Constructions and Properties of the Heighway Dragon Curve Formalized in Agda

Ting-Wu Chang

Abstract

The Heighway dragon curve is generated by iteratively applying transformation to a unit line segment. Two constructions of the dragon curve—the "unfolding" and "expansion" constructions—are proved equivalent in Agda. Additionally, we outline a proof for the property that the dragon curve never overlaps itself.

Introduction The Heighway dragon curve is a fractal curve constructed by iteratively applying a sequence of transformations to a unit line segment. Here, two types of transformations—unfolding and expansion [1, p. 3]—that can be used to form a construction are introduced and formalized.

Unfolding The "unfolding" construction of the dragon curve starts from a unit line segment with start and end points. We repeatedly 1. make a copy of the curve, 2. rotate the copy 90 degrees clockwise (R) or counterclockwise (L), 3. connect the two curves at the end points, and 4. make the starting point of the copy the new end point for the whole curve, as demonstrated in Fig. 1.

This informal construction can be formalized by introducing a language consisting of lists of the letters $+$ and $-$. Each list provides instructions for drawing a curve. For $-$, we turn the drawing direction 90 degrees clockwise, while $+$ turns it counterclockwise. At the beginning and after each letter, we draw a unit line segment in the current direction. With this language, dragon curves can be generated by a recursive function, `dragonU`, that operates on a list of "operations" R and L:

```
dragonU : List Op → List Letter
dragonU []          = []
dragonU (op :: ops) = dragonU ops ++ choose op :: invert (reverse (dragonU ops))
```

where `choose` is a function that maps R to $+$ and L to $-$, and `invert` swaps $+$ and $-$ [2].

The function represents the unfolding construction since, when a copy of the curve is made and rotated, the drawing order of the copy is reversed, and each turn is inverted. The turn between the original curve and the copy is determined by the operation at that iteration, which is realized with `choose`. For instance, `dragonU [L, R, R]` outputs `[+, +, -, -, +, -, -]`, which is `dragonU [R, R] ++ choose L :: invert (reverse (dragonU [R, R]))` and corresponds to Fig. 1.

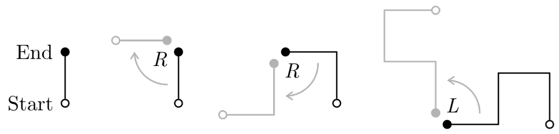

Figure 1: The unfolding construction

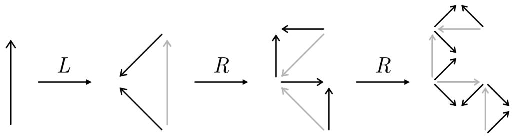

Figure 2: The expansion construction

Expansion The "expansion" construction starts with a line segment with a direction. At each iteration, each segment is expanded to its right (R) or left (L), creating two new segments. The three form a 45-45-90 triangle with the original segment being the hypotenuse, and the directions of the new segments facing each other, as demonstrated in Fig. 2.

The expansion construction can also be formalized by a recursive function that generates drawing instructions. More specifically, it can be described using an L-system, a type of system that iteratively replaces each letter in a list with a predefined sequence of letters. First, we extend the alphabet with X and Y. Then, we define the rules of the L-system: for R, the rules are $X \to X+Y$ and $Y \to X-Y$, and for L, the rules are $X \to X-Y$ and $Y \to X+Y$ (list notation omitted). Finally, we can define `dragonE` that takes an initial list and a list of operations.

```
dragonE : List Letter → List Op → List Letter
dragonE seed []          = seed
dragonE seed (op ∷ ops) = l-help op (dragonE seed ops)
```

where `l-help` implements the L-system [3, p. 11]. To interpret the output of `dragonE`, we can simply ignore the X and Y as they only serve the purpose of discriminating the direction of lines.

`dragonE [X]` represents the expansion construction because the alternating appearance of X and Y in the list corresponds to the alternating directions of segments in the curve. The direction of the segment determines whether a left or right turn will appear, similar to how X and Y determine the addition of $+$ and $-$ after expansion. For instance, `dragonE [X] [R, R, L]` outputs `X+Y+X-Y-X+Y-X-Y`, which is expanded from `X+Y-X-Y` under R. This corresponds to Fig. 2.

Construction Equivalence One property of the dragon curve is that when given the same sequence of operations, only in reverse order, the two constructions generate the same curve (as shown in Fig. 1 and 2). In Agda, we can express this theorem as follows:

```
E[X]≡U : ∀ (ops : List Op) → strip (dragonE [X] (reverse ops)) ≡ dragonU ops
```

where `strip` removes all X and Y. This theorem is proven by decomposing both sides and showing the equivalence on each component inductively. A complementary theorem describing the relation between `dragonE [Y]` and `dragonU` is used and proven together through mutual induction.

Non-Overlapping Property (NOP) Another property of the dragon curve is that, regardless of the sequence of operations, the generated curve never overlaps itself, formalized as follows:

```
no-overlap-in-dragonG : ∀ {cur : Curve} → DragonG cur → Non-OL-Curve cur
```

Here, `Curve` is defined as `List Seg` (a list of segments) on an integer coordinate plane, where each segment has its start and end points. `DragonG` uses the expansion construction to inductively construct the dragon curve as a `Curve`, while `Non-OL-Curve` requires proof that no two segments in the curve overlap each other. We attempt to follow the proof of this property mentioned in Ryde's work [1, p. 4] by defining a `Pattern` predicate on `Seg`. By showing that curves whose segments all follow the pattern preserve the NOP and that the dragon curve follows the pattern, we can inductively prove the NOP of the dragon curve. However, the proof is still incomplete.

Future Work Future work includes completing the proof for the non-overlapping property, generalizing the proof outline to include the NOPs of other fractal curves defined by L-systems or expansion of line segments, or further exploration of the relationships between the language for drawing instructions (`dragonU`, `dragonE`) and their graphical representations (`DragonG`).

References

[1] K. Ryde. *Iterations of the Dragon Curve*. Draft 23. 2021.

[2] S. Tabachnikov. "Dragon Curves Revisited". In: *The Mathematical Intelligencer* 36.1 (2014), pp. 13–17.

[3] P. Prusinkiewicz and A. Lindenmayer. *The Algorithmic Beauty of Plants*. Springer-Verlag, 1990.

Proof Sketch

Construction Equivalence We prove the construction equivalence:

```
E[X]≡U : ∀ (ops : List Op)
  → strip (dragonE [X] (reverse ops)) ≡ dragonU ops
```

inductively on the list of operations. For the base case, `E[X]≡U` holds trivially. For the inductive case, we first observe that the LHS `strip (dragonE [X] (reverse (op :: ops)))` can be decomposed into `A ++ (choose op) :: B`, where A is `strip (dragonE [X] (reverse ops))` and B is `strip (dragonE [Y] (reverse ops))`. We prove this decomposition by observing that `dragonE`, when given a list of operations `ops` $::^r$ `op` and an initial list `[X]`, produces `X :: (choose op) :: [Y]` after the first operation (the last item) is applied.

Similarly, the RHS `dragonU (op :: ops)` can also be decomposed into `C ++ (choose op) ++ D`, where C is `dragonC ops` and D is `invert (reverse (dragonC ops))` by definition.

$A \equiv C$ is trivial from the inductive hypothesis, while $B \equiv D$ requires another theorem:

```
E[Y]≡U : ∀ (ops : List Op)
  → strip (dragonE [Y] (reverse ops)) ≡ invert (reverse (dragonU ops))
```

`E[Y]≡U` is similar to `E[X]≡U` but is given `[Y]` instead of `[X]` as the initial list for `dragonE`. To prove `E[Y]≡U`, we take a similar approach to the proof of `E[X]≡U` by decomposing both sides of the equivalence and then showing the equivalence on each component. This process will, in turn, make use of `E[X]≡U`. In fact, `E[X]≡U` and `E[Y]≡U` are proven together through mutual induction.

Non-Overlapping Property We attempt to prove the NOP of dragon curves:

```
no-overlap-in-dragonG : ∀ {curve} → DragonG curve → Non-OL-Curve curve
```

where `DragonG` is defined inductively using the expansion construction, and `Non-OL-Curve` is defined as follows:

```
Non-OL-Curve : Curve → Set
Non-OL-Curve cur = ∀ i j → i ≢ j → ¬ Overlap (lookup cur i) (lookup cur j)
```

where `Overlap` is a predicate of whether two segments start and end at the same points, regardless of direction.

`DragonG` uses the `expand-curve` function to formalize the expansion construction. At each iteration, it first performs a 45-degree rotation and a scaling by $\sqrt{2}$ on the curve. It then expands each segment in the curve to its left or right side according to the inputted operation, as shown in Fig. 3. The purpose of the linear transformation before the expansion is to ensure that the

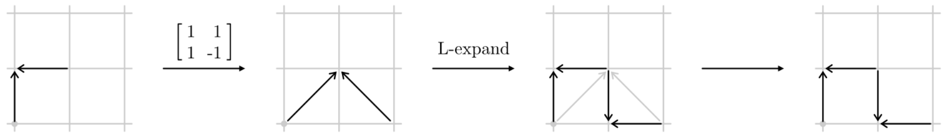

Figure 3: A curve expanded by the `expand-curve` function

endpoints of the segments remain integers. It is also proven that the transformation preserves the similarity between the original and resulting curves.

With the definitions, we attempt to prove the NOP of the dragon curve by observing that all the segments in the dragon curve seem to be in the form of

```
Pattern : Seg → Set
Pattern record {s = record {x = sx ; y = sy} ; e = record {x = ex ; y = ey}}
  = Even (sx + sy) × Odd (ex + ey) × ((ex - sx) ² + (ey - sy) ² ≡ 1)
```

If we can prove the following lemmas for this pattern:

```
pattern-preserve :   ∀ {cur} → ∀ op
  → All Pattern cur → All Pattern (expand-curve op cur)

NOC-preserve : ∀ {cur} → ∀ op
  → Non-OL-Curve cur → All Pattern cur → Non-OL-Curve (expand-curve op cur)

dragon-on-pattern : ∀ {cur} → DragonG cur → All Pattern cur
```

then the NOP can be proven inductively:

```
no-overlap-in-dragonG : ∀ {cur} → DragonG cur → Non-OL-Curve cur
```

Among the lemmas, `dragon-on-pattern` can be proven with `pattern-preserve` trivially. However, the proofs for `pattern-preserve` and `NOC-preserve` are still incomplete.