Horn axioms \((p(t) \iff \varphi)\) and co-Horn axioms \((p(t) \Rightarrow \varphi)\) defining functions, relations or non-deterministic functions (transition systems),

- logical **top-down derivations** into True or another **solved formula** (constraint):
  - **prove** \(\varphi\): \(\varphi \vdash True\)
  - **solve** \(\varphi\): \(\varphi \vdash \) solved formula
  - **refute** \(\varphi\): \(\neg \varphi \vdash True\)
  - **verify** \(p\): \(p(x) \Rightarrow \varphi \vdash True\)
  - **evaluate** \(p\): \(p(x) \iff \varphi \vdash True\)
  - **evaluate** \(t\): \(t \equiv x \vdash c \equiv x\)
• rewrite sequences that generate/manipulate/normalize functional terms or check Kripke models,

• rules at 3 levels of automation/interaction:
  • Simplifications are equivalence transformations that partially evaluate terms and formulas.
  • Narrowing and rewriting apply all or some axioms to goals, exhaustively or selectively, interactively or automatically, stepwise or iteratively.
  • Induction, coinduction and other proper expansions are applied interactively and stepwise. (Fixpoint) induction and coinduction apply goals (as hypotheses) to axioms and thus show the former by solving the latter.
The semantics is given by the initial resp. final model of the axioms.

Relations and predicates are interpreted as the least or greatest solutions of their Horn resp. co-Horn axioms.

These dualities admit the uniform treatment of constructor- and destructor-based data types, finite and infinite objects, positive and negative propositions.

3 representations of a formula/term:
- text, tree (rooted graph) and picture (list of 2-dimensional widgets).
All representations can be edited, moved and scaled.
The pictorial ones can also be rotated and connected by arcs of different shapes.
Expander’s representations of terms/formulas and derivations
Swinging types

- Sums $\bigsqcup_{i \in I} t_i$ formalize/implement selection and case analysis.
- Products $\prod_{i \in I} t_i$ formalize/implement tupling and relationships.
- A recursively defined type $T$ is created from

<table>
<thead>
<tr>
<th>constructors</th>
<th>destructors</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c : \text{composed type} \to T$</td>
<td>$d : T \to \text{composed type}$</td>
</tr>
<tr>
<td>(initial) algebras</td>
<td>(final) coalgebras</td>
</tr>
<tr>
<td>context-free languages</td>
<td>transition systems</td>
</tr>
</tbody>
</table>

Constructor-based types are called **visible**. Destructor-based types are called **hidden**.

- More constructors lead to **supertypes**. More destructors lead to **subtypes**.
- Functions $f : T \to \text{composed type}$ on a visible type $T$ are defined by **recursion**.
- Functions $g : \text{composed type} \to T$ into a hidden type $T$ are defined by **corecursion**.
Swinging types

- **Horn clauses** $r(t) \iff \varphi$ define **least relations**
  (least solution of $r(t) \iff \varphi$ in $r$).
- **Co-Horn clauses** $r(t) \Rightarrow \varphi$ define **greatest relations**
  (greatest solution of $r(t) \Rightarrow \varphi$ in $r$).

Properties of least relations are proved by **induction**.
Properties of greatest relations are proved by **coinduction**.

- Least or greatest **congruences** $\equiv: t \times t$ and **quotients** $A/\equiv^A$
  formalize/implement (visible or hidden) abstraction.
- Least or greatest **invariants** $all: t$ and **substructures** $all^A \subseteq A$
  formalize/implement (visible or hidden) restriction.
The standard models are **initial/least** or **final/greatest** solutions of domain equations \((A \leq B \text{ iff } \exists A \rightarrow B)\), constructed as **suprema/colimits** or **infima/limits** of **ascending** or **descending** chains.

An element of the initial model for constructors \(c_i : s_{i,1} \times \ldots \times s_{i,n_i} \rightarrow s_i\) (left) versus an element of the final model for destructors \(d_i : s_i \rightarrow s_{i,1} + \cdots + s_{i,n_i}\) (right).
Picture generation

- **text**
- **compile**
- **abstract syntax tree**
  - **modify**
  - **simplify**
  - **rewrite**
- **design language**
- **picture with state**
  - **single widget**
    - **mkWidg**
    - **move**
    - **rotate**
    - **setState**
    - **position orientation color lightness**
  - **multiple widget**
    - **mkPict**
    - **widget**
      - **widget 1**
      - **widget n**
      - **move**
      - **scale**
      - **rotate**
      - **edit arcs**
      - **change lightness**

- **Haskell types & functions**
- **Tcl/Tk**
  - **wysiwyg**
  - **fast or slow**
  - **scale & draw**
  - **add positions**
  - **add cross points**
  - **add intersections**
  - **add unions**
  - **add convex hulls**
Several interpreters translate a tree into a picture:
- alignment, linear equations, matrices, matrix solution,
- partition, polygons, polygon solution, rectangles

They create a widget in an initial state
(position (0,0), orientation 0, color, lightness 0)

Some operations on a list of pictures $ps$, the contents of a file $F$, a single widget $w$ or a list of actions $acts$:
- $color(c,ps)$, $dark(ps)$, $file(F)$, $flipH/V(ps)$, $gif(F)$, $grow(ps)$, $grow5/R(n,ps)$, $hframe(ps)$, $light(i,ps)$, $matrix(w)$, $meet(n,ps)$, $odots(ps)$, $outline(ps)$, $place(w,points)$, $rainbow(w,n,d,a)$, $rainbow2(w,n)$, $reverse(ps)$, $rframe(ps)$, $shelves(n,d,ps)$, $shineB/W(w,d,a)$, $shuffle(ps)$, $split(ps)$, $splitS(sc,ps)$, $tabA/B(n,d,ps)$, $tabAS/BS(n,d,sc,ps)$, $turt(acts)$, $turt(ps)$
- animators: $fadeB/W(w)$, $fast(w)$, $flash(w)$, $new$, $old$, $osciL/P/W(...)$, $peaks(w,m)$, $pulse(w)$, $repeat(ps)$, $rotate/C(w,a)$
- Further operations on list terms are provided by the simplifier.
The Haskell types

type Picture = [Widget_

type Arcs = [[Int]]

data Widget_ = Arc Color ArcStyleType Point Float (Float,Float) |
Arc0 State ArcStyleType Float Float |
Arc0, Path0 and Tree0 are abstract versions of Arc, Path resp. Tree which they are turned into before being displayed.
Bunch Widget_ [Int] |
Bunch w ns represents widget w together with arcs leading from w to the widgets at positions ns.
ms++ns is the list of direct successors of Bunch w ms ns.
Circ State Float | CircA State Float | Dot Color Point |
CircA and RectA ignore the scale of enclosing turtles.
Fast Widget_ | File_ String | Gif String Point |
New | Old | Path Color Int [Point] |
Path0 State Int Point [Point] |
Poly State Int [Float] Float |
Rect State Float Float | RectA State Float Float |
Repeat Widget_ | Snow State Int Float |
Text_ State [String] |
Tree Color Color (Term TNode) |
Tree0 State String Color [Term TNode] |
Tria State Float |
Turtle State Float [TurtleAct] | White
The Haskell types

data TurtleAct = Move Float | MoveA Float | Jump Float | JumpA Float |
  MoveA and JumpA ignore the scale of the enclosing turtle.
Turn Float | Open Color Int | Scale Float |
Close | Draw | Widg Widget_ | WidgB Widget_
  Widg w ignores the orientation of the enclosing turtle,
  WidgB w adds it to the orientation of w.

type State = (Point,Float,Color,Int)
type TNode = (String,Point)
type Point = (Float,Float)
Turtle actions

drawAt ps = f [open] p0 ps
  where f acts p (q:ps) = f (acts++acts’) q ps
    where acts’ = if p == q then [Widg w]
      else [Turn a,Jump d,Turn (-a),Widg w]
      (a,d) = (angle p q,distance p q)
    f acts _ _ _ = acts++[Close]
From turtle actions to a picture

\[\text{mkPict} \ (\text{Turtle} \ (p, a, c, i) \ \text{sc} \ \text{acts}) = g \ \text{pict} \ c' \ n \ \text{ps} \]
\[
\text{where} \ (\text{pict},(_,_\text{c'},n,_\text{ps}):_) = \text{foldl} \ f \ ([],[(p,a,c,0,\text{sc},[p])]) \ \text{acts} \]
\[
f \ (\text{pict},(p,a,c,n,\text{sc},\text{ps}):\text{s}) (\text{Move} \ d) = \\
(\text{pict},(q,a,c,n,\text{sc},\text{ps}++[q]):\text{s}) \\
\text{where} \ q = \text{successor} \ p \ a \ (d*\text{sc})
\]
\[
f \ (\text{pict},(p,a,c,n,\text{sc},\text{ps}):\text{s}) (\text{Jump} \ d) = \\
(g \ \text{pict} \ c \ n \ \text{ps},(q,a,c,n,\text{sc},[q]):\text{s}) \\
\text{where} \ q = \text{successor} \ p \ a \ (d*\text{sc})
\]
\[
f \ (\text{pict},(p,a,c,n,\text{sc},\text{ps}):\text{s}) (\text{Turn} \ b) = \\
(\text{pict},(p,a+b,c,n,\text{sc},\text{ps}):\text{s})
\]
\[
f \ (\text{pict},s@((p,a,c,m,\text{sc},_):_)) (\text{Open} \ d \ n) = \\
(\text{pict},(p,a,d,n,\text{sc},[p]):\text{s})
\]
\[
f \ (\text{pict},s@((p,a,c,n,\text{sc},\text{ps}):_)) (\text{Scale} \ \text{sc'}) = \\
(\text{pict},(p,a,c,n,\text{sc*sc'},\text{ps}):\text{s})
From turtle actions to a picture

\[
f (\text{pict}, (_,_,c,n,_,ps):s) \text{ Close } = \\
(\text{pict++}[\text{Path (mkLight i c) n ps}], \\
s)
\]

\[
f (\text{pict}, (p,a,c,n,sc,ps):s) \text{ Draw } = \\
(\text{pict++}[\text{Path (mkLight i c) n ps}], \\
(p,a,c,n,sc,[p]):s)
\]

\[
f (\text{pict}, s@( (p,a,_,_,sc,_):_)) \text{ (Widg w) } = \\
(\text{pict++}[\text{scaleWidg sc (moveWidg p a w)}], \\
s)
\]
Recursive drawing

drawPict pict = action
    if fast || all isFast pict then mapM_ drawWidget pict
    else let scan = head scans
        run <- scan.isRunning
        if run then scan.addScan pict
            else scan.startScan0 delay pict

drawWidget (Circ ((x,y),_,c,i) r) = action
    canv.oval (round2 (x-r,y-r)) (round2 (x+r,y+r))
        [Outline (outColor c i), Fill (fillColor c i)]
    done

... drawWidget w | isWidg w = drawWidget (mkWidg w)
    | isPict w = drawPict (mkPict w)
**Picture scanner**

```haskell
struct Scanner = startScan0 :: Int -> Picture -> Action

...

scanner :: TkEnv -> (Widget_ -> Action) -> Template Scanner
scanner tk act =
    template (as,run,running) := ([],undefined,False)
    in let startScan0 n bs = action as := bs; startScan n
        startScan n = action if running then run.stop
        run0 <- tk.periodic n loop
        run := run0; run.start; running := True
        loop = action case as of a:s -> if noRepeat a then as := s
        act a; if isFast a then loop
        _ -> stopScan
    addScan bs = action as := bs++as
    stopScan = action if running then run.stop; running := False
    isRunning = request return running

    in struct ..Scanner
```
Oscillable widgets

```plaintext
struct Oscillable = maxheight :: Float
    actseq :: Float -> [TurtleAct]

oscillate obj = f h++g 1
    where f a = if a == 0 then [] else acts a++f (a-1)
        g a = if a == h then [Fast (w h)]
            else acts a++g (a+1)
        acts a = [Fast v,wait,Fast (delWidg v)]
            where v = w a
        w = turtle . obj.actseq
        h = obj.maxheight
```
oleafF h c = struct maxheight = h
    actseq a = leafF h a c

oplait n d c c’ = struct maxheight = 85
    actseq a = f a c++f (-a) c’
    where f = wave True 3 n d

owave n d c = struct maxheight = 85
    actseq a = wave False 3 n d a c

leafF h d c = [open, Jump y, Turn 90, Jump (-x), Turn a, Widg w, Turn (-a),
    Jump (x+x), Turn (-a), Widg (flipWidg False w), Close]
    where p@(x,y) = ((h*h-d*d)/(d+d), h/2)
    (dist,a) = (angle p0 p, distance p0 p)
    w = Arc0 (p0,0,c,0) Chord dist (a+a)

wave b k n d a c = Open c k:
    if b then right: Jump (-y/2): left: acts else acts
    where right = Turn 90; left = Turn (-90)
    acts = Jump (-fromInt n*x): right: Jump (-5):
        left: border a++border (-a)++[Close]
border a = foldl1 (<++>) (take n (repeat (step a)))++
   [right, Move 10, right]
step a = [Turn a, Move d, Turn (-a-a), Move d, Turn a]
(x, y) = successor p0 a d
Putting it all together

scaleAndDraw = action
    mapM_ (.stopScan) scans
    scan <- scanner tk drawWidget
    scans := [scan]
    let pict = pictures!!curr
    sizes <- mkSizes font (stringsInPict pict)
    (ns,ws) <- getEnclosed pict
    let (pict1,(x1,y1,x2,y2)) = f pict 0
        f (w:pict) i = (w’:pict’,minmax4 (widgFrame sizes w’) bds)
            where w’ = scaleWidg (sc i) w
                    (pict’,bds) = f pict (i+1)
        f _ _ = ([],(0,0,0,0))
    sc i = if just rect && i ‘elem‘ ns then rscale else scale
pict2 = map (transXY (5-x1) (5-y1)) pict1
pict3 = filter (not . isRedDot) pict2
compl = map (hullLines sizes) . minus1 pict3
anchor w = Dot c p where p = coords w
    c = if any (interior p) (compl w)
        then RGB 150 150 150 else black
(hull,rs) = convexPath sizes qs pict3
qs = if just rect then filter ('inRect'(get rect)) ps else ps
    where ps = map coords pict3
hullNos = zipWithIndices addNo rs
    where addNo i p = Text_ (p,0,dark red,0) [show i]
hulls = concatMap (hullPoints sizes) (removeSingles sizes pict3)
pictures := updList pictures curr
   (zipWithIndices (scaleWidg (recip (sc pict2)))))
widthX := max 100 (round (x2-x1+10))
widthY := max 100 (round (y2-y1+10))
canv.set [ScrollRegion (0,0) (widthX,widthY)]
if partBit then drawPict pict2
else case drawMore of
  0 -> drawPict pict2
  1 -> drawPict (pict3++map anchor pict3)
  2 -> drawPict (pict3++hull++hullNos)
  3 -> drawPict (pict3++markCross hulls)
  4 -> drawPict (pict3++meetHulls (showStrands True) hulls)
  5 -> drawPict (pict3++meetHulls (showStrands False) hulls)
  n -> drawPict (pict3++uniquePaths (meetHulls f hulls))
      where f = mergeStrands Path (n-6)
mapM_ drawArrow (getArcs sizes pict2 (edges!!curr))
if just rect then drawWidget (get rect)
Example: NDA (Examples/TRANS0)

defuncts: states
fovars: n
axioms: states = [0..10] &
(n < 6 & n \text{ mod } 2 = 0 \implies n \rightarrow [n,n+1]) &
(n < 6 & n \text{ mod } 2 \neq 0 \implies n \rightarrow n+1) &
6 \rightarrow [1,3,5,7..10]
Example: Five queens (Examples/QUEENS)

preds: cmp loop queens
fovars: n x y xs ys ps s s'

axioms: (x /= y+n & x /= y-n ==> cmp(x)(y,n)) 
(xs 'gives' x & zipAll(cmp(x))(ys)[1..length(ys)] 
  ==> (xs,ys) -> (xs-x,x:ys)) 
loop(xs,([],ys),zip(xs)(ys)) 
(s -> s' ==> (loop(xs,s,ps) <= loop(xs,s',ps))) 
(xs = [1..n] ==> (queens(n,ps) <= loop(xs,(xs,[]),ps)))
conjects: queens(5, ps)

ps = [(1, 4), (2, 2), (3, 5), (4, 3), (5, 1)]
| ps = [(1, 3), (2, 5), (3, 2), (4, 4), (5, 1)]
| ps = [(1, 5), (2, 3), (3, 1), (4, 4), (5, 2)]
| ps = [(1, 4), (2, 1), (3, 3), (4, 5), (5, 2)]
| ps = [(1, 5), (2, 2), (3, 4), (4, 1), (5, 3)]
| ps = [(1, 1), (2, 4), (3, 2), (4, 5), (5, 3)]
| ps = [(1, 2), (2, 5), (3, 3), (4, 1), (5, 4)]
| ps = [(1, 1), (2, 3), (3, 5), (4, 2), (5, 4)]
| ps = [(1, 3), (2, 1), (3, 4), (4, 2), (5, 5)]
| ps = [(1, 2), (2, 4), (3, 1), (4, 3), (5, 5)]
Example: Labelled transition relation (Examples/TRANS1)

counts: a b
defuncts: states
axioms: states = [1,2,3,4] & labels = [a,b] &
(2,b) -> [1,3] & (3,b) -> 3 & (3,a) -> 4 & (4,b) -> 3
Example: Turtle (Examples/turt)
Example: Table (Examples/polytab)
Example: Shelves (Examples/shelvesA)
Example: Shelves (Examples/shelvesB)
Example: Decorations (Examples/hull)
Example: Rainbows (Examples/raintabB)
Example: Pathfinder (Examples/ROBOT)

preds: loop
constructs: turt path place circ red
defuncts: cs
fovars: x’ y’ p q ps pa
axioms:

cs = [(2,6),(6,2)]

((p = (x+2,y) | p = (x,y+2)) & p 'NOTin' cs ==> (x,y) -> p) &

loop((8,12), path(ps), path(ps++[(8,12)])) &
(p < (8,12) & p -> q
  ==> (loop(p, path(ps), p) <=== loop(q, path(ps++[p]), p)))) &

((x,y) < (x,y') <=== y < y') &
((x,y) < (x',y) <=== x < x') &
((x,y) < (x',y') <=== x < x' & y < y')

conjects:

Any pa: (loop((0,0), path[], p) & turt(p:place(circ(2,red), cs)) = z)
Example: Plan formation (Examples/ROBOTACTS)

preds: loop

constructs: turt pathS place circ red 0 C blue M R L
defuncts: cs
fovars: x' y' s s' act act' acts acts1 acts2

axioms:

(s = (x+2,y) & s ‘NOTin’ cs ==> (x,y) -> (s,[M(2)])) &
(s = (x,y+2) & s ‘NOTin’ cs ==> (x,y) -> (s,[R,M(2),L])) &

loop((8,12),acts,acts) &
(s < (8,12) & s -> (s’,acts)
   ==> (loop(s,acts1,acts2) <=< loop(s’,acts1++acts,acts2))) &

conjects:

Any acts: (loop((0,0),[],acts) &
   turt(0(blue):acts++[C,place(circ(2,red),cs)]) = z)
Example: Pythagorean trees (Examples/PYTREE)

fovars: x y
axioms: trunk -> flipV(trunk) &
trunk -> grow(trunk,trunk) &
flipV(flipV(x)) -> x
conjects: trunk ++>
flipV(trunk) ++>
grow(trunk,trunk) ++>
grow(trunk,flipV(trunk)) ++>
pytree1 ++>
pytree2 ++>
file(pytree1code)
trunk c = path0 c 2 [p0, (-15, 0), (-15, -30), (-3, -45), (15, -30), (15, 0), p0]
grow c acts1 acts2 =
    Widg (trunk c): Turn 180: open: Jump 15: Turn 90: Jump 30: Turn 38.6598: Scale 0.640313: Jump 15: acts1++ close2++ open: Jump 3: Turn 90: Jump 45: Turn 129.806: Scale 0.781023: Jump 15: acts2++ close2

fractal "pytree" n c = open: f n c ++ [Close]
where f 0 c = []
    f n c = grow c acts acts where acts = f (n-1) (nextCol n c)

fractal "pytree" 12 blue ∼→
Example: Various trees (Examples/NICETREE)

fovars: n x

axioms:  
rhomb -> leaf(1.5,20)  
rhomb -> leafF(15,6)  
rhomb -> turt(blosF(10,5,2,red),blosF(5,3,1,yellow))  
rhomb -> polyR(5,[9,3])  
rhomb -> rhomb5(1)  
rhomb -> flipV(rhomb)  
rhomb -> grow5(1,rhomb,rhomb,rhomb,rhomb,rhomb)  
rhomb -> growR(1,rhomb,rhomb,rhomb,rhomb,rhomb)  
...  
flipV(flipV(x)) -> x
Example: Tattoo (Examples/tattoo)
Example: Stars (Examples/shinepoly3)
Example: Shuttles (Examples/shuttles2)
Example: Windmill (Examples/shineplait4)
Example: Carpet (Examples/CARPET)
Example: Yin & Yang (Examples/yinyang)
Example: Yin & Yang (Examples/yinyangR)
Example: Yin & Yang (Examples/yinyangRR)
Example: Yin & Yang (Examples/yinyangRR2)
Example: Concatenation (Examples/circtab)
Example: Concatenation (Examples/circtabA)
Example: Shuffling (Examples/circtabAS)
Example: Concatenation (Examples/circtabB)
Example: Shuffling (Examples/circtabBS)
Example: Shuffling (Examples/circtabBR2S)
Example: Reversal and shuffling (Examples/circtabBSR)