s'$  bzw.  $s \neq s'$  zu überprüfen, braucht *sort* dann nur den Wert des flags abzufragen. Zeigen Sie, dass Ihr optimiertes *bubble* zum *bubble* von BUBBLESORT äquivalent ist!

### [13], Beispiel 7.2.8: Rucksackproblem

Gegeben sind ein Rucksack der Größe  $b$  und  $n > 0$  Objekte mit Größen  $a_1, \dots, a_n$  und Nutzwerten  $c_1, \dots, c_n$ . Gesucht ist eine Teilmenge  $M$  von  $\{1, \dots, n\}$  mit  $\sum_{i=1}^n a_i x_i \leq b$  und maximalem Gesamtnutzen  $c(\vec{x}) =_{def} \sum_{i=1}^n c_i x_i$ .  $\vec{x} \in \{0, 1\}^n$  ist der charakteristische Vektor von  $M$ , d.h.  $x_i = 1 \Leftrightarrow i \in M$ . Der Aufruf  $knap(n, b)$  des folgenden, leider exponentiellen Haskell-Programms  $knap$  berechnet  $\vec{x}$  und  $c(\vec{x})$  rekursiv. Vektoren werden hier als Listen implementiert.

```

knap(1, b) = if a(1) > b then ([0], 0) else ([1], c(1))
knap(n, b) = if d1 > d2' then (v1++[0], d1) else (v2++[1], d2')
            where (v1, d1) = knap(n-1, b)
                  (v2, d2) = knap(n-1, b-a(n))
                  d2' = d2+c(n)
    
```

☞ Zeigen Sie die Korrektheit von  $knap$  durch Fixpunktinduktion! Die zu beweisende Behauptung lautet als Formel wie folgt:

$$knap(k, a) = \vec{x} \implies \left( \sum_{i=1}^n a_i x_i \leq b \wedge \forall \vec{y} : \left( \sum_{i=1}^n a_i y_i \leq b \implies \sum_{i=1}^n c_i y_i \leq \sum_{i=1}^n c_i x_i \right) \right).$$

☞ Zeigen Sie  $unsorted([1, 2, 3, 5, 4])$  erstens mit Coinduktion über  $unsorted$  und zweitens durch Widerlegung, d.h. durch Ableitung der Formel  $sorted([1, 2, 3, 5, 4]) \Rightarrow False$  nach  $True$ !

[13], Beispiel 7.3.1: MAP implementiert STACK, cont.

$$i > j \implies (\text{upd}(i, x, f), j) \sim (f, j)$$

Applying coinduction w.r.t.

$$s \sim s' \implies \text{top}(s) = \text{top}(s') \ \& \ \text{pop}(s) \sim \text{pop}(s')$$

at position [] of the preceding formula leads to

$$\text{All } i \ j \ x \ f: (i > j \implies \text{top}(\text{upd}(i, x, f), j) = \text{top}(f, j)) \ \&$$

$$\text{All } i \ j \ x \ f: (i > j \implies$$

$$\text{Any } i_0 \ j_0 \ x_0 \ f_0: (i_0 > j_0 \ \& \ \text{pop}(\text{upd}(i, x, f), j) = (\text{upd}(i_0, x_0, f_0), j_0) \ \& \\ \text{pop}(f, j) = (f_0, j_0)))$$

Reducts have been simplified.

Applying the axioms

$$\text{pop}(f, i) = (f, \text{pred}(i))$$

$$\& \text{top}(f, i) = \text{get}(f, i)$$

at positions [1,0,1,0,2,0], [1,0,1,0,1,0], [0,0,1,1], [0,0,1,0]

of the preceding formula leads to

All i j:(i > j ==> i > pred(j)) &  
All i j x f:(i > j ==> get(upd(i,x,f),j) = get(f,j))

Reducts have been simplified.

Applying the axioms

get(upd(i,x,f),i) = entry(x)  
& (get(upd(i,x,f),j) = get(f,j) <==> i =/ j)

at position [1,0,1,0] of the preceding formula leads to

All i j:(i > j ==> i > pred(j))

Reducts have been simplified.

Applying the theorem

i > j ==> i > pred(j)

at position [0,0] of the preceding formula leads to

True

### [13], Beispiel 7.4.1: Merge

Wir zeigen:

$$\text{sorted}(s_1) \wedge \text{sorted}(s_2) \Rightarrow \text{sorted}(\text{merge}(s_1, s_2)). \quad (1)$$

(1) beschreibt eine typische **Invarianzbedingung**: Wird *merge* auf sortierte Listen angewendet, dann ist auch die Ergebnisliste sortiert. Zum Beweis mit Fixpunktinduktion muss (1) generalisiert werden:

$$\text{merge}(s_1, s_2) = s \wedge \text{sorted}(s_1) \wedge \text{sorted}(s_2) \Rightarrow \text{sorted}(s) \wedge s \sim s_1 + + s_2. \quad (2)$$

(2) ist äquivalent zur Konjunktion von Lemma (1) und Lemma (3) aus Beispiel 7.2.6.

$$\text{merge}(s_1, s_2) = s \wedge \text{sorted}(s_1) \wedge \text{sorted}(s_2) \Rightarrow \text{sorted}(s) \wedge s \sim s_1 + + s_2$$

Shifting subformulas at positions [0,1], [0,2] of the preceding formula leads to

$$\text{merge}(s_1, s_2) = s \Rightarrow (\text{sorted}(s_1) \wedge \text{sorted}(s_2) \Rightarrow \text{sorted}(s) \wedge s \sim s_1 + + s_2)$$

Applying fixpoint induction w.r.t.

$$\begin{aligned} \text{merge}([], s) &= s \wedge \text{merge}(s, []) = s \wedge \\ (\text{merge}(x:s, y:s') &= x:z3 \iff x \leq y \wedge \text{merge}(s, y:s') = z3) \wedge \\ (\text{merge}(x:s, y:s') &= y:z3 \iff x > y \wedge \text{merge}(x:s, s') = z3) \end{aligned}$$

at position [] of the preceding formula leads to

$$\begin{aligned} \text{All } s \ x \ y \ s' \ z3: & ((\text{sorted}[] \wedge \text{sorted}(s) \Rightarrow \text{sorted}(s) \wedge s \sim [] + + s) \wedge \\ & (\text{sorted}(s) \wedge \text{sorted}[] \Rightarrow \text{sorted}(s) \wedge s \sim s + + []) \wedge \end{aligned}$$

$$\begin{aligned}
& ((\text{sorted}(x:s) \ \& \ \text{sorted}(y:s') \implies \\
& \quad \text{sorted}(x:z3) \ \& \ x:z3 \sim x:s+y:s') \iff \\
& \quad x \leq y \ \& \\
& \quad (\text{sorted}(s) \ \& \ \text{sorted}(y:s') \implies \text{sorted}(z3) \ \& \ z3 \sim s+y:s')) \ \& \\
& ((\text{sorted}(x:s) \ \& \ \text{sorted}(y:s') \implies \\
& \quad \text{sorted}(y:z3) \ \& \ y:z3 \sim x:s+y:s') \iff \\
& \quad x > y \ \& \\
& \quad (\text{sorted}(x:s) \ \& \ \text{sorted}(s') \implies \text{sorted}(z3) \ \& \ z3 \sim x:s+s')) \\
\end{aligned}$$

Simplifying the preceding formula (33 steps) leads to

$$\begin{aligned}
& \text{All } s \ x \ y \ s' \ z3: (x \leq y \ \& \ (\text{sorted}(s) \ \& \ \text{sorted}(y:s') \implies \text{sorted}(z3)) \ \& \\
& \quad (\text{sorted}(s) \ \& \ \text{sorted}(y:s') \implies z3 \sim s+y:s') \ \& \ \text{sorted}(x:s) \ \& \\
& \quad \text{sorted}(y:s') \implies \\
& \quad \text{sorted}(x:z3)) \ \& \\
& \text{All } s \ x \ y \ s' \ z3: (x \leq y \ \& \ (\text{sorted}(s) \ \& \ \text{sorted}(y:s') \implies \text{sorted}(z3)) \ \& \\
& \quad (\text{sorted}(s) \ \& \ \text{sorted}(y:s') \implies z3 \sim s+y:s') \ \& \ \text{sorted}(x:s) \ \& \\
& \quad \text{sorted}(y:s') \implies \\
& \quad x:z3 \sim x:s+y:s') \ \& \\
& \text{All } s \ x \ y \ s' \ z3: (x > y \ \& \ (\text{sorted}(x:s) \ \& \ \text{sorted}(s') \implies \text{sorted}(z3)) \ \& \\
& \quad (\text{sorted}(x:s) \ \& \ \text{sorted}(s') \implies z3 \sim x:s+s') \ \& \ \text{sorted}(x:s) \ \& \\
& \quad \text{sorted}(y:s') \implies \\
& \quad \text{sorted}(y:z3)) \ \& \\
& \text{All } s \ x \ y \ s' \ z3: (x > y \ \& \ (\text{sorted}(x:s) \ \& \ \text{sorted}(s') \implies \text{sorted}(z3)) \ \& \\
& \quad (\text{sorted}(x:s) \ \& \ \text{sorted}(s') \implies z3 \sim x:s+s') \ \& \ \text{sorted}(x:s) \ \&
\end{aligned}$$

```
sorted(y:s') ==>  
y:z3 ~ x:s++y:s')
```

Applying the axiom resp. theorem

```
sorted(s) <==> sorted(x:s)
```

at positions [0,0,2,0,0], [0,0,1,0,0] of the preceding formula leads to 4 factors.  
The current factor is given by

```
All s x y s' z3:(x <= y & (Any x0:(sorted(x0:s)) & sorted(y:s') ==> sorted(z3)) &  
(Any x1:(sorted(x1:s)) & sorted(y:s') ==> z3 ~ s++y:s') &  
sorted(x:s) & sorted(y:s') ==>  
sorted(x:z3))
```

Simplifying the preceding factors (2 steps) leads to the factor

```
All s x y s' z3:(sorted(x:s) & sorted(y:s') & x <= y & sorted(z3) &  
z3 ~ s++y:s' ==>  
sorted(x:z3))
```

Applying the axiom resp. theorem

```
sorted(x:s1) <==>  
sorted(x:s) & sorted(y:s') & x <= y & sorted(s1) & s1 ~ s++y:s'
```

at position [0,1] of the preceding factors leads to the factor

```
All s x y s' z3:(sorted(x:s) & sorted(y:s') & x <= y & sorted(z3) &
z3 ~ s++y:s' ==>
Any s9 y0 s'0:(sorted(x:s9) & sorted(y0:s'0) & x <= y0 &
sorted(z3) & z3 ~ s9++y0:s'0))
```

Simplifying the preceding factors (2 steps) leads to 3 factors.

The current factor is given by

```
All s x y s' z3:(x <= y & (sorted(s) & sorted(y:s') ==> sorted(z3)) &
(sorted(s) & sorted(y:s') ==> z3 ~ s++y:s') & sorted(x:s) &
sorted(y:s') ==>
x:z3 ~ x:s++y:s')
```

Applying the axiom resp. theorem

```
sorted(s) <==> sorted(x:s)
```

at position [0,0,2,0,0] of the preceding factors leads to the factor

```
All s x y s' z3:(x <= y & (sorted(s) & sorted(y:s') ==> sorted(z3)) &
(Any x3:(sorted(x3:s)) & sorted(y:s') ==> z3 ~ s++y:s') &
sorted(x:s) & sorted(y:s') ==>
```

$x:z3 \sim x:s++y:s')$

Simplifying the preceding factors leads to the factor

All  $s x y s' z3 : (sorted(x:s) \& sorted(y:s') \& x \leq y \&$   
 $(sorted(s) \& sorted(y:s') ==> sorted(z3)) \& z3 \sim s++y:s' ==>$   
 $x:z3 \sim x:s++y:s')$

Decomposing the atom at position [0,1] of the preceding factors leads to the factor

All  $s x y s' z3 : (sorted(x:s) \& sorted(y:s') \& x \leq y \&$   
 $(sorted(s) \& sorted(y:s') ==> sorted(z3)) \& z3 \sim s++y:s' ==>$   
 $z3 \sim s++y:s')$

Simplifying the preceding factors (2 steps) leads to 2 factors.

The current factor is given by

All  $s x y s' z3 : (x > y \& (sorted(x:s) \& sorted(s') ==> sorted(z3)) \&$   
 $(sorted(x:s) \& sorted(s') ==> z3 \sim x:s++s') \& sorted(x:s) \&$   
 $sorted(y:s') ==>$   
 $sorted(y:z3))$

Applying the theorem

$x > y ==> y \leq x$

at position [0,0,0] of the preceding factors leads to the factor

```
All s x y s' z3:(y <= x & (sorted(x:s) & sorted(s') ==> sorted(z3)) &
                  (sorted(x:s) & sorted(s') ==> z3 ~ x:s++s') & sorted(x:s) &
                  sorted(y:s') ==>
                  sorted(y:z3))
```

Applying the axiom resp. theorem

```
sorted(s) <==> sorted(x:s)
```

at positions [0,0,2,0,1], [0,0,1,0,1] of the preceding factors leads to the factor

```
All s x y s' z3:(y <= x & (sorted(x:s) & Any x5:(sorted(x5:s')) ==> sorted(z3)) &
                  (sorted(x:s) & Any x6:(sorted(x6:s')) ==> z3 ~ x:s++s') &
                  sorted(x:s) & sorted(y:s') ==>
                  sorted(y:z3))
```

Simplifying the preceding factors (2 steps) leads to the factor

```
All s x y s' z3:(sorted(x:s) & sorted(y:s') & y <= x & sorted(z3) &
                  z3 ~ x:s++s' ==>
                  sorted(y:z3))
```

Applying the axiom resp. theorem

sorted(x:s1) <====  
sorted(x:s) & sorted(y:s') & x <= y & sorted(s1) & s1 ~ s++y:s'

at position [0,1] of the preceding factors leads to the factor

All s x y s' z3:(sorted(x:s) & sorted(y:s') & y <= x & sorted(z3) &  
z3 ~ x:s++s' ==>  
Any s13 y2 s'1:(sorted(y:s13) & sorted(y2:s'1) & y <= y2 &  
sorted(z3) & z3 ~ s13++y2:s'1))

Simplifying the preceding factors (3 steps) leads to the factor

All s x y s' z3:(sorted(x:s) & sorted(y:s') & y <= x & sorted(z3) &  
z3 ~ x:s++s' ==>  
Any s13 y2 s'1:(sorted(y:s13) & z3 ~ s13++y2:s'1 &  
sorted(y2:s'1) & y <= y2))

Applying the theorem

s'++x:s ~ x:s++s'

at position [0,0,4] of the preceding factors leads to the factor

```

All s x y s' z3:(sorted(x:s) & sorted(y:s') & y <= x & sorted(z3) &
z3 = s'++x:s ==>
Any s13 y2 s'1:(sorted(y:s13) & z3 ~ s13++y2:s'1 &
sorted(y2:s'1) & y <= y2))

```

Simplifying the preceding factors (2 steps) leads to the factor

```

All s x y s':(sorted(x:s) & sorted(y:s') & y <= x & sorted(s'++x:s) ==>
Any s13 y2 s'1:(sorted(y:s13) & s'++x:s ~ s13++y2:s'1 &
sorted(y2:s'1) & y <= y2))

```

Substituting s' for s13 at position [0,1,0,1,0,0] of the preceding factors  
leads to the factor

```

All s x y s':(sorted(x:s) & sorted(y:s') & y <= x & sorted(s'++x:s) ==>
Any s13 y2 s'1:(sorted(y:s') & s'++x:s ~ s'++y2:s'1 &
sorted(y2:s'1) & y <= y2))

```

Substituting x for y2 at position [0,1,0,1,0,1,0] of the preceding factors  
leads to the factor

```

All s x y s':(sorted(x:s) & sorted(y:s') & y <= x & sorted(s'++x:s) ==>
Any s13 y2 s'1:(sorted(y:s') & s'++x:s ~ s'++x:s'1 &
sorted(x:s'1) & y <= x))

```

Substituting s for s' at position [0,1,0,1,0,1,1] of the preceding factors leads to the factor

All s x y s':(sorted(x:s) & sorted(y:s') & y <= x & sorted(s'++x:s) ==>  
Any s13 y2 s'1:(sorted(y:s') & s'++x:s ~ s'++x:s & sorted(x:s) &  
y <= x))

Simplifying the preceding factors (6 steps) leads to a single formula, which is given by

All s x y s' z3:(x > y & (sorted(x:s) & sorted(s') ==> sorted(z3)) &  
(sorted(x:s) & sorted(s') ==> z3 ~ x:s++s') & sorted(x:s) &  
sorted(y:s') ==>  
y:z3 ~ x:s++y:s')

Applying the theorem

$x > y \implies y \leq x$

at position [0,0,0] of the preceding formula leads to

All s x y s' z3:(y <= x & (sorted(x:s) & sorted(s') ==> sorted(z3)) &  
(sorted(x:s) & sorted(s') ==> z3 ~ x:s++s') & sorted(x:s) &  
sorted(y:s') ==>  
y:z3 ~ x:s++y:s')

Applying the axiom resp. theorem

$\text{sorted}(s) \leqslant \text{sorted}(x:s)$

at position [0,0,2,0,1] of the preceding formula leads to

All  $s\ x\ y\ s'\ z3:(y \leq x \ \& (\text{sorted}(x:s) \ \& \ \text{sorted}(s') \implies \text{sorted}(z3)) \ \&$   
 $(\text{sorted}(x:s) \ \& \ \text{Any } x10:(\text{sorted}(x10:s')) \implies z3 \sim x:s++s') \ \&$   
 $\text{sorted}(x:s) \ \& \ \text{sorted}(y:s') \implies$   
 $y:z3 \sim x:s++y:s')$

Simplifying the preceding formula leads to

All  $s\ x\ y\ s'\ z3:(\text{sorted}(x:s) \ \& \ \text{sorted}(y:s') \ \& \ y \leq x \ \&$   
 $(\text{sorted}(x:s) \ \& \ \text{sorted}(s') \implies \text{sorted}(z3)) \ \& \ z3 \sim x:s++s' \implies$   
 $y:z3 \sim x:s++y:s')$

A transitivity axiom at position [0,1] of the preceding formula leads to

All  $s\ x\ y\ s'\ z3:(\text{sorted}(x:s) \ \& \ \text{sorted}(y:s') \ \& \ y \leq x \ \&$   
 $(\text{sorted}(x:s) \ \& \ \text{sorted}(s') \implies \text{sorted}(z3)) \ \& \ z3 \sim x:s++s' \implies$   
 $\text{Any } z4:(y:z3 \sim z4 \ \& \ z4 \sim x:s++y:s'))$

Applying the theorem

$y:x:s++s' \sim x:s++y:s'$

at position [0,1,0,1] of the preceding formula leads to

All  $s\ x\ y\ s'\ z3:(sorted(x:s) \ \& \ sorted(y:s') \ \& \ y \leq x \ \&$   
 $(sorted(x:s) \ \& \ sorted(s') \ ==> sorted(z3)) \ \& \ z3 \sim x:s++s' ==>$   
Any  $z4:(y:z3 \sim z4 \ \& \ z4 = y:x:s++s'))$

Substituting  $y:x:s++s'$  for  $z4$  at position [0,1,0,1,1] of the preceding formula leads to

All  $s\ x\ y\ s'\ z3:(sorted(x:s) \ \& \ sorted(y:s') \ \& \ y \leq x \ \&$   
 $(sorted(x:s) \ \& \ sorted(s') \ ==> sorted(z3)) \ \& \ z3 \sim x:s++s' ==>$   
Any  $z4:(y:z3 \sim y:x:s++s' \ \& \ y:x:s++s' = y:x:s++s'))$

Simplifying the preceding formula (6 steps) leads to

All  $s\ x\ y\ s'\ z3:(sorted(x:s) \ \& \ sorted(y:s') \ \& \ y \leq x \ \&$   
 $(sorted(x:s) \ \& \ sorted(s') \ ==> sorted(z3)) \ \& \ z3 \sim x:s++s' ==>$   
 $y:z3 \sim y:x:s++s')$

Decomposing the atom at position [0,1] of the preceding formula leads to

All  $s\ x\ y\ s'\ z3:(sorted(x:s) \ \& \ sorted(y:s') \ \& \ y \leq x \ \&$

```
(sorted(x:s) & sorted(s') ==> sorted(z3)) & z3 ~ x:s++s' ==>
z3 ~ x:s++s')
```

Simplifying the preceding formula (2 steps) leads to

True

Number of proof steps: 31

[13], Beispiel 7.4.2: 2-3-Bäume *insert* trägt ein Element in einen balancierten binären Baum ein und rebalanciert anschließend den Ergebnisbaum.

2-3-TREE[ORD(s)] and NAT then

sorts	tree	
constructs	mt:->tree	- leerer Baum
	_#_#:tree*s*tree->tree	- Wurzel hat 2 Nachfolger
	_#_#_#_#:tree*s*tree*s*tree->tree	- Wurzel hat 3 Nachfolger
defuncts	insert:s*tree->(tree*nat)	
	h:tree->nat	- Baumhöhe
preds	balanced:tree	
vars	x,y,z,z':s T1,T2,T3,T4,T5:tree	
axioms	insert(x,mt) = (mt#x#mt,1)	
	insert(x,T1#x#T2) = (T1#x#T2,0)	
	insert(x,T1#x#T2#y#T3) = (T1#x#T2#y#T3,0)	
	insert(x,T1#y#T2#x#T3) = (T1#y#T2#x#T3,0)	
	insert(x,T1#y#T2) = (T3#y#T2,0) ⇐ x < y ∧ insert(x,T1) = (T3,0)	

```

insert(x,T1#y#T2) = (T1#y#T3,0)    ⇐   y < x ∧ insert(x,T2) = (T3,0)
insert(x,T1#y#T2#z#T3) = (T4#y#T2#z#T3,0)
                                ⇐   x < y ∧ insert(x,T1) = (T4,0)
insert(x,T1#y#T2#z#T3) = (T4#y#T2#z#T3,0)
                                ⇐   y < x < z ∧ insert(x,T2) = (T4,0)
insert(x,T1#y#T2#z#T3) = (T1#y#T2#z#T4,0)
                                ⇐   z < x ∧ insert(x,T3) = (T4,0)
insert(x,T1#y#T2) = (T3#z#T4#y#T2,0)
                                ⇐   x < y ∧ insert(x,T1) = (T3#z#T4,1)
insert(x,T1#y#T2) = (T1#y#T3#z#T4,0)
                                ⇐   y < x ∧ insert(x,T2) = (T3#z#T4,1)
insert(x,T1#y#T2#z#T3) = (T4#y#(T2#z#T3),1)
                                ⇐   x < y ∧ insert(x,T1) = (T4#z'#T5,1)
insert(x,T1#y#T2#z#T3) = ((T1#y#T4)#z' #(T5#z#T3),1)
                                ⇐   y < x < z ∧ insert(x,T2) = (T4#z'#T5,1)
insert(x,T1#y#T2#z#T3) = ((T1#y#T2)#z#T4,1)
                                ⇐   z < x ∧ insert(x,T3) = (T4#z'#T5,1)

h(mt) = 0
h(T1#x#T2) = max(h(T1),h(T2))+1
h(T1#x#T2#y#T3) = max(h(T1),h(T2),h(T3))+1
balanced(mt)
balanced(T1#x#T2) ⇐ h(T1) = h(T2) ∧ balanced(T1) ∧ balanced(T2)
balanced(T1#x#T2#y#T3)
                ⇐ h(T1) = h(T2) = h(T3) ∧ balanced(T1) ∧ balanced(T2)
                                ∧ balanced(T3)

```

☞ Zeigen Sie folgende Invarianzbedingungen durch Fixpunktinduktion über *insert*:

$$\begin{aligned} \text{insert}(x, T) \equiv (T', 0) \wedge \text{balanced}(T) &\Rightarrow h(T) \equiv h(T') \wedge \text{balanced}(T') \\ \text{insert}(x, T) \equiv (T', 1) \wedge \text{balanced}(T) &\Rightarrow h(T) + 1 \equiv h(T') \wedge \text{balanced}(T'). \end{aligned}$$

[13], Beispiel 7.4.3: AVL-Bäume *insert* trägt ein Element in einen balancierten binären Baum ein und rebalanciert anschließend den Ergebnisbaum.

```

AVL-TREE[ORD(s)] = NAT then
  sorts      bintree    nonempty
  constructs mt:->tree
               _ : nonempty->tree
               _#_#_ : tree*s*tree->nonempty
               □#_#_ : nonempty*s*tree->nonempty
               _#_#□ : tree*s*nonempty->nonempty
  defuncts  insert : s*tree->nonempty
               □#_#_ : nonempty*s*tree->nonempty
               _#_#□ : tree*s*nonempty->nonempty
               h : tree->nat
  preds      balanced : tree
  vars       x,y,z:s   T1,T2,T3,T4:tree
  axioms     insert(x,mt) = mt#x#mt
             insert(x,T1#x#T2) = T1#x#T2
             insert(x, □#x#T2) = □#x#T2
             insert(x,T1#x#□) = T1#x#□
             insert(x,T1#y#T2) = T3#y#T2  ⇐  x < y ∧ insert(x,T1) = T3

```

- leerer Baum
- nichtleerer Baum
- ausgeglichener Baum
- linkslastiger Baum
- rechtslastiger Baum
- Rotation von links her
- Rotation von rechts her
- Baumhöhe

$$\begin{aligned}
 & \wedge h(T1) = h(T3) \\
 \text{insert}(x, T1 \# y \# T2) = T1 \# y \# T3 & \Leftarrow y < x \wedge \text{insert}(x, T2) = T3 \\
 & \wedge h(T1) = h(T3) \\
 \text{insert}(x, \boxed{T1} \# y \# T2) = \boxed{T3} \# y \# T2 & \Leftarrow x < y \wedge \text{insert}(x, T1) = T3 \\
 & \wedge h(T1) = h(T3) \\
 \text{insert}(x, \boxed{T1} \# y \# T2) = \boxed{T1} \# y \# T3 & \Leftarrow y < x \wedge \text{insert}(x, T2) = T3 \\
 & \wedge h(T1) = h(T3) \\
 \text{insert}(x, T1 \# y \# \boxed{T2}) = T3 \# y \# \boxed{T2} & \Leftarrow x < y \wedge \text{insert}(x, T1) = T3 \\
 & \wedge h(T1) = h(T3) \\
 \text{insert}(x, T1 \# y \# \boxed{T2}) = T1 \# y \# \boxed{T3} & \Leftarrow y < x \wedge \text{insert}(x, T2) = T3 \\
 & \wedge h(T1) = h(T3) \\
 \text{insert}(x, T1 \# y \# T2) = \boxed{T3} \# y \# T2 & \Leftarrow x < y \wedge \text{insert}(x, T1) = T3 \\
 & \wedge h(T1) \neq h(T3) \\
 \text{insert}(x, T1 \# y \# T2) = T1 \# y \# \boxed{T3} & \Leftarrow y < x \wedge \text{insert}(x, T2) = T3 \\
 & \wedge h(T2) \neq h(T3) \\
 \text{insert}(x, \boxed{T1} \# y \# T2) = \boxed{T3} \# y \# T2 & \Leftarrow x < y \wedge \text{insert}(x, T1) = T3 \\
 & \wedge h(T1) \neq h(T3) \\
 \text{insert}(x, \boxed{T1} \# y \# T2) = T1 \# y \# T3 & \Leftarrow y < x \wedge \text{insert}(x, T2) = T3 \\
 & \wedge h(T2) \neq h(T3) \\
 \text{insert}(x, T1 \# y \# \boxed{T2}) = T3 \# y \# T2 & \Leftarrow x < y \wedge \text{insert}(x, T1) = T3 \\
 & \wedge h(T1) \neq h(T3) \\
 \text{insert}(x, T1 \# y \# \boxed{T2}) = T1 \# y \# \boxed{T3} & \Leftarrow y < x \wedge \text{insert}(x, T2) = T3 \\
 & \wedge h(T2) \neq h(T3)
 \end{aligned}$$

$$(1) \quad \boxed{T1 \# x \# T2} \# y \# T3 = T1 \# x \# \boxed{T2} \# y \# T3$$

$$(2) \quad \boxed{T1 \# x \# T2} \# y \# T3 = T1 \# x \# (T2 \# y \# T3)$$

$$(3) \quad T1\#x\# \boxed{T2\#y\#T3} \#z\#T4 = (T1\#x\#T2)\#y\#(T3\#z\#T4)$$

$$(4) \quad T1\#x\# \boxed{T2\#y\#T3} \#z\#T4 = (T1\#x\#T2)\#y\#(T3\#z\#\boxed{T4})$$

$$(5) \quad T1\#x\# \boxed{T2\#y\#T3} \#z\#T4 = (\boxed{T1}\#x\#T2)\#y\#(T3\#z\#T4)$$

$$(6) \quad T1\#x\# \boxed{T2\#y\#T3} = \boxed{T1\#x\#T2}\#y\#T3$$

$$(7) \quad T1\#x\# \boxed{T2\#y\#T3} = (T1\#x\#T2)\#y\#T3$$

$$(8) \quad T1\#x\# \boxed{T2\#y\#T3} \#z\#T4 = (T1\#x\#T2)\#y\#(T3\#z\#T4)$$

$$(9) \quad T1\#x\# \boxed{T2\#y\#T3} \#z\#T4 = (\boxed{T1}\#x\#T2)\#y\#(T3\#z\#T4)$$

$$(10) \quad T1\#x\# \boxed{T2\#y\#T3} \#z\#T4 = (T1\#x\#T2)\#y\#(T3\#z\#\boxed{T4})$$

$$h(mt) = 0$$

$$h(T1\#x\#T2) = \max(h(T1), h(T2)) + 1$$

$$h(\boxed{T1}\#x\#T2) = \max(h(T1), h(T2)) + 1$$

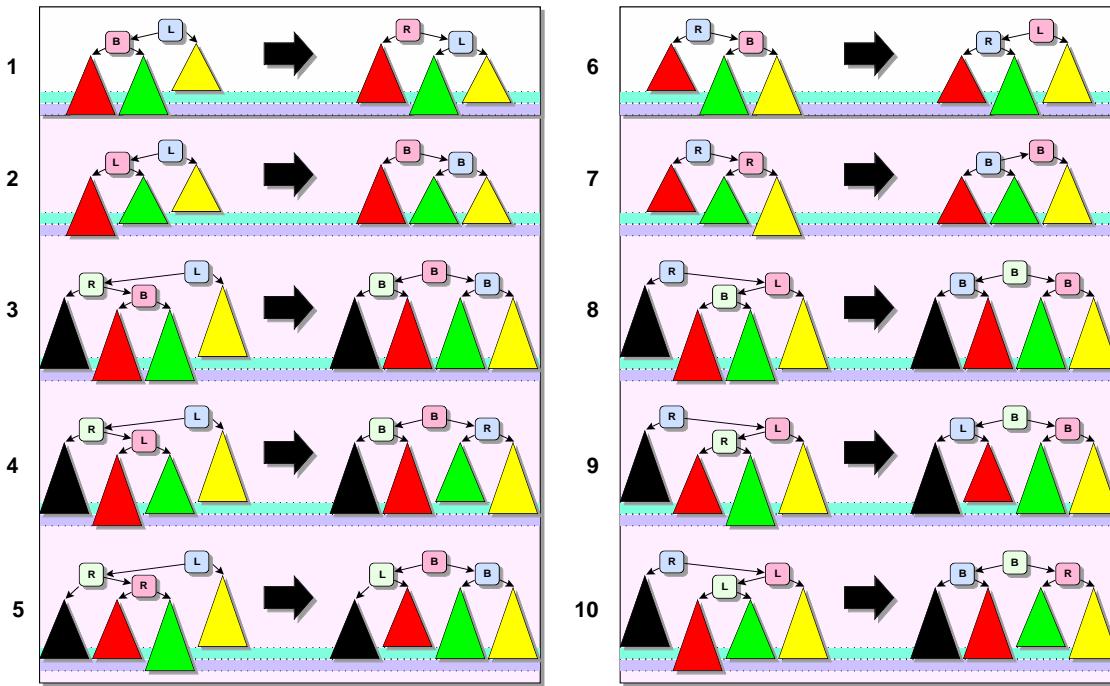
$$h(T1\#x\#\boxed{T2}) = \max(h(T1), h(T2)) + 1$$

`balanced(mt)`

$$\text{balanced}(T1\#x\#T2) \Leftarrow h(T1) = h(T2) \wedge \text{balanced}(T1) \wedge \text{balanced}(T2)$$

$$\text{balanced}(\boxed{T1}\#x\#T2) \Leftarrow h(T1) = h(T2) + 1 \wedge \text{balanced}(T1) \wedge \text{balanced}(T2)$$

$$\text{balanced}(T1\#x\#\boxed{T2}) \Leftarrow h(T1) + 1 = h(T2) \wedge \text{balanced}(T1) \wedge \text{balanced}(T2)$$



Die 10 Rotationsfälle (nach A. Hallmann)

Wieder ist die wesentliche Anforderung an *insert* eine Invarianzbedingung:

$$\text{balanced}(T) \Rightarrow \text{balanced}(\text{insert}(x, T)).$$

☞ Zeigen Sie durch Fixpunktinduktion über *insert* und die beiden Rotationsfunktionen:

$$\text{insert}(x, T) \equiv T' \Rightarrow (\text{balanced}(T) \Rightarrow \text{balanced}(T'))$$

$$T1 \# x \# T2 \equiv T \Rightarrow ((\text{balanced}(T1) \wedge \text{balanced}(T2) \wedge h(T1) \equiv h(T2) + 2) \Rightarrow \text{balanced}(T))$$

$$T1 \# x \# T2 \equiv T \Rightarrow ((\text{balanced}(T1) \wedge \text{balanced}(T2) \wedge h(T1) + 2 \equiv h(T2)) \Rightarrow \text{balanced}(T)).$$

- ☞ Optimieren Sie – ähnlich der Einfügeoperation bei 2-3-Bäumen (s. Bsp. ??) – *insert* durch Erweiterung des Wertebereiches der Funktion um einen Booleschen Wert (*flag*), der angibt, ob der jeweilige Aufruf von *insert* die Höhe des Argumentbaumes verändert hat! Anstatt die Bedingungen  $h(T1) = h(T3)$ ,  $h(T1) \neq h(T3)$ , etc., zu überprüfen, braucht *insert* dann nur den Wert des flags abzufragen. Zeigen Sie, dass Ihr optimiertes *insert* zum *insert* von AVL-TREE äquivalent ist!
- ☞ Spezifizieren Sie eine Operation  $\text{remove}: s \times \text{nonempty} \rightarrow \text{tree}$ , die Einträge entfernt und wie *insert* die Balanciertheit von Bäumen erhält! Da eine Löschoperation die Höhe eines Baumes verringern kann, müssen ggf. Teilbäume mehrfach rotiert werden, was zu rekursiven Aufrufen der Rotationsfunktionen führt.

### [13], Beispiel 7.4.4: Baumordnungen

Infixordnung (auch Suchbaumeigenschaft genannt) und partielle Ordnung sind zwei Arten, die Struktur eines binären Baumes auszunutzen, um Einträge zu sortieren und damit die Zugriffszeit zu verkürzen.<sup>2</sup> Für Suchbäume bieten sich Konstruktoren an, die den leeren Baum ausschließen.

```

TREE-ORDERS [ORD(s)] = AVL-TREE [ORD(s)] then
    sorts      searchtree
    constructs leaf:s -> searchtree
                _&_:_:searchtree*s*searchtree -> searchtree
    defuncts   minT,maxT:searchtree -> s
                bin:searchtree -> tree
    preds      infix:searchtree
                parti:tree
  
```

<sup>2</sup> Alle Axiome mit dem Konstruktor  $\#-\#-$  gibt es implizit auch für  $\square\#-\#-\square$  und  $\#-\#\square$ .

```

axioms      infix(leaf(x))
            infix(T&x&T') <== infix(T) /\ maxT(T) <= x /\ x <= minT(T') /\ in
            minT(leaf(x)) = x
            minT(T&x&T') = (min(minT(T),min(x,minT(T'))))
            maxT(leaf(x)) = x
            maxT(T&x&T') = (max(maxT(T),max(x,maxT(T'))))
            bin(leaf(x)) = mt#x#mt
            bin(T&x&T') = bin(T)#x#bin(T')
            parti(mt)
            parti(mt#x#(T1#y#T2))           <== x <= y /\ parti(T1#x#T2)
            parti((T1#x#T2)#y#mt)          <== y <= x /\ parti(T1#x#T2)
            parti((T1#x#T2)#y#(T3#z#T4)) <== y <= x /\ parti(T1#x#T2) /\
                                         y <= z /\ parti(T3#z#T4)

```

☞ Zeigen Sie, dass die Invarianzbedingung

$$infix(T) \wedge insert(x, bin(T)) \equiv bin(T') \Rightarrow infix(T')$$

ein induktives Theorem von TREE-ORDERS ist, dass also *insert* die Infixordnung erhält!

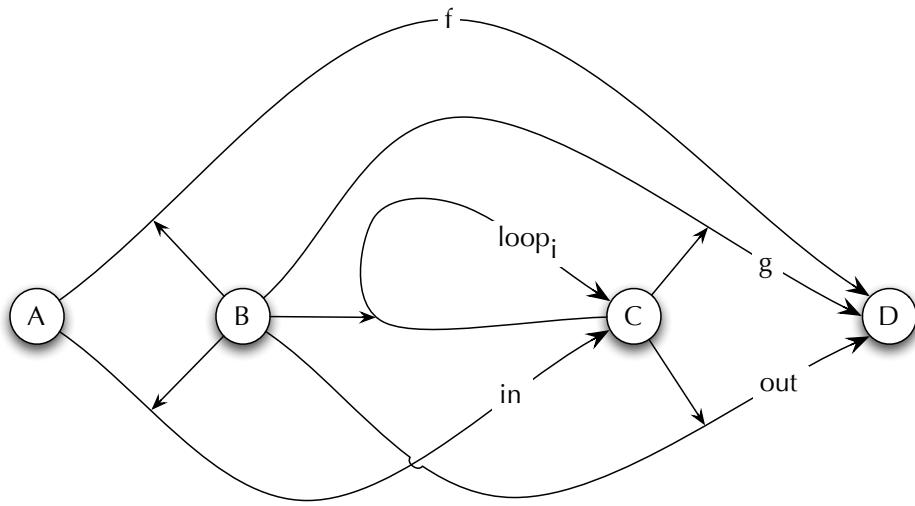
☞ Zeigen Sie, dass auch Ihre spezifizierte Löschfunktion *remove* Suchbäume in Suchbäume überführt!

☞ Zeigen Sie, dass die Invarianzbedingung

$$parti(T1) \wedge parti(T2) \Rightarrow parti(sift(T1#x#T2))$$

ein induktives Theorem von HEAP ist (siehe [13], Beispiel 6.4.5)! Folgern Sie daraus die Gültigkeit von *parti(heapify(T))*!

## [13], §7.5: Schleifenvarianten



Die Funktionen eines iterativen Programms: die Ausgangsfunktion  $f$ , deren Schleifenfunktion  $g$ , die Initialisierung  $in$  der Schleifenvariablen, deren mögliche Modifikationen  $loop_i$  und die Projektion vom Schleifenbereich  $C$  in den Ausgabebereich  $D$ . Eine bestimmte Struktur der Axiome für eine Funktion oder ein Prädikat nennt man auch **Programmschema**. Eins der gebräuchlichsten ist das für iterative Programme charakteristische, das aus vier Sorten  $A$ ,  $B$ ,  $C$  und  $D$ , einer Funktion  $f : A \times B \rightarrow D$ , einer **Schleifenfunktion**  $g : B \times C \rightarrow D$ , Hilfsfunktionen  $in : A \times B \rightarrow C$ ,  $out : B \times C \rightarrow D$  und

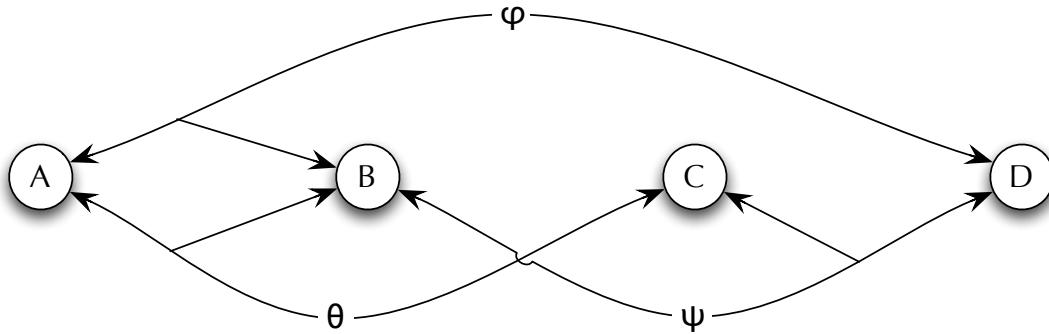
$loop_1, \dots, loop_n : B \times C \rightarrow C$  sowie einer Axiomenmenge der Form

$$\begin{aligned} f(a, b) &\equiv g(b, in(a, b)) \\ g(b, c) &\equiv g(b, loop_1(b, c)) \Leftarrow \delta_1(b, c) \\ \dots \\ g(b, c) &\equiv g(b, loop_n(b, c)) \Leftarrow \delta_n(b, c) \\ g(b, c) &\equiv out(b, c) \Leftarrow \delta(b, c) \end{aligned} \tag{1}$$

besteht. Dass (1) tatsächlich einer iterativen Definition entspricht, zeigt die folgende Implementierung der Axiome als Funktionsprozedur mit Schleifenrumpf:

```
function f(a:A,b:B):D;
begin c:C;
  c:=in(a,b);
  while δ_1(b,c) ∨ … ∨ δ_n(b,c)
    do if δ_1(b,c) then c:=loop_1(b,c);
       ...
       if δ_n(b,c) then c:=loop_n(b,c)
    od;
    if δ(b,c) then return out(b,c)
end
```

Welche Bedingung muss an die Spezifikation, in die (1) eingebettet ist, gestellt werden, damit diese Implementierung von (1) korrekt ist?



Die Relationen eines iterativen Programms: die (gewünschte) Ein/Ausgaberelation  $\varphi$  von  $f$ , die Hoare-Invariante  $\theta$  und die Subgoal-Invariante  $\psi$ , die der (gewünschten) Ein/Ausgaberelation der Schleifenfunktion  $g$  entspricht.

Der Verifikation von Schleifenprogrammen dienen zwei komplementäre Verfahren: **Hoare-Induktion** und **Subgoal-Induktion**.

Subgoal-Induktion ist Fixpunktinduktion über  $f$  (Beweis der E/A-Relation  $\varphi(a, b, d)$  von  $f$ ) mit anschließender Fixpunktinduktion über  $g$  (Beweis der **Subgoal-Invarianten**). Die Subgoal-Invariante  $\psi(b, c, d)$  setzt Werte der **Schleifenvariablen**  $b, c$  mit Werten der **Ausgabevariable**  $d$  in Beziehung. Die an  $\psi$  gestellten Bedingungen bilden die Konklusion folgender Regel:

$$\boxed{\text{Subgoal-Invarianten-Erzeugung}} \quad \frac{f(a, b) \equiv d \Rightarrow \varphi(a, b, d)}{\psi(b, \text{in}(a, b), d) \Rightarrow \varphi(a, b, d)} \uparrow \wedge \quad g(b, c) \equiv d \Rightarrow \psi(b, c, d)$$

Der zweite Faktor der Konklusion lässt sich dann durch Fixpunktinduktion über  $g$  beweisen (siehe §7.2). Die Axiome für  $g$  sind durch (1) gegeben.

Dual dazu beschreibt eine **Hoare-Invariante**  $\theta(a, b, c)$  einen Zusammenhang zwischen Werten der **Eingabeveriablen**  $a, b$  und Werten der Schleifenvariable  $c$ . Eine Hoare-Invariante  $\theta$  lässt sich folgendermaßen aus einer Subgoal-Invarianten  $\psi$  konstruieren:

$$\theta(a, b, c) \iff_{def} \forall d : (\psi(b, c, d) \Rightarrow \varphi(a, b, d))$$

Umgekehrt liefert jede Hoare-Invariante  $\theta$  eine Subgoal-Invariante  $\psi$ :

$$\psi(b, c, d) \iff_{def} \forall a : (\theta(a, b, c) \Rightarrow \varphi(a, b, d))$$

Mit dieser Beziehung erhält man die Korrektheit Regel zur Erzeugung von Hoare-Invarianten aus der Korrektheit der o.g. Regel zur Erzeugung von Subgoal-Invarianten:

<b>Hoare-Invarianten-Erzeugung</b>	$\frac{f(a, b) \equiv d \Rightarrow \varphi(a, b, d)}{\theta(a, b, in(a, b))}$ $\wedge \quad g(b, c) \equiv d \Rightarrow (\theta(a, b, c) \Rightarrow \varphi(a, b, d))$
------------------------------------	---

Der zweite Faktor der Konklusion lässt sich wieder durch Fixpunktinduktion über  $g$  beweisen.

Beide Regeln sind natürlich auch dann korrekt, wenn die Axiome für  $g$  nicht dem iterativen Schema (1) folgen.  $f$  muss allerdings wie in (1) durch das Axiom  $f(a, b) \equiv g(b, in(a, b))$  definiert sein.

**Beispiel** Wir beweisen die Korrektheit eines iterativen Programms zur Zerlegung einer Liste in Teillisten fester Länge mit Hoare-Induktion.

```
PARTN = PARTITION and NAT then
  defuncts  partn:nat*list->list(list)
            g:nat*list*nat*list*list(list)->list(list)
            length:list->nat
```

```

preds      lg0k:nat*list(list)
vars       k,n:nat  x:s  s,s':list  p,p':list(list)
          partn(s,n) = g(n,s,0,[],[])
          g(n,[],k,s,p) = p++[s]
          g(n,x:s,k,s',p) = g(n,s,k+1,s'++[x],p) <== k < n
          g(n,x:s,k,s',p) = g(n,x:s,0,[],p++[s']) <== k > n
          length[] = 0
          length(x:s) = length(s)+1
          lg0k(n,[])
          lg0k(n,[s]) <== length(s) <= n
          lg0k(n,s:s':p) <== length(s) = n /\ lg0k(n,s':p)
theorems: flatten(p++p') = flatten(p)++flatten(p')
          lg0k(n,p++[s]) <== lg0k(n,p) /\ length(s) <= n
conjects: partn(s,n) = p ==> s = flatten(p)
          partn(s,n) = p ==> lg0k(n,p)

```

Die beiden letzten Formeln sind die Korrektheitsbedingungen an *partn*. Aus den Argumentsorten von *partn* und *g* folgt, dass eine Hoare-Invariante den Typ

$$nat \times list \times list \times nat \times list \times list(list)$$

haben muss. Zum Beweis von (3) definieren wir sie durch das Axiom:

$$INV(s, n, s_1, k, s_2, p) \Leftarrow s \equiv flatten(p ++ [s_2 ++ s_1]).$$

$$\text{partn}(s, n) = p \Rightarrow s = \text{flatten}(p)$$

Hoare invariant creation for the iterative program

$\text{partn}(s, n) = g(n, s, 0, [], [])$

at position [] of the preceding formula leads to

$\text{INV}(s, n, s, 0, [], []) \ \&$   
 $(g(n, z3, z4, z5, z6) = p \ \& \ \text{INV}(s, n, z3, z4, z5, z6) ==> s = \text{flatten}(p))$

Narrowing at position [0] of the preceding formula (3 steps) leads to

True  $\&$   $(g(n, z3, z4, z5, z6) = p \ \& \ \text{INV}(s, n, z3, z4, z5, z6) ==> s = \text{flatten}(p))$

Reducts have been simplified.

Simplifying the preceding formula leads to

$g(n, z3, z4, z5, z6) = p \ \& \ \text{INV}(s, n, z3, z4, z5, z6) ==> s = \text{flatten}(p)$

Shifting subformulas at position [0,1] of the preceding formula leads to

$g(n, z3, z4, z5, z6) = p ==> (\text{INV}(s, n, z3, z4, z5, z6) ==> s = \text{flatten}(p))$

Applying fixpoint induction w.r.t.

```

g(n, [] ,k,s,p) = p++[s] &
(g(n,x:s,k,s',p) = z7 <==> k < n & g(n,s,k+1,s'++[x],p) = z7) &
(g(n,x:s,k,s',p) = z7 <==> k > n & g(n,x:s,0,[] ,p++[s']) = z7)

```

at position [] of the preceding formula leads to

```

All n k s p x s' z7:(All s6:(INV(s6,n,[] ,k,s,p) ==> s6 = flatten(p++[s])) &
                           (All s7:(INV(s7,n,x:s,k,s',p) ==> s7 = flatten(z7)) <==>
                            k < n &
                            All s7:(INV(s7,n,s,k+1,s'++[x],p) ==> s7 = flatten(z7))) &
                            (All s8:(INV(s8,n,x:s,k,s',p) ==> s8 = flatten(z7)) <==>
                            k > n &
                            All s8:(INV(s8,n,x:s,0,[] ,p++[s']) ==> s8 = flatten(z7))))

```

Applying the axiom resp. theorem

```
INV(s,n,s1,k,s2,p) <==> s = flatten(p++[s2++s1])
```

at positions [0,2,1,1,0,0],[0,2,0,0,0],[0,1,1,1,0,0],[0,1,0,0,0],[0,0,0,0] of the preceding formula leads to

```

All n k s p x s' z7:(All s6:(s6 = flatten(p++[s++[]])) ==> s6 = flatten(p++[s])) &
                           (All s7:(s7 = flatten(p++[s'++x:s])) ==> s7 = flatten(z7)) <==>
                            k < n &
                            All s7:(s7 = flatten(p++[s'++[x]++s])) ==>

```

```

        s7 = flatten(z7))) &
(All s8:(s8 = flatten(p++[s'++x:s]) ==> s8 = flatten(z7)) <===
  k > n &
  All s8:(s8 = flatten(p++[s'])++[[[]++x:s]) ==>
    s8 = flatten(z7)))

```

Simplifying the preceding formula (29 steps) leads to

```

All n k s p x s' z7:(k > n & flatten(p++[s',x:s]) = flatten(z7) ==>
  flatten(p++[s'++x:s]) = flatten(z7))

```

Applying the axiom resp. theorem

$$\text{flatten}(p++p') = \text{flatten}(p)++\text{flatten}(p')$$

at positions [0,1],[0,0,1] of the preceding formula leads to

```

All n k s p x s' z7:(k > n & flatten(p)++flatten[s',x:s] = flatten(z7) ==>
  flatten(p)++flatten[s'++x:s] = flatten(z7))

```

Applying the axiom resp. theorem

$$\text{flatten}(s:p) = s++\text{flatten}(p)$$

at positions [0,1],[0,0,1] of the preceding formula leads to

```
All n k s p x s' z7:(k > n & flatten(p)++s'++flatten[x:s] = flatten(z7) ==>
                      flatten(p)++s'++x:s++flatten[] = flatten(z7))
```

Applying the axioms

```
flatten[] = []
& flatten(s:p) = s++flatten(p)
```

at positions [0,1],[0,0,1] of the preceding formula leads to

```
All n k s p x s' z7:(k > n & flatten(p)++s'++x:s++flatten[] = flatten(z7) ==>
                      flatten(p)++s'++x:s++[] = flatten(z7))
```

Applying the axiom resp. theorem

```
flatten[] = []
```

at position [0,0,1] of the preceding formula leads to

```
All n k s p x s' z7:(k > n & flatten(p)++s'++x:s++[] = flatten(z7) ==>
                      flatten(p)++s'++x:s++[] = flatten(z7))
```

Simplifying the preceding formula (4 steps) leads to

True

Number of proof steps: 12

Zum Beweis der zweiten Bedingung an *partn* definieren wir eine Hoare-Invariante durch das Axiom:

$$INV(s, n, s_1, k, s_2, p) \Leftrightarrow lgOk(n, p) \wedge k = length(s_2) \wedge k \leq n$$

*partn(s,n) = p*  $\Rightarrow$  *lgOk(n,p)*

Hoare invariant creation for the iterative program

*partn(s,n) = g(n,s,0,[] ,[] )*

at position [] of the preceding formula leads to

*INV(s,n,s,0,[] ,[] ) &*  
*(g(n,z3,z4,z5,z6) = p & INV(s,n,z3,z4,z5,z6)  $\Rightarrow$  lgOk(n,p))*

Narrowing at position [0] of the preceding formula (2 steps) leads to

True & *(g(n,z3,z4,z5,z6) = p & INV(s,n,z3,z4,z5,z6)  $\Rightarrow$  lgOk(n,p))*

Reducts have been simplified.

Simplifying the preceding formula leads to

$g(n, z3, z4, z5, z6) = p \ \& \ \text{INV}(s, n, z3, z4, z5, z6) ==> \text{lg0k}(n, p)$

Shifting subformulas at position [0,1] of the preceding formula leads to

$g(n, z3, z4, z5, z6) = p ==> (\text{INV}(s, n, z3, z4, z5, z6) ==> \text{lg0k}(n, p))$

Applying fixpoint induction w.r.t.

$g(n, [], k, s, p) = p++[s] \ \&$   
 $(g(n, x:s, k, s', p) = z7 <==> k < n \ \& \ g(n, s, k+1, s'++[x], p) = z7) \ \&$   
 $(g(n, x:s, k, s', p) = z7 <==> k > n \ \& \ g(n, x:s, 0, [], p++[s']) = z7)$

at position [] of the preceding formula leads to

All n k s p s4: ( $\text{INV}(s4, n, [], k, s, p) ==> \text{lg0k}(n, p++[s])$ )  $\&$

All n k s p x s' z7 s5: ( $k < n \ \&$   
                          All s5: ( $\text{INV}(s5, n, s, k+1, s'++[x], p) ==> \text{lg0k}(n, z7)$ )  $\&$   
                           $\text{INV}(s5, n, x:s, k, s', p) ==>$   
                           $\text{lg0k}(n, z7)$ )  $\&$

All n k s p x s' z7 s6: ( $k > n \ \&$   
                          All s6: ( $\text{INV}(s6, n, x:s, 0, [], p++[s']) ==> \text{lg0k}(n, z7)$ )  $\&$   
                           $\text{INV}(s6, n, x:s, k, s', p) ==>$   
                           $\text{lg0k}(n, z7)$ )

Reducts have been simplified.

Applying the axiom resp. theorem

$\text{INV}(s, n, s1, k, s2, p) \iff \text{lg0k}(n, p) \ \& \ k = \text{length}(s2) \ \& \ k \leq n$

at positions [2,0,0,2], [2,0,0,1,0,0], [1,0,0,2], [1,0,0,1,0,0], [0,0,0]  
of the preceding formula leads to

All  $n \ s \ p : (\text{length}(s) \leq n \ \& \ \text{lg0k}(n, p) \implies \text{lg0k}(n, p++[s])) \ \&$   
All  $n \ p \ s' \ z7 : (\text{length}(s') \leq n \ \& \ \text{length}(s') < n \ \&$   
 $(\text{lg0k}(n, p) \ \& \ \text{length}(s')+1 \leq n \implies \text{lg0k}(n, z7)) \ \& \ \text{lg0k}(n, p) \implies$   
 $\text{lg0k}(n, z7))$

Reducts have been simplified.

Applying the axiom resp. theorem

$\text{lg0k}(n, p++[s]) \iff \text{lg0k}(n, p) \ \& \ \text{length}(s) \leq n$

at position [0,0,1] of the preceding formula leads to

All  $n \ p \ s' \ z7 : (\text{length}(s') \leq n \ \& \ \text{length}(s') < n \ \&$   
 $(\text{lg0k}(n, p) \ \& \ \text{length}(s')+1 \leq n \implies \text{lg0k}(n, z7)) \ \& \ \text{lg0k}(n, p) \implies$   
 $\text{lg0k}(n, z7))$

Applying the theorem

$$k < n \implies k+1 \leq n$$

at position [0,0,1] of the preceding formula leads to

True

Reducts have been simplified.

Number of proof steps: 8

**Beispiel** Wir beweisen die Korrektheit eines iterativen Programms zur Berechnung von Quotient und Rest der Division natürlicher Zahlen.

```
DIVLOOP = NAT then
    defuncts  div:nat->nat*nat
              loop:nat*nat*nat->nat*nat
    preds      INV:nat*nat*nat*nat          -- Hoare-Invariante
    vars       x,y,q,r:nat
    axioms     div(x,y) = loop(y,0,x)
              loop(y,q,r) = (q,r) <== r < y
              loop(y,q,r) = loop(y,q+1,r-y) <== r >= y
              INV(x,y,q,r) <== x = (y*q)+r
```

$\text{div}(x, y) = (q, r) \implies x = (y*q) + r \ \& \ r < y$

Hoare invariant creation for the iterative program

$\text{div}(x, y) = \text{loop}(y, 0, x)$

at position [] of the preceding formula leads to

$\text{INV}(x, y, 0, x) \ \&$

$(\text{loop}(y, z1, z2) = z0 \ \& \ \text{INV}(x, y, z1, z2) \implies (z0 = (q, r) \implies x = (y*q) + r \ \& \ r < y))$

Narrowing at position [0] of the preceding formula leads to

True &

$(\text{loop}(y, z1, z2) = z0 \ \& \ \text{INV}(x, y, z1, z2) \implies (z0 = (q, r) \implies x = (y*q) + r \ \& \ r < y))$

Reducts have been simplified.

Simplifying the preceding formula leads to

$\text{loop}(y, z1, z2) = z0 \ \& \ \text{INV}(x, y, z1, z2) \implies (z0 = (q, r) \implies x = (y*q) + r \ \& \ r < y)$

Shifting subformulas at position [0,1] of the preceding formula leads to

$\text{loop}(y, z1, z2) = z0 \implies$

$(INV(x,y,z1,z2) \implies (z0 = (q,r) \implies x = (y*q)+r \ \& \ r < y))$

Applying fixpoint induction w.r.t.

$(loop(y,q,r) = (q,r) \iff r < y) \ \&$   
 $(loop(y,q,r) = z3 \iff r \geq y \ \& \ loop(y,q+1,r-y) = z3)$

at position [] of the preceding formula leads to

All  $y \ q \ r \ z3: ((\text{All } x \ q0 \ r0: (INV(x,y,q,r) \implies$   
 $\quad ((q,r) = (q0,r0) \implies x = (y*q0)+r0 \ \& \ r0 < y)) \iff$   
 $\quad r < y) \ \&$   
 $\quad (\text{All } x \ q1 \ r1: (INV(x,y,q,r) \implies$   
 $\quad \quad (z3 = (q1,r1) \implies x = (y*q1)+r1 \ \& \ r1 < y)) \iff$   
 $\quad \quad r \geq y \ \&$   
 $\quad \quad \text{All } x \ q1 \ r1: (INV(x,y,q+1,r-y) \implies$   
 $\quad \quad \quad (z3 = (q1,r1) \implies x = (y*q1)+r1 \ \& \ r1 < y))))$

Applying the axiom resp. theorem

$INV(x,y,q,r) \iff x = (y*q)+r$

at positions [0,1,1,1,0,0], [0,1,0,0,0], [0,0,0,0,0] of the preceding formula leads to

All  $y \ q \ r \ z3: ((\text{All } x \ q0 \ r0: (x = (y*q)+r \implies$

$$\begin{aligned}
& ((q, r) = (q_0, r_0) \implies x = (y*q_0) + r_0 \ \& \ r_0 < y) \iff \\
& r < y \ \& \\
& (\text{All } x \ q_1 \ r_1 : (x = (y*q) + r \implies \\
& z_3 = (q_1, r_1) \implies x = (y*q_1) + r_1 \ \& \ r_1 < y)) \iff \\
& r \geq y \ \& \\
& \text{All } x \ q_1 \ r_1 : (x = (y*(q+1)) + (r-y) \implies \\
& z_3 = (q_1, r_1) \implies x = (y*q_1) + r_1 \ \& \ r_1 < y)))
\end{aligned}$$

Simplifying the preceding formula (57 steps) leads to

$$\begin{aligned}
& \text{All } y \ q \ r \ q_1 \ r_1 : (r \geq y \ \& \ \text{All } q_1 \ r_1 : ((y*(q+1)) + (r-y) = (y*q_1) + r_1) \ \& \\
& \text{All } r_1 : (r_1 < y) \implies \\
& (y*q) + r = (y*q_1) + r_1)
\end{aligned}$$

Applying the axiom resp. theorem

$$(x*(y+1)) + (z-x) = (x*y) + z$$

at position [0,0,1,0] of the preceding formula leads to

$$\begin{aligned}
& \text{All } y \ q \ r \ q_1 \ r_1 : (r \geq y \ \& \ \text{All } q_1 \ r_1 : ((y*q) + r = (y*q_1) + r_1) \ \& \ \text{All } r_1 : (r_1 < y) \implies \\
& (y*q) + r = (y*q_1) + r_1)
\end{aligned}$$

Simplifying the preceding formula (2 steps) leads to

True

Number of proof steps: 9

## Beispiele

Es folgen einige z.T. nichttriviale iterative Programme mit ihrer jeweiligen Korrektheitsbedingung  $\varphi$  und Hoare-Invarianten  $\theta$ . Hilfsfunktionen werden nicht explizit spezifiziert. Jede vom Programm nicht veränderte gemeinsame Komponente von Eingabe- und Schleifenvariable taucht nur einmal als Argument von  $\theta$  auf.

### Summe

```
sum(x) = sum1(x,0)
sum1(y,z) = if y > 0 then sum1(y-1,y+z) else z
 $\varphi(x,z) \iff_{def} z = \sum_{i=1}^x i.$ 
 $\theta(x,y,z) \iff_{def} \sum_{i=1}^x i = (\sum_{i=1}^y i) + z.$ 
```

### Fakultät

```
fact(x) = fact1(x,1)
fact1(y,z) = if y > 0 then fact1(y-1,y*z) else z
 $\varphi(x,z) \iff_{def} z = x!.$ 
 $\theta(x,y,z) \iff_{def} x! = y! * z.$ 
```

### Fibonacci-Zahlen

```
fib(x) = fib1(x,0,1)
fib1(x,y,z) = if x > 0 then fib1(x-1,z,y+z) else y
 $\varphi(x,z) \iff_{def} z \text{ ist die } x\text{-te Fibonacci-Zahl.}$ 
 $\theta(n,x,y,z) \iff_{def} n \geq x \text{ und } y \text{ ist die } (n-x)\text{-te und } z \text{ die } (n-x+1)\text{-te Fibonacci-Zahl.}$ 
```

## Ganzzahlige Wurzel

$\text{root}(x) = \text{root1}(x, 0, 1)$

$\text{root1}(x, y, z) = \text{if } x \geq z \text{ then } \text{root1}(x, y+1, z+2y+3) \text{ else } y$

$\varphi(x, z) \iff_{\text{def}} z = \lceil \sqrt{x} \rceil.$

$\theta(x, y, z) \iff_{\text{def}} x \geq y^2 \wedge z = (y + 1)^2.$

## Größter gemeinsamer Teiler

$\text{gcd}(x, y) = \text{gcd1}(x, y)$

$\text{gcd1}(x, y) = \text{if } x > y \text{ then } \text{gcd1}(x-y, y) \text{ else if } x < y \text{ then } \text{gcd1}(x, y-x) \text{ else } y$

$\varphi(x, y, z) \iff_{\text{def}} z = ggT(x, y).$

$\theta(m, n, x, y) \iff_{\text{def}} ggT(m, n) = ggT(x, y).$

## Quotient und Rest (Beispiel ??)

$\text{div}(x, y) = \text{div1}(y, 0, x)$

$\text{div1}(y, q, r) = \text{if } r \geq y \text{ then } \text{div1}(y, q+1, r-y) \text{ else } (q, r)$

$\varphi(x, y, q, r) \iff_{\text{def}} x = y * q + r \wedge r < y.$

$\theta(x, y, q, r) \iff_{\text{def}} x = y * q + r.$

## Kürzeste Wege in Graphen des Typs $\text{nat} \times \text{nat} \rightarrow \text{nat}$ nach Floyd und Warshall (vgl. [11], §6.2)

$\text{minGraph}(G) = f(G, 1)$

$f(G, i) = \text{if } i \leq N \text{ then } f(\lambda(j, k). \min(G(j, k), G(j, i) + G(i, k)), i+1) \text{ else } G$

$\varphi(G, G') \iff_{\text{def}} G \text{ und } G' \text{ sind markierte gerichtete Graphen mit der Knotenmenge } \{1, \dots, N\}.  
Jede Kante } (j, k) \text{ von } G' \text{ ist markiert mit der Länge des kürzesten Weges,}\\ \text{der in } G \text{ von } j \text{ nach } k \text{ führt.}$

$\theta(G, G', i) \iff_{\text{def}} G \text{ und } G' \text{ sind markierte gerichtete Graphen mit der Knotenmenge } \{1, \dots, N\}.  
Jede Kante } (j, k) \text{ von } G' \text{ ist markiert mit der Länge des kürzesten Weges,}\\ \text{der in } G \text{ von } j \text{ nach } k \text{ führt und nur Knoten der Menge } \{1, \dots, i-1\} \text{ passiert.}$

Tiefensuche in Graphen des Typs  $s \rightarrow list(s)$  (vgl. [11], §6.3)

search(G,L) = search1(G,L,[])

search1(G,[],V) = V

search1(G,x:L,V) = if x in V then search1(G,L,V) else search1(G,G(x)++L,x:V)

$\varphi(G, L, V) \iff_{def} V$  ist eine Liste aller (von einem Knoten) von  $L$  aus erreichbaren Knoten von  $G$ .

$\theta(G, L, L', V) \iff_{def}$  Jeder von  $L$  aus erreichbare Knoten von  $G$  liegt in  $V$  oder ist von  $L'$  aus aus einem Weg erreichbar, der keinen Knoten von  $V$  passiert.

Eine Subgoal-Invariante ist hier leichter zu finden:

$\psi(G, L, V, V') \iff_{def} V'$  besteht aus den Knoten von  $V$  und den von  $L \setminus V$  aus erreichbaren Knoten von  $G$ .

## Minimaler Spannbaum nach Kruskal

```
KRUSKAL = LIST{nat/s}[NAT] and ORD(label) and LIST{edge/s}[NAT]
           and LIST{list(nat)/s}[LIST{nat/s}[NAT]] then
  sorts      edge = nat*label*nat
             graph = list(edge)
             partition = list(list(nat))
  defuncts   maxnode:->nat
             firstPart:->partition
             spanTree:graph->graph
             spanTree1:graph*partition*graph->graph
  preds       _<=_:edge*edge
  vars        i,j,m,n:nat  x,y:label  G,T:graph  L,L',L1,L2:list(nat)  P:partition
  axioms     (i,x,j) <= (m,y,n)  <==  x <= y
```

```

firstPart = [[1], ..., [maxnode]]
spanTree(G) = spanTree1(sort(G), firstPart, [])
spanTree1([], L:L':P, T) = []      -- Hier ist G ist unzusammenhaengend.
spanTree1(G, [L], T) = T
spanTree1((i,x,j):G,L:L':P,T)
= if L1 = L2 then spanTree1(G,L:L':P,T)
            else spanTree1(G,(L1++L2):remove(L1,remove(L2,L:L':P)),(:
<== i in L1 /\ j in L2

```

$\varphi(G, T)$   $\iff_{def}$   $G$  ist ein markierter Graph mit der Knotenmenge  $\{1, \dots, N\}$  und  $T$  ist ein minimaler Spannbaum von  $G$ .

$\theta(G, G', P, T)$   $\iff_{def}$   $G$  und  $G'$  sind markierte Graphen mit der Knotenmenge  $\{1, \dots, N\}$  und  $P$  enthält alle Knoten von  $G$ . Jede Kante von  $G \setminus G'$  verbindet zwei Knoten der Liste von  $P$ .

Für alle Knotenlisten  $L$  von  $P$  ist der von  $L$  erzeugte Teilgraph von  $T$  ein minimaler Spannbaum des von  $L$  erzeugten Teilgraphen von  $G$ .

## Optimaler Binärcode nach Huffman

```

BINTREE = LIST[TRIV(s)] then
  sorts      tree
  constructs leaf:list(s)->tree
              _#_#:tree*list(s)*tree->tree
  defuncts   root:tree->list(s)
  vars       T,T':tree  cL:list(s)
  axioms    root(T#cL#T') = cL

```

```

HUFFMAN = LIST{tree/s}[BINTREE] and NAT then
  defuncts   minOf2:(s->nat)*tree*tree->tree
              minOfList:(s->nat)*list(tree)->tree
              firstList:list(s)->list(tree)
              newList:list(s)*list(tree)->list(tree)
              Huffman:(s->nat)*list(s)->tree
              Huffman1:(s->nat)*list(tree)->tree
  vars       T,T',T1,T2:tree  fr:s->nat  c:s  cL:list(s)  TL:list(tree)
  axioms    minOf2(fr,T,T') = if sum(map(fr)(root(T))) <= sum(map(fr)(root(T'))))
                        then T else T'
            minOfList(fr,T:TL) = if TL = [] then T
                                      else minOf2(fr,T,minOfList(fr,TL))
            firstList([]) = []
            firstList(c:cL) = [c]:firstList(cL)
            newList(fr,T:TL) = (T1#(root(T1)++root(T2))#T2):remove(T2,TL')
                        <== minOfList(fr,T:TL) = T1 /\ 
                        remove(T1,T:TL) = TL' /\ 
                        minOfList(fr,TL') = T2
            Huffman(fr,cL) = Huffman1(fr,firstList(cL))
            Huffman1(fr,T:TL) = if TL = [] then T
                                    else Huffman1(fr,newList(fr,T:TL))

```

$\varphi(fr, cL, T)$   $\iff_{def}$   $T$  ist ein Codebaum so, dass für alle Zeichen  $c, c' \in cL$  und Wege  $w, w'$  von der Wurzel von  $T$  zu dem Blatt,

das mit  $c$  bzw.  $c'$  markiert ist, gilt:

$$fr(c) < fr(c') \Rightarrow length(w) \geq length(w').$$

$\theta(fr, cL, TL) \iff_{def} TL$  ist eine Codebaumliste derart, dass für alle Zeichen  $c, c' \in cL$  und Wege  $w, w'$  von der Wurzel der Bäume  $T, T'$  zu dem Blatt, das mit  $c$  bzw.  $c'$  markiert ist, gilt:

$$fr(c) < fr(c') \Rightarrow length(w) \geq length(w'),$$

$$fr(c) < fr(c') \wedge length(w) = length(w') \wedge T \neq T'$$

$$\Rightarrow \sum_{c'' \in root(T)} fr(c'') < \sum_{c'' \in root(T')} fr(c'').$$

## References

- [1] S. Antoy, R. Echahed, M. Hanus, *A Needed Narrowing Strategy*, Journal of the ACM 47 (2000) 776-822
- [2] M. Bonsangue, J. Rutten, A. Silva, A Kleene Theorem for Polynomial Coalgebras, Proc. FOSSACS 2009, Springer LNCS 5504 (2009) 122–136
- [3] E. M. Clarke, O. Grumberg, D.A. Peled, Model Checking, MIT Press 1999
- [4] M. Huth, M. Ryan, Logic in Computer Science, Cambridge University Press 2004
- [5] R. Y. Kain, Automata theory: machines and languages, McGraw-Hill 1972
- [6] F. Moller, G. Struth, *Modelling Computing Systems* - Mathematics for Computer Science, Springer 2013

- [7] P. Padawitz, Computing in Horn Clause Theories, EATCS Monographs on Theoretical Computer Science 16, Springer-Verlag, 1988
- [8] P. Padawitz, *Deduction and Declarative Programming*, Cambridge Tracts in Theoretical Computer Science 28, Cambridge University Press, 1992
- [9] P. Padawitz, *Inductive Theorem Proving for Design Specifications*, J. Symbolic Computation 21 (1996) 41-99
- [10] P. Padawitz, *Proof in Flat Specifications*, in: E. Astesiano, H.-J. Kreowski, B. Krieg-Brückner, eds., *Algebraic Foundations of Systems Specification*, IFIP State-of-the-Art Report, Springer (1999) 321-384
- [11] P. Padawitz, *Einführung ins funktionale Programmieren*, Vorlesungsskript, TU Dortmund 1999
- [12] P. Padawitz, *Modellieren und Implementieren in Haskell*, Folienskript, TU Dortmund, 2019
- [13] P. Padawitz, *Formale Methoden des Systementwurfs*, Vorlesungsskript, TU Dortmund, 2006
- [14] P. Padawitz, *Expander2 as a Prover and Rewriter*, TU Dortmund, 2021
- [15] P. Padawitz, *Expander2: Program Verification between Interaction and Automation*, Folien, TU Dortmund, 2021

- [16] P. Padawitz, *Expander2-Manual*, TU Dortmund, 2021
- [17] P. Padawitz, *Algebraic Model Checking*, in: F. Drewes, A. Habel, B. Hoffmann, D. Plump, eds., Manipulation of Graphs, Algebras and Pictures, *Electronic Communications of the EASST* Vol. 26 (2010)
- [18] P. Padawitz, *Fixpoints, Categories, and (Co)Algebraic Modeling*, TU Dortmund 2019
- [19] P. Padawitz, *Swinging Types At Work*, TU Dortmund, 2008
- [20] P. Padawitz, *Übersetzerbau*, Folienskript, TU Dortmund, 2017
- [21] P. Padawitz, *Übersetzerbau*, Vorlesungsskript, TU Dortmund, 2010