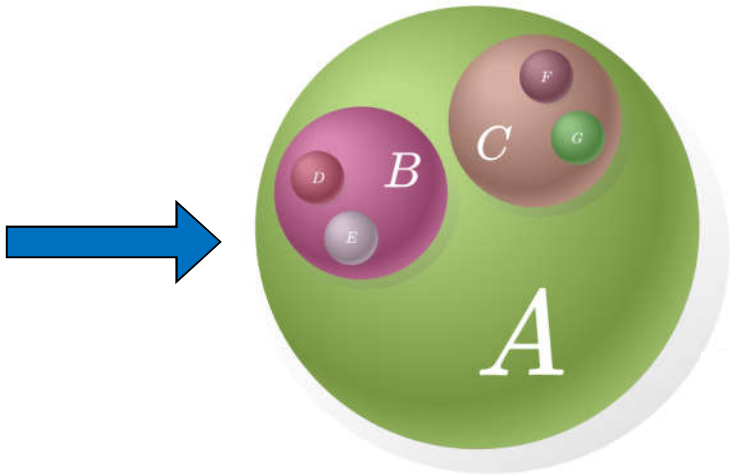
Penrose: From Mathematical Notation to Beautiful Diagrams

Jonathan Aldrich, Carnegie Mellon University

Joint work with Katherine Ye, Wode Ni, Max Krieger, Dor Ma'ayan, Jenna Wise, Yumeng Du, Lily Shellhammer, Joshua Sunshine, and Keenan Crane

Set A, B, C, D, E, F, G
B ⊂ A
C ⊂ A
D ⊂ B
E ⊂ B
F ⊂ C
G ⊂ C
E ∩ D = ∅
F ∩ G = ∅
B ∩ C = ∅



Diagrams are useful, but too rare

- Diagrams are useful…
  - Diagrams help in solving math problems [Larkin&Simon]
  - High-impact papers have many figures [Lee et al.]

- But rare: just 39% of arXiv math papers contain diagrams
  - And even those contain only 1 figure for every 10 pages

"People have very powerful facilities for taking in information visually... On the other hand, they do not have a good built-in facility for turning an internal spatial understanding back into a two-dimensional image. [So] mathematicians usually have fewer and poorer figures in their papers and books than in their heads."

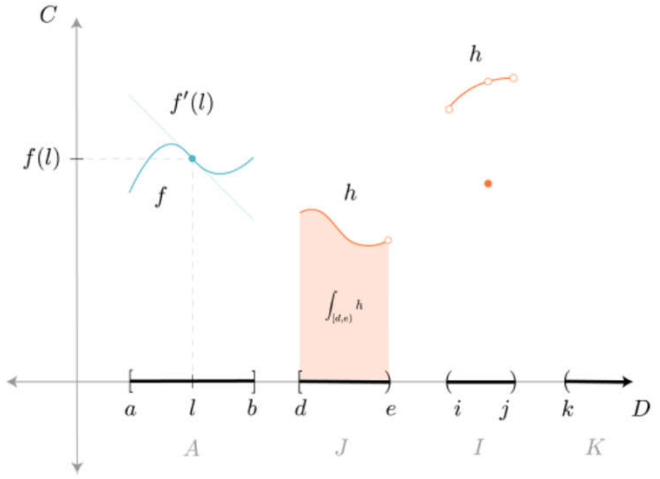
- Fields medalist William Thurston

The Penrose Vision

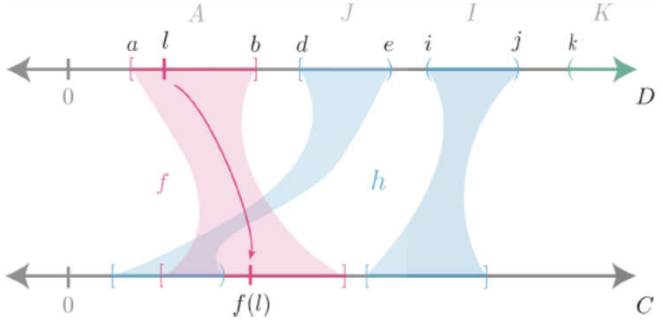
You write this:

```
a, b, d, e, i, j, k, inf ∈ ℝ
b IsLessThan k
A := [a, b] ⊆ ℝ
J := [d, e) ⊆ ℝ
I := (i, j) ⊆ ℝ
K := (k, inf) ⊆ ℝ

f : A → ℝ
f IsDifferentiable

U := (J ∪ I) ⊆ ℝ
h : U → ℝ
h IsDiscontinuous

l ∈ A
p2 := Pt(l, f(l))
dfl := f'(l)
ih := ∫_J h
```

Penrose generates this:



Or, if you prefer, this:



Penrose in Action

- Linear algebra – simple intro
- Linear algebra – sugar and direct manipulation
- SIGGRAPH teaser video

Can we create a LaTeX for Diagrams?

LaTeX

- Describe document content (.tex) separate from layout
- Extensible formatting styles (.sty)
- Extensible with new document structuring concepts (macros)
- Optimizes (mostly textual) layout of documents

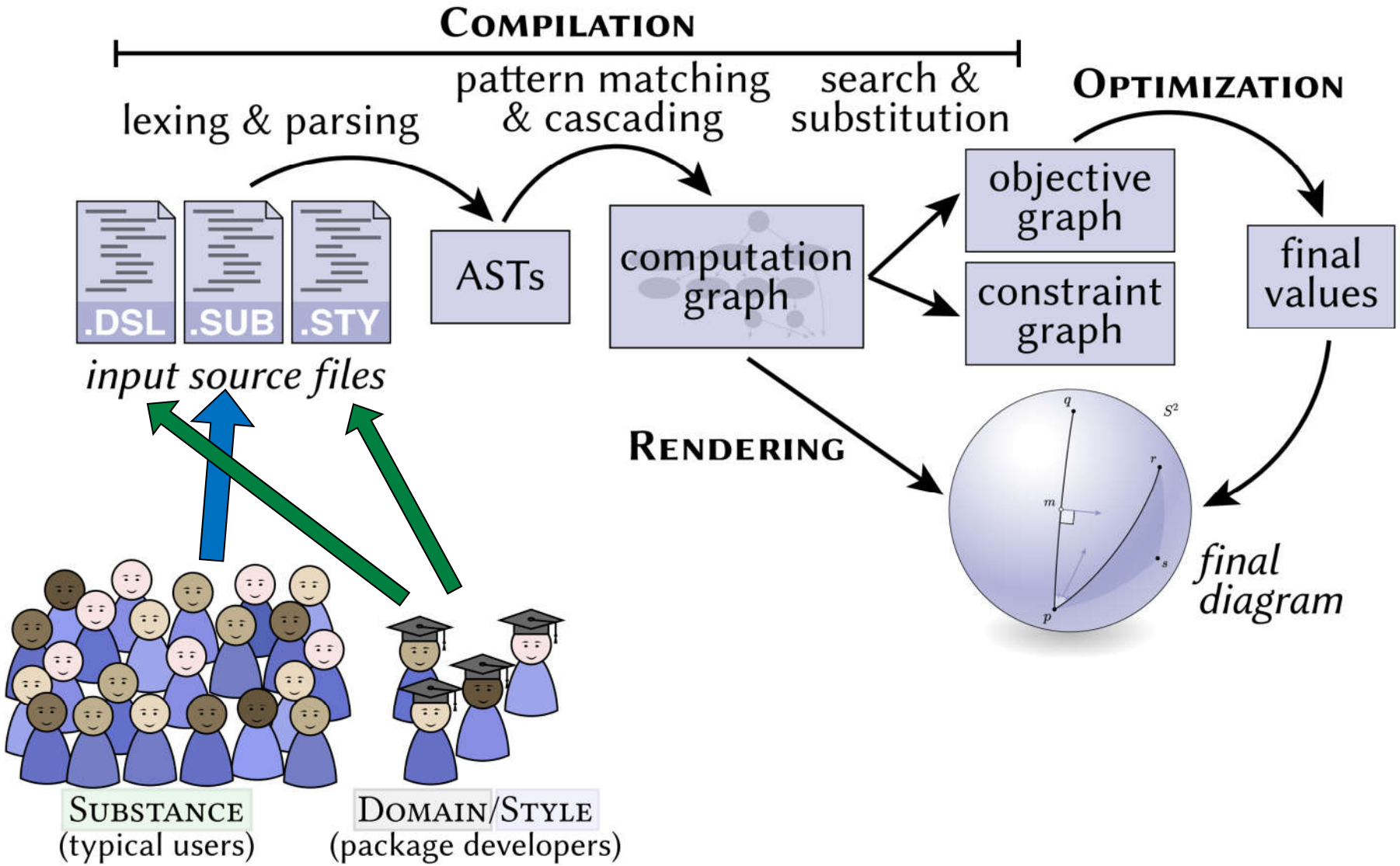
Penrose

- Describe mathematical content (Substance) separate from visual representation
- Extensible rendering (Style)
- Extensible with new math domains (Domain)
- Optimizes (graphical) layout of diagrams

Existing tools are inadequate

- Graphing calculators (e.g. Wolfram Alpha)
  - Visualize concrete data or functions
  - Don't understand, can't visualize mathematical abstractions
- Drawing tools (e.g. Adobe Illustrator, TikZ)
  - Require laborious specification of low-level details
  - Don't understand semantics
- Domain-specific visualizations (e.g. Group Explorer)
  - Work well for a particular domain, but are not extensible

The Penrose Architecture and Users



Anatomy of a Substance Program

Object U of type VectorSpace

VectorSpace U

Vector $u_1, u_2, u_3, u_4, u_5 \in U$

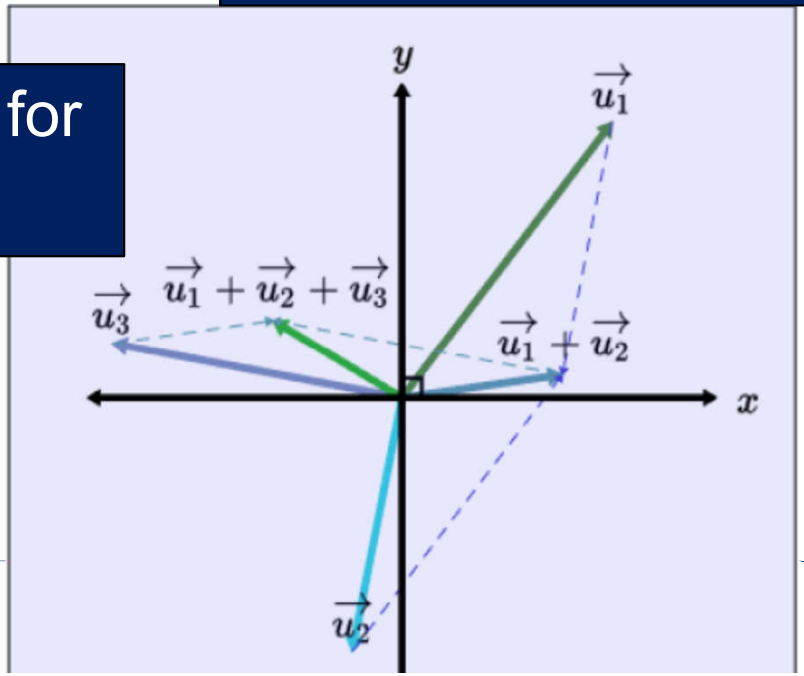
$u_3 := u_1 + u_2$

$u_5 := u_3 + u_4$

Syntactic sugar declares variables and relationships $In(u_i, U)$ for each u_i

Syntactic sugar for $AddV(u_3, u_4)$

Declares that u_5 is equal to $AddV(u_3, u_4)$



The Domain Language

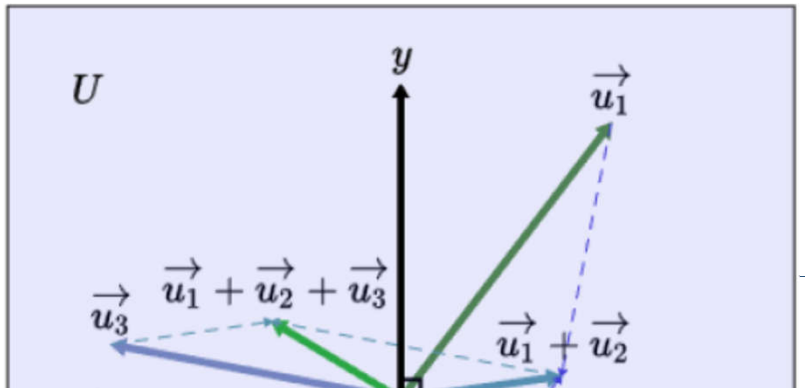
```
type Vector
type VectorSpace
predicate In: Vector * VectorSpace V
function addV: Vector * Vector -> Vector

notation "v1 + v2" ~ "addV(v1, v2)"
notation "Vector a ∈ U" ~ "Vector a; In(a, U)"
```

Declares type VectorSpace. Constructors may eventually have arguments.

Declares a predicate and its type

Declares an operator and its type

Declares syntactic sugar



Substance and Domain Design Features

- Separate, reusable domain extensions
  - New types, predicates, operators
  - New domain-specific notation
    - Similar to Coq notation extension

- Generic, typed object model
  - Check that substance programs are well-formed
  - Match on types in style programs

The Style Language

```
Vector v
with VectorSpace U
where v ∈ U {
   v.shape = Arrow {
      start = U.shape.center
   }

   encourage nearHead(v.shape, v.text)
   ensure contains(U.shape, v.shape)
}

Vector u
with Vector v, w; VectorSpace U
where u := v + w; u, v, w ∈ U {
   u.shape.end = v.shape.end + w.shape.end

   u.slider_v = Arrow {
      start = w.shape.end
      end  = u.shape.end
      style = "dashed"
   }

   u.slider_w = Arrow { ... }
}
```

Match once per Vector; call it v

For each v there must be at least one U

And In(v, U) must hold

The Style Language

```
Vector v
with VectorSpace U
where v ∈ U {
   v.shape = Arrow {
      start = U.shape.center
   }

   encourage nearHead(v.shape, v.text)
   ensure contains(U.shape, v.shape)
}

Vector u
with Vector v, w; VectorSpace U
where u := v + w; u, v, w ∈ U {
   u.shape.end = v.shape.end + w.shape.end

   u.slider_v = Arrow {
      start = w.shape.end
      end  = u.shape.end
      style = "dashed"
   }

   u.slider_w = Arrow { ... }
}
```

For each match create an Arrow graphical object. Attach it to the shape field of v.

Set a feature of v.shape
The runtime can optimize other features

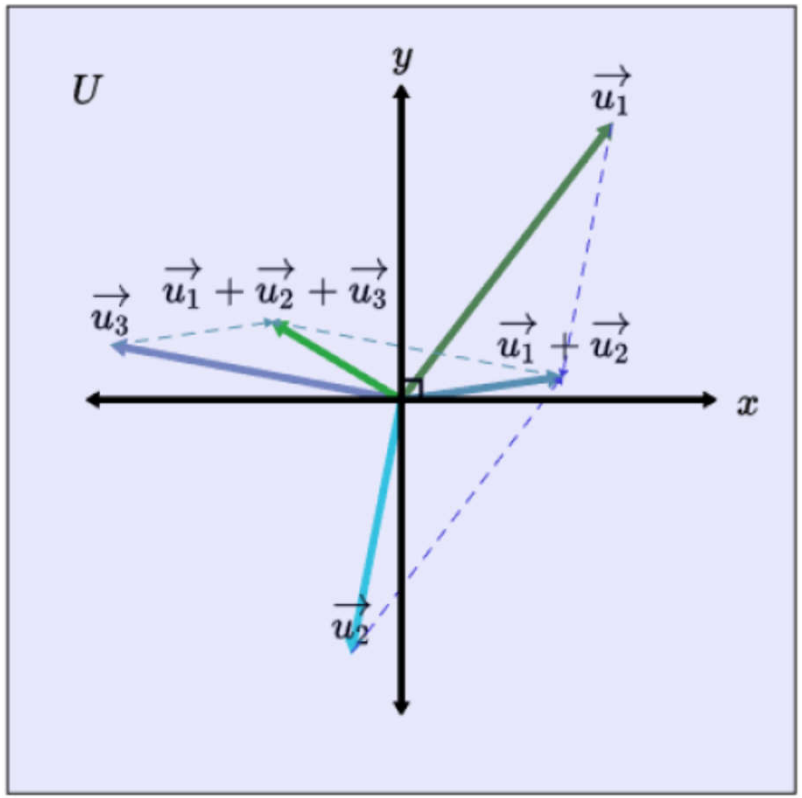
encourage contributes to objective function. **ensure** is a constraint.

Refines a shape created earlier

Creates more shapes for vectors when there is predicate relating them to other vectors

Substance, Style, and Output

VectorSpace U
Vector $u_1, u_2, u_3, u_4, u_5 \in U$
$u_4 := u_1 + u_2$
$u_5 := u_4 + u_3$




```
Vector v
with VectorSpace U
where v ∈ U {
   v.shape = Arrow {
      start = U.shape.center
   }

   encourage nearHead(v.shape, v.text)
   ensure contains(U.shape, v.shape)
}

Vector u
with Vector v, w; VectorSpace U
where u := v + w; u, v, w ∈ U {
   u.shape.end = v.shape.end + w.shape.end

   u.slider_v = Arrow {
      start = w.shape.end
      end  = u.shape.end
      style = "dashed"
   }

   u.slider_w = Arrow { ... }
}
```

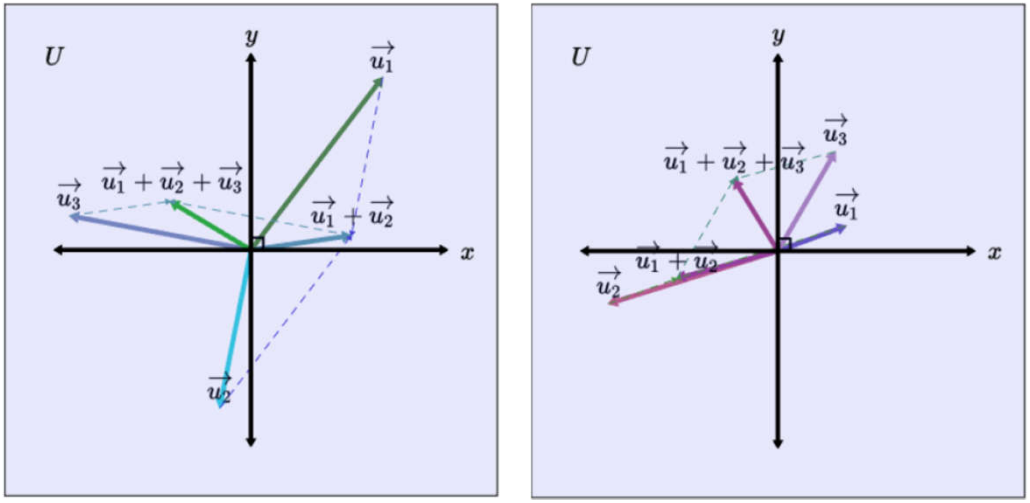
Style Design Characteristics

- Extensible and reusable
  - Many styles per domain
  - Use different styles with the same substance program
  - Typical end-users need not understand style programs
    - But expert users can edit them or write new ones if they want to

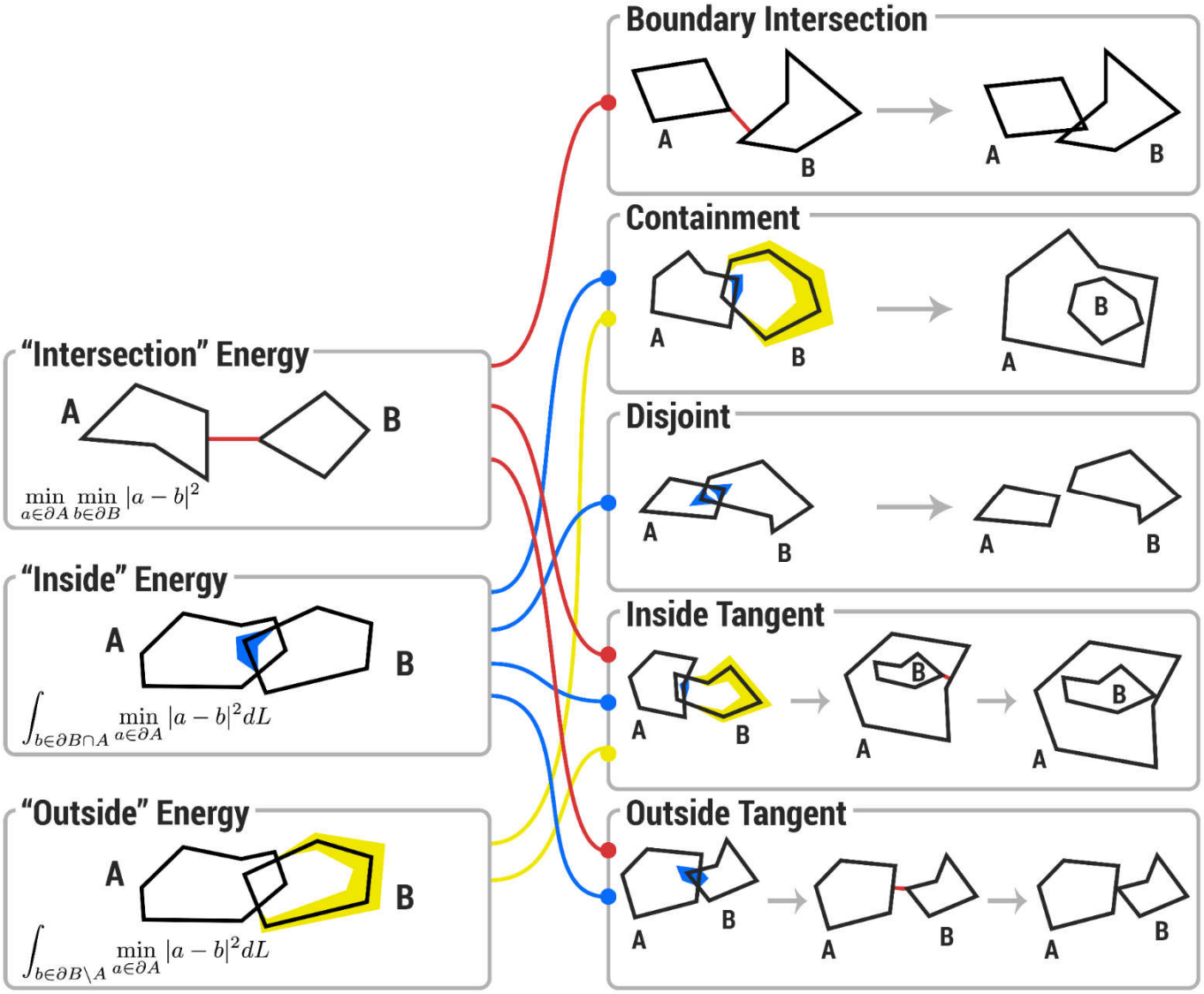
- Provides a *visual semantics* for substance programs
  - Pattern matches over logical objects, generates graphical objects
  - Generates objectives and constraints for later optimization
  - Later matches can refine the semantics provided by earlier ones

Optimization

- Basically hill-climbing to solve constraints and maximize objectives
- All Penrose functions are end-to-end differentiable
  - Can take the derivative and modify the input(s) in the direction(s) that improve the composite objective function
- Can run multiple times
  → multiple diagrams



Mathematics Underlying the Constraints

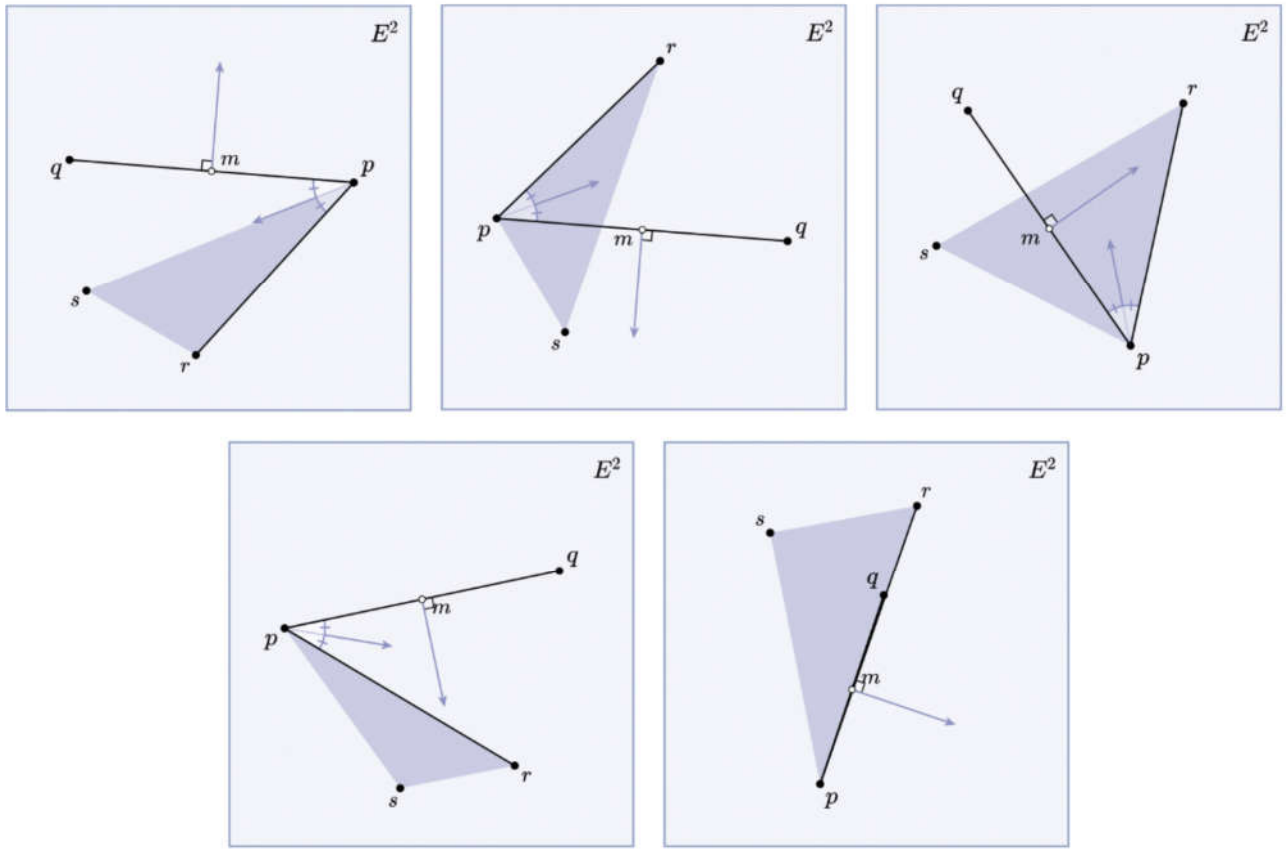


Euclidean Geometry

```
Point p, q, r, s
Segment a := p, q
Segment b := p, r
Point m := Midpoint(a)
Angle theta := ∠(q, p, r)
Triangle t := p, r, s
Ray w := Bisector(theta)
Ray h := PerpendicularBisector(a)
```

How do these relationships look if we assume that two parallel lines never meet?

Euclidean Geometry




```
Point p, q, r, s
Segment a := p, q
Segment b := p, r
Point m := Midpoint(a)
Angle theta := ∠(q, p, r)
Triangle t := p, r, s
Ray w := Bisector(theta)
Ray h := PerpendicularBisector(a)
```

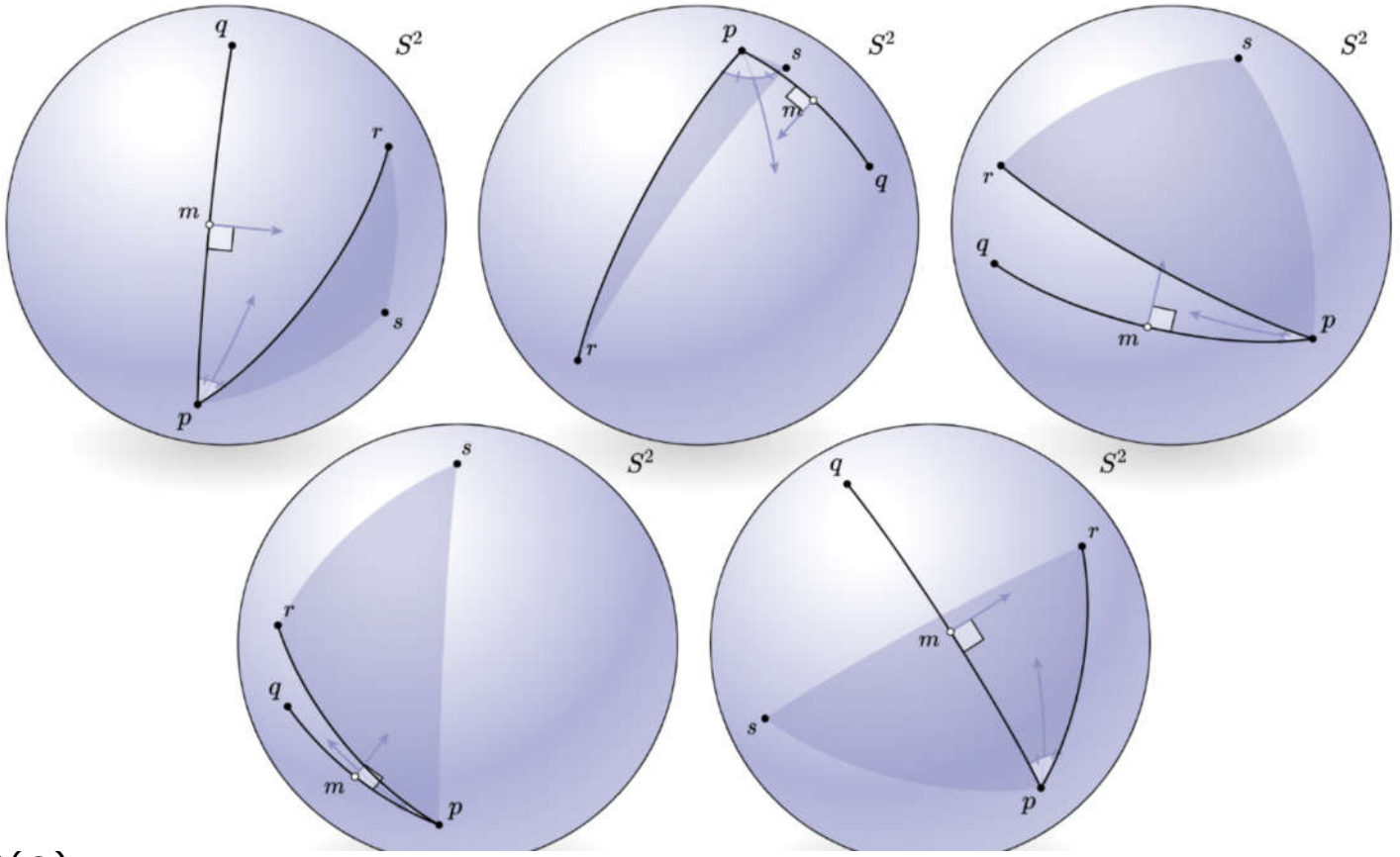
drawn in euclidean geometry
(assuming parallel postulate)

Non-Euclidean Geometry

```
Point p, q, r, s
Segment a := p, q
Segment b := p, r
Point m := Midpoint(a)
Angle theta := ∠(q, p, r)
Triangle t := p, r, s
Ray w := Bisector(theta)
Ray h := PerpendicularBisector(a)
```

What if the parallel postulate doesn't hold?
How would we visualize these relationships on, say, a sphere?

Non-Euclidean Geometry

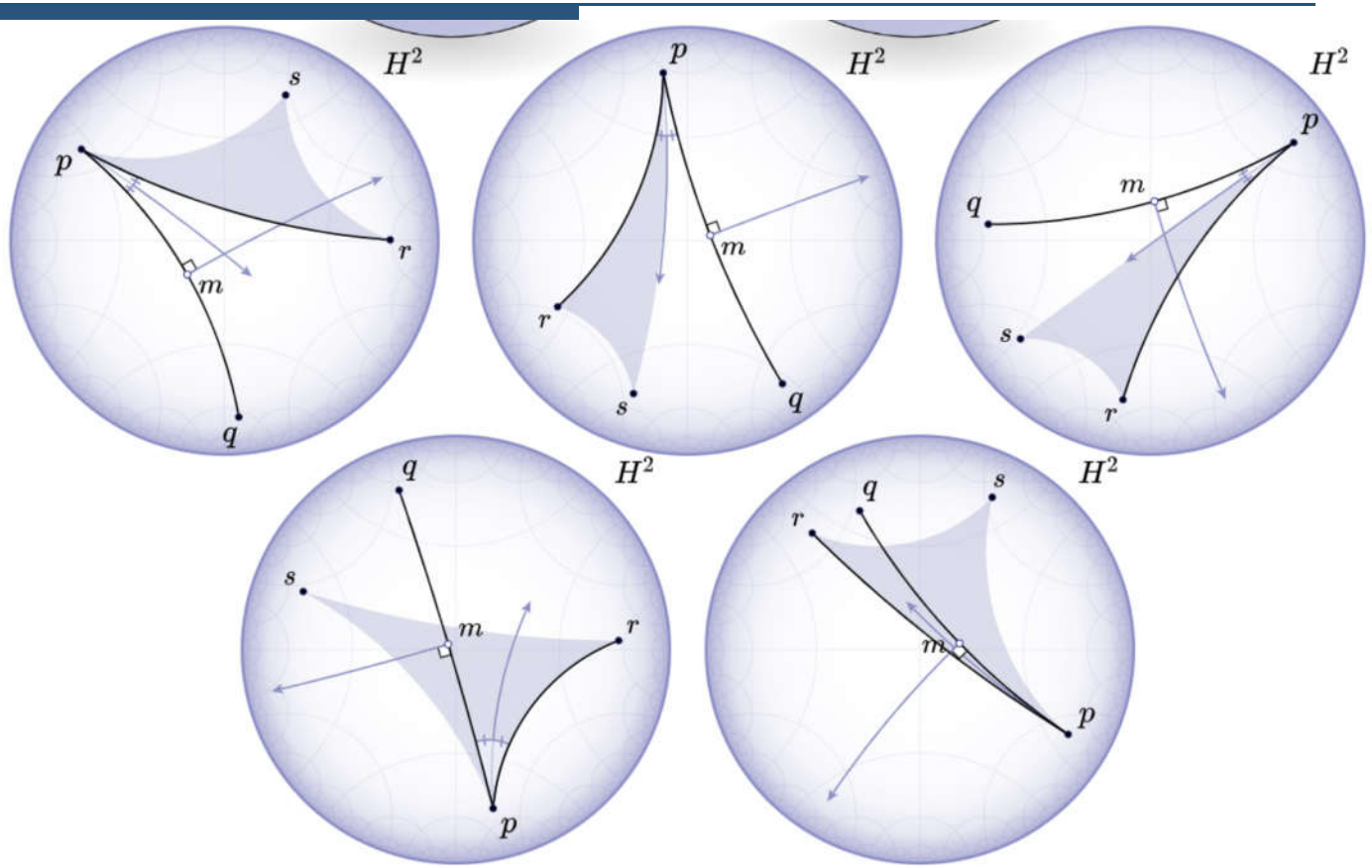



```
Point p, q, r, s
Segment a := p, q
Segment b := p, r
Point m := Midpoint(a)
Angle theta := ∠(q, p, r)
Triangle t := p, r, s
Ray w := Bisector(theta)
Ray h := PerpendicularBisector(a)
```

(different samples of the same Substance program, not a rotated sphere of the same diagram)

here's the style program

Non-Euclidean Geometry



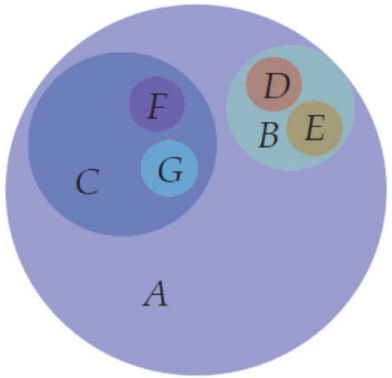

```
Point p, q, r, s
Segment a := p, q
Segment b := p, r
Point m := Midpoint(a)
Angle theta := ∠(q, p, r)
Triangle t := p, r, s
Ray w := Bisector(theta)
Ray h := PerpendicularBisector(a)
```

(different samples of the same Substance program)

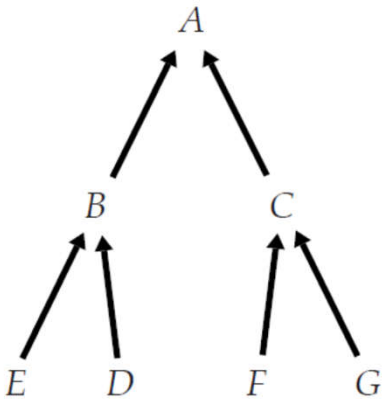
More Penrose Demonstrations

- Set theory
  - tree style
  - Venn style
- Real analysis
  - parallel axis style
  - perpendicular axis style
- Any live requests?
  - Set theory
  - Linear algebra
  - Real analysis

```
Set A, B, C, D, E, F, G
Subset B A
Subset C A
Subset D B
Subset E B
Subset F C
Subset G C
NoIntersect E D
NoIntersect F G
NoIntersect B C
```

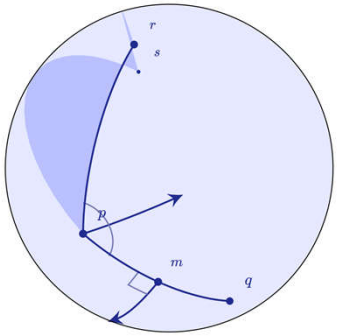


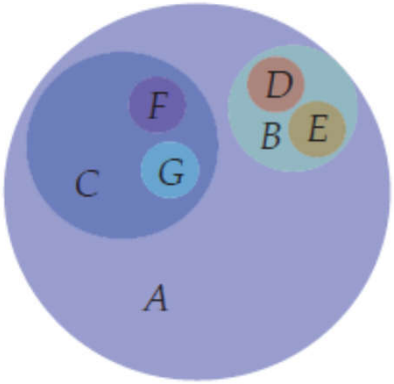

```
Set x {
  shape = Circle { }
  ensure x contains x.label
}
NoIntersect x y {
  ensure x notOverlap y
}
Subset x y {
  ensure y contains x
  ensure x smallerThan y
  ensure y.label outside x
}
```



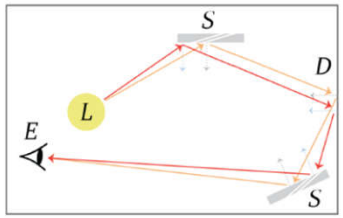

```
Set x { shape = Text{ } }
Subset x y {
  encourage y above x
  encourage x sameX y
  shape =  Arrow {
      start = x.shape
      end   = y.shape
}}
Set x, Set y {
  encourage x repel y
}
```

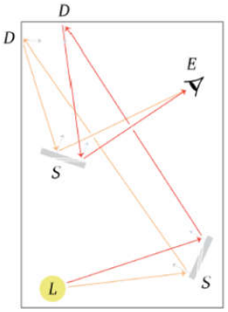
Penrose: customizable visual semantics for concept-level expressions in an extensible set of domains



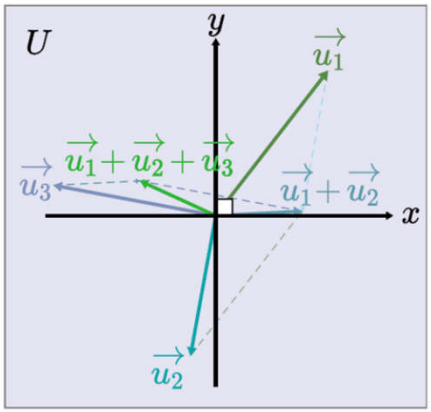


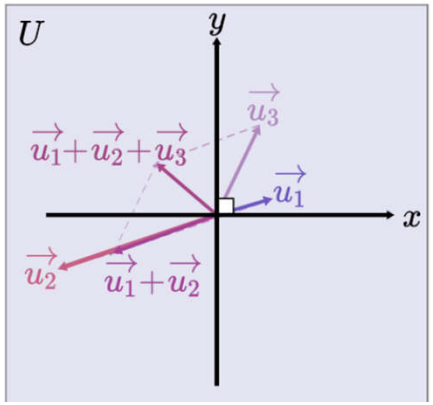

```
-- regex for a caustic
pathType = [L, S, D, S, E]
path1 = sample(pathType)
path2 = sample(pathType)
```

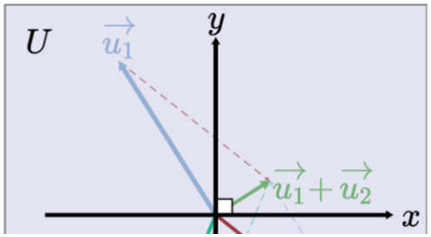




VectorSpace U
Vector $u_1, u_2, u_3,$
$u_4, u_5 \in U$
u_3 := u_1 + u_2
u_5 := u_3 + u_4

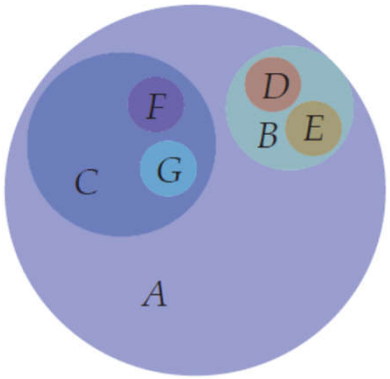




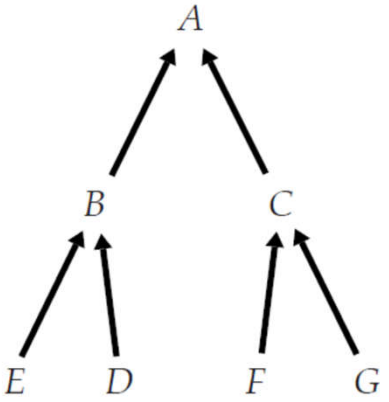


More examples

```
Set A, B, C, D, E, F, G
Subset B A
Subset C A
Subset D B
Subset E B
Subset F C
Subset G C
NoIntersect E D
NoIntersect F G
NoIntersect B C
```




```
Set x {
  shape = Circle { }
  ensure x contains x.label
}
NoIntersect x y {
  ensure x notOverlap y
}
Subset x y {
  ensure y contains x
  ensure x smallerThan y
  ensure y.label outside x
}
```




```
Set x { shape = Text{ } }
Subset x y {
  encourage y above x
  encourage x sameX y
  shape =  Arrow {
      start = x.shape
      end   = y.shape
}}
Set x, Set y {
  encourage x repel y
}
```