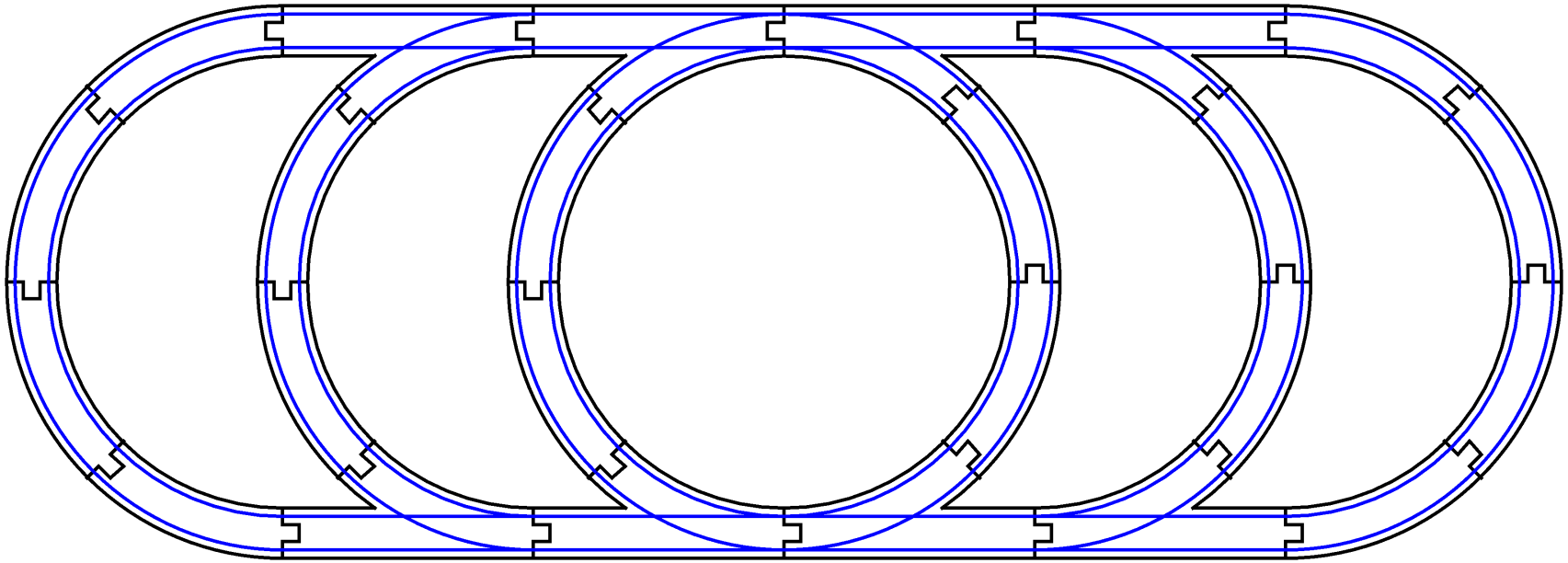
Trains, Pictures and Types

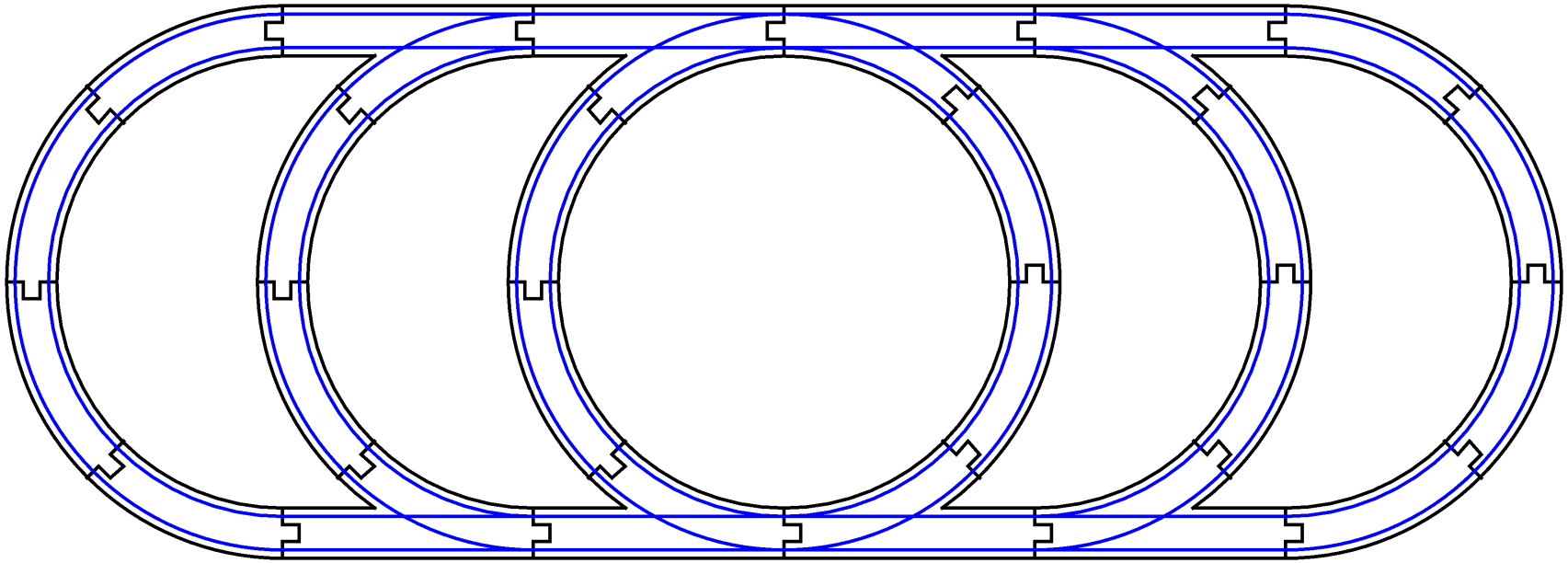
Professor Simon Gay
School of Computing Science
University of Glasgow, UK

The story so far: trains and types

Consider a model train layout:



Consider a model train layout:

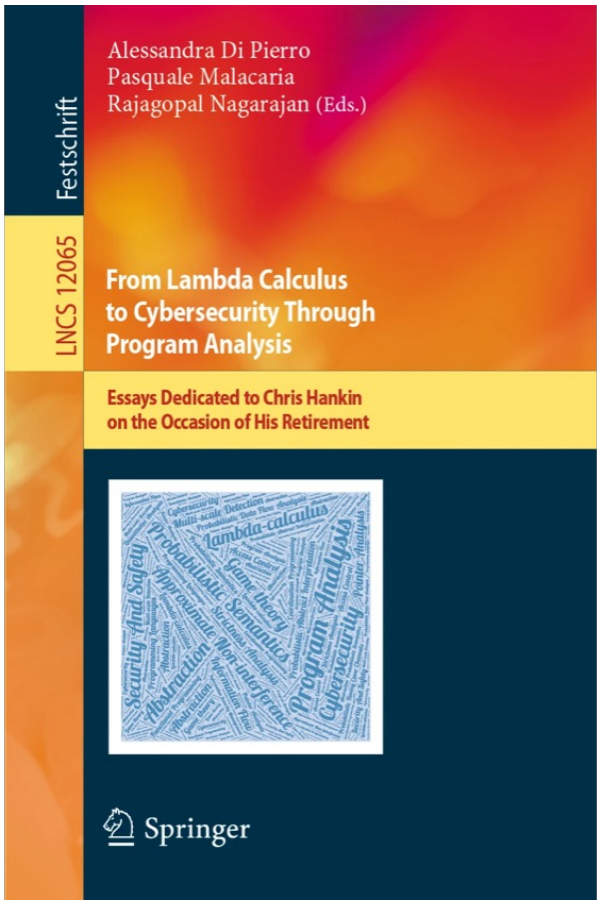


Safety property:

If we start with two trains running in the same direction, there can never be a head-on collision.

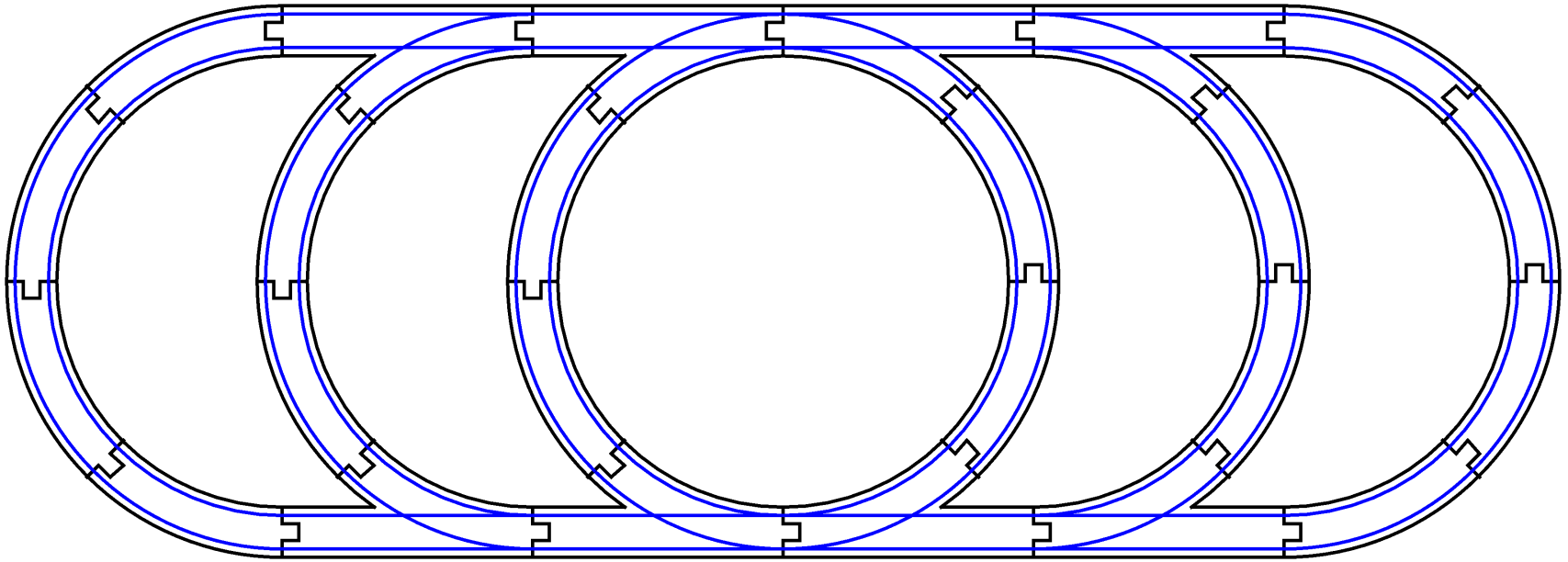
This can be proved with a static type system.

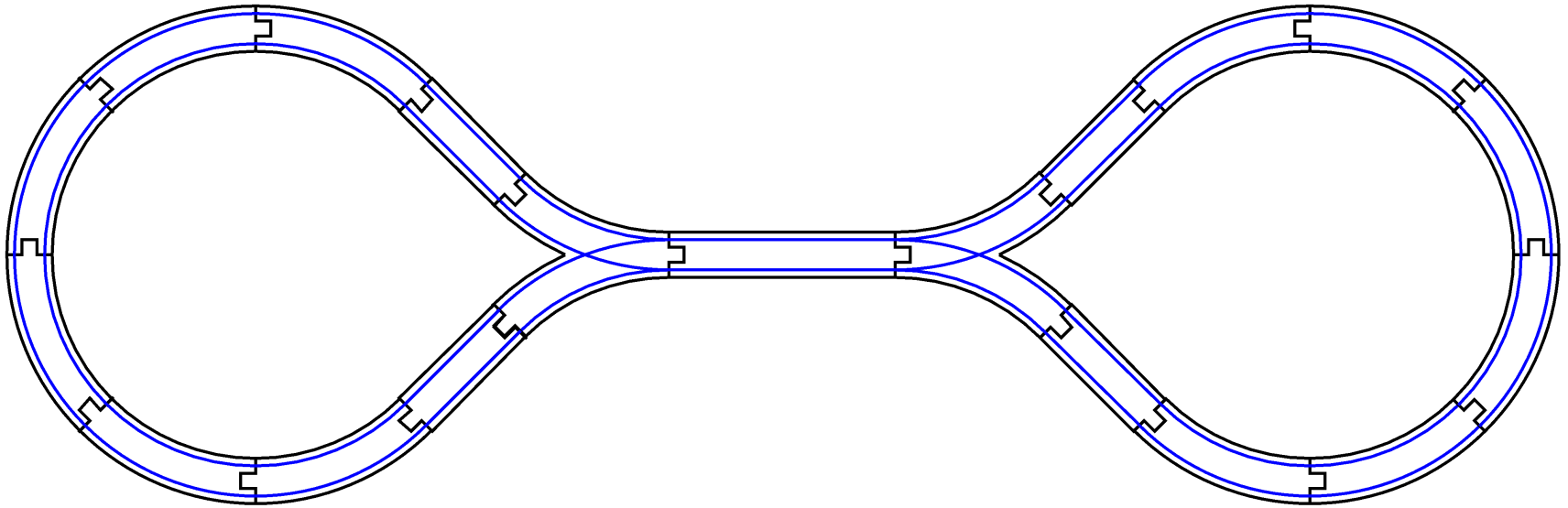
Every section of track is associated with a unique direction, and this property is inductively preserved when pieces are connected together.

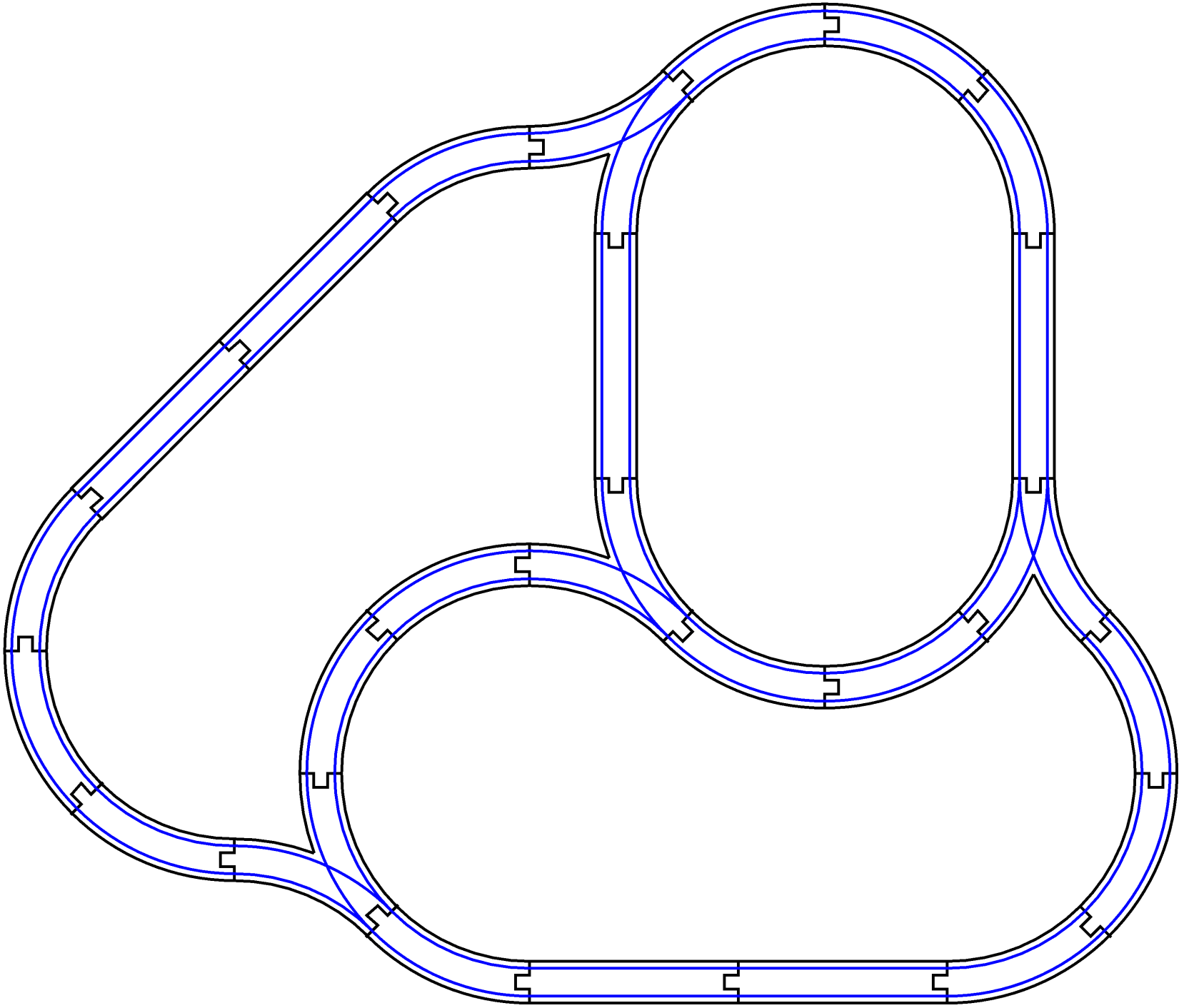


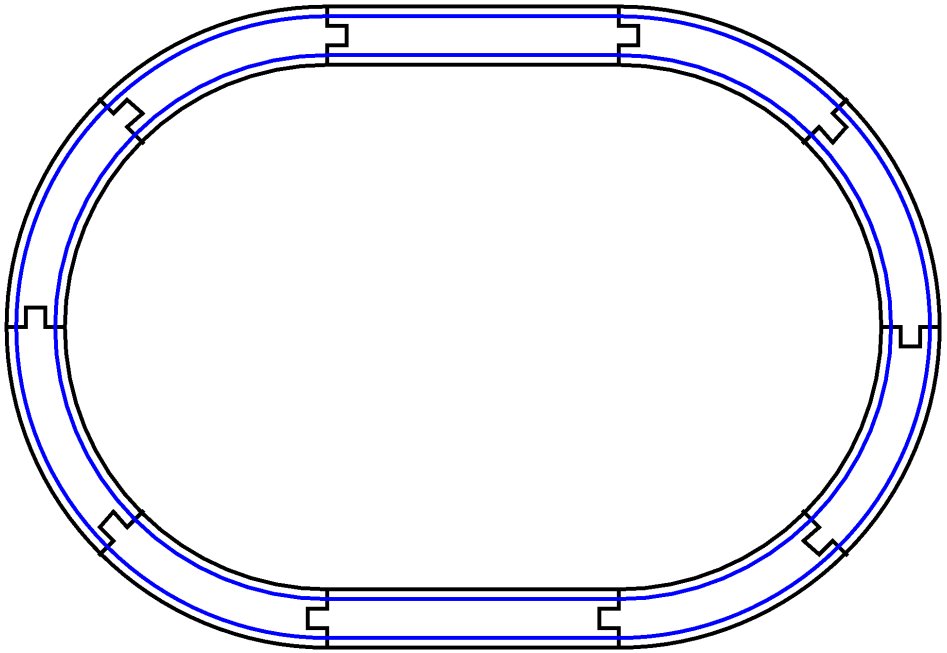
Cables, Trains and Types, 2020.

Continuing the journey: pictures and types

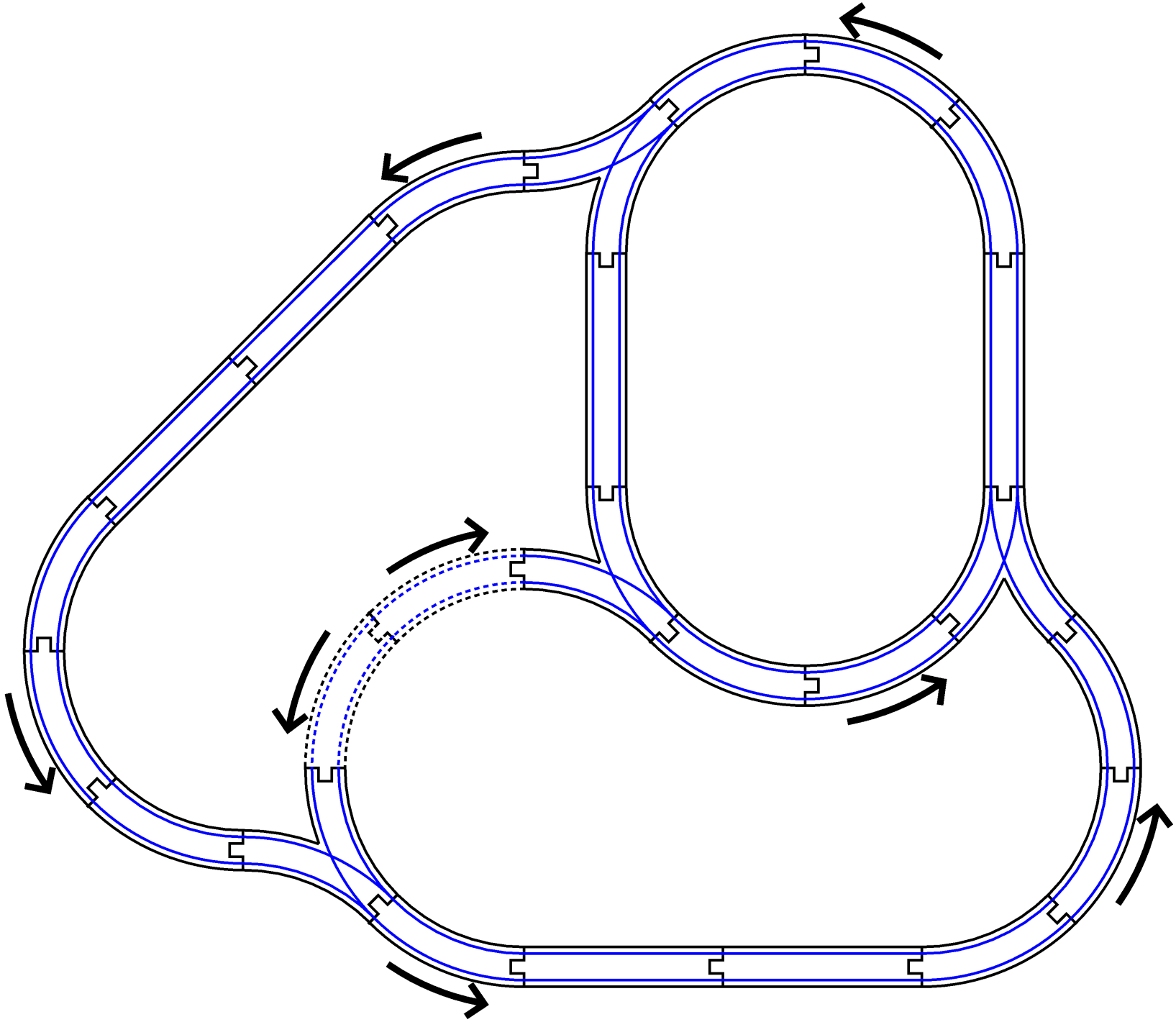




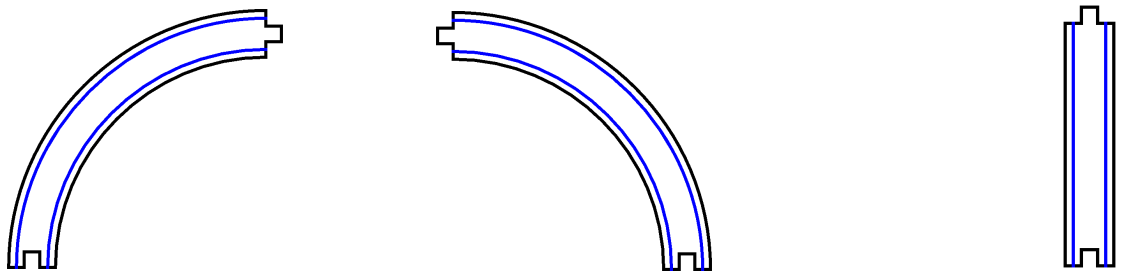




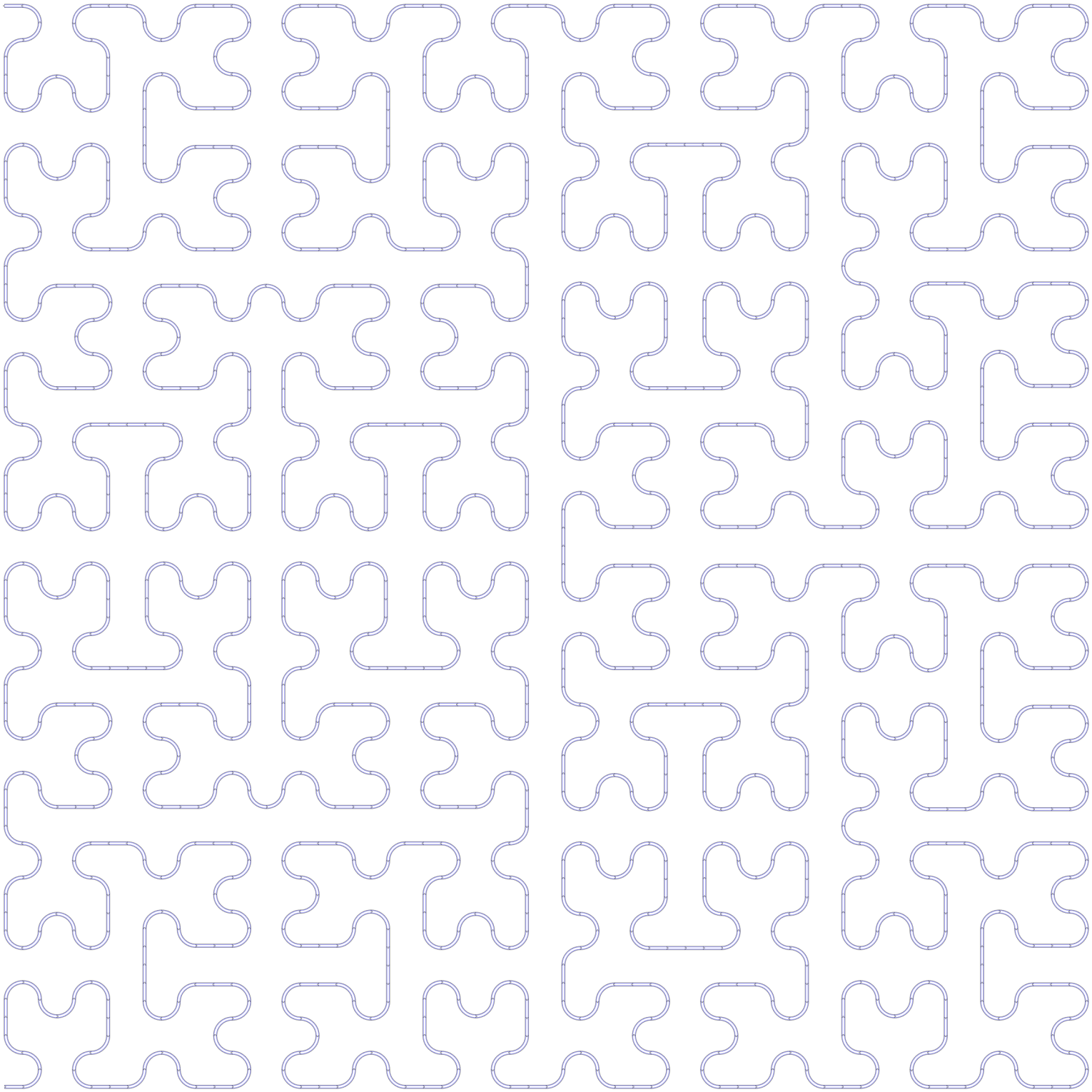
```
(new Diagram() {
        void draw() {
                forward();
                right();
                right();
                right();
                right();
                forward();
                right();
                right();
                right();
                right();
        }
}
).save("oval.png");
```



Let’s restrict to track layouts with no branching, 90º curves and straight pieces.



We can still produce lots of interesting layouts.



There's a correctness property for constructing tracks, before we consider running trains:

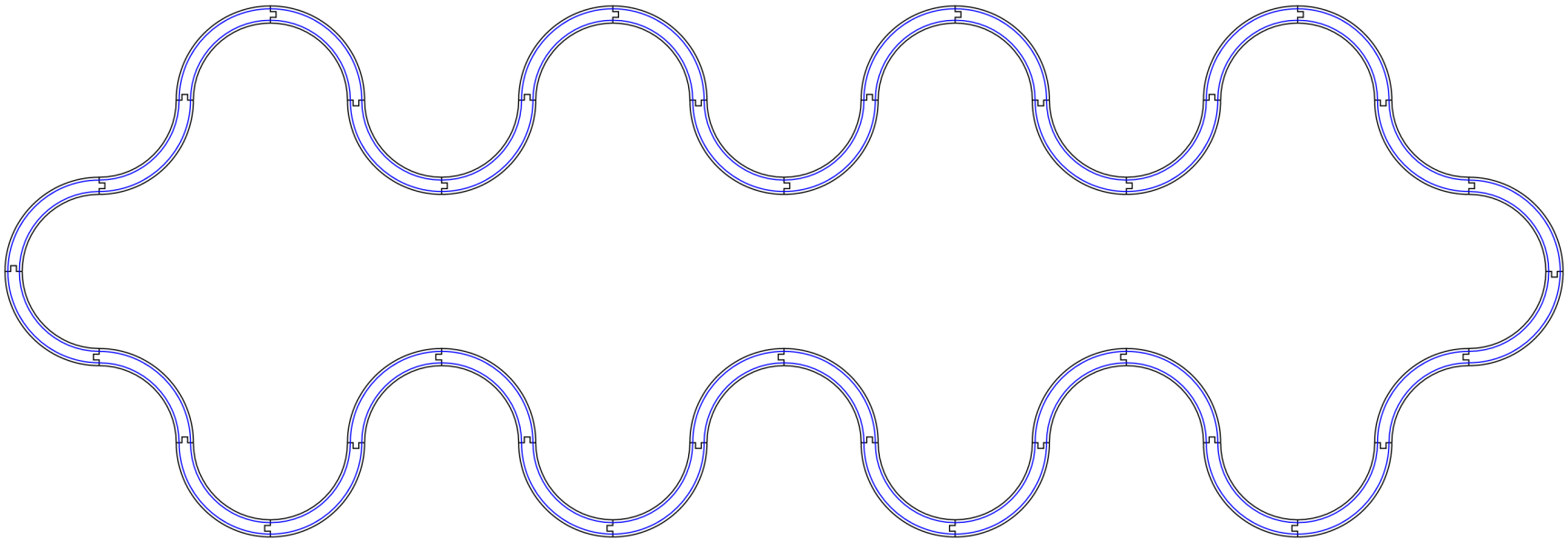
we want a closed loop that joins together smoothly.

A sequence of drawing commands produces a smooth closed track if and only if:

- the end point is the same as the starting point, and
- the direction at the end is the same as the direction at the beginning.

Exercise: if we restrict to 90^o curves and don't use straight pieces, the first property implies the second property.

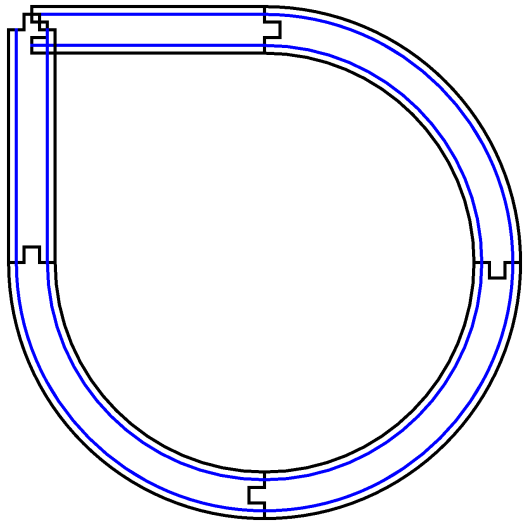
Exercise: if we restrict to 90º curves and don't use straight pieces, the first property implies the second property.



A sequence of drawing commands produces a smooth closed track if and only if:

- the end point is the same as the starting point, and
- the direction at the end is the same as the direction at the beginning.

In general we need both properties.

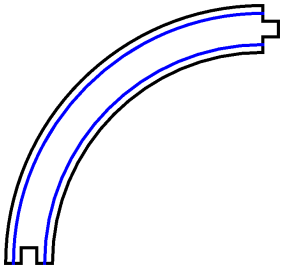


We can use a dependent type system to check at compile-time that a layout joins up smoothly.

The type `Track (m,n) θ` is the type of a layout that produces displacement `(m,n)` from the starting point (using coordinates based on the initial direction of travel) and rotates the direction of travel by `θ` .

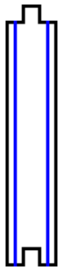
A single right curve has type

`Track (1,1) -90°`



A straight section has type

`Track (0,1) 0°`



It’s easiest to represent angles by rotation matrices:

0^o $\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$

90^o $\begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}$

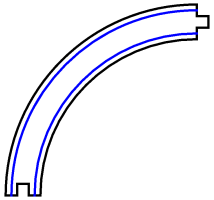
-90^o $\begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}$

180^o $\begin{pmatrix} -1 & 0 \\ 0 & -1 \end{pmatrix}$

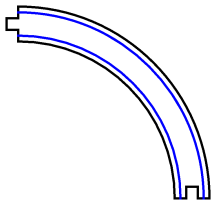
and then displacements become column vectors.

```
data Track : Vector → Angle → Type where
```

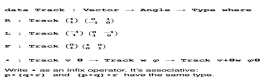
```
R : Track (1, 1) (0  1)
                 (-1 0)
```



```
L : Track (-1, 1) (0  -1)
                  (1   0)
```



```
F : Track (0, 1) (1  0)
                 (0  1)
```



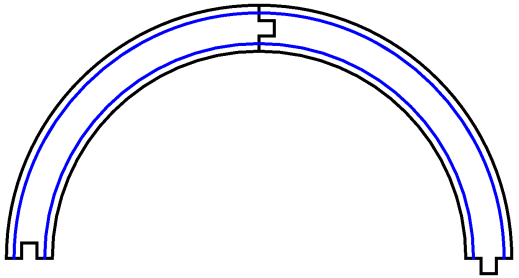
Implementing in Idris.

Representing the structure – different from Java.

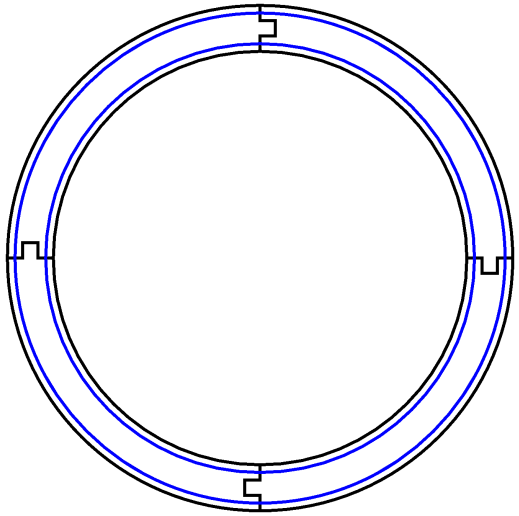
```
• : Track v θ → Track w φ → Track v+θw φθ
```

Write • as an infix operator. It's associative:
`p•(q•r)` and `(p•q)•r` have the same type.

$\mathbf{R \bullet R}$: Track $\begin{pmatrix} 2 \\ 0 \end{pmatrix}$ $\begin{pmatrix} 0 & -1 \\ -1 & 0 \end{pmatrix}$



$\mathbf{R \bullet R \bullet R \bullet R}$: Track $\begin{pmatrix} 0 \\ 0 \end{pmatrix}$ $\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$



Finally, define

$$\texttt{draw : Track } \begin{pmatrix}0\\0\end{pmatrix} \begin{pmatrix}1 & 0\\0 & 1\end{pmatrix} \rightarrow \texttt{ Unit}$$

and then

```
draw (R • R)
```

is a type error.

Now we are drawing – but we could do other things, e.g. count the pieces of each shape.

The type of a track has constant size, independently of the length of the track.

The type of a track has constant size, independently of the length of the track.

Think of a type as a “surface” on which we interact with a “solid” term. The type is smaller than the term.

The type of a track has constant size, independently of the length of the track.

Think of a type as a “surface” on which we interact with a “solid” term. The type is smaller than the term.

It’s bigger on the inside!

The type of a track has constant size, independently of the length of the track.

Think of a type as a “surface” on which we interact with a “solid” term. The type is smaller than the term.

It’s bigger on the inside!

Types
Abstract,
Radically
Decreasing
In
Size

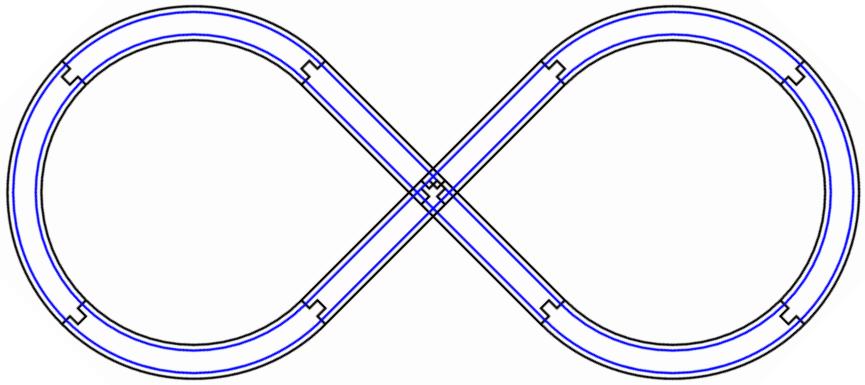
Soundness and Completeness

- It's obvious that this type system is sound and complete for the property of being a closed track with smooth joins.

- This is only one notion of correctness, and other things can go wrong.

  `R•R•R•F•F•L•L•L•F•F`

  has type **`Track`** $\binom{0}{0}$ $\begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix}$ but intersects itself.



Avoiding Self-Intersection

Refine the type of a track by adding a list of the coordinates that it visits (including the initial point but not the final point).

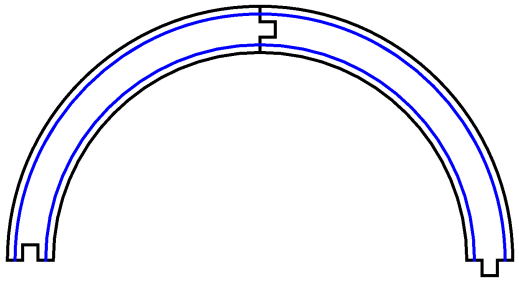
```
data Track : Vector → Angle →
                    List Vector → Type where
```

$$\texttt{R : Track } \begin{pmatrix}1\\1\end{pmatrix} \begin{pmatrix}0 & 1\\-1 & 0\end{pmatrix} \texttt{ [} \begin{pmatrix}0\\0\end{pmatrix} \texttt{]}$$

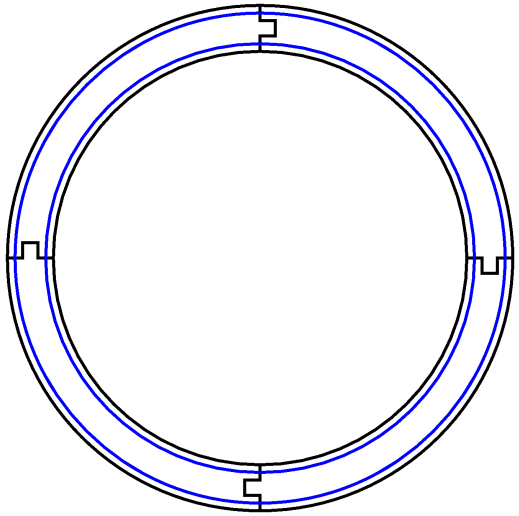
$$\texttt{L : Track } \begin{pmatrix}-1\\1\end{pmatrix} \begin{pmatrix}0 & -1\\1 & 0\end{pmatrix} \texttt{ [} \begin{pmatrix}0\\0\end{pmatrix} \texttt{]}$$

$$\texttt{F : Track } \begin{pmatrix}0\\1\end{pmatrix} \begin{pmatrix}1 & 0\\0 & 1\end{pmatrix} \texttt{ [} \begin{pmatrix}0\\0\end{pmatrix} \texttt{]}$$

$\mathbf{R \bullet R : Track} \begin{pmatrix} 2 \\ 0 \end{pmatrix} \begin{pmatrix} 0 & -1 \\ -1 & 0 \end{pmatrix} \left[\begin{pmatrix} 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 \\ 1 \end{pmatrix} \right]$



$\mathbf{R \bullet R \bullet R \bullet R : Track} \begin{pmatrix} 0 \\ 0 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \left[\begin{pmatrix} 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \begin{pmatrix} 2 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 \\ -1 \end{pmatrix} \right]$



$\bullet$: **Track v θ vs** $\rightarrow$ **Track w** φ **ws** $\rightarrow$
Track v+θw φ**θ (vs ++ map (**λ**x.v+θx) ws)**

There are conditions for using $\bullet$:

1. **vs** and **map (**λ**x.v+θx) ws** are disjoint

2. $\mathbf{w} \neq \binom{0}{0}$

The condition $\mathbf{v} \neq \binom{0}{0}$ is not needed because if $\mathbf{v} = \binom{0}{0}$ then the first point of the second track is the same as the first point of the first track. Condition 1 eliminates this situation.

To define composition in Idris, we now require proofs of these conditions.

```
Compose: Track v θ vs →
         Track w φ ws →
         w ≠ (0 0) →
         Disjoint vs (map (λx.v+θx) ws) →
         Track v+θw φθ (vs ++ map (λx.v+θx) ws)
```

The key now is to define **`v ≠ w`** in a positive way, so that Idris can automatically find proofs for all required instances of **`v ≠ w`** and **`Disjoint vs ws`**.

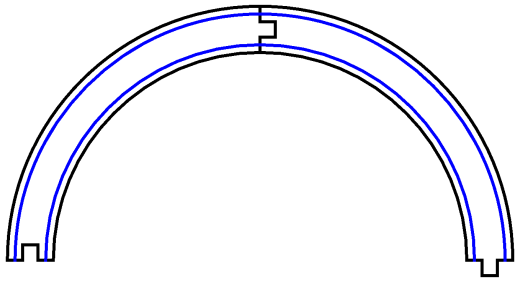
```
data NotEqual : Vector -> Vector -> Type where
    FstNE : So (not (a == x)) -> NotEqual (V a b) (V x y)
    SndNE : So (not (b == y)) -> NotEqual (V a b) (V x y)

NotIn : Vector -> List Vector -> Type
NotIn v vs = All (NotEqual v) vs

data Disjoint : List Vector -> List Vector -> Type where
  Empty  : Disjoint Nil ws
  Extend : Disjoint vs ws -> NotIn v ws -> Disjoint (v::vs) ws
```

Jan de Muijnck-Hughes showed me how to do this. 😀

Now we can define a composition operator in which all the proofs are implicit and hidden.

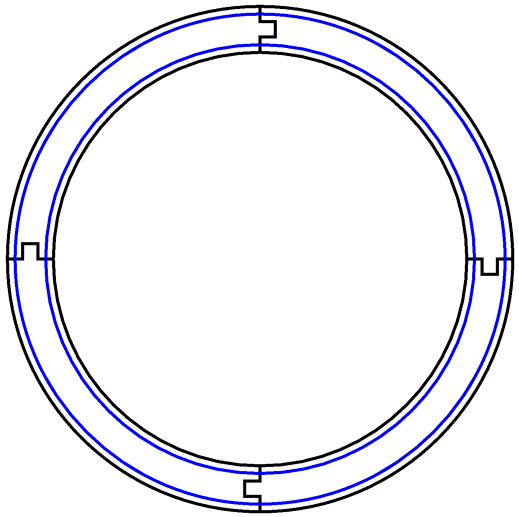
```
infixr 9 .

(.) : Track v theta vs ->
           Track w phi ws ->
           (prfa : NotEqual w (V 0 0)) =>
           (prfb : Disjoint vs (mapShift v theta ws)) =>
           Track (v + (rotate theta w)) (phi * theta)
                 (vs ++ mapShift v theta ws)

(.) {prfa} {prfb} s t = Compose s t prfa prfb
```

`R•R : Track` $\binom{2}{0}$ $\begin{pmatrix} 0 & -1 \\ -1 & 0 \end{pmatrix}$ `[` $\binom{0}{0}$`,` $\binom{1}{1}$ `]`



`draw (R•R)` is a type error

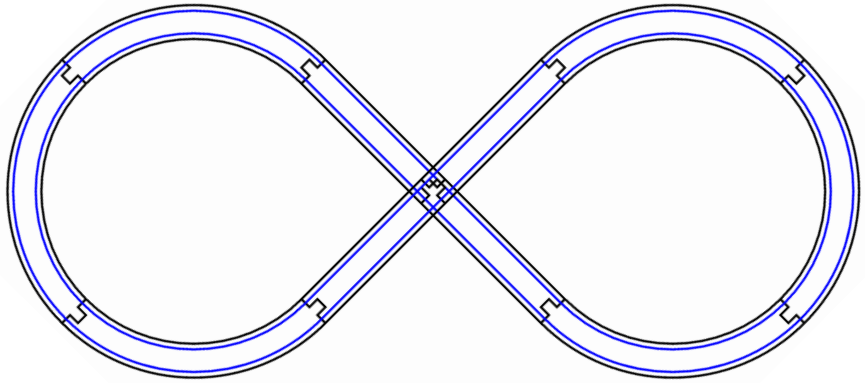
`R•R•R•R : Track` $\binom{0}{0}$ $\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$ `[` $\binom{0}{0}$`,` $\binom{1}{1}$`,` $\binom{2}{0}$`,` $\binom{1}{-1}$ `]`



`draw (R•R•R•R)` is OK

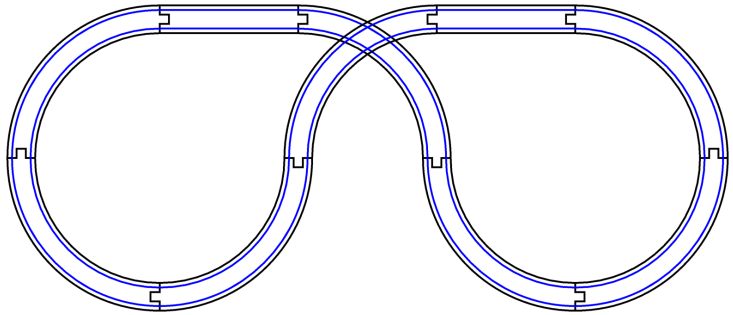
`draw (R•R•R•F•F•L•L•L•F•F)` is a type error because

`R•R•R•F•F•L•L•L•F•F` is a type error.



Avoiding Self-Intersection: The Missing Case

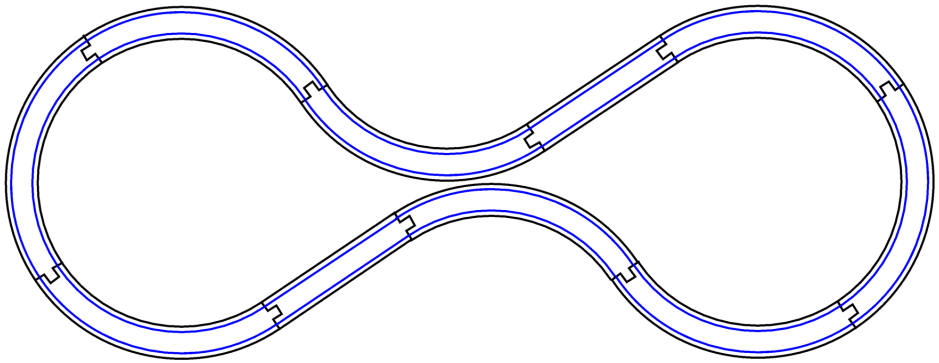
Alert members of the audience might realise that we have not succeeded in eliminating all self-intersections.



To deal with this, we need to record the grid squares that are crossed diagonally by curved pieces. Represent them by their centre coordinates, in the same list as the endpoint coordinates. To avoid working with half-integers, multiply everything by 2. Everything goes through without difficulty.

Are We Sound for Self-Intersection?

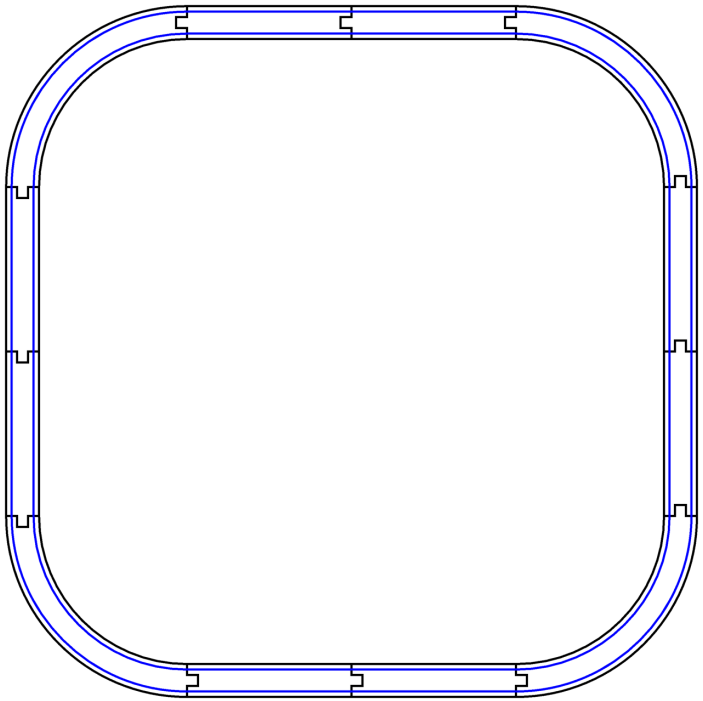
This layout is typable, but it would self-intersect in a different way if the track pieces were wider in relation to their length.



Our type system is a significant abstraction from the details of geometry.

Repetition

We would like to be able to use simple loops:



```
repeat 4 (F . F . L)
```

Repetition

$$\texttt{Null} : \texttt{Track} \begin{pmatrix} 0 \\ 0 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

```
repeat 0     track = Null
repeat (S n) track = track . (repeat n track)
```

Repetition

```
Null : Track (0) (1 0)
             (0) (0 1)

repeat : (n:Nat) -> Track v theta ->
   Track (repShift n v theta) (repMult n theta)

repeat 0     track = Null
repeat (S n) track = track . (repeat n track)
```

Repetition

```
repMult : Nat -> Angle -> Angle

repMult 0     theta = 
```

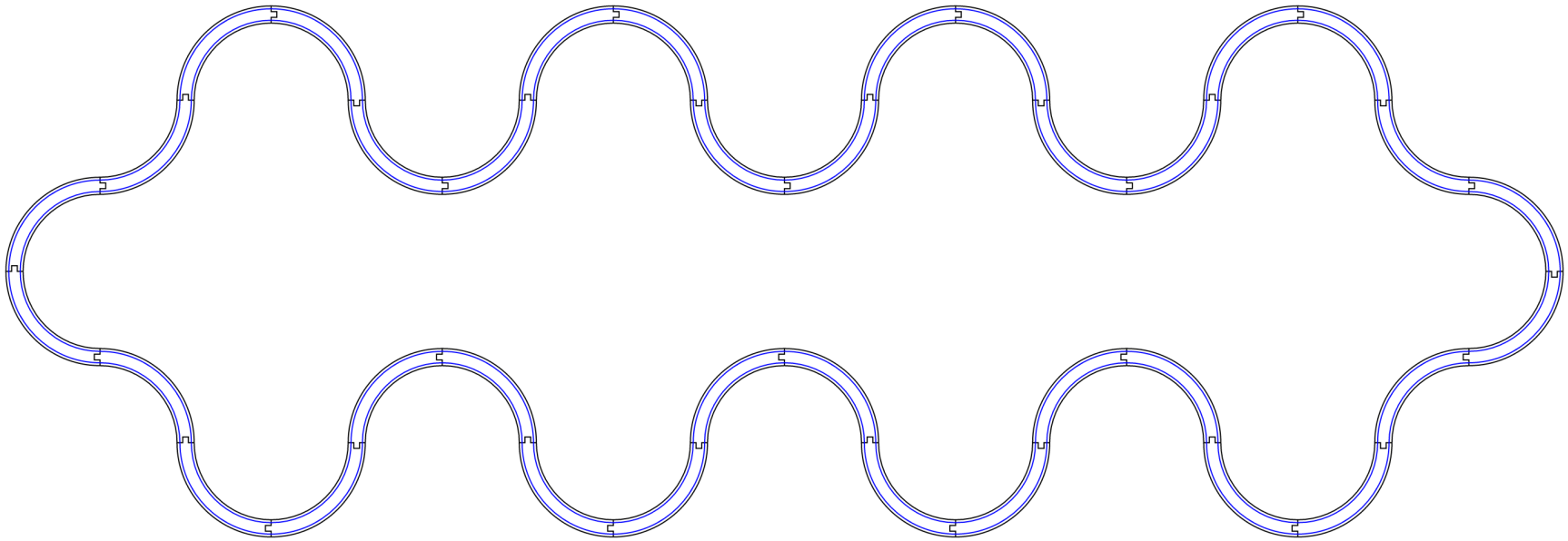
$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

```
repMult (S n) theta = (repMult n theta) * theta
```

Repetition

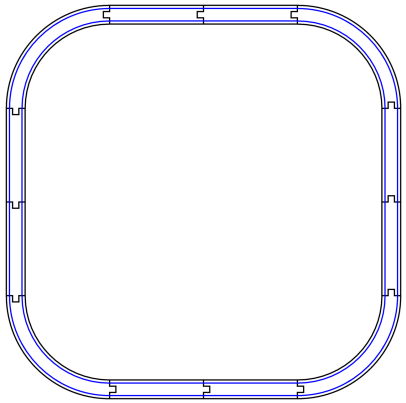
```
repShift : Nat -> Vector -> Angle -> Vector

repShift 0     v theta = (0 0)

repShift (S n) v theta =
        v + (rotate theta (repShift n v theta))
```

Repetition



```
repeat 2 (L.(repeat 3 (R.R.L.L)).R.R.L.R.R)
```

Parameterised Definitions



We would like to generate this layout by `square 2`

where `square : Nat -> Track` $\binom{0}{0}\ \binom{1\ \ 0}{0\ \ 1}$

is defined by

```
square n = repeat 4 (side n)
```

and `side : (n:Nat) -> Track` $\binom{-1}{n}\ \binom{0\ \ -1}{1\ \ 0}$

is defined by

```
side 0     = L
side (S n) = F . (side n)
```

Parameterised Definitions

Here we run into a common problem with practical dependently typed programming.

The step from `side n : Track` $\binom{-1}{n}\begin{pmatrix}0 & -1\\ 1 & 0\end{pmatrix}$

to `F . (side n) : Track` $\binom{-1}{n+1}\begin{pmatrix}0 & -1\\ 1 & 0\end{pmatrix}$

requires a proof that

$$\binom{0}{1} + \begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix}\binom{-1}{n} = \binom{-1}{n+1}$$

which the Idris typechecker is unable to construct automatically.

(To be fair to Idris, exactly the same thing happens with Agda).

Helping the Typechecker

```
-- A sequence of (n+1) F pieces, using a big hack to
-- force typechecking to work.
-- Note that the equality type in magic is the opposite
-- way around to the function type in forwardProof

magic : (m : Nat) ->
     (V 0 (prim__cast_IntegerInt
      (prim__add_Integer 1 (natToInteger (plus m 1)))))
  = (V (prim__add_Int 0 (prim__add_Int 0
      (prim__mul_Int 0 (prim__cast_IntegerInt
        (natToInteger (plus m 1))))))
     (prim__add_Int 1 (prim__add_Int 0 (prim__mul_Int 1
    (prim__cast_IntegerInt (natToInteger (plus m 1)))))))

magic m = believe_me m
```

Helping the Typechecker

```
forwardProof :
  (m : Nat) -> Track (V (prim__add_Int 0
                 (prim__add_Int 0 (prim__mul_Int 0
                 (prim__cast_IntegerInt (natToInteger
                 (plus m 1)))))) (prim__add_Int 1
                 (prim__add_Int 0 (prim__mul_Int 1
                 (prim__cast_IntegerInt (natToInteger
                 (plus m 1)))))))) (M 1 0 0 1)
               -> Track (V 0 (prim__cast_IntegerInt
                 (prim__add_Integer 1 (natToInteger
                 (plus m 1))))) (M 1 0 0 1)

forwardProof m result = rewrite (magic m) in result
```

So why isn't this definition a problem?

This was similar or maybe more complicated, but it worked!

```
repeat : (n:Nat) -> Track v theta ->
   Track (repShift n v theta) (repMult n theta)

repeat 0     track = Null
repeat (S n) track = track . (repeat n track)
```

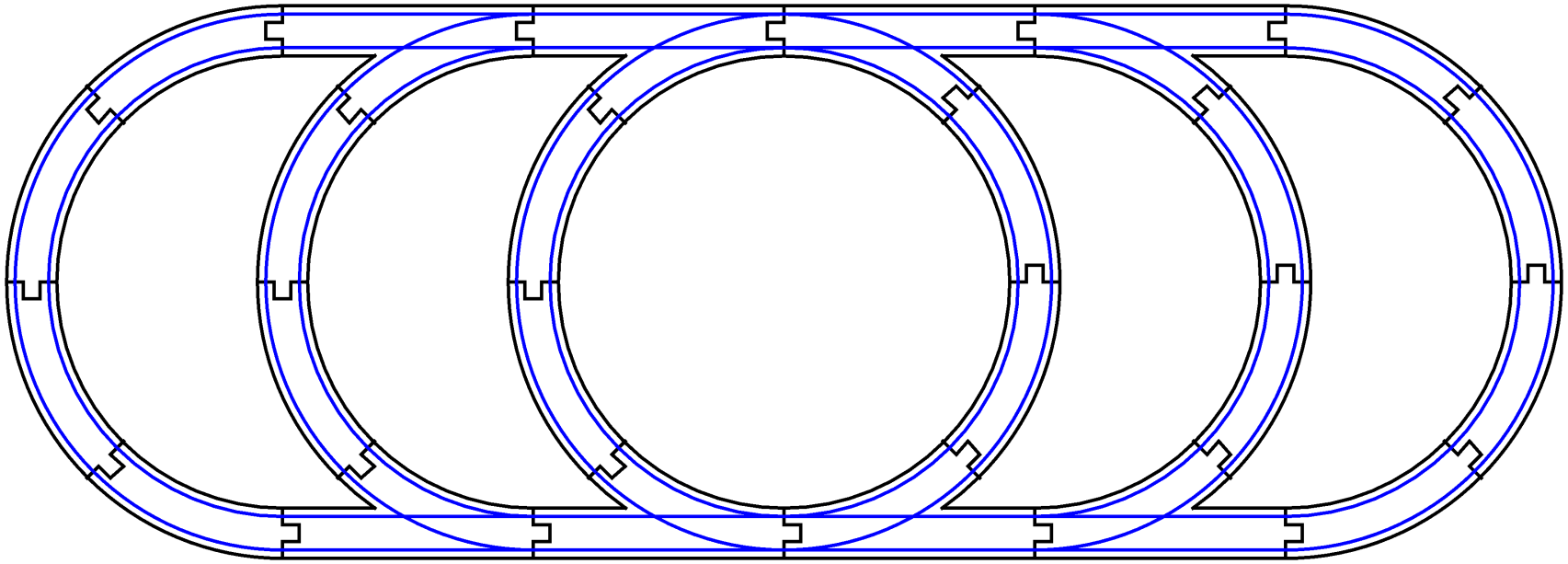
In this case, the recursive structure in the functions closely matches the recursive structure in the types. The types that need to match are syntactic identities (or very very close).

Summary

- A dependent type system can track the position of the components of a railway layout.
- If a sequence of constructors produces a layout that doesn't join up properly, this can be detected as a type error at compile time.
- With a little more work (and bigger types), typechecking can also detect self-intersection of layouts.
- I emphasised the diagrams, but the type system is really about correctness of the structure of the track.
- The step from terms-as-data to terms-as-proofs is significant in learning dependently typed programming.
- We encounter the limitations of automated symbolic reasoning in Idris (or Agda).

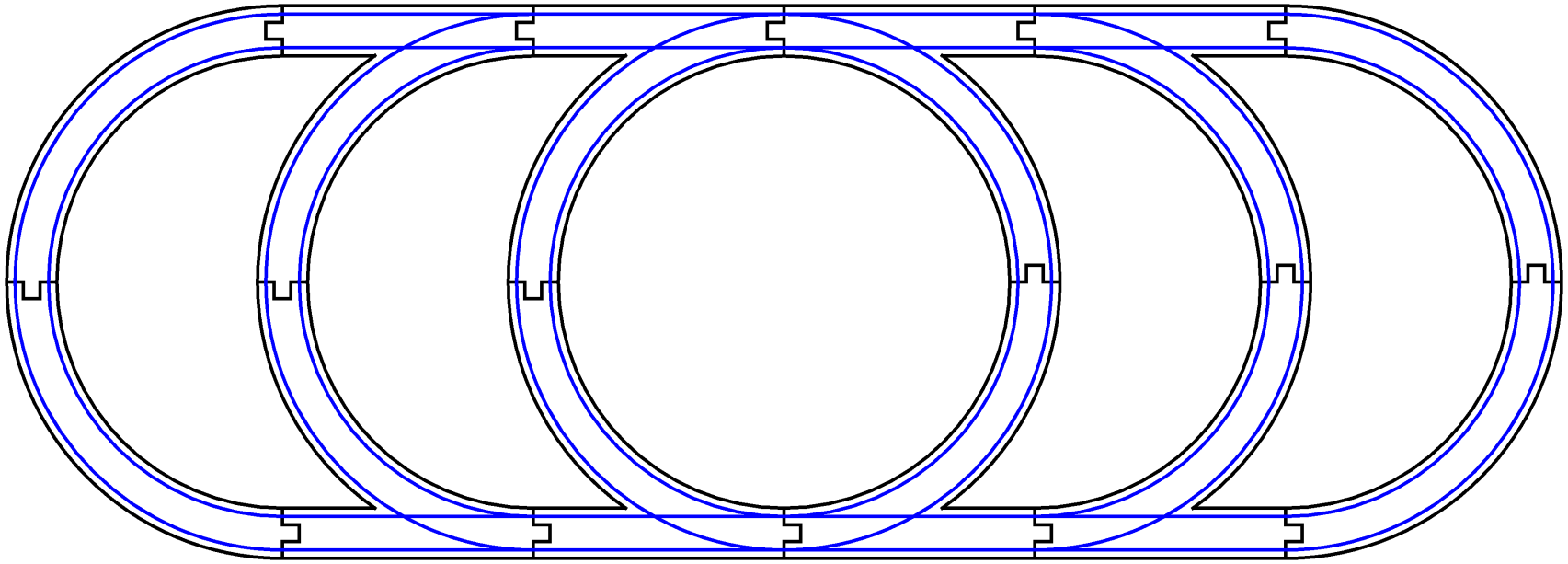
Branching

We omitted branching pieces for simplicity, but can we bring them back?



Branching

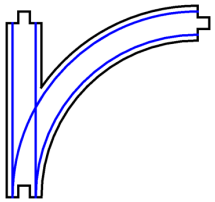
Keep the restriction to 90º curves.



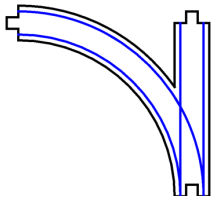
Ignore self-intersection, just check for smooth closed tracks.

The type of a piece now includes a list of “loose ends”.

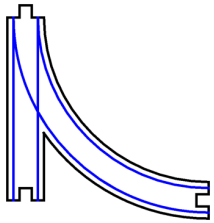
RB : Track $\binom{0}{1}$ **0° [(**$\binom{1}{1}$**,-90°)]**



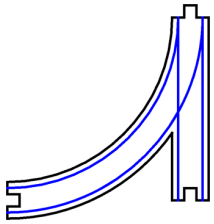
LB : Track $\binom{0}{1}$ **0° [(**$\binom{-1}{1}$**,90°)]**



RM : Track $\binom{0}{1}$ **0° [(**$\binom{1}{0}$**,90°)]**

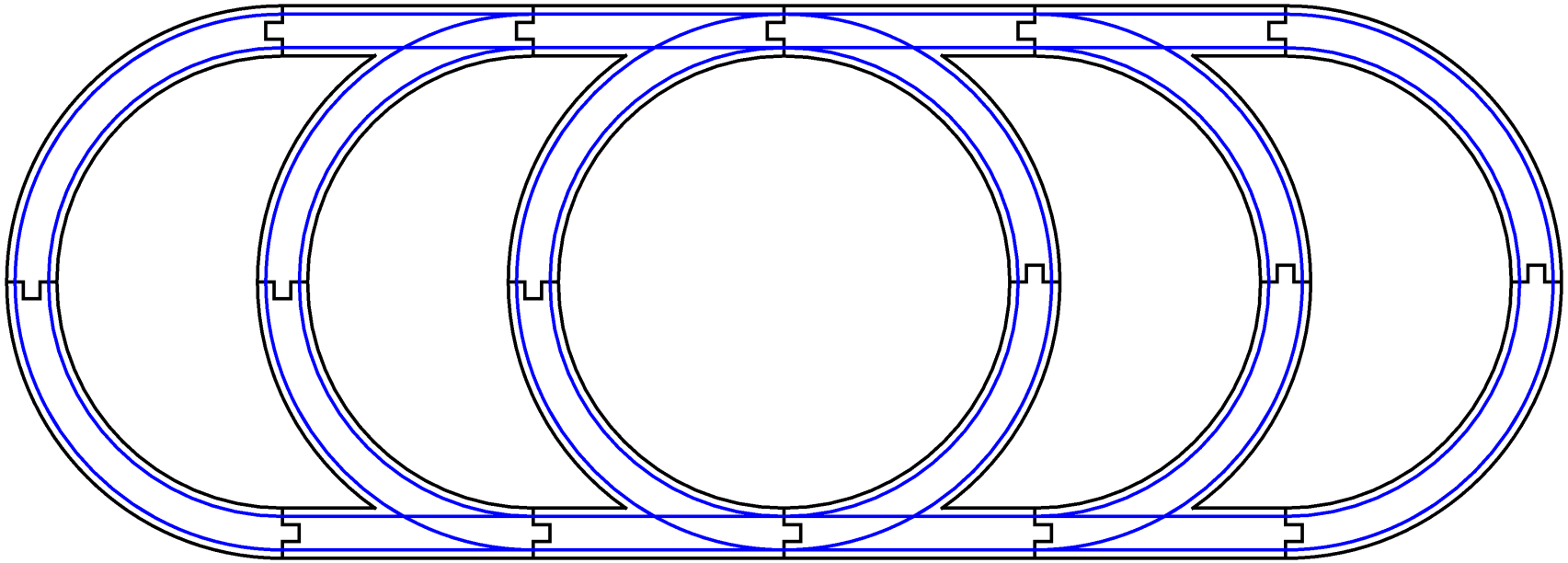


LM : Track $\binom{0}{1}$ **0° [(**$\binom{-1}{0}$**,-90°)]**



Construct this layout by following the outer path:

```
LB.LB.L.L.LM.LM.LB.LB.L.L.LM.LM
```



The pieces are in the correct places, but the type is

```
Track (0,0) 0° [((-1,n),90°),((-1,n),90°),((-1,-n),90°),((-1,-n),90°) | 1 <= n <= 3]
```

We introduce a new constructor for **`Track`**:

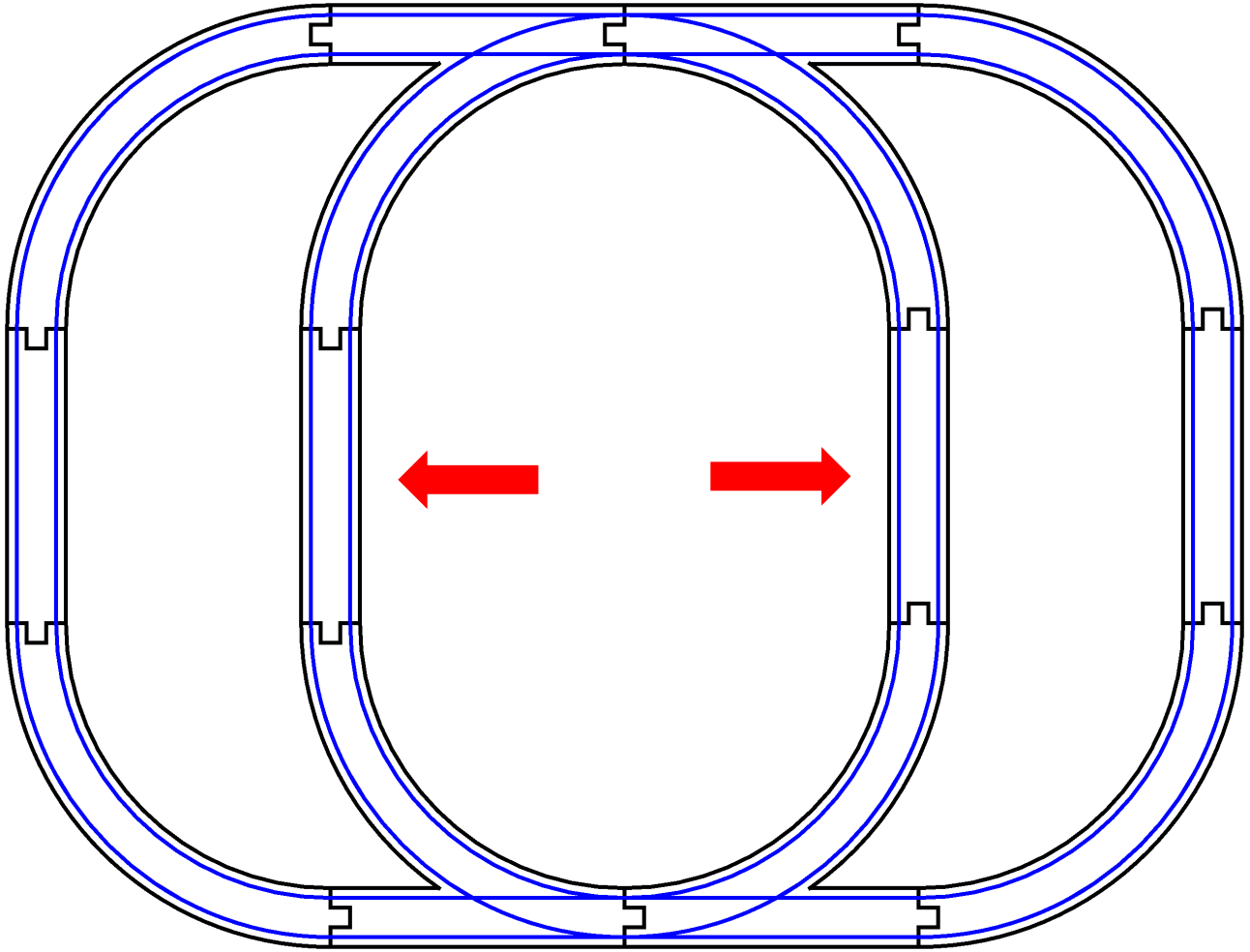
```
Close : Track v θ vs → MatchPairs vs → Track v θ []
```

It checks that loose ends occur in matching pairs, and removes them.

We wrap the constructor in a function **`close`** that automatically finds the term of type **`MatchPairs vs`**.

```
close (LB.LB.L.L.LM.LM.LB.LB.L.L.LM.LM)
```

We were lucky with the previous layout, but for this variation we need to add pieces at the loose ends:



One more constructor:

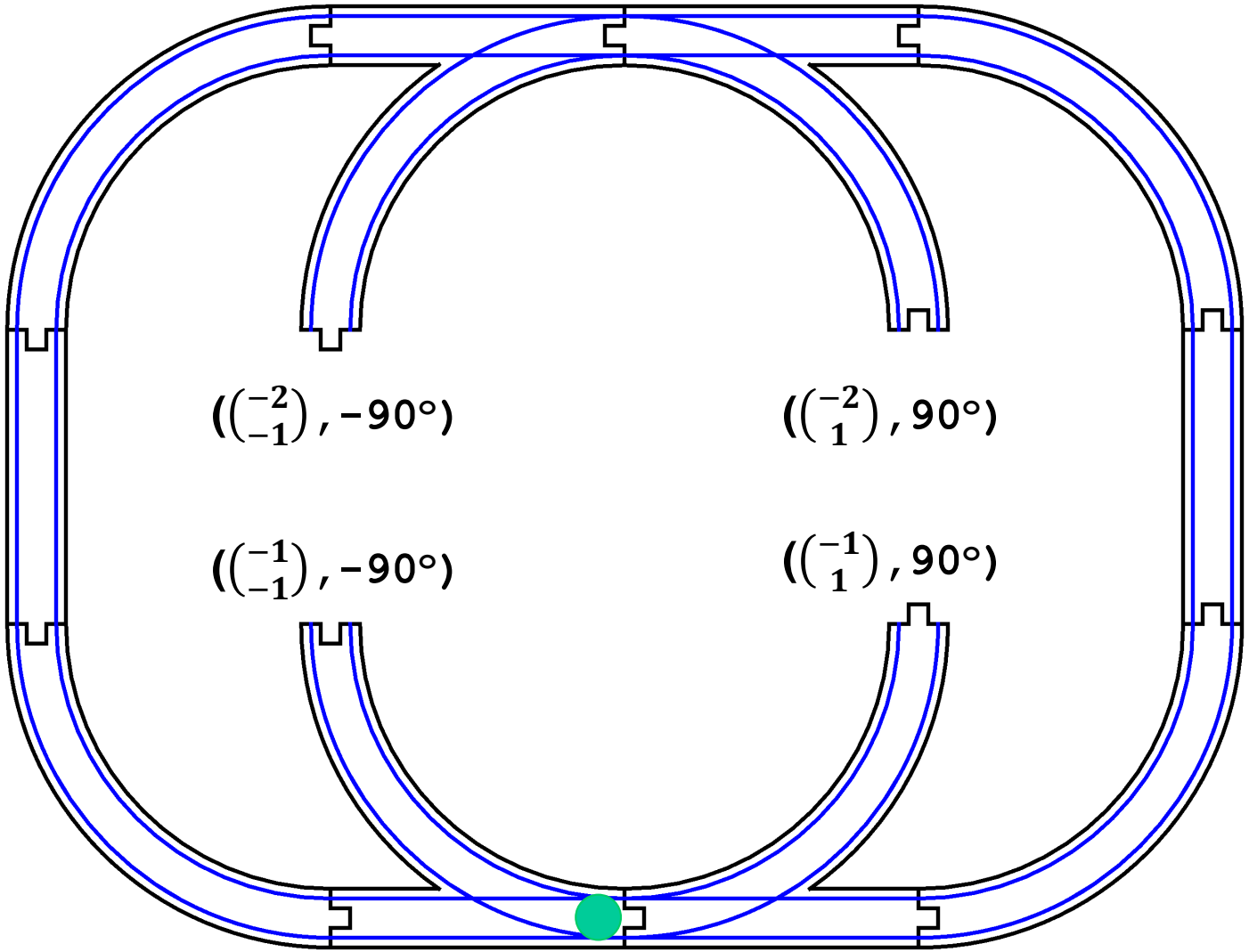
```
Swap : (n : Nat) ->
       (vs : Vect (S n) Vertex) ->
       (m : Fin (S n)) ->
       Track v θ vs ->
       Track (fst (index m vs)) (snd (index m vs))
             (replaceAt m (v,θ) vs)
```

```
Vertex = (Vector,Angle)
```

We wrap the constructor in a function **`swap`** that automatically finds **`n`** and **`vs`**.

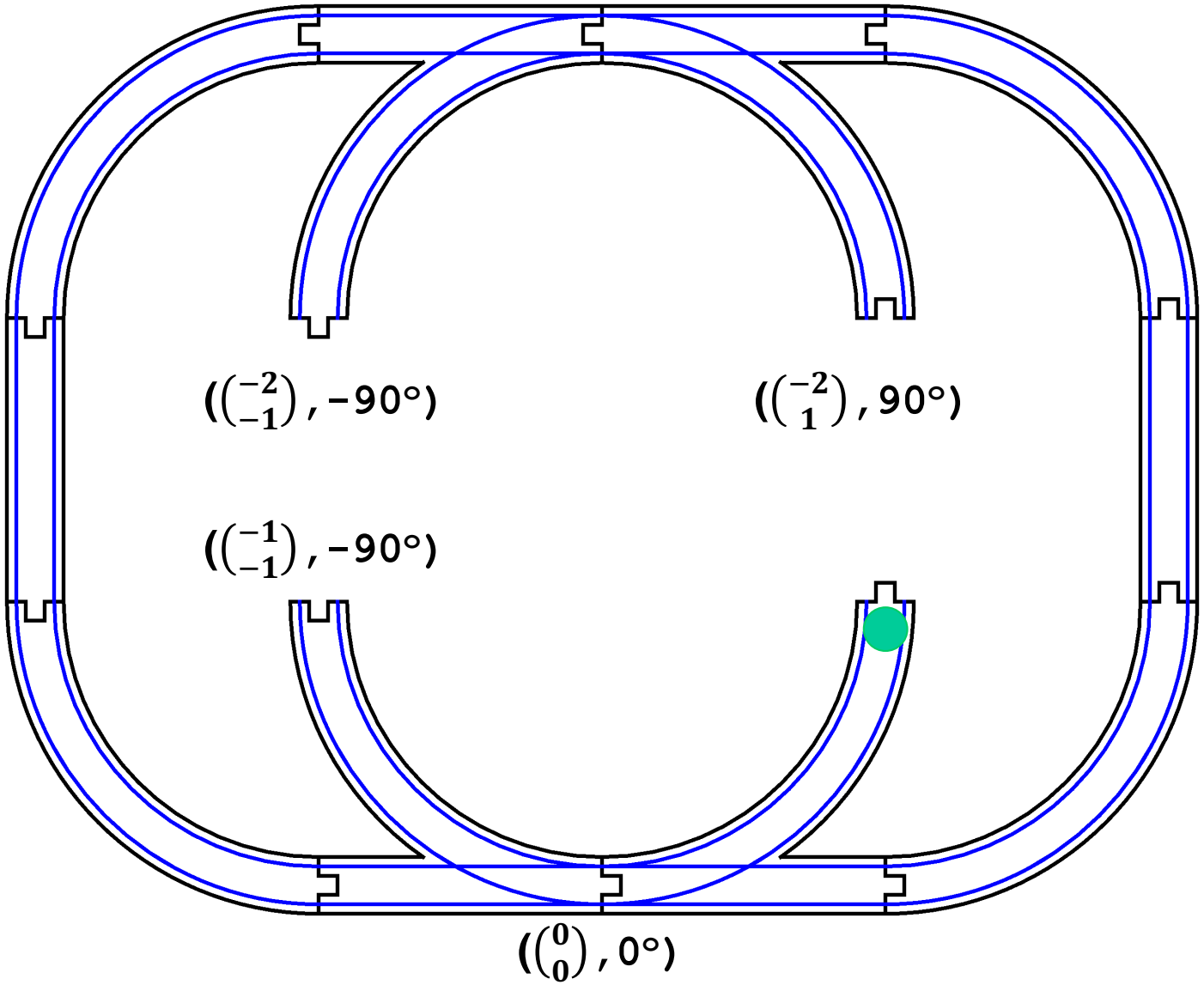
LB.L.F.L.LM.LB.L.F.L.LM :

Track $\binom{0}{1}$ **0°** [$\left(\binom{-1}{1}, 90°\right)$, $\left(\binom{-2}{1}, 90°\right)$, $\left(\binom{-2}{-1}, -90°\right)$, $\left(\binom{-1}{-1}, -90°\right)$]



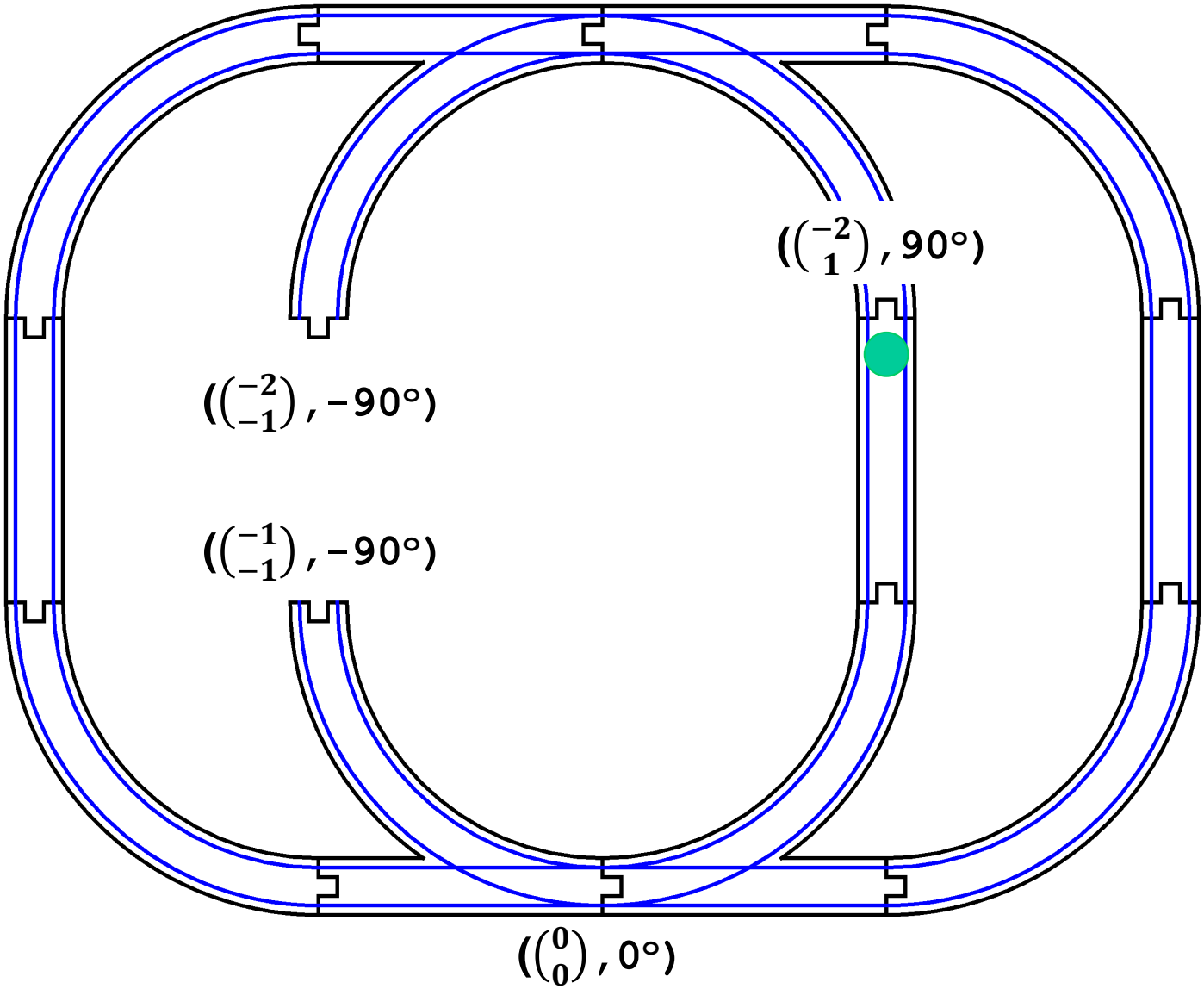
swap 0 (LB.L.F.L.LM.LB.L.F.L.LM) :

Track $\binom{-1}{1}$ 90° [($\binom{0}{0}$,0°), ($\binom{-2}{1}$,90°), ($\binom{-2}{-1}$,-90°), ($\binom{-1}{-1}$,-90°)]



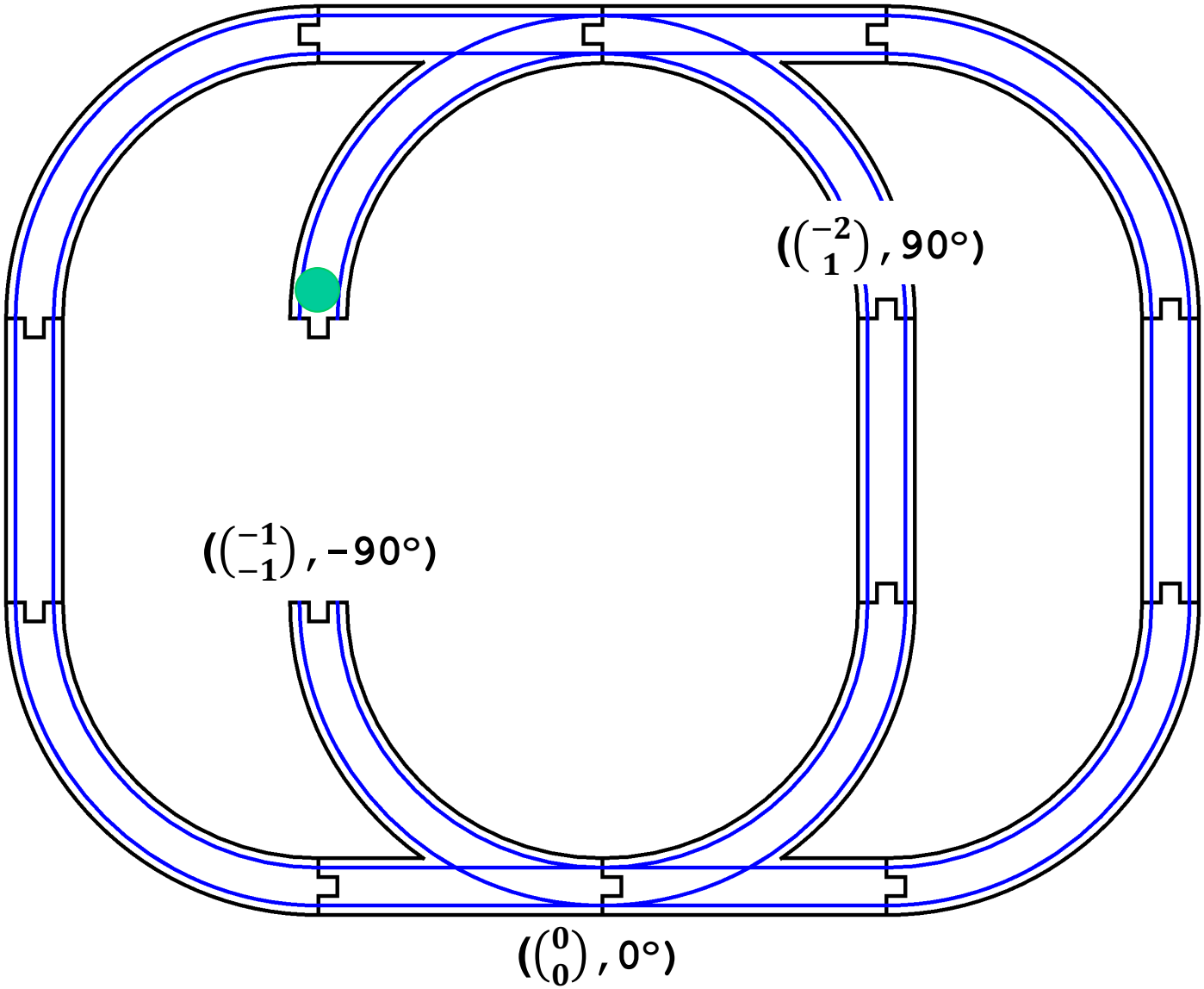
(swap 0 (LB.L.F.L.LM.LB.L.F.L.LM)).F :

Track $\binom{-2}{1}$ 90° [($\binom{0}{0}$,0°), ($\binom{-2}{1}$,90°), ($\binom{-2}{-1}$,-90°), ($\binom{-1}{-1}$,-90°)]

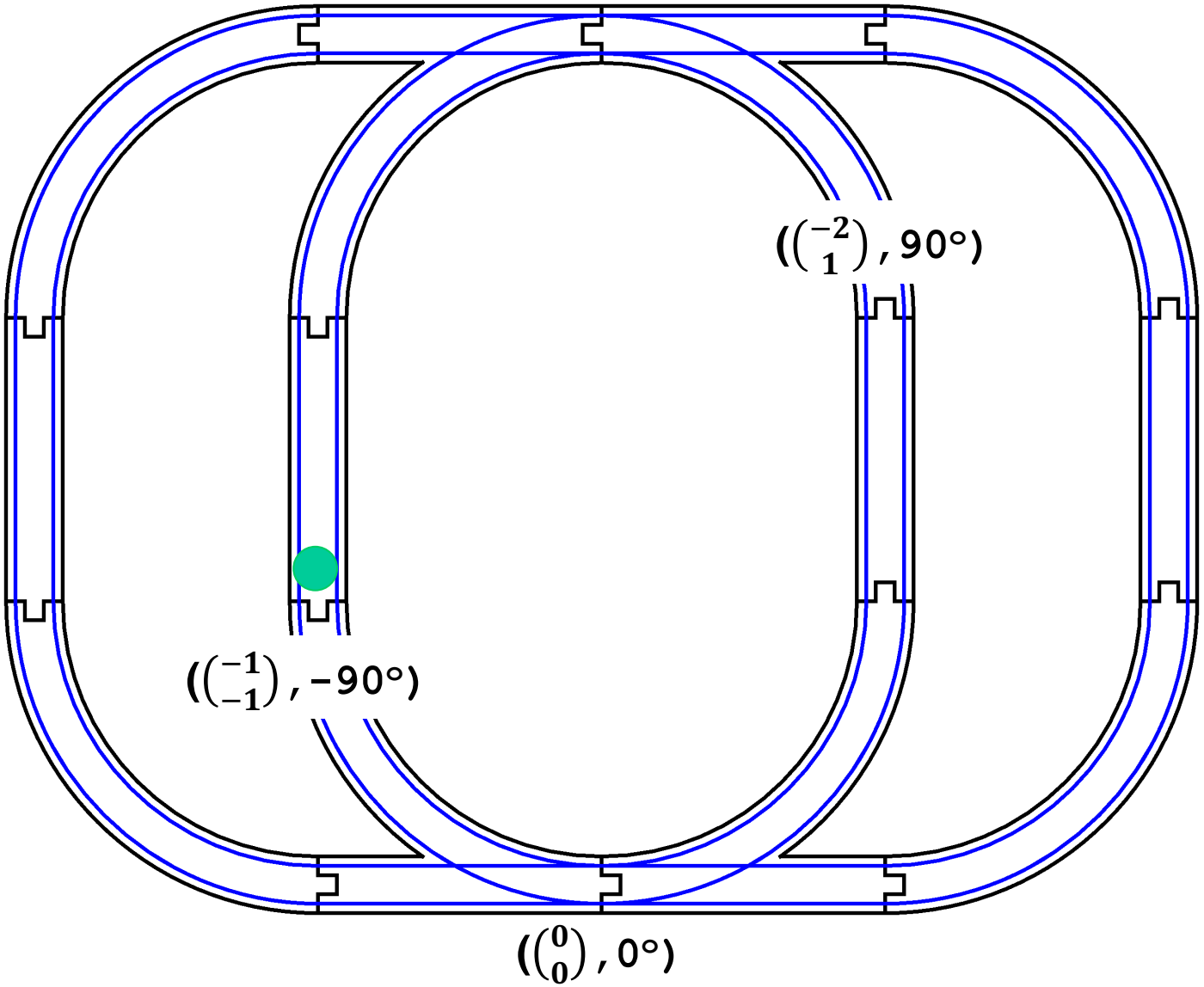


swap 2 ((swap 0 (LB.L.F.L.LM.LB.L.F.L.LM)).F) :

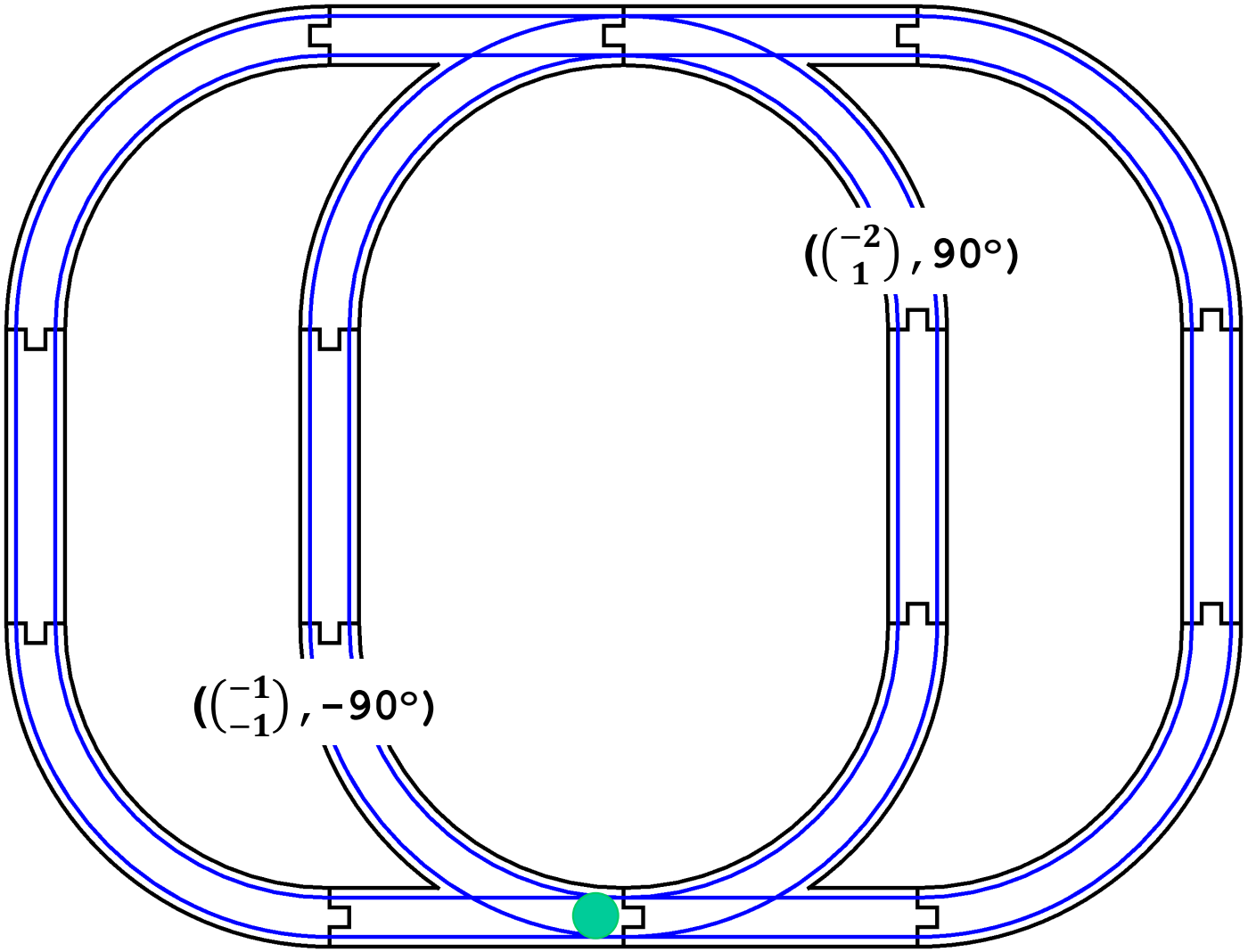
Track $\binom{-2}{-1}$ −90o [($\binom{0}{0}$,0°),($\binom{-2}{1}$,90°),($\binom{-2}{1}$,90°),($\binom{-1}{-1}$,-90°)]



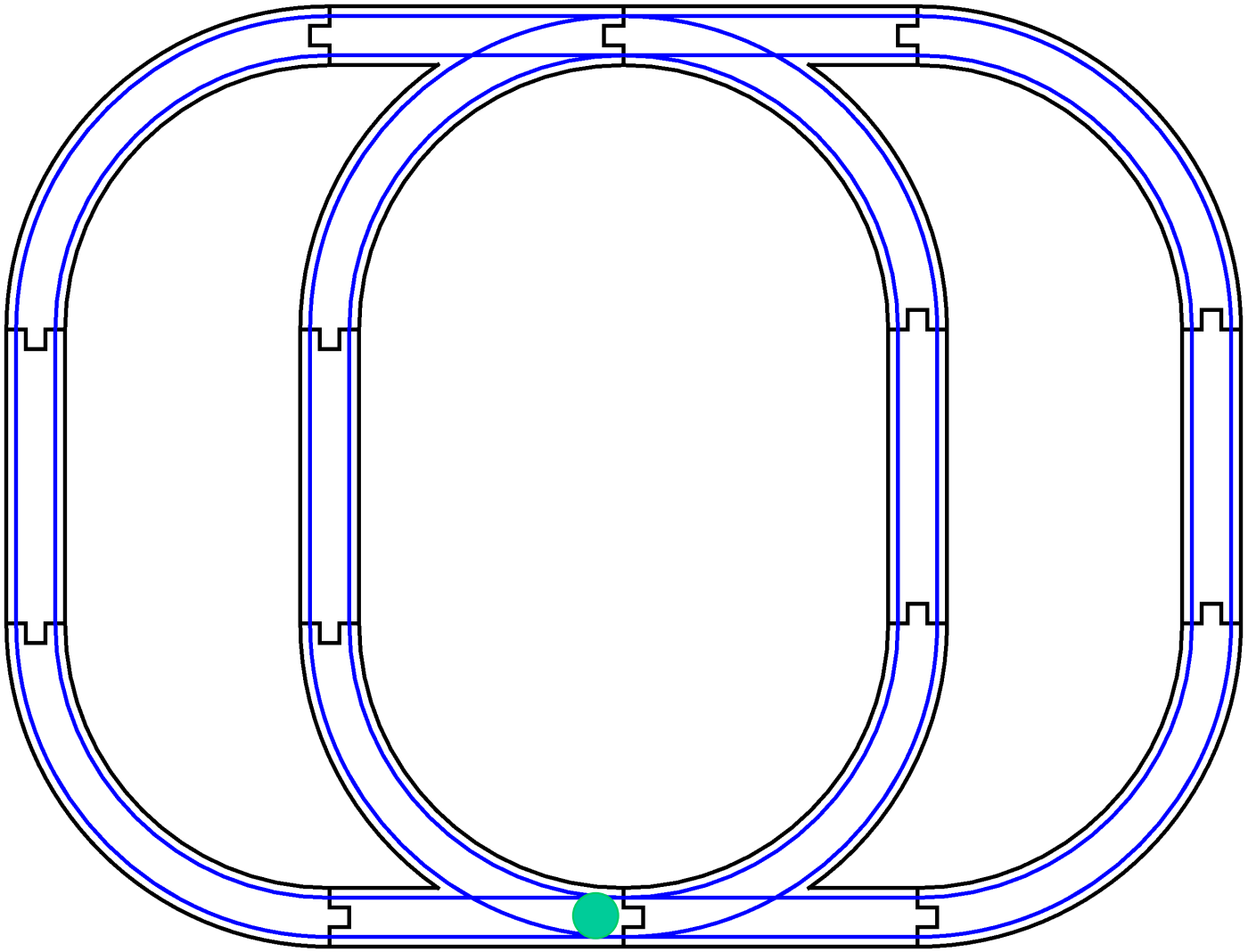
```
(swap 2 ((swap 0 (LB.L.F.L.LM.LB.L.F.L.LM)).F)).F :
Track (-1,-1) -90° [((0,0),0°),((-2,1),90°),((-2,1),90°),((-1,-1),-90°)]
```



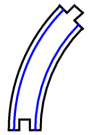
```
swap 0 ((swap 2 ((swap 0 (LB.L.F.L.LM.LB.L.F.L.LM)).F)).F) :
Track (0,0) 0° [((-1,-1),-90°),((-2,1),90°),((-2,1),90°),((-1,-1),-90°)]
```

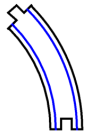


```
close (swap 0 ((swap 2 ((swap 0(LB.L.F.L.LM.LB.L.F.L.LM)).F)).F)) :
Track (0 0) 0° []
```



Adding 45° Curves

R : Track $\begin{pmatrix} 1-\frac{\sqrt{2}}{2} \\ \frac{\sqrt{2}}{2} \end{pmatrix} \begin{pmatrix} \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \\ -\frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \end{pmatrix}$ 

L : Track $\begin{pmatrix} \frac{\sqrt{2}}{2}-1 \\ \frac{\sqrt{2}}{2} \end{pmatrix} \begin{pmatrix} \frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{2} \\ \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \end{pmatrix}$ 

All the coordinates we need are integer linear combinations of 1 and $\frac{\sqrt{2}}{2}$. We can represent $a + b\frac{\sqrt{2}}{2}$ by (a,b) and rework everything, noting that $a + b\frac{\sqrt{2}}{2} = c + d\frac{\sqrt{2}}{2}$ iff $a = c$ and $b = d$.

Checking for self-intersection becomes much harder (impossible?).

Summary

- A dependent type system can track the position of the components of a railway layout.
- If a sequence of constructors produces a layout that doesn't join up properly, this can be detected as a type error at compile time.
- With a little more work (and bigger types), typechecking can also detect self-intersection of layouts.
- I emphasised the diagrams, but the type system is really about correctness of the structure of the track.
- The step from terms-as-data to terms-as-proofs is significant in learning dependently typed programming.
- We encounter the limitations of automated symbolic reasoning in Idris (or Agda).