O’Haskell constructs and selected Expander2 code

Peter Padawitz

TU Dortmund, Germany

May 13, 2017
| 1. Data types       | 4 |
| 2. Records         | 5 |
| 3. Sub- and supertyping | 6 |
| 4. Templates       | 7 |
| 5. Main module of Expander2 | 8 |
| 6. Trees in Expander2 | 9 |
| 7. The solver template | 14 |
| 8. Proof step finalization | 17 |
| 9. Graphs in Expander2 | 20 |
| 10. Compiling polygons to paths | 25 |
| 11. Compiling polygons to pictures | 27 |
| 12. Compiling turtle actions to pictures | 29 |
| 13. Interpreting widget terms as pictures | 31 |
| 14. Template for processing widgets | 49 |
| 15. The painter template | 50 |
| 16. Draw actions of the painter template | 52 |
17. System.hs
18. The Tk environment
Data types

data Datatype = constructor1 type11 ... type1n1 |
    constructor2 type21 ... type2n2 |
    ...

  a = constructor1 term11 ... term1n1
  b = constructor2 term21 ... term2n2
Records

struct Record = selector1 :: type1 -> type1’
    selector2 :: type2 -> type2’

record = struct selector1 t1 = term1 (non-recursive)
    selector2 t2 = term2 (non-recursive)

OR

record = struct selector1 = selector1
    selector2 = selector2
     where selector1 t1 = term1 (recursive)
        selector2 t2 = term2 (recursive)

a = record.selector1
b = record.selector2
Sub- and supertyping

struct RecordS < Record = selectorS1 :: typeS1
    selectorS2 :: typeS2

data DatatypeS > Datatype = constructorS1 typeS11 ... typeS1nS1 |
    constructorS2 typeS21 ... typeS2nS2 |

Action < Cmd ()
Request a < Cmd a
Template a < Cmd a

struct Methods = method1 :: type11 ... type1n1 -> Action
    method2 :: type21 ... type2n2 -> Request type2
Templates (= object classes)

class :: type1 -> type2 -> ... -> Template Methods

class x1 x2 ... = template stateVar1 := term1
    stateVar2 := term2
    in struct method1 = action monad_term1 (non-recursive)
    method2 = request monad_term2 (non-recursive)
    where <local definitions>

OR

class x1 x2 ... = template stateVar1 := term1
    stateVar2 := term2
    in let <local definitions including
        recursive actions or requests>
        method1 = action monad_term1 (recursive)
        method2 = request monad_term2 (recursive)
        in struct ..Methods
    where <local definitions>

a <- class a1 a2 ...
module Ecom where

import System

main tk = do
    mkdir $ home ++ fileSeparator:"ExpanderLib"
    mkdir libPix
    mv "Painter.js" libPix
    win1 <- tk.window []
    win2 <- tk.window []
    fix solve1 <- solver tk "Solver1" win1 solve2 "Solver2" enum1 paint1
    solve2 <- solver tk "Solver2" win2 solve1 "Solver1" enum2 paint2
    paint1 <- painter 820 tk "Solver1" solve1 "Solver2" solve2
    paint2 <- painter 820 tk "Solver2" solve2 "Solver1" solve1
    enum1 <- enumerator tk solve1
    enum2 <- enumerator tk solve2
    solve1.buildSolve (0,20)
    solve2.buildSolve (20,20)
    win2.iconify
data Term a = V a | F a [Term a] | Hidden Special deriving (Show, Eq, Ord)

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

type TermS = Term String

type Simplification = (TermS, [TermS], TermS)

class Root a where undef :: a

instance Root Color where undef = white
instance Root Int where undef = 0
instance Root Float where undef = 0.0
instance Root [a] where undef = []
instance (Root a, Root b) => Root (a, b) where undef = (undef, undef)
instance (Root a, Root b, Root c) => Root (a, b, c) where undef = (undef, undef, undef)
isV (V _) = True
isV _ = False

isF (F _ _) = True
isF _ = False

ishidden = not . (isV ||| isF)

root :: Root a => Term a -> a
root (V x) = x
root (F x _) = x
root t = undef

subterms (F _ ts) = ts
subterms _ = []

-- label t p returns the root of the subterm at position p of t.
label :: Root a => Term a -> [Int] -> a
label t [] = root t
label (F _ ts) (n:p) | n < length ts = label (ts!!n) p
label _ _ = undef

-- getSubterm t p returns the subterm at position p of t.
getSubterm t [] = t
getSubterm (F _ ts) (n:p) | n < length ts = getSubterm (ts!!n) p
getSubterm \( t \_ \) = Hidden ERR

-- dropFromPoss \( p \_ \) removes the prefix \( p \) from each pointer of \( t \) below \( p \).

\[
\text{dropFromPoss} \( p \_ \) = \text{if null} \ p \ \text{then id else mapT} \ f \\
\text{where} \ f \ x = \text{if isPos} \ x \ \&\& \ p \ \ll\ q \\
\text{then mkPos0} \$ \ \text{drop} \ (\text{length} \ p) \ q \ \text{else} \ x \\
\text{where} \ q = \text{getPos} \ x
\]

-- getSubterm1 \( t \_ \) \( p \) returns the subterm \( u \) at position \( p \) of \( t \) and replaces each
-- pointer \( p++q \) in \( u \) by \( q \).

\[
\text{getSubterm1} \ t \ p = \text{dropFromPoss} \ p \$ \ \text{getSubterm} \ t \ p
\]

-- addToPoss \( p \_ \) \( t \) adds the prefix \( p \) to all pointers of \( t \) that point to subterms
-- of \( t \).

\[
\text{addToPoss} \ p \ t = \text{if null} \ p \ \text{then} \ t \ \text{else mapT} \ f \ t \\
\text{where} \ f \ x = \text{if isPos} \ x \ \&\& \ q \ \text{’elem’ positions} \ t \\
\text{then} \ \text{mkPos0} \$ \ p++q \ \text{else} \ x \ \text{where} \ q = \text{getPos} \ x
\]

-- changePoss \( p \_ \) \( q \) \( t \) replaces the prefix \( p \) of all pointers of \( t \) with prefix \( p \) by
-- \( q \).

\[
\text{changePoss} \ p \ q = \text{mapT} \ f \ \text{where} \ f \ x = \text{if isPos} \ x \ \&\& \ p \ \ll\ r \\
\text{then} \ \text{mkPos0} \$ \ q++\text{drop} \ (\text{length} \ p) \ r \ \text{else} \ x \\
\text{where} \ r = \text{getPos} \ x
\]
-- changeLPoss p q ts applies changePoss p(i) q(i) to ts for all 0 <= i <= |ts| - 1.

changeLPoss p q ts = map f ts where f t = foldl g t $ indices_ ts where
g t i = changePoss (p i) (q i) t

-- replace t p u expands t at all pointers into the subterm v of t at position
-- p. Pointers to the same subterm are expanded only once, the others are
-- redirected. Afterwards v is replaced by u.

replace t p0 u = f [] t
  where f p _ | p == p0 = u
    f p (F x ts)       = F x $ zipWithSucs f p ts
    f p (V x) | isPos x && p0 <<= q && not (p0 <<= p)
      = if p == r then movePoss t q p
        else mkPos r
          where q = getPos x
            Just r = lookup q $ g [] t
    f _ t = t
  g p _ | p == p0 = []
  g p (F x ts) = concat $ zipWithSucs g p ts
  g p (V x) | isPos x && p0 <<= q && not (p0 <<= p)
    = [(q,p)] where q = getPos x
  g _ t = []
-- replace1 t p u applies replace t p to u after all pointers of u into the
-- subterm of t at position p have been expanded.

replace1 t p = replace t p . addToPoss p

-- replace2 t p u q copies the subterm at position p of t to position q of u and
-- replaces each pointer p++r in the modified term by q++r.

replace2 t p0 u q0 = replace u q0 $ changePoss p0 q0 $ f [] $ getSubterm t p0
  where f p (F x ts) = F x $ zipWithSucs f p ts
        f p (V x) | isPos x && q0 << q && not (p0 <<= q) = movePoss t q p where q = getPos x
        f _ t = t
The solver template

struct Solver =
  addSpec :: Bool -> Action -> String -> Action
  backWin, bigWin, checkInSolver, drawCurr, forwProof, showPicts, skip, stopRun :: Action
  buildSolve :: Pos -> Action
  enterPT :: Int -> [Step] -> Action
  enterText :: String -> Action
  enterFormulas :: [TermS] -> Action
  enterTree :: Bool -> TermS -> Action
  getEntry, getSolver, getText :: Request String
  getFont :: Request TkFont
  getSignatureR :: Request Sig
  getTree :: Request (Maybe TermS)
  isSolPos :: Int -> Request Bool
  labBlue, labRed, labGreen :: String -> Action
  narrow :: Action -> Action
  saveGraphDP :: Bool -> Canvas -> Action
  setCurrInSolve :: Int -> Action -> Action
  setForw, setQuit :: [ButtonOpt] -> Action
  setNewTrees :: [TermS] -> String -> Action
  setSubst :: (String -> TermS, [String]) -> Action
  simplify :: Bool -> Action -> Action
data Step = ApplySubst | ApplySubstTo String TermS | ApplyTransitivity |
  BuildKripke Int | CollapseStep | ComposePointers |
  CopySubtrees | CreateIndHyp | CreateInvariant Bool |
  DecomposeAtom | DeriveMode Bool Bool | EvaluateTrees |
  ExpandTree Bool Int | FlattenImpl | Generalize [TermS] |
  Induction Bool Int | Mark [[Int]] | Match Int | Minimize |
  Narrow Int Bool | NegateAxioms [String] [String] | RandomLabels |
  RandomTree | ReleaseNode | ReleaseSubtree | ReleaseTree |
  RemoveCopies | RemoveEdges Bool | RemoveNode | RemoveOthers |
  RemovePath | RemoveSubtrees | RenameVar String |
  ReplaceNodes String | ReplaceOther |
  ReplaceSubtrees [[Int]] [TermS] | ReplaceText String |
  ReplaceVar String TermS [Int] | ReverseSubtrees | SafeEqs |
  SetAdmitted Bool [String] | SetCurr String Int | SetDeriveMode |
  SetMatch | ShiftPattern | ShiftQuants | ShiftSubs [[Int]] |
  Simplify Bool Int Bool | SplitTree | StretchConclusion |
  StretchPremise | SubsumeSubtrees | Theorem Bool TermS |
  UnifySubtrees | POINTER Step |

deriving Show

solver :: TkEnv -> String -> Window -> Solver -> String -> Enumerator |
  -> Painter -> Template Solver |

solver tk this win solve other enum paint =

template (backBut, canv, canvSlider, deriveBut, treeSlider, ent, fastBut, font, 
  forwBut, hideBut, interpreterBut, lab, matchBut, narrowBut, quit, safeBut, 
  simplButD, simplButB, splitBut, subToBut, tedit, termBut, lab2)
:= (undefined, undefined, undefined, undefined, undefined, undefined, undefined, undefined, undefined, undefined, undefined, undefined, undefined, undefined, undefined, undefined, undefined, undefined, undefined, undefined, undefined, undefined)

(tree, node, penpos, subtree, isSubtree, suptree, osci) := (Nothing, Nothing, Nothing, Nothing, Nothing, Nothing, Nothing)

(fast, firstMove, formula, showState, joined, safe, wtree) := (True, True, True, True, True, True, True)

(checking, checkingP, simplifying, refuting, collSimpls, newTrees, restore) := (False, False, False, False, False, False, False)

(canvSize, corner, counter, curr, curr1, hideVals, matching, proofPtr, proofTPtr, picNo, stateIndex) := ((0,0), (20,20), const 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)

(axioms, checkers, conjects, indClauses, iniStates, matchTerm, oldTreeposs, proof, proofTerm, refuteTerm, ruleString, simplRules, simplTerm, solPositions, specfiles, terms, theorems, transRules, treeposs, trees) := ([], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [])

numberedExps := ([], True); constraints := (True, [])

drawFun, picEval, picDir := ("", "tree", "picDir")
signatureMap := (id, []); newPreds := nil2; part := (id, [])

proofStep := ApplySubst; substitution := (V, []); treeMode := "tree"
symbols := iniSymbols; rand := seed; sizeState := sizes0

spread := (10, 30); times := (0, 300); maxHeap := 100; speed := 500

varCounter := const 0; perms := \n -> [0..n-1]
kripke := ([], [], [], [], [], [], [], [], [], [], [], [], [], [])

in let ... in struct ..Solver
setProof correct postSimpl msg ps labMsg = action
let oldProofElem = proof!!proofPtr
    t = trees!!curr
    n = counter 'd'
msg1 = msg 'elem' words "ADMITTED EQS"
msg2 = msg 'elem' words "MOVED SPLIT JOIN"
str = if msg1 then labMsg
    else if msg2 then labMsg ++ showCurr fast t formula
    else if newTrees
        then showNew fast (length trees) t msg n ps formula
        else showPre fast t msg n ps formula
str0 = "\nThe axioms have been MATCHED against their redices."
    ‘onlyif‘ matching < 2
str1 = "\nThe reducts have been simplified." ‘onlyif‘ simplifying
str2 str = "\nFailure "++ str ++" have been removed."
    ‘onlyif‘ refuting
str3 = if correct then case ruleString of
    "NARROWING" -> str0++str1++str2 "atoms"
    "REWRITING" -> str1++str2 "terms"
    _ -> str1 ‘onlyif‘ postSimpl
else "\nCAUTION: This step may be semantically incorrect!"
(msgP,msgL) = if null str3 then (str,labMsg)
    else (str++’\n’:str3,labMsg++str3)
msg3 = msgL ++ if newTrees || msg1 || msg2 || notnull msgL &&
    head msgL == ' ' || trees /= oldProofElem.trees
    then "" else "\nCAUTION: The "++ formString formula
++" has not been modified."

u = joinTrees treeMode trees
us = map (joinTrees treeMode . (.trees)) proof
cycle = search (eqTerm u) us
i = get cycle

cmsg i = "\nTHIS GOAL COINCIDES WITH GOAL NO. " ++ show i
msg4 = if just cycle then msg3 ++ cmsg i else msg3

if null ruleString || n > 0 then
proofPtr := proofPtr+1
let proof' = if nothing cycle then proof
    else updList proof $ extendMsg (cmsg proofPtr)
        $ proof!!i

next = struct msg = if just cycle then msgP ++ cmsg i else msgP
    msgL = msg4; treeMode = treeMode; trees = trees
treePoss = ps; curr = curr; perms = perms
varCounter = varCounter; newPreds = newPreds
solPositions = solPositions
substitution = substitution
constraints = constraints; joined = joined
safe = safe

proof := take proofPtr proof'++[next]
{-case u of F x ts | just cycle && permutative x
    -> let n = length ts
        if n > 1 then

perms := upd perms n $ nextPerm $ perms n
trees := [F x [ts!!i | i <- perms n]]
-- trees := [F x $ tail ts++[head ts]]
-- trees := [F x $ reverse ts]
curr := 0
        _ -> done-}
else picNo := picNo-1
newTrees := False; ruleString := ""
labColorToPaint green $ show proofPtr ++ ". " ++ msg4
Graphs in Expander2

type Point    = (Float,Float)
type Point3   = (Float,Float,Float)
type Line_    = (Point,Point)
type Lines    = [Line_]

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

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

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

type TurtleActs = [TurtleAct]
data TurtleAct = Close | Draw |
   -- Close and Draw finish a polygon resp. path starting at the
   -- preceding Open command.
   Jump Float | JumpA Float | Move Float | MoveA Float |
   -- JumpA and MoveA ignore the scale of the enclosing turtle.
   Open Color Int | Scale Float | Turn Float | Widg Bool Widget_
   -- The open mode ‘elem’ [0..5] (see drawWidg Path0)
   -- determines the mode of the path ending when the next
   -- Close/Draw command is reached.
   -- Widg False w ignores the orientation of w, Widg True w
   -- adds it to the orientation of the enclosing turtle.
deriving (Show,Eq)

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

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

type WidgTrans = Widget_ -> Widget_
instance Eq ArcStyleType where
  Chord == Chord = True
  Pie == Pie = True
  Perimeter == Perimeter = True
  _ == _ = False

isWTree (WTree _) = True
isWTree _ = False

p0 :: Point
p0 = (0,0)

st0 :: Color -> State
st0 c = (p0,0,c,0)

st0B :: State
st0B = st0 black

path0 :: Color -> Int -> Path -> Widget_
path0 = Path . st0

widg = Widg False

wait = widg Skip

noRepeat (Repeat _) = False
noRepeat _ = True
isFast (Fast _) = True
isFast _       = False

wfast = widg . fast

fast (Turtle st sc acts) = Fast $ Turtle st sc $ map f acts
  where f (Widg b w) = Widg b $ fast w
         f act         = act

fast (Bunch w is)    = Bunch (fast w) is
fast (Fast w)        = fast w
fast w                = Fast w

posWidg x = Text_ stOB 0 [x] [0]

Move 0<:>acts       = acts
Move a<:>(Move b:acts) = Move (a+b):acts
MoveA 0<:>acts      = acts
MoveA a<:>(MoveA b:acts) = MoveA (a+b):acts
Jump 0<:>acts       = acts
Jump a<:>(Jump b:acts) = Jump (a+b):acts
JumpA 0<:>acts      = acts
JumpA a<:>(JumpA b:acts) = JumpA (a+b):acts
Turn 0<:>acts       = acts
Turn a<:>(Turn b:acts) = Turn (a+b):acts
act<:>(act’:acts)   = act:act’:acts
act<:>_             = [act]
(act:acts) ++ acts' = act :> acts ++ acts'
     acts          = acts

reduceActs (act:acts) = act :> reduceActs acts
reduceActs _        = []

turtle0 :: Color -> TurtleActs -> Widget_
turtle0 c = Turtle (st0 c) 1

turtle0B, turtle1 :: TurtleActs -> Widget_
turtle0B     = turtle0 black
turtle1 acts = (case acts of Open c _:_ -> turtle0 c
                      Widg _ w:_ -> turtle0 $ getCol w
                      _        -> turtle0B) $ reduceActs acts

up    = Turn $ -90
down  = Turn 90
back  = Turn 180

open  = Open black 0
close2 = [Close,Close]

text0 (n,width) x = Text_ st0B n strs $ map width strs where strs = words x

(x',y') 'inRect' Rect ((x,y),_,_,_) b h = x-b <= x' && x' <= x+b &&
                      y-h <= y' && y' <= y+h
Compiling polygons to paths

-- Each widget is turned into a picture consisting of Arcs, Dots, Gifs,
-- horizontal or vertical Ovals, Path0s, Text_s and Trees before being drawn.

-- mkWidg (w (p,a,c,i) ...) rotates widget w around p by a.
-- mkWidg is used by drawWidget and hulls.

mkWidg :: WidgTrans
mkWidg (Dot c p) = Oval (p,0,c,0) 5 5
mkWidg (Oval (p,a,c,i) rx ry) = Path0 c i (filled c) $ map f [0,5..360]
    where f = rotate p a . successor2 p rx ry
mkWidg (Path (p,a,c,i) m ps) = Path0 c i m $ map (rotate p a . add2 p) ps
mkWidg (Poly (p,a,c,i) m rs b) = Path0 c i m $ last ps:ps
    where ps = circlePts p a b rs
mkWidg (Rect (p@(x,y),a,c,i) b h) = Path0 c i (filled c) $ last qs:qs
    where ps = [(x+b,y-h),(x+b,y+h),
               (x-b,y+h),(x-b,y-h)]
    qs = map (rotate p a) ps
mkWidg (Tria (p@(x,y),a,c,i) r) = Path0 c i (filled c) $ last qs:qs
    where ps = [(x+lg,z),(x-lg,z),(x,y-r)]
    lg = r*0.86602 -- r*3/(2*sqrt 3)
         = sidelength/2
    z = y+lg*0.57735 -- y+lg*sqrt 3/3
    qs = map (rotate p a) ps
circlePts :: Point -> Float -> Float -> [Float] -> Path
circlePts p a inc = fst . foldl f ([],a)
    where f (ps,a) 0 = (ps,a+inc)
              f (ps,a) r = (successor p r a:ps,a+inc)
Compiling polygons to pictures

mkPict :: Widget_ -> Picture

-- mkPict (Poly (p,a,c,i) mode rs b) with mode > 5 computes triangles or chords
-- of a rainbow polygon with center p, orientation a, inner color c, lightness
-- value i, radia rs and increment angle b.

mkPict (Poly (p,a,c,i) m (r:rs) b) = pict
  where (pict,_,_,_,_,_) = foldl f ([],successor p r a,a+b,c,1,False) $ rs++[r]
    lg = length rs+1
    f (pict,q@(x,y),a,c,k,d) r = if r == 0 then (pict,q,a+b,c,k+1,False)
    else (pict++new,p',a+b,c',1,d')
    where p'@(x',y') = successor p r a
           (new,c',d') = if m < 9
                         then if d then (pict',c,False)
                         else (pict',hue (m-5) c (lg 'div' 2) k,True)
                         else if m < 12
                              then (mkPict $ w c,hue (m-8) c lg k,d)
                              else if m < 15
                                   then (pict',hue (m-11) c lg k,d)
                                   else (mkPict $ w $ h 1,h $ k+k,d)
           pict' = fst $ iterate g ([],q)!!k
    g (pict,q) = (pict++[Path0 c i 4 [p,q,q'],q'],q')
    where q' = add2 q $ apply2 (/n) (x'-x,y'-y)
\[ h = \text{hue} \left( m - 14 \right) c \cdot 2 \cdot \log \]
\[ n = \text{fromInt} \ k \]
\[ w \ c' = \text{Turtle} \ (p, 0, c, i) \ 1 \ $ \text{Turn} \ (a - b \ast (n - 1) / 2) : \text{leafC} \ h \ d \ c \ c' \]
\[ \text{where} \ h = r / 2; \ d = n \ast \text{distance} \ (h, 0) \ (\text{successor} \ p0 \ h \ b) / 2 \]
Compiling turtle actions to pictures

-- mkPict (Turtle (p,a,c,i) sc acts) translates acts into the picture drawn by a
turtle that executes acts, starting out from point p with scale factor sc,
orientation a, color c and lightness value i.

mkPict (Turtle (p,a,c,i) sc acts) =
case foldl f iniState acts of (pict,(_,c,m,_,ps):_) -> g pict c m ps
_ -> []
where iniState = ([],[(a,c,0,sc,[p])])
f (pict,states@(a,c,m,sc,ps):s)) act =
case act of
  Jump d -> (g pict c m ps,(a,c,m,sc,[q]):s)
    where q = successor p (d*sc) a
  JumpA d -> (g pict c m ps,(a,c,m,sc,[q]):s)
    where q = successor p d a
  Move d -> (pict,(a,c,m,sc,ps++[q]):s)
    where q = successor p (d*sc) a
  MoveA d -> (pict,(a,c,m,sc,ps++[q]):s)
    where q = successor p d a
  Turn b -> (pict,(a+b,c,m,sc,ps):s)
  Open c m -> (pict,(a,c,m,sc,[p]):states)
  Scale sc' -> (pict,(a,c,m,sc*sc',[p]):states)
    -- or ps instead of [p] ?
  Close -> (g pict c m ps,s)
  Draw -> (g pict c m ps,(a,c,m,sc,[p]):s)
Widg b w -> (pict++[moveTurnScale b p a sc w],
    states)
_ -> (pict,states)

where p = last ps

g pict c m ps = if length ps < 2 then pict
    else pict++[Path0 c i m $ reduceP ps]

mkPict w = [w]
type Interpreter = Sizes -> Pos -> TermS -> Maybe Picture

jturtle :: TurtleActs -> Maybe Picture
jturtle = Just . single . turtle1

jfile = Just . single . File_

-- searchPic eval sizes spread t recognizes the maximal subtrees of t that are
-- interpretable by eval and combines the resulting pictures into a single one.

searchPic :: Interpreter -> Interpreter
searchPic eval sizes spread t = g [] $ expand 0 t []
    where g p t = case eval sizes spread t of
        pict@(Just _) -> pict
        _ -> do F _ ts <$> Just t
            concatJust $ zipWithSucs g p ts

-- solPic sig eval sizes spread t recognizes the terms of a solution t that are
-- interpretable by eval and combines the resulting pictures into a single one.

solPic :: Sig -> Interpreter -> Interpreter
solPic sig eval sizes spread t = do sol <- parseSol (solAtom sig) t
    let f = eval sizes spread . getTerm
concatJust $ map f sol

partition :: Int -> Interpreter
partition mode sizes _ = f where f (F "file" [F file []]) = jfile file
    f t = jturtle $ drawPartition sizes mode t

alignment,dissection,linearEqs,matrix,widgetTree,widgets :: Interpreter

alignment sizes _ = f
    where f (F "file" [F file []]) = jfile file
        f t
            = do ali <- parseAlignment t
                jturtle $ drawAlignment sizes ali

dissection _ _ (F "file" [F file []]) = jfile file
dissection _ _ (Hidden (Dissect quads)) = jturtle $ drawDissection quads
dissection _ _ t = do quads <- parseList parseIntQuad t
    jturtle $ drawDissection quads

linearEqs sizes _ = f
    where f (F "file" [F file []]) = jfile file
        f (F x [t]) | x 'elem' words "bool gauss gaussI" = f t
        f t
            = do eqs <- parseLinEqs t
                jturtle $ matrixTerm sizes $ g eqs 1
                g ((poly,b):eqs) n = map h poly++(str,"=",mkConst b):g eqs (n+1)
                where h (a,x) = (str,x,mkConst a); str = show n
        g _ _ = []
matrix sizes spread = f
  where f (Hidden (BoolMat dom1 dom2 pairs@(_:_)))
    = jturtle $ matrixBool sizes dom1 dom2
      $ deAssoc0 pairs
  f (Hidden (ListMat dom1 dom2 trips@(_:_)))
    = jturtle $ matrixList sizes dom1 dom
      $ map g trips
    where g (a,b,cs) = (a,b,map leaf cs)
        dom = mkSet [b | (_,b,:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_:_ (!_!!!)
ps = deAssoc2 $ get us
        (dom1,dom2) = sortDoms ps

f (F _ ts) | just us = jturtle $ matrixList sizes dom1 dom2 trs

where us = mapM parseConsts2Terms ts
        trs = deAssoc3 $ get us
        (dom1,dom2) = sortDoms2 trs

f _ = Nothing

widgetTree _ _ (F "file" [F file []]) = jfile file

widgetTree sizes spread t = Just [WTree $ f [] t]
where f :: [Int] -> TermS -> TermW
        f p (F "<+>" ts) = F Skip $ zipWithSucs f p ts
        f p (F "widg" ts@(::_)) = F w $ zipWithSucs f p $ init ts
            where v = expand 0 t $ p++[length ts-1]
                    w = case widgets sizes spread v of
                          Just [v] -> v
                          _ -> text $ showTerm0 v
        f p (F x ts) = F (text x) $ zipWithSucs f p ts
        f _ (V x) = V $ if isPos x then posWidg x else text x
        f _ _ = F (text "blue_hidden") []

text = text0 sizes

widgets sizes@(n,width) spread t = f black t
where next = nextColor 1 $ depth t
        f c (F "$" [t,u]) | just tr
            = do [w] <- fs c u; Just [get tr w]
                where tr = widgTrans t
            
        f c (F x ts) | just tr = f c (F (text x) ts)
        f c _ = f c (F (text "blue_hidden") [])

        fs c u = [v | t <- f c u, t /= []]
        get tr w = [w] ++ t
f c (F x []) | x `elem` words "TR SQ PE PY CA HE LB RB LS RS PS"
  = Just [mkTrunk c x]
f c (F x [n]) | x `elem` fractals
  = do n <- parsePnat n; jturtle $ fractal x n c
f c (F "anim" [t]) = do pict <- fs c t
  jturtle $ init $ init $ concatMap onoff pict
f c (F "arc" [r,a]) = do r <- parseReal r; a <- parseReal a
  Just [Arc (st0 c) Perimeter r a]
f c (F "bar" [i,h]) = do i <- parseNat i; h <- parsePnat h
  guard $ i <= h; jturtle $ bar sizes n i h c
f c (F x [t]) | z == "base"
  = do [w] <- fs c t
        w' <- mkBased (notnull mode) c w
        Just [w']
  where (z,mode) = splitAt 4 x

-- Based widgets are polygons with a horizontal line of 90 pixels
-- starting in (90,0) and ending in (0,0). mkBased and mkTrunk generate
-- based widgets.

f c (F x [n,r,a]) | z == "blos"
  = do hue:mode <- Just mode
        hue <- parse nat [hue]
        m <- search (== mode) leafmodes
        n <- parsePnat n; r <- parseReal r
        a <- parseReal a
        let next1 = nextColor hue n
next2 = nextColor hue $ 2*n
if m < 4 then
  jturtle $ blossom next1 n c
  $ case m of
    0 -> \c -> leafD r a c c
    1 -> \c -> leafA r a c c
    2 -> \c -> leafC r a c c
    _ -> leafS r a
  else jturtle $ blossomD next2 n c
  $ case m of
    4 -> leafD r a
    5 -> leafA r a
    _ -> leafC r a
where (z,mode) = splitAt 4 x

f c (F x [n]) | z == "cantP"
  = do mode <- search (== mode) pathmodes
       n <- parsePnat n
       Just [path0 c mode $ map fromInt2 $ take (n*n) $ iterate (cantor n) (0,0)]
where (z,mode) = splitAt 5 x

f c (F "center" [t]) = do [w] <- fs c t; Just [shiftWidg (center w) w]
f c (F "chord" [r,a]) = do r <- parseReal r; a <- parseReal a
                         Just [Arc (st0 c) Chord r a]
f c (F "chordL" [h,b]) = do h <- parseReal h; b <- parseReal b
                         jturtle $ chord True h b c
f c (F "chordR" [h,b]) = do h <- parseReal h; b <- parseReal b
                         jturtle $ chord False h b c
f c (F "circ" [r]) = do r <- parseReal r; Just [Oval (st0 c) r r]
f _ (F "colbars" [c]) = do c <- parseColor c
    jturtle $ colbars sizes n c
f c (F "dark" [t]) = do pict <- fs c t
    Just $ map (shiftLight $ -16) pict
f c (F "$" [F "dots" [n],t])
    = do n <- parsePnat n; pict <- fs c t
        Just $ dots n pict
f c (F "fadeB" [t]) = do [w] <- fs c t; jturtle $ fade False w
f c (F "fadeW" [t]) = do [w] <- fs c t; jturtle $ fade True w
f c (F "fast" [t]) = do pict <- fs c t; Just $ map fast pict
f c (F "fern2" [n,d,r])
    = do n <- parsePnat n; d <- parseReal d
        r <- parseReal r; jturtle $ fern2 n c d r
f c (F "file" [F file []])
    = jfile file
f c (F "flash" [t]) = do [w] <- fs c t; jturtle $ flash w
f c (F "flipH" [t]) = do pict <- fs c t; Just $ flipPict True pict
f c (F "flipV" [t]) = do pict <- fs c t; Just $ flipPict False pict
f c (F "$" [F "flower" [mode],u])
    = do mode <- parseNat mode; pict <- fs (next c) t
        jturtle $ flower c mode pict
f c (F "fork" [t]) = do pict <- fs c t; guard $ all isTurtle pict
    jturtle $ tail $ concatMap h pict
    where h (Turtle _ _ as) = widg New:as
          h _ = []
f c (F x [t]) | z == "frame"
    = do mode <- search (== mode) pathmodes
pict <- fs c t
Just $ map (addFrame c mode) pict
where (z,mode) = splitAt 5 x
f c (F "gif" [F file [],b,h])
  = do b <- parseReal b; h <- parseReal h
     Just [Gif c p0 file b h]
f c (F "gifs" [d,n,b,h])
  = do d <- parseConst d; n <- parsePnat n
      b <- parseReal b; h <- parseReal h
      let gif i = Gif c p0 (d++fileSeparator:d++
             '_':show i) b h
         Just $ map gif [1..n]
f c (F "grav" [t])  = do [w] <- fs c t
                               Just [shiftWidg (gravity w) w]
f c (F "$" [F "grow" [t],u])
  = do [w] <- fs c t; pict <- fs (next c) u
       jturtle $ grow id (updCol c w)
             $ map getActs pict
f c (F "$" [F "growT" [t,u],v])
  = do tr <- widgTrans t; [w] <- fs c u
       pict <- fs (next c) v
       jturtle $ grow tr (updCol c w)
             $ map getActs pict
f c (F x [n]) | z == "hilbP"
  = do mode <- search (== mode) pathmodes
       n <- parsePnat n
       Just [turtle0 c $ hilbert n East]
where \((z, \text{mode}) = \text{splitAt}~5~x\)

\[
f_c\ (F\ x\ [t])\ |\ z\ ==\ "hue"\ =\ \text{do}\ \text{acts} <- \text{parseList'}\ ((\text{parseAct}\ c)\ t)\ \\text{hue} <- \text{search}\ ((==\ \text{hue})\ \text{huemodes})\ \\text{let}\ \text{acts}' = \text{mkHue}\ ((\text{nextColor}\ $\ \text{hue} + 1)\ c\ \text{acts})\ \\text{Just}\ [\text{turtle0}\ c\ \text{acts}']\ \text{where}\ \ (z,\text{hue}) = \text{splitAt}\ 3\ x\
\]

\[
f_c\ (F\ x\ [t])\ |\ z\ ==\ "join"\ =\ \text{do}\ \text{mode} <- \text{parse}\ \text{pnat}\ \text{mode}\ \\text{guard}\ $\ \text{mode} '\text{elem'}\ [6..14];\ \text{pict} <- \text{fs}\ c\ t\ \\text{Just}\ [\text{mkTurt}\ p0\ 1\ $\ \text{extendPict}\ \text{mode}\ \text{pict}]\ \text{where}\ \ (z,\text{mode}) = \text{splitAt}\ 4\ x\
\]

\[
f_c\ (F\ x\ [r,a])\ |\ y\ ==\ "leaf"\ =\ \text{do}\ m <- \text{search}\ ((==\ \text{mode})\ \text{leafmodes})\ \\text{r} <- \text{parseReal}\ r;\ \text{a} <- \text{parseReal}\ a\ \\text{let}\ c' = \text{complColor}\ c\ \text{jturtle}$\ \text{case}\ m\ \text{of}\ \begin{align*}
& 0 \rightarrow \text{leafD}\ r\ a\ c\ c
& 1 \rightarrow \text{leafA}\ r\ a\ c\ c
& 2 \rightarrow \text{leafC}\ r\ a\ c\ c
& 3 \rightarrow \text{leafS}\ r\ a\ c
& 4 \rightarrow \text{leafD}\ r\ a\ c\ c'
& 5 \rightarrow \text{leafA}\ r\ a\ c\ c'
& _ \rightarrow \text{leafC}\ r\ a\ c\ c'
\end{align*}\ \text{where}\ \ (y,\text{mode}) = \text{splitAt}\ 4\ x
\]

\[
f_c\ (F^\ "light"\ [t])\ =\ \text{do}\ \text{pict} <- \text{fs}\ c\ t\ \text{Just}\ $\ \text{map}\ (\text{shiftLight}\ 21)\ \text{pict}$
\]

\[
f_\ _\ (F^\ "matrix"\ [t])\ =\ \text{matrix}\ \text{sizes}\ (0,0)\ t
\]

39
f c (F "\$" [F x [n],t]) | z == "morph"
    = do hue:mode <- Just mode
        hue <- parse nat [hue]
        guard $ hue ‘elem’ [1,2,3]
        mode <- search (== mode) pathmodes
        n <- parsePnat n; pict <- fs c t
        Just $ morphPict mode hue n pict
        where (z,mode) = splitAt 5 x

f _ (F "new" []) = Just [New]
f c (F "oleaf" [n]) = do n <- parsePnat n; jturtle $ oleaf n c
f c (F x [n,d,m]) | z == "owave"
    = do mode <- search (== mode) pathmodes
        n <- parsePnat n; d <- parseReal d
        m <- parseInt m
        jturtle $ owave mode n d m c
        where (z,mode) = splitAt 5 x

f c (F "outline" [t]) = do pict <- fs c t; Just $ outline pict
f c (F "oval" [rx,ry]) = do rx <- parseReal rx; ry <- parseReal ry
    Just [Oval (st0 c) rx ry]

f c (F x ps) | z == "path"
    = do mode <- search (== mode) pathmodes
        ps@((x,y):_) <- mapM parseRealReal ps
        let h (i,j) = (i-x,j-y)
            Just [path0 c mode $ map h ps]
        where (z,mode) = splitAt 4 x

f c (F x rs@(._:_)) | z == "peaks"
    = do m:mode <- Just mode
mode <- search (== mode) polymodes
rs <- mapM parseReal rs
guard $ head rs /= 0
jturtle $ peaks (m == 'I') mode c rs

where (z,mode) = splitAt 5 x

f c (F x (n:rs@(_:_))) | z == "pie"
  = do mode:hue <- Just mode
       let m = case mode of 'A' -> Perimeter
                   'C' -> Chord
                   _ -> Pie
       hue <- search (== hue) huemodes
       n <- parsePnat n; rs <- mapM parseReal rs
       jturtle $ pie m (nextColor $ hue+1) c
       $ concat $ replicate n rs
       where (z,mode) = splitAt 3 x

f _ (F "pile" [h,i]) = do h <- parsePnat h; i <- parseNat i
                          guard $ i <= h; jturtle $ pile h i
f _ (F "pileR" [t]) = do h:is <- parseList parseNat t
                        guard $ all (< h) is; jturtle $ pileR h is

f c (F "$" [F "place" [x,y],t])
  = do [w] <- fs c t; x <- parseReal x
       y <- parseReal y
       jturtle $ shiftTo (x,y) ++ [widg w]

f c (F x [n,d,m]) | z == "plait"
  = do mode <- search (== mode) pathmodes
       n <- parsePnat n; d <- parseReal d
       m <- parsePnat m
jturtle $ plait mode n d m c
where (z,mode) = splitAt 5 x
f c (F "\$" [F "planar" [n],t])
    = do maxmeet <- parsePnat n; [w] <- fs c t
        Just [planarWidg maxmeet w]
f c (F x (n:rs@(:_:_))) | z == "poly"
    = do mode <- search (== mode) polymodes
        n <- parsePnat n; rs <- mapM parseReal rs
        let k = n*length rs; inc = 360/fromInt k
        guard $ k > 1
        Just [Poly (st0 c) mode
            (take k $ cycle rs) inc]
        where (z,mode) = splitAt 4 x
f c (F "pulse" [t])
    = do pict <- fs c t; jturtle $ pulse pict
f c t
    = g c t
f c (F "rect" [b,h])
    = do b <- parseReal b; h <- parseReal h
        Just [Rect (st0 c) b h]
g c (F "repeat" [t])
    = do pict <- fs c t
        Just [Repeat $ turtle0B $ map widg pict]
g c (F "revpic" [t])
    = do pict <- fs c t; Just $ reverse pict
f c (F "rhomb" [])
    = Just [rhombV c]
g c (F "$" [F "rotate" [a],t])
    = do a <- parseReal a; guard $ a /= 0
        pict <- fs c t; jturtle $ rotatePict a pict
f c (F "$" [F "scale" [sc],t])
    = do sc <- parseReal sc; pict <- fs c t
        Just $ scalePict sc pict
g c (F "$" [F x (n:s),t]) | x ‘elem’ ["shelf","tower","shelfS","towerS"]
    = do n <- parsePnat n
         pict <- fs c t
         let k = if last x == 'S' then square pict
             else n
             c = if take 5 x == "shelf" then '1'
                 else '2'
         h d a b = Just $ fst $ shelf (pict,[]) k
         (d,d) a b False ['m',c]
    case s of
        d:s -> d <- parseReal d    -- spacing
        case s of
            a:s -> a <- parseChar a    -- alignment
            case s of
                b:s -> b <- parseChar b
                h d a $ b == 'C'
                _ -> h d a False
                _ -> h d 'M' False
                _ -> h 0 'M' False
    g _ (F "skip" [])         = Just [Skip]
    g c (F "slice" [r,a])    = do r <- parseReal r; a <- parseReal a
                                      Just [Arc (st0 c) Pie r a]
    g c (F "smooth" [t])     = do pict <- fs c t; Just $ smooth pict
    g c (F x [d,m,n,k,t]) | z == "snow"
                            = do hue <- search (== mode) huemodes
                               d <- parseReal d; m <- parsePnat m
                               n <- parsePnat n; k <- parsePnat k
\[ w \] <- \texttt{fs c t}
\[
\text{Just [mkSnow True \((\text{hue}+1)\) \(d \ m \ n \ k \ w\)]}
\text{where \((z,\text{mode}) = \text{splitAt} \ 4 \ x\)}
\]
\[
g \ c \ (\text{F "spline" } [t]) \quad = \text{do pict} <- \text{fs c t}; \text{Just} \ [\text{splinePict pict}]
\]
\[
g \ c \ (\text{F "star" } [n,r,r'])
\quad = \text{do} \ n <- \text{parsePnat} \ n; \ r <- \text{parseReal} \ r
\quad \quad r' <- \text{parseReal} \ r'; \text{jturtle} \ $ \ \text{star} \ n \ c \ r \ r'
\]
\[
g \ c \ (\text{F "$" [F "table" } [n,d],t])
\quad = \text{do} \ n <- \text{parsePnat} \ n; \ d <- \text{parseReal} \ d
\quad \quad \text{pict} <- \text{fs c t}; \text{Just} \ [\text{table pict d n}]
\]
\[
g \ c \ (\text{F "taichi" } s)
\quad = \text{jturtle} \ $ \ \text{taichi sizes} \ s \ c
\]
\[
g \ c \ (\text{F "text" } ts)
\quad = \text{do} \ \text{guard} \ $ \ \text{nonnull} \ strs
\quad \quad \text{Just} \ [\text{Text_ (st0 c) n strs $ map width strs}]
\quad \quad \text{where} \ strs = \text{words} \ $ \ \text{showTree False}$
\quad \quad \quad \ $\ \text{colHidden} $ \ \text{mkTup} \ ts
\]
\[
g \ c \ (\text{F "tree" } [t])
\quad = \text{Just} \ [\text{Tree st0B n c $ mapT h ct}]
\quad \quad \text{where} \ ct = \text{coordTree width spread}
\quad \quad \quad \quad (20,20) \ $ \ \text{colHidden} \ t
\quad \quad \quad \quad (_,\(x,y\)) = \text{root} \ ct
\quad \quad \quad \quad h \ (a,\(i,j\)) = (a,\text{fromInt2} \ (i-x,j-y),
\quad \quad \quad \quad \quad \text{width} \ a)
\]
\[
g \ c \ (\text{F "tria" } [r])
\quad = \text{do} \ r <- \text{parseReal} \ r; \text{Just} \ [\text{Tria (st0 c) r}]
\]
\[
g \ c \ (\text{F "$" [F "turn" } [a],t])
\quad = \text{do} \ a <- \text{parseReal} \ a; \text{pict} <- \text{fs c t}
\quad \quad \text{Just} \ $ \ \text{turnPict a pict}
\]
\[
g \ c \ (\text{F "turt" } [t])
\quad = \text{do} \ \text{acts} <- \text{parseList'} \ (\text{parseAct c}) \ t$
\quad \quad \text{Just} \ [\text{turtle0 c acts}]
\]
g c (F x [n,d,a]) | z == "wave"
   = do mode <- search (== mode) pathmodes
       n <- parsePnat n; d <- parseReal d
       a <- parseReal a
       jturtle $ wave mode n d a c
       where (z,mode) = splitAt 4 x

g _ (F x [t]) | just c = f (get c) t where c = parse color x
g _ _
   = Nothing

fs c t = do picts <- parseList' (f c) t; Just $ concat picts

parseAct c (V x) | isPos x = parseAct c $ getSubterm t $ getPos x
parseAct c (F "A" [t]) = widgAct True c t
parseAct _ (F "B" []) = Just back
parseAct _ (F "C" []) = Just Close
parseAct _ (F "D" []) = Just Draw
parseAct _ (F "J" [d]) = do d <- parseReal d; Just $ Jump d
parseAct _ (F "L" []) = Just up
parseAct _ (F "M" [d]) = do d <- parseReal d; Just $ Move d
parseAct c (F "O" []) = Just $ Open c 0
parseAct _ (F "O" [c]) = do c <- parseColor c; Just $ Open c 0
parseAct c (F "OS" []) = Just $ Open c 1
parseAct _ (F "OS" [c]) = do c <- parseColor c; Just $ Open c 1
parseAct c (F "OF" []) = Just $ Open c 2
parseAct c (F "OFS" []) = Just $ Open c 3
parseAct _ (F "OF" [c]) = do c <- parseColor c; Just $ Open c 4
parseAct _ (F "OFS" [c]) = do c <- parseColor c; Just $ Open c 5
parseAct _ (F "R" []) = Just down
parseAct _ (F "SC" [sc]) = do sc <- parseReal sc; Just $ Scale sc
parseAct _ (F "T" [a]) = do a <- parseReal a; Just $ Turn a
parseAct c t = widgAct False c t

widgAct b c t = do [w] <- fs c t ++ Just [text0 sizes $ showTerm0 t]
                Just $ Widg b w

huemodes = "":words "2 3 4 5 6"
pathmodes = "":words "S W SW F SF"
polymodes = pathmodes ++ words "R R1 R2 L L1 L2 T T1 T2 LT LT1 LT2"
trackmodes = words "asc sym ang slo"
leafmodes = words "D A C S D2 A2 C2"

fractals = words "bush bush2 cant cactus dragon fern gras grasL grasA grasC" ++
           words "grasR hexa hilb koch penta pentaS pytree pytreeA wide"

depth (F "$" [F "flower" _,t]) = depth t+1
depth (F "$" [F "grow" _,t]) = depth t+1
depth (F "$" [F "growT" _,t]) = depth t+1
depth (F _ ts) = maximum $ 1:map depth ts
depth _ = 1

-- The following widget transformations may occur as arguments of growT (see
-- widgets). They do not modify the outline of a widget and can thus be applied
-- to based widgets.
widgTrans :: TermS -> Maybe WidgTrans
widgTrans t = f t
  where f (F "." [t,u]) = do tr1 <- widgTrans t; tr2 <- widgTrans u
      Just $ tr1 . tr2

f (F x [F mode []]) | init z == "trac"
  = do guard $ typ 'elem' trackmodes
       m <- search (== m) pathmodes
       hue <- search (== hue) huemodes
       let h = if last z == 'c' then coords
           else gravity
       Just $ track h typ m $ nextColor $ hue+1
       where (z,hue) = splitAt 5 x
             (typ,m) = splitAt 3 mode

f (F x (n:s)) | z == "rainbow"
  = do n <- parsePnat n
       hue <- search (== hue) huemodes
       let next = nextColor (hue+1) n
       if null s then Just $ rainbow n 0 0 next
       else [a,d] <- mapM parseReal s
       Just $ rainbow n a d next
       where (z,hue) = splitAt 7 x

f (F "shine" (i:s)) = do i <- parseInt i
    guard $ abs i 'elem' [1..42]
    if null s then Just $ shine i 0 0
    else [a,d] <- mapM parseReal s
    Just $ shine i a d

f (F "inCenter" [tr]) = do tr <- widgTrans tr; Just $ inCenter tr
f _ = Nothing
Template for processing widgets

```
struct Scanner = startScan0 :: Int -> Picture -> Action
    startScan :: Int -> Action
    addScan :: Picture -> Action
    stopScan0 :: Action
    stopScan :: Action
    isRunning :: Request Bool

scanner :: TkEnv -> (Widget_ -> Action) -> Template Scanner
scanner tk act =
    template (run,running,as) := (undefined,False,[])
    in let startScan0 delay bs = action as := bs; startScan delay
        startScan delay = action if running then run.stop
            run0 <- tk.periodic delay loop
            run := run0; run.start; running := True
        loop = action case as of w:s -> if noRepeat w then as := s
            act w
            if isFast w then loop
        _ -> stopScan
        addScan bs = action as := bs++as
        stopScan0 = action stopScan; as := []
        stopScan = action if running then run.stop; running := False
        isRunning = request return running
    in struct ..Scanner
```
The painter template

struct Painter =
    callPaint :: [Picture] -> [Int] -> Bool -> Bool -> Int -> String
        -> Action -> Action

    labSolver :: String -> Action

    remote :: Action -> Action


    setCurrInPaint :: Int -> Action

    setEval :: String -> Pos -> Action

    setFast :: Bool -> Action

painter :: Int -> TkEnv -> String -> Solver -> String -> Solver
                  -> Template Painter
painter pheight tk solveName solve solveName2 solve2 =

    template (canv,combiBut,fastBut,edgeBut,font,lab,modeEnt,narrowBut,
                  pictSlider,saveEnt,colorScaleSlider,simplifyD,simplifyB,
                  spaceEnt,stopBut,win)
        := (undefined,undefined,undefined,undefined,undefined,undefined,
            undefined,undefined,undefined,undefined,undefined,undefined,
            undefined,undefined,undefined,undefined)
    (cols,curr,drawMode,grade,noOfGraphs,canvSize,spread,colorScale)
        := (0,0,0,0,0,(0,0),(0,0),(0,[]))
    (delay,oldRscale,rscale,yscale) := (1,1,1,1)
(arrangeMode, picEval, bgcolor) := ("", ",", white)
(changedWidgets, oldGraph) := (nil2, nil2)
(fast, connect, onlySpace, open, subtrees, isNew)
 := (False, False, False, False, False, True)
(edges, permutation, pictures, rectIndices, scans, solverMsg, treeNumbers)
 := ([], [], [], [], [], [], []
(oldRect, osci, penpos, rect, source, target, bunchpict)
 := (Nothing, Nothing, Nothing, Nothing, Nothing, Nothing, Nothing, Nothing)
in let ... in struct ..Painter
Draw actions of the painter template

drawPict pict = action
  if fast || all isFast pict then mapM_ drawWidget pict
  else let lgs = length scans
       (picts1,picts2) = splitAt lgs pict
       g sc pict = do run <- sc.isRunning
                      if run then sc.addScan pict else h sc pict
       h sc = sc.startScan0 delay
       zipWithM_ g scans pict1
       if lgp > lgs then scs <- accumulate $ replicate (lgp-lgs)
                         $ scanner tk drawWidget
                      zipWithM_ h scs pict2
       scans := scans++scs
  where pict = if New ‘elem’ pict then f pict [] [] else [pict]
       f (New:pict) pict’ = f pict (picts++[pict’]) []
       f (w:pict) pict’ = f pict pict’ (pict’++[w])
       f _ pict’ = pict’++[pict’]
       lgp = length pict

drawText (p,c,i) x = do
  let col = if deleted c then bgcolor
          else mkLight i $ case parse colPre x of
                      Just (c’,_) | c == black -> c’
                      _ -> c
canv.text (round2 p) [Text $ delQuotes x, NamedFont font, Fill col, Justify CenterAlign]

drawTree n (F cx@(x,q,lg) cts) ct nc c p = action
drawText (q,nc,0) x; drawTrees n x q lg cts ct nc c $ succsInd p cts
drawTree _ (V cx@(x,q,_) ) _ nc _ _ = action drawText (q,nc,0) x; done

drawTrees n x xy lg (ct:cts) ct0 nc c (p:ps) = action
canv.line [q,r] [Fill c]; drawTree n ct ct0 nc c p
drawTrees n x xy lg cts ct0 nc c ps
where (z,pz,lgz) = root ct
v = Text_ (xy,0,black,0) n [x] [lg]
w = Text_ (pz,0,black,0) n [z] [lgz]
q = round2 $ hullCross (pz,xy) v
r = round2 $ hullCross (xy,pz) w
drawTrees _ _ _ _ _ _ _ _ _ = done

drawWidget (Arc ((x,y),a,c,i) t r b) = action
let out = outColor c i bgcolor
fill = fillColor c i bgcolor
canv.arc (round2 (x-r,y-r)) (round2 (x+r,y+r)) $
[Angles $ round2 (-a,b), ArcStyle t, Outline out] ++
if t == Perimeter then [Fill out,Width $ round $ r/10]
else [fill]
doneda
drawWidget (Fast w) = action
if isPict w then mapM_ drawWidget $ mkPict w else drawWidget w
drawWidget (Gif c p file b h) = action
   if deleted c then drawWidget $ hull c $ Rect (p,0,c,0) b h
   else pic <- loadPhoto tk file
      canv.image (round2 p) [Img pic]
   done

drawWidget (Oval ((x,y),0,c,i) rx ry) = action
   canv.oval (round2 (x-rx,y-ry)) (round2 (x+rx,y+ry))
   [Outline $ outColor c i bgcolor,fillColor c i bgcolor]
   done

drawWidget (Path0 c i m ps) = action
   let fill = fillColor c i bgcolor
      out = outColor c i bgcolor
   in
   optsL :: Int -> [LineOpt]
   optsL 0 = [Fill out]
   optsL 1 = [Fill out,Smooth True]
   optsL 2 = [Fill out,Width 2]
   optsL _ = [Fill out,Width 2,Smooth True]
   optsP :: Int -> [PolygonOpt]
   optsP 4 = [Outline out,fill]
   optsP _ = [Outline out,fill,Smooth True]
   if m < 4 then act canv.line $ optsL m
      else act canv.polygon $ optsP m
   where act f opts = mapM_ (flip f opts . map round2) $ splitPath ps
   -- do flip f opts $ map round2 ps; done

drawWidget (Repeat w) = drawWidget w

drawWidget (Saved file hull) = action
   w <- loadWidget tk file
drawWidget $ moveWidg (coords hull) w
drawWidget Skip = action done
drawWidget (Text_ (p,_,c,i) n strs lgs) = action
    zipWithM_ f [0..] strs where (_,_,ps) = textblock p n lgs
    f k = drawText (ps!!k,c,i)
drawWidget (Tree (p@(x,y),a,c,i) n c’ ct) = action
    drawTree n ct’ ct’ (outColor c i bgcolor) c’ []
    where ct’ = mapT3 f ct; f (i,j) = rotate p a (i+x,j+y)
drawWidget w | isWidg w = drawWidget $ mkWidg w
    | isPict w = drawPict $ mkPict w
drawWidget _ = action done

scaleAndDraw msg = action
    mapM_ (.stopScan0) scans; canv.clear
    sc <- scanner tk drawWidget; scans := [sc]
    stopBut.set [Text "stop", Command $ interrupt True]
n <- saveEnt.getValue
    let maxmeet = case parse pnat n of Just n -> n; _ -> 200
        graph = (pictures!!curr,edges!!curr)
        reduce = planarAll maxmeet graph
        (graph’,is) = if drawMode == 15 &&
            msg /= "A subgraph has been selected."
            then if just rect
                then reduce rect rectIndices rscale
            else reduce Nothing [] scale
            else (graph,rectIndices)
        (pict,arcs) = bunchesToArcs graph’
(pict1, bds) = foldr f ([], (0, 0, 0, 0)) $ indices_ pict
f i (ws, bds) = (w:ws, minmax4 (widgFrame w) bds)
    where w = scaleWidg (sc i) $ pict!!i
sc i = if i 'elem' is then rscale else scale
(x1, y1, x2, y2) = if just rect
    then minmax4 (widgFrame $ get rect) bds else bds
size = apply2 (max 100 . round . (+10)) (x2 - x1, y2 - y1)
translate = transXY (-x1, -y1)
pict2 = map translate pict1
g = scaleWidg . recip . sc
pictures := updList pictures curr $ zipWith g [0..] pict2
edges := updList edges curr arcs
canvSize := size
canv.set [ScrollRegion (0, 0) size]
let pict3 = map (transXY (5, 5)) pict2
    pict4 = h pict3
    h = filter propNode
ws = if just rect then h $ map (pict3!!) is else pict4
(hull, qs) = convexPath (map coords ws) pict4
drawArrow ps = do canv.line (map round2 ps)
    $ if arrangeMode == "d1" then [Smooth True]
    else [Arrow Last, Smooth True]
k = treeNumbers!!curr
if drawMode 'elem' [0, 15] then drawPict pict3
else case drawMode of
    1 -> drawPict pict4
    2 -> drawPict $ h $ colorLevels True pict3 arcs

56
3 -> drawPict $ h $ colorLevels False pict3 arcs
4 -> drawPict $ pict4++hull
5 -> (n,wid) <- mkSizes font $ map show qs
   let addNo x p = Text_ (p,0,dark red,0) n [x] [wid x]
   drawPict $ pict4++hull++zipWith (addNo . show) [0..] qs
_ -> drawPict $ extendPict drawMode pict4
if arrangeMode /= "d2"
  then mapM_ drawArrow $ buildAndDrawPaths (pict3,arcs)
if just rect then let (x1,y1,x2,y2) = pictFrame $ map (pict2!!) is
  (b,h) = (abs (x2-x1)/2,abs (y2-y1)/2)
  r = Rect ((x1+b,y1+h),0,black,0) b h
  rect := Just r; draw55 [r]
solver <- solve.getSolver; b <- solve.isSolPos k
let str1 = if subtrees then subtreesMsg solver
            else treesMsg k noOfGraphs solver b
    add str = if null str then "" else '\n':str
labGreen $ str1 ++ add solverMsg ++ add msg
module System where

import Tk

data ExitCode = ExitSuccess | ExitFailure Int deriving (Eq,Ord,Read,Show)

primitive primSystem :: String -> Request Int -- IO Int
primitive doesFileExist :: FilePath -> Cmd Bool -- IO Bool
primitive doesDirectoryExist :: FilePath -> Cmd Bool
primitive createDirectory :: FilePath -> Cmd () -- IO ()
primitive getDirectoryContents :: FilePath -> Cmd [FilePath]
primitive primGetAppDirectory :: FilePath
primitive primGetFileSeparator :: Char
primitive primGetOS :: Int

home = primGetAppDirectory

fileSeparator = primGetFileSeparator

expanderLib = home ++ fileSeparator:"ExpanderLib" ++ [fileSeparator]

libPix = expanderLib ++ "Pix"
pixpath file = libPix ++ fileSeparator:file

mkdir, rmdir :: FilePath -> Request ExitCode
mkdir dir = system $ "mkdir " ++ dir -- rawSystem "mkdir" [dir]
rmdir dir = system $ "rm -rf " ++ dir

mv :: FilePath -> FilePath -> Request ExitCode
mv file dir = system $ "mv -n " ++ file ++ ' ' :dir

system :: String -> Request ExitCode -- IO ExitCode
system cmd = do ec <- primSystem cmd
               return $ if ec == 0 then ExitSuccess else ExitFailure ec

savePng :: Canvas -> String -> Cmd FilePath
savePng canv file = do canv.save file1
                       system $ "convert " ++ file1 ++ ' ' :file2
                       system $ "convert " ++ file2 ++ " -trim " ++ file2
                       system $ "rm -f " ++ file1
                       return file2
                       where file1 = file ++ " .eps"
                           file2 = file ++ " .png"

lookupExamples :: TkEnv -> FilePath -> Cmd String
lookupExamples tk file = tk.readFile (homeExamples ++ file) 'catch' handler
                        where handler _ = tk.readFile ("Examples" ++ fileSeparator:file)
                           'catch' const (return "")
data OSType = Unknown | Windows | Unix | Dos | RiscOS
  deriving (Eq, Read, Show, Enum, Ord)

getOS :: OSType
getOS = toEnum primGetOS
module Tk where

struct Tk =
    window :: [WindowOpt] -> Request Window
    bitmap :: [BitmapOpt] -> Request ConfBitmap
    photo :: [PhotoOpt] -> Request Photo
    delay :: Int -> (String -> Cmd ()) -> Request String
    periodic :: Int -> Cmd () -> Request Runnable
    bell :: Action

primTk :: Template Tk
primTk =
    template in
        let window opts = request
            x <- primGetPath
            primExTcl_["toplevel",x]
            winsetcmd x opts
            win x
        bell = primExTcl_ ["bell"]
        delay t a = request
            n <- primNextCallBack
            tag <- primExTcl ["after",show t, 
            
            let tag' = drop 6 tag -- all tags start with "after#"
primAddCallBack (\_ -> a tag')
return tag'
periodic t a = request
    n <- primAddCallBack (\_ -> a)
    let ln = "loop"++show n
    primExTcl_['proc',ln,'{args} {haskellEvent ",show n,
        "\nupdate\nafter",show t,ln,"}"
    ]
    hnd ln
    bitmap opts = request
        os <- textOpts opts
        nm <- primExTcl['image create bitmap",os]
        btmp nm
    photo opts = request
        os <- textOpts opts
        nm <- primExTcl['image create photo",os]
        phto nm
    in struct ..Tk

primExTcl  = primExecuteTcl . unwords
primExTcl_ = primExecuteTcl_ . unwords

primitive primExecuteTcl "primExecuteTcl" :: String -> Request String
primitive primExecuteTcl_ "primExecuteTcl_" :: String -> Action
primitive primGetPath "primGetPath" :: Request String
primitive primAddCallBack "primAddCallBack" :: (String -> Cmd ()) -> Request Int
primitive primNextCallBack "primNextCallBack" :: Request Int
-- Windows

struct BasicWindow a < ConfWidget a =
    button :: [ButtonOpt] -> Request Button
    canvas :: [CanvasOpt] -> Request Canvas
    checkButton :: [CheckButtonOpt] -> Request CheckButton
    entry :: [EntryOpt] -> Request Entry
    frame :: [FrameOpt] -> Request Frame
    label :: [LabelOpt] -> Request Label
    listBox :: [ListBoxOpt] -> Request ListBox
    menuButton :: [MenuButtonOpt] -> Request MenuButton
    radioButton :: [RadioButtonOpt] -> Request RadioButton
    scrollBar :: [ScrollBarOpt] -> Request ScrollBar
    slider :: [SliderOpt] -> Request Slider
    textEditor :: [TextEditorOpt] -> Request TextEditor

type Pos = (Int,Int)

struct ManagedWindow =
    getGeometry :: Request (Pos,Pos) -- size,position
    setSize :: Pos -> Action
    setPosition :: Pos -> Action
    iconify :: Action
    deiconify :: Action
-- top level windows

struct Window < BasicWindow WindowOpt, ManagedWindow

-- Images

struct Image =
    imageName :: String

struct Bitmap < Image

struct ConfBitmap < Bitmap, Configurable BitmapOpt

struct PredefBitmap < Bitmap

struct Photo < Image, Configurable PhotoOpt =
    blank :: Action
    putPixel :: Pos -> Color -> Action
    getPixel :: Pos -> Request Color
    copyFrom :: Photo -> Action -- to be refined
    saveAs :: FilePath -> Action

struct Runnable =
    start :: Action
    stop :: Action

struct TkEnv < Tk, StdEnv
-- General widget structures

struct Widget =
    ident :: String
    destroy :: Action
    exists :: Request Bool
    focus, raise, lower :: Action
    bind :: [Event] -> Action

struct Configurable a =
    set :: [a] -> Action

struct ConfWidget a < Widget, Configurable a a

-- Structures for subtyping by WWidgets

struct Cell a =
    setValue :: a -> Action
    getValue :: Request a

struct LineEditable =
    lines :: Request Int
    getLine :: Int -> Request String
    deleteLine :: Int -> Action
    insertLines :: Int -> [String] -> Action
struct Invokable =
    invoke :: Action

struct Packable =
    packIn :: String -> Dir -> Stretch -> Expansion -> Cmd ()
    wname :: String

struct Scannable a =
    mark :: a -> Action
    drag :: a -> Action

struct WWidget a < ConfWidget a, Packable

struct ScrollWidget a < WWidget a =
    xview :: Request (Double,Double)
    yview :: Request (Double,Double)

-- Window widgets

struct Frame < BasicWindow FrameOpt, WWidget FrameOpt

struct Slider < WWidget SliderOpt, Cell Int

struct Button < WWidget ButtonOpt, Invokable =
    flash :: Action
struct CheckButton < Button =
  toggle  :: Action
  checked :: Request Bool

struct RadioButton < Button =
  select  :: Action
  deselect :: Action

struct MenuButton < WWidget MenuButtonOpt =
  menu :: [MenuOpt] -> Request Menu

struct Label < WWidget LabelOpt

struct ListBox < ScrollWidget ListBoxOpt, LineEditable, Cell [Int],
  Scannable Pos =
  view :: Int -> Action

struct TextEditor < ScrollWidget TextEditorOpt, LineEditable, Scannable Pos

struct Entry < ScrollWidget EntryOpt, Cell String, Scannable Int =
  cursorPos :: Request Int
struct Canvas < ScrollWidget CanvasOpt, Scannable Pos =

  oval  :: Pos -> Pos -> [OvalOpt]   -> Request Oval
  arc   :: Pos -> Pos -> [ArcOpt]    -> Request Arc
  rectangle :: Pos -> Pos -> [RectangleOpt] -> Request Rectangle
  line  :: [Pos]    -> [LineOpt]     -> Request Line
  polygon :: [Pos]  -> [PolygonOpt]  -> Request Polygon
  text   :: Pos      -> [CTextOpt]   -> Request CText
  image  :: Pos      -> [CImageOpt]  -> Request CImage
  cwindow :: Pos     -> [CWindowOpt] -> Request CWindow
  clear   :: Action
  save    :: FilePath -> Action

struct ScrollBar < WWidget ScrollBarOpt =

  attach :: ScrollWidget BasicWOpt -> Dir -> Action
-- Canvas Widgets

struct CWidget a < ConfWidget a =
    getCoords :: Request [Pos]
    setCoords :: [Pos] -> Action
    move :: Pos -> Action

struct Arc < CWidget ArcOpt
struct Oval < CWidget OvalOpt
struct Rectangle < CWidget RectangleOpt
struct Line < CWidget LineOpt
struct Polygon < CWidget PolygonOpt
struct CText < CWidget CTextOpt
struct CImage < CWidget CImageOpt
struct CWindow < CWidget WindowOpt, BasicWindow WindowOpt

-- Menus

struct Menu < ConfWidget MenuOpt =
    mButton :: [MBottonOpt] -> Request MButton
    cascade :: [MButtonOpt] -> Request Menu

struct MButton < Configurable MButtonOpt, Invokable
-- Color

data Color = RGB Int Int Int deriving Eq

black = RGB 0 0 0
white = RGB 255 255 255
red = RGB 255 0 0
green = RGB 0 255 0
blue = RGB 0 0 255
yellow = RGB 255 255 0

-- Auxiliary types for options

data None = None

data AnchorType = NW | N | NE | W | C | E | SW | S | SE

data ReliefType = Raised | Sunken | Flat | Ridge | Solid | Groove

data VertSide = Top | Bottom

data WrapType = NoWrap | CharWrap | WordWrap

data SelectType = Single | Multiple

data Align = LeftAlign | CenterAlign | RightAlign

data Round = Round

data ArcStyleType = Pie | Chord | Perimeter

data CapStyleType = Round = Butt | Proj

data JoinStyleType = Round = Bevel | Miter

data ArrowType = None = First | Last | Both
-- Options

data Anchor = Anchor AnchorType
...

-- widget option types

data BasicOpt > Background, BorderWidth, Cursor, Relief
data BasicWOpt > BasicOpt, Width
data DimOpt > Height, Width
data StdOpt > BasicWOpt, DimOpt
data FontOpt > Font, Foreground, Anchor, Justify
data PadOpt > Padx, Pady

data WindowOpt > BasicOpt, Title
data PhotoOpt > DimOpt, File
data BitmapOpt > Background, Foreground, File, BitmapData

data ButtonOpt > MenuButtonOpt, Command
data CanvasOpt > StdOpt, ScrollRegion
data CheckButtonOpt > ButtonOpt, Indicatoron, SelectColor
data EntryOpt > BasicWOpt, Justify, Font, Foreground, Enabled
type FrameOpt = StdOpt
data LabelOpt > StdOpt, FontOpt, PadOpt, Img, Btmp, Underline, Text
data ListBoxOpt > StdOpt, Font, Foreground, SelectMode
data MenuButtonOpt > LabelOpt, Enabled
type RadioButtonOpt = CheckButtonOpt
type ScrollBarOpt = StdOpt

data SliderOpt > BasicWOpt, From, To, Orientation, Length,
                  Font, Foreground, CmdInt, Enabled

data TextEditorOpt > StdOpt, Font, Foreground, PadOpt, Wrap, Enabled

data CBasicOpt > Fill, Width, Stipple

data CImageOpt > Anchor, Img, Btmp

data CTextOpt > Font, Justify, Text, Anchor, Fill

data CWindowOpt > DimOpt, Anchor

data LineOpt > CBasicOpt, Arrow, Smooth, CapStyle, JoinStyle

data PolygonOpt > OvalOpt, Smooth

data ArcOpt > OvalOpt, ArcStyle, Angles

data OvalOpt > CBasicOpt, Outline

data RectangleOpt > OvalOpt

data MenuOpt > WindowOpt, Enabled

data MButtonOpt > StdOpt, FontOpt, PadOpt, Img, Btmp, Underline,
                  CLabel, Enabled, Command

data AllOpt > MenuOpt, CheckButtonOpt, TextEditorOpt, FrameOpt,
              LineOpt, WindowOpt, ArcOpt, PolygonOpt,
              OvalOpt, CTextOpt, RectangleOpt, SliderOpt, MButtonOpt,
              CanvasOpt, ListBoxOpt, BitmapOpt, PhotoOpt, CImageOpt,
              EntryOpt, CWindowOpt, ButtonOpt, MenubuttonOpt,
              LabelOpt

72
--- Events

data ButtonPress = ButtonPress Int (Pos -> Cmd ())
  | AnyButtonPress (Int -> Pos -> Cmd ())

data MouseEvent > ButtonPress =
  ButtonRelease Int (Pos -> Cmd ())
  | AnyButtonRelease (Int -> Pos -> Cmd ())
  | Motion Int (Pos -> Cmd ())
  | AnyMotion (Pos -> Cmd ())
  | Double ButtonPress
  | Triple ButtonPress

data WindowEvent = Enter (Cmd ())
  | Leave (Cmd ())
  | Configure (Pos -> Cmd ())

data SimpleKeyEvent = KeyPress String (Cmd ())
  | KeyRelease String (Cmd ())
  | AnyKeyPress (String -> Cmd ())

data KeyEvent > SimpleKeyEvent = Mod [Modifier] SimpleKeyEvent

data DestroyEvent = Destroy (Cmd ())

data Event > MouseEvent, KeyEvent, WindowEvent, DestroyEvent