Spaken

Compass-and-straightedge Constructions in Haskell

Martijn van Steenbergen & Jeroen Leeuwestein
Utrecht University
Dutch Haskell Users Group
Monday, 8 February 2010

Welcome, everyone! Tonight I will introduce you to Spaken, a project by Jeroen and I that started back in 2008. If you're wondering, the name Spaken means "spokes" in Dutch. The name is only marginally related to the subject matter of the project; Jeroen and I like to use nonsensical names and logos (this one was made by Jeroen) that are even less related.
Spaken is about compass-and-straightedge constructions.

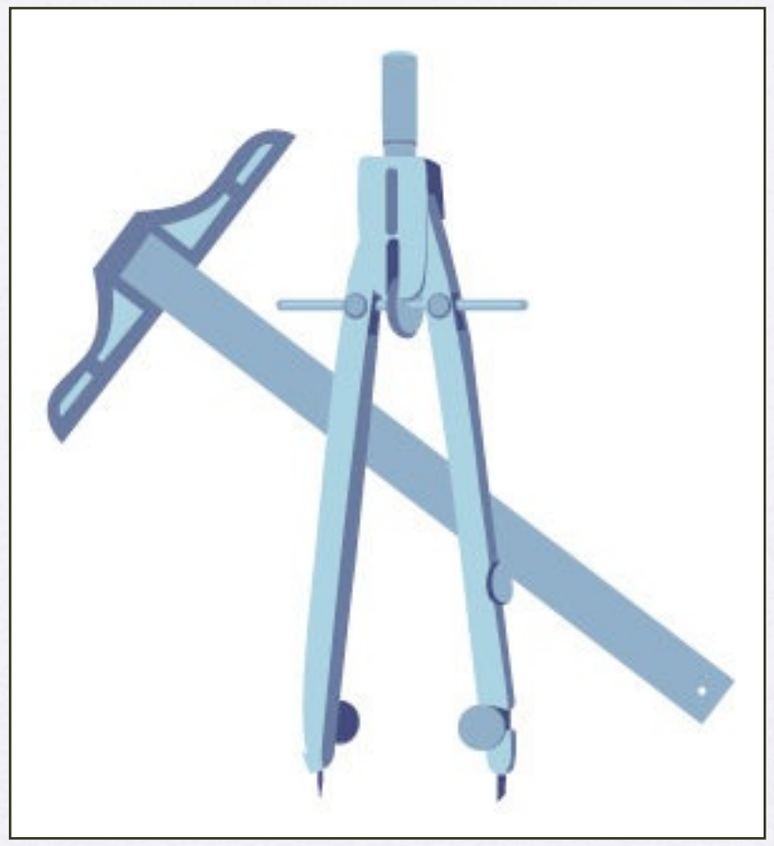
Overview

- Introduction to compass-and-straightedge constructions (CASCs)
- Live demo
- A DSL for constructions
- Serializing and deserializing patterns

We will start by taking a look at these constructions: what are they? There is no better way of explaining them than showing an interactive demo, so we will do just that in a moment. Then we will see how to model constructions in Haskell in such a way that we can serialize and deserialize them.

What are CASCs?

- The compass is used to draw **circles**.
- The straightedge is used to draw straight **lines**. It is unmarked: you cannot use it to measure lengths.
- Constructions must be **exact**.
- What shapes can be created within these limits?



By flickr.com/photos/sakraft1

CASC refers to the two tools that you are allowed to use when drawing geometric constructions. The compass ("passer" in Dutch) is used to draw circles, given a center point and a point on the circle's radius. The straightedge is like a ruler, except it has no markings and you're not allowed to measure distances with it. Instead, it may only be used to draw straight lines.
Constructions must be exact: you're not allowed to estimate distances, for example. Because of these rules, it becomes somewhat of a puzzle game: what shapes can you create within these limits, and what shapes are impossible to construct?

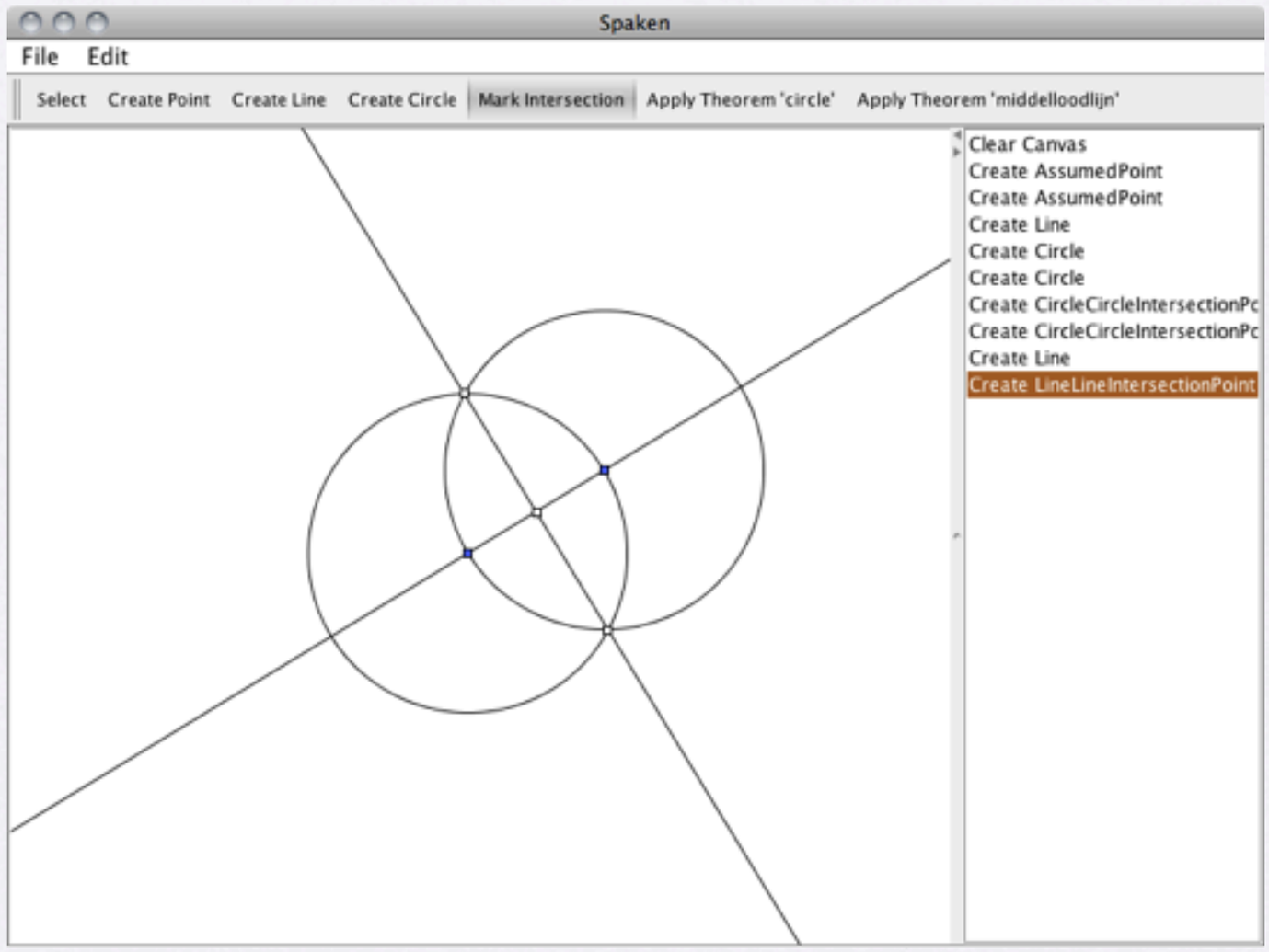
What are CASCs?

- Mathematically, it's an **axiomatic system**.
  - A finite set of **primitives**: the *axioms*.
  - Axioms are **composed** to make greater constructions: the *theorems*.
- The axioms roughly are:
  - Draw an arbitrary **point**.
  - Given two points A and B, draw the **line** that goes through A and B.
  - Given two points A and B, draw a **circle** centered at A through B.
  - Mark the **intersections** of two elements.

If you look at it from a mathematical point-of-view, these rules form an axiomatic system. There are a few primitive constructions; these form the axioms. You can combine axioms in certain ways; these form theorems. The theorems themselves may be used again in other theorems.
These primitives operations are roughly: draw an arbitrary point, draw a line, draw a circle and mark an intersection point between a line and a line, a line and a circle or two circles. I say roughly, because as we will see when we start modelling the constructions in Haskell, this is still too vague.

Demonstration

```
% svn co http://spaken.googlecode.com/svn/trunk/ spaken
% cd spaken
% rake run
```



Before we continue, let's look at an example. Jeroen and I have built a Java program that allows you to draw a limited set of constructions. If you want, you can try it yourself later: the code is available online. Entering these three commands in a terminal should do it. You will need Subversion, Java 5 and Ruby.

Interesting problems

- Can you construct an ***n*-sided polygon** in a given circle?
  Answer: only if *n* is the product of a power of 2 and any number of distinct Fermat primes. (Carl Friedrich Gauss, 1801 and Pierre Wantzel, 1837)
- Can you **divide a line segment** in *n* equal subsegments?
  Answer: yes.
- Can you **divide an angle α** in *n* equal subangles?
  Answer: it depends on **α** and on *n*.
- Can you construct a **golden rectangle**?
  Answer: yes.

Famous theorems

- The **compass equivalence theorem** says that the compass may transfer distances even if it collapses. (Euclid's *Elements*, book I, proposition II, 300 BC)
- The **Mohr–Mascheroni theorem** says that you don't need a straightedge. (Georg Mohr, 1672 and Lorenzo Mascheroni, 1797)
  - **Napeleon's problem**: finding the center of a circle with only a compass.
- The **Poncelet–Steiner theorem** says that you don't need a compass, if you are given a single circle and its center. (Jean Victor Poncelet, 1822 and Jakob Steiner, 1833)

Famous unsolvable problems

- **Squaring the circle**: draw a square with the same area as a given circle. (Ferdinand von Lindemann, 1882)
- **Doubling the cube**: given the side of a cube with volume V, construct the side of a cube with volume 2V. (Pierre Wantzel, 1837)
- **Trisecting the angle**: equally divide an arbitrary angle in three. (Pierre Wantzel, 1837)

Goal of Spaken

- To create **constructions**.
- To store a created construction as a **pattern**:
  - mark input;
  - mark output.
- To **reuse** stored patterns in bigger constructions.

So why did we build Spaken? There is plenty of software out there already that allows you to create constructions. However, building the application was unexpectedly satisfying: it's a lot of fun to build something so elegant yet interactive.
But there was a goal to Spaken: I wanted to be able to save user-created constructions as "procedures", give them names and use them again in bigger constructions.
In Java because Java2D is a joy to use.
For some reason we didn't manage to build this. Either it is a very difficult problem, or Jeroen and I suck at programming (or both).

Modelling constructions in Haskell

- Constructions can be modelled using **functions**:
  - Define a finite set of primitive functions.
  - Theorems are functions defined in terms of these primitives.
  - Invoking a theorem equals calling a function.
- For example:

```
bisection :: Point -> Point -> Line
parallel  :: Line  -> Point -> Line
```


So I decided to do something that had helped before in other situations: model the problem in Haskell. Somehow in modelling problems, the stronger types guide you towards a solution.
In the next few slides we will develop a DSL for constructions in Haskell (tackling one problem per slide).
Constructions (or axiomatic systems) and Haskell make a good match in general, because you can model theorems as functions. The axioms, then, are a limited set of primitive functions, and theorems are composed of axioms and other theorems.

Monadic constructions

- `arbitraryPoint :: ??`
- Modelling constructions using monads has advantages:
  - The creation of elements is **explicit and observable**.
  - Consequently, the reuse of created elements is explicit and observable.
- These properties are also important in order to serialize constructions efficiently.

```
instance Monad Spaken

arbitraryPoint :: Spaken Point

example = arbitraryPoint >>= \p -> ...
```


However, functions alone are not enough. What should be the type of arbitraryPoint? It doesn't take any input, so should it just be "Point"? But how do you distinguish between using the same arbitrary point twice, and constructing two different arbitrary points?
To model this, we need a system that is able to model effects. We know of three in Haskell: applicatives, arrows and monads. I've gone with monads (or actually a restricted form as we will later see). I will say a bit more about this later on.
Let's call our monad Spaken. We can then think of a suitable type for arbitraryPoint and use it like this. There is now a clear distinction between the creation of the arbitrary point, and the created point itself (which is called "p" here).
But one of our goals was storing patterns, and serializing monads is a lot harder than serializing a normal datatype.

Axioms in their own type class

- Why capture the axioms in a type class?
  - Clear distinction between axioms and theorems.
  - Allows multiple implementations.
  - A polymorphic type proves independence of specific implementations.

```
class Monad m => MonadSpaken m where
  arbitraryPoint ::                      m Point
  line           :: Point  -> Point  -> m Line
  circle         :: Point  -> Point  -> m Circle
```


In the next few slides we will develop a set of functions to model constructions. The first thing we will do is clearly separate the axioms from the theorems by putting the axioms in a type class. Not only is this a clean separation, it also allows us to supply different semantics for the constructions by writing several instances of the type class. Also, if a construction is still polymorphic in the type class, we can tell it doesn't depend on any specific implementation.
Let's call our class MonadSpaken. It contains three functions: one for each elemental shape. We will look at intersections in a moment. First, we should ask ourselves: the types Point, Line and Circle are all concrete types. What should their implementations be?

Abstract element types

- But `Point`, `Line` and `Circle` are still concrete types. What should their implementations be?
- Defer implementation: turn them into indices.

```
data Point
data Line
data Circle
```

```
class Monad m => MonadSpaken m r | m -> r where
  arbitraryPoint ::                          m (r Point)
  line           :: r Point  -> r Point  -> m (r Line)
  circle         :: r Point  -> r Point  -> m (r Circle)
```

- Kinds of type parameters:
  - `m :: * -> *` (as required by `class Monad`)
  - `r :: Element -> *`
  - `datakind Element = Point | Line | Circle`

We don’t know; it will depend on the instance of the class, and we didn’t want to commit to any instances yet. So we introduce an extra type constructor r and use Point, Line and Circle as indexes. Because we give no extra information about r, polymorphic constructions cannot do anything with the values other than pass them to other functions.

Note that the kind of m is * -> *, like any monad type constructor. And (as least to Haskell) the kind of r is also * -> *, but in practice we will only pass it one of the element types, so you can think of it as having kind Element -> *.

We’re also added a functional dependency that says that the r is uniquely determined by the choice of m.

Intersections

- What result type should intersections have?
- Intersections can yield 0, 1, 2 or infinitely many points.

```
class Monad m => MonadSpaken m r | m -> r where
  arbitraryPoint ::                          m (r Point)
  line           :: r Point  -> r Point  -> m (r Line)
  circle         :: r Point  -> r Point  -> m (r Circle)
  intersectLL    :: r Line   -> r Line   -> m ??
  intersectLC    :: r Line   -> r Circle -> m ??
  intersectCC    :: r Circle -> r Circle -> m ??
```


How should we model intersections? What should their return type be?

Intersections

- What result type should intersections have?
- Intersections can yield 0, 1, 2 or infinitely many points.
- Solution: intermediate index `Points`.

```
class Monad m => MonadSpaken m r | m -> r where
  arbitraryPoint :: m (r Point)
  line           :: r Point  -> r Point  -> m (r Line)
  circle         :: r Point  -> r Point  -> m (r Circle)
  intersectLL    :: r Line   -> r Line   -> m (r Points)
  intersectLC    :: r Line   -> r Circle -> m (r Points)
  intersectCC    :: r Circle -> r Circle -> m (r Points)
```


Depending on a line or circle’s position, the intersection might not exist, or consist of only 1 or 2 points. Or, if lines or circles coincide, the result is an infinite number of points.

Intersections

- What result type should intersections have?
- Intersections can yield 0, 1, 2 or infinitely many points.
- Solution: intermediate index `Points`.
- Introduce projection functions (they might fail).

```
class Monad m => MonadSpaken m r | m -> r where
  arbitraryPoint :: m (r Point)
  line           :: r Point  -> r Point  -> m (r Line)
  circle         :: r Point  -> r Point  -> m (r Circle)
  intersectLL    :: r Line   -> r Line   -> m (r Points)
  intersectLC    :: r Line   -> r Circle -> m (r Points)
  intersectCC    :: r Circle -> r Circle -> m (r Points)
  point1         :: r Points ->             m (r Point)
  point2         :: r Points ->             m (r Point)
```


To solve this, we introduce a fourth index that represents a set of points. To be able to use the elements of such a set in further steps, we also add two projection functions. Obviously m should be able to represent failure in some way. There might be better ways to model this, but this way suffices for now.

Example: Bisection

```
bisection :: MonadSpaken m r =>
  r Point -> r Point -> m (r Line)
bisection p1 p2 = do
  c1 <- circle p1 p2
  c2 <- circle p2 p1
  ps <- intersectCC c1 c2
  i1 <- point1 ps
  i2 <- point2 ps
  line i1 i2

class Monad m => MonadSpaken m r | m -> r where
  arbitraryPoint :: m (r Point)
  line           :: r Point  -> r Point  -> m (r Line)
  circle         :: r Point  -> r Point  -> m (r Circle)
  intersectLL    :: r Line   -> r Line   -> m (r Points)
  intersectLC    :: r Line   -> r Circle -> m (r Points)
  intersectCC    :: r Circle -> r Circle -> m (r Points)
  point1         :: r Points ->             m (r Point)
  point2         :: r Points ->             m (r Point)
```


With these primitives, we have enough to model the bisection we saw during the demo earlier. It looks like this. To keep the examples in the following slides small, we only keep those axioms we used in bisection.

Rendering elements

- Let's write an instance of `MonadSpaken`: rendering the various shapes in the 2D plane, i.e. computing the positions of the various elements.
- By choosing a data family for `r :: Element -> *`, we can build a datatype for the different element types.

```
data family   Mk element :: *
data instance Mk Point   =  MkPoint  Double Double
data instance Mk Line    =  MkLine   (Mk Point) (Mk Point)
data instance Mk Circle  =  MkCircle (Mk Point) (Mk Point)
data instance Mk Points  =  MkPoints (Mk Point) (Mk Point)
```


We are going to write in instance of MonadSpaken that computes the position of elements in the 2D plane.
Let's first think about what r we're going to use. The data we want to track depends on the index of r, so we can use a data family for this. For points, we keep track of the x- and y-coordinates. For lines and circles, we keep track of the points that defined them.
For a set of points, we will cheat a bit as assume there are always two points in the set.

Rendering elements

- Rendering of elements requires no side-effects, so set `m = Identity`.

```
newtype Render a = Render (Identity a)
  deriving (Functor, Monad)

instance MonadSpaken Render Mk where
  line   p1 p2             = return (MkLine   p1 p2)
  circle p1 p2             = return (MkCircle p1 p2)
  point1 (MkPoints p1 _) = return p1
  point2 (MkPoints _ p2) = return p2
  intersectCC c1 c2        = return (MkPoints ... ...)
```


Computing these positions don’t require any side-effects, so we choose m to be Identity. Or rather, Identity wrapped in a newtype, because otherwise we get in trouble with the functional dependency.
The first four functions are straightforward. intersectCC is more interesting but a bit too long to include on the slide and not important for the story.

Serializing patterns

- What is a pattern?
  - A function with type
    `MonadSpaken m r =>`
    `  r e1 -> r e2 -> ... r en -> m (r e0)`
- The primitives are patterns, too.
- What do we mean by serializing?
  - Converting to bytes, or
  - converting to a datatype that has `Read` and `Show` instances.
- Serializing throws away type information; deserializing recovers type information.
- What is the type of `serialize`?

```
data SerialPattern = ??
  deriving (Read, Show, Eq)

serialize :: ?? -> SerialPattern
```


Now the more interesting part. Let's think about serializing patterns.
First: what do we mean by patterns? An example of a pattern is bisection: it takes some elements as arguments and yields a new element. So we define pattern as a function in terms of MonadSpaken, of the following shape. Note that the axioms are patterns too, according to this definition.
Second: what do we mean by serializing? Technically it means converting it to a series of bytes, but I am content with converting it to a datatype that has Read and Show instances, because if we have those, converting to and from bytes is trivial. If you look at it like this, then serializing is like throwing away all type information, while deserializing recovers that type information again.
But if a pattern can take arguments, what is the type of serialize?
In the next few slides we will look in detail at the code responsible for the serialization.

Swallowing arguments

```
data Pattern m r where
  Ret :: Ty ty -> m (r ty)                -> Pattern m r
  Arg :: Ty ty -> (r ty -> Pattern m r) -> Pattern m r


-- Ty :: Element -> *

data Ty ty where
  TyPoint  :: Ty Point
  TyLine   :: Ty Line
  TyCircle :: Ty Circle
  TyPoints :: Ty Points
```


We can fix this by using a GADT to hide the arguments. Let's call it Pattern. It has two constructors: one that swallows and argument and one that says: we're done now, there are no more arguments to swallow. The m and r are type arguments to Pattern, and will probably have the MonadSpaken constraint applied to them when they're used in functions. Note that the return type is existentially quantified: we don't want to see it in Pattern's return type, because we want to unify the types of all patterns. But at the same time, we don't really want to lose the type information.

For that reason, we also store a value of datatype Ty. It's a GADT with a constructor for every possible element index. One of the nice things about GADTs is that pattern matching on such a constructor can refine an existing type, so by storing a Ty value in Ret, we can later recover the type we existentially quantified.

In Arg we also quantify over a type, namely the argument index. It takes a function whose result is yet another Pattern, so we eventually we end up with a Ret constructor. Let's look at an example.

Swallowing arguments

```
bisectionPattern :: MonadSpaken m r => Pattern m r
bisectionPattern =
  Arg TyPoint $ \p1 ->
  Arg TyPoint $ \p2 ->
  Ret TyLine  $ bisection p1 p2


unwrapBisection :: MonadSpaken m r => Pattern m r ->
  r Point -> r Point -> m (r Line)
unwrapBisection arg p1 p2 = case arg of
  Arg TyPoint f           -> case f p1 of
    Arg TyPoint g         -> case g p2 of
      Ret TyLine l        -> l
```


This is what bisection looks like as a Pattern. It takes two arguments, so it uses the Arg constructor twice.

To see that no type information is lost, below is the code for transforming the pattern back to its former type. By pattern matching on the Ty values, this function is well-typed.

Using references

- Patterns are composed of primitives.
- Every primitive pushes exactly 1 new element on the stack.
- Push pattern arguments onto the stack first.
- Set `r = Ref`. A `Ref` indexes the stack.

```
newtype Ref a = Ref { getRef :: Int }

bisection (Ref 0) (Ref 1) = do
  Ref 2 <- circle       (Ref 0) (Ref 1)
  Ref 3 <- circle       (Ref 1) (Ref 0)
  Ref 4 <- intersectCC (Ref 2) (Ref 3)
  Ref 5 <- point1       (Ref 4)
  Ref 6 <- point2       (Ref 4)
  Ref 7 <- line         (Ref 5) (Ref 6)
  return                (Ref 7)
```


That solves one of the problems. Now we still need to figure out what to set m and r to.

Note that every primitive creates one new element. We will model a pattern using a stack of created elements, so every primitive pushes one new element on the stack. We set r to Ref. What Ref does is ignore its index and always store an integer. This integer is an index into the stack.

To make this a bit clearer, let's look at bisection with the Refs filled in. Note that its arguments get pushed onto the stack first: they get indices 0 and 1. Then every statement uses references to elements created in the lines before.

Type SerialPattern

```
data SerialPattern = SerialPattern
    { serialArgTypes     :: [SomeTy]
    , serialConstruction :: [AxiomInvoc]
    , serialReturnType   :: SomeTy
    , serialReturnRef    :: Int
    }
  deriving (Eq, Show, Read)

type AxiomInvoc = (AxiomType, [Int])

data AxiomType
  = AxLine | AxCircle | AxIntersectCC | AxPoint1 | AxPoint2
      deriving (Eq, Show, Read)

data SomeTy where
  SomeTy :: Ty ty -> SomeTy

instance Eq   SomeTy where ...
instance Show SomeTy where ...
instance Read SomeTy where ...
```


Now we have a good idea what the result type of serialize looks like. Let's call it SerialPattern.

The stack at the end of the construction is modelled by a list of axiom invocations. An AxiomInvoc is a combination of an axiom type and its arguments, which are always other elements on the stack and therefore modelled as indexes.

We also need to store how many arguments the pattern takes. Not only do we store how many arguments, we also store their types, lest we need to do type inferencing. For the return value, we store again an index and its return type.

Types are modelled using Ty, but wrapped in a constructor that discards the index so that read :: String -> SomeTy is well-typed.

Serialized bisections

```
> serialize bisectionPattern
SerialPattern
  { serialArgTypes     = [TyPoint,TyPoint]
  , serialConstruction =
    [ (AxCircle,      [0,1])
    , (AxCircle,      [1,0])
    , (AxIntersectCC, [2,3])
    , (AxPoint1,      [4])
    , (AxPoint2,      [4])
    , (AxLine,        [5,6])
    ]
, serialReturnType   = TyLine
, serialReturnRef    = 7
}
```


Again, let’s look at what a bisection would look like in this form.

If you count the elements in the list, you can see that they match the Refs from earlier.

State monad to the rescue

```
serialize :: Pattern Serialize Ref -> SerialPattern

newtype Serialize a = Serialize (State SerializeState a)
  deriving (Monad, MonadState SerializeState)

type SerializeState = ([SomeTy], [AxiomInvoc])

doSerialize :: Pattern Serialize Ref -> Serialize SerialPattern
doSerialize pat = case pat of
  Ret ty sp -> do
    Ref r <- sp
    (argTypes, invocs) <- get
    return (SerialPattern argTypes invocs (SomeTy ty) r)
  Arg ty f -> do
    argTypes <- gets fst
    modify (first (++ [SomeTy ty]))
    let newRef = Ref (length argTypes)
    doSerialize (f newRef)
```


Now we have a good idea what to choose for m. A simple state monad will do. We wrap it in a newtype called Serialize.

For the state we keep track of the argument types and the stack of invocations. We take a Pattern as input (with m and r specialized) and pattern match on the constructors: does the pattern take any more arguments?

If so, we add the argument type to the first list. Then we compute a new reference, pass it to the stored function and go in recursion on the result.

If we find the Ret node, we execute the MonadSpaken action, which will push the remaining items on the stack. After that we have the final state and are able to construct and create a SerialPattern value.

State monad to the rescue

```
instance MonadSpaken Serialize Ref where
  line        (Ref p1) (Ref p2) = mkInvoke (AxLine,        [p1, p2])
  circle      (Ref pc) (Ref pr) = mkInvoke (AxCircle,      [pc, pr])
  intersectCC (Ref c1) (Ref c2) = mkInvoke (AxIntersectCC, [c1, c2])
  point1      (Ref ps)          = mkInvoke (AxPoint1,      [ps])
  point2      (Ref ps)          = mkInvoke (AxPoint2,      [ps])

mkInvoke :: AxiomInvoc -> Serialize (Ref a)
mkInvoke invoc = do
  (argTypes, invocs) <- get
  let newRef = Ref (length argTypes + length invocs)
  put (argTypes, invocs ++ [invoc])
  return newRef
```


The only thing that remains, then, is our MonadSpaken instance.

It’s not very complex. Every axiom receives some references as arguments and pushes a new element onto the stack and returns a newly created reference that depends on the current size of the stack.

Deserialization: a sketch

- `deserialize :: MonadSpaken m r =>`
  `SerialPattern -> Pattern m r`
- Uses a stack of results of axiom invocations.
- First process argument types: every argument results in a `Ret` constructor. The lambda pushes its argument onto the stack.
- Then process the body of the pattern.
  Deserialization of one invocation:
  - Look up axiom pattern in axiom table.
  - Look up axiom's arguments on the stack.
  - Invoke axiom; bind result.
  - Push result onto stack.

```
MonadSpaken m r => StateT [StackEl r] m ()

data StackEl r where
  StackEl :: Ty ty -> r ty -> StackEl r
```


It feels like SerialPattern stores enough information, but the only way to be sure is to see if we can write a deserialize. We already know its type.

We won't look at the code in detail; we've already seen enough code for today. But I will sketch how it works.

No MonadSpaken instance is necessary for deserialization. Instead, we build a result that is polymorphic in But like before we will keep a stack of created elements. Except this time it will hold typed values, polymorphic in MonadSpaken. First we process the arguments: every argument results in a Ret constructor. Remember, Ret takes a function. So the lambda we create pushes its argument on the stack, and then we continue processing the rest.

Then we process the list of AxiomInvocs. The trick here is that we use a StateT over MonadSpaken. Note that we stay polymorphic in MonadSpaken. Then at the end we call execStateT which gives us the final stack. We look up the return index in the stack and return that value.

For every AxiomInvoc we look up the corresponding axiom action in a table.

The axiom table

```
axioms :: MonadSpaken m r => [(AxiomType, Pattern m r)]
axioms =
  [ (AxLine,
      Arg TyPoint  $ \p1 ->
      Arg TyPoint  $ \p2 ->
      Ret TyLine   $ line p1 p2)
  , (AxCircle,
      Arg TyPoint  $ \p1 ->
      Arg TyPoint  $ \p2 ->
      Ret TyCircle $ circle p1 p2)
, (AxIntersectCC,
    Arg TyCircle $ \c1 ->
    Arg TyCircle $ \c2 ->
    Ret TyPoints $ intersectCC c1 c2)
...
```


This is what the table looks like. It maps from AxiomTypes to Patterns.

The patterns indicate how many arguments we should expect. As we pattern match on each Arg, we look up the corresponding value in the stack, check that its type matches the type of the argument, and call the function.

Why is this possible?

- Only `r`-wrapped arguments.
- Only `r`-wrapped results.
- Effectively, `(>>=) :: MonadSpaken m r =>`
  `m (r a) -> (r a -> m (r b)) -> m (r b)`.
- Because `r`-values are abstract, the body of a pattern cannot depend on the values of its arguments, so `serialize` can run a computation with placeholder values as arguments and see where they end up.
- The possible indexes of `r` are known at compile-time. This allows existentially quantified types to be recovered.

So how come we can store monadic computations? In general, we cannot serialize binds because the right-hand side of a bind is a function.

Well, class MonadSpaken has some important properties. All the arguments are wrapped in r's. Also, all the return types are wrapped in r's. This effectively restricts the type of the bind.

Future work

- Allow patterns to return multiple elements.

```
tuple :: MonadSpaken m r => r a -> r b -> m (r (a, b))
fst   :: MonadSpaken m r => r (a, b)   -> m (r a)
snd   :: MonadSpaken m r => r (a, b)   -> m (r b)
```

- Investigate the use of arrows instead of monads.
- More powerful `Points` projections.
  - Arbitrary point on {Line, Circle}.
  - That point which is on the same side of the line as this point.
- Support for angles.
- Port the general idea to Java.
- Improve interaction design in the Java application.

Applicatives are probably too weak to make the distinction between creation and reuse of elements.

Conclusion

- By using monads, the distinction between the construction of a new element and the reuse of a previously constructed element is made explicit.
- By restricting the type of the bind operator, the resulting monadic computations are deterministic and permit serialization, among other things.
- Constructive geometry is fun!