Well-Behaved (Co)algebraic Semantics of Regular Expressions in Dafny

Jan 12, 2024 • Stefan Zetzsche and Wojciech Różowski

Introduction

Regular expressions are one of the most ubiquitous formalisms of theoretical computer science. Commonly, they are understood in terms of their *denotational* semantics, that is, through formal languages — the *regular* languages. This view is *inductive* in nature: two primitives are equivalent if they are *constructed* in the same way. Alternatively, regular expressions can be understood in terms of their *operational* semantics, that is, through finite automata. This view is *coinductive* in nature: two primitives are equivalent if they are *deconstructed* in the same way. It is implied by Kleene's famous theorem that both views are equivalent: regular languages are precisely the formal languages accepted by finite automata. In this blogpost, we utilise Dafny's built-in inductive and coinductive reasoning capabilities to show that the two semantics of regular expressions are *well-behaved*, in the sense they are in fact one and the same, up to pointwise bisimulation.

Denotational Semantics

In this section, we define regular expressions and formal languages, introduce the concept of bisimilarity, formalise the *denotational* semantics of regular expressions as a function from regular expressions to formal languages, and prove that the latter is an algebra homomorphism.

Regular Expressions as Datatype

We define the set of regular expressions parametric in an alphabet `A` as an inductive `datatype`:

```
datatype Exp<A> = Zero | One | Char(A) | Plus(Exp, Exp) | Comp(Exp, Exp) | Star(Exp)
```

Note that above, and later, we make use of Dafny's type parameter completion.

The definition captures that a regular expression is either a primitive character `Char(a)`, a non-deterministic choice between two regular expressions `Plus(e1, e2)`, a sequential composition of two regular expressions `Comp(e1, e2)`, a finite number of self-iterations `Star(e)`, or one of the constants `Zero` (the unit of `Plus`) and `One` (the unit of `Comp`). At a higher level, the above defines `Exp<A>` as the *smallest* algebraic structure that is equipped with two constants, contains all elements of type `A`, and is closed under two binary operations and one unary operation.

Formal Languages as Codatatype

We define the set of formal languages parametric in an alphabet `A` as a coinductive `codatatype`:

```
codatatype Lang<!A> = Alpha(eps: bool, delta: A -> Lang<A>)
```

Note that we used the type-parameter mode `!`, which indicates that there could be strictly more values of type `Lang<A>` than values of type `A`, for any type `A`, and that there is no subtype relation between `Lang<A>` and `Lang<B>`, for any two types `A, B`. A more detailed explanation of the topic can be found here.

To some, the choice above might seem odd at first sight. If you are familiar with the topic, you likely have expected a formal language to be defined more concretely as a set of finite sequences (sometimes called *words*), `iset<seq<A>>`. Rest assured, we agree — up to an appropriate notion of equality! Whereas you characterise languages intrinsically, we treat them extrinsically, in terms of their universal property: it is well known that `iset<seq<A>>` forms the *greatest* coalgebraic structure (think of a deterministic automaton without initial state) `S` that is equipped with functions `eps: S -> bool` and `delta: S → (A → S)`. Indeed, for any set `U` of finite sequences, we can verify whether `U` contains the empty sequence, `U.eps == ([] in U)`, and for any `a: A`, we can transition to the set `U.delta(a) == (iset s | [a] + s in U)`. Here, we choose the more abstract perspective on formal languages as it hides irrelevant specifics and thus allows us to write more elegant proofs.

An Algebra of Formal Languages

If you think of formal languages as the set of all sets of finite sequences, you will soon realise that languages admit quite a bit of algebraic structure. For example, there exist two languages of distinct importance (can you already guess which ones?), and one can obtain a new language by taking e.g. the union of two languages. In fact, if you think about it for a bit longer, you'll realise that formal languages admit exactly the same type of algebraic structure as the one you've encountered when we defined regular expressions!

First, there exists the empty language `Zero()` that contains no words at all. Under the view above, we find `Zero().eps == false` and `Zero().delta(a) == Zero()`, since the empty set does not contain the empty sequence, and the derivative `iset s | [a] + s in iset{}` with respect to any `a: A` yields again the empty set, respectively. We thus define:

```
function Zero<A>(): Lang {
  Alpha(false, (a: A) => Zero())
}
```

Using similar reasoning, we additionally derive the following definitions. In order, we formalise i) the language `One()` that contains only the empty sequence; ii) for any `a: A` the language `Singleton(a)` that consists of only the word `[a]`; iii) the language `Plus(L1, L2)` which consists of the union of the languages `L1` and `L2`; iv) the language `Comp(L1, L2)` that consists of all possible concatenation of words in `L1` and `L2`; and v) the language `Star(L)` that consists of all finite compositions of `L` with itself. Our definitions match what is well-known as *Brzozowski derivatives*.

```
function One<A>(): Lang {
  Alpha(true, (a: A) => Zero())
}

function Singleton<A(==)>(a: A): Lang {
  Alpha(false, (b: A) => if a == b then One() else Zero())
}

function {:abstemious} Plus<A>(L1: Lang, L2: Lang): Lang {
  Alpha(L1.eps || L2.eps, (a: A) => Plus(L1.delta(a), L2.delta(a)))
}

function {:abstemious} Comp<A>(L1: Lang, L2: Lang): Lang {
  Alpha(
    L1.eps && L2.eps,
    (a: A) => Plus(Comp(L1.delta(a), L2), Comp(if L1.eps then One() else Zero(), L2.delta(a)
  )
}

function Star<A>(L: Lang): Lang {
  Alpha(true, (a: A) => Comp(L.delta(a), Star(L)))
}
```

Note that the `{:abstemious}` attribute above signals that a function does not need to unfold a codatatype instance very far (perhaps just one destructor call) to prove a relevant property. Knowing this is the case can aid the proofs of properties about the function. In this case, it is needed to convince Dafny that the corecursive calls in `Comp` and `Star` are logically consistent.

Denotational Semantics as Induced Morphism

The denotational semantics of regular expressions can now be defined through induction, as a function `Denotational: Exp -> Lang`, by making use of the operations on languages we have just defined:

```
function Denotational<A(==)>(e: Exp): Lang {
  match e
  case Zero => Languages1.Zero()
  case One => Languages2.One()
  case Char(a) => Languages2.Singleton(a)
  case Plus(e1, e2) => Languages2.Plus(Denotational(e1), Denotational(e2))
  case Comp(e1, e2) => Languages2.Comp(Denotational(e1), Denotational(e2))
  case Star(e1) => Languages2.Star(Denotational(e1))
}
```

Bisimilarity and Coinduction

Let us briefly introduce a notion of equality between formal languages that will be useful soon. A binary relation `R` between languages is called a *bisimulation*, if for any two languages `L1`, `L2` related by `R` the following holds: i) `L1` contains the empty word iff `L2` does; and ii) for any `a: A`, the derivatives `L1.delta(a)` and `L2.delta(a)` are again related by `R`. As it turns out, the union of two bisimulations is again a bisimulation. In consequence, one can combine all possible bisimulations into a single relation: the *greatest* bisimulation. In Dafny, we can formalise the latter as a `greatest predicate`:

```
greatest predicate Bisimilar<A(!new)>[nat](L1: Lang, L2: Lang) {
  && (L1.eps == L2.eps)
  && (forall a :: Bisimilar(L1.delta(a), L2.delta(a)))
}
```

It is instructive to think of a `greatest predicate` as pure syntactic sugar. Indeed, under the hood, Dafny's compiler uses the body above to implicitly generate i) for any `k: nat`, a *prefix predicate* `Bisimilar#[k](L1, L2)` that signifies that the languages `L1` and `L2` concur on the first `k`-unrollings of the definition above; and ii) a predicate `Bisimilar(L1, L2)` that is true iff `Bisimilar#[k](L1, L2)` is true for all `k: nat`:

```
/* Pseudo code for illustration purposes */

predicate Bisimilar<A(!new)>(L1: Lang, L2: Lang) {
  forall k: nat :: Bisimilar#[k](L1, L2)
}

predicate Bisimilar#<A(!new)>[k: nat](L1: Lang, L2: Lang)
  decreases k
{
  if k == 0 then
    true
  else
    && (L1.eps == L2.eps)
    && (forall a :: Bisimilar#[k-1](L1.delta(a), L2.delta(a)))
}
```

Now that we have its definition in place, let us establish a property about bisimilarity, say, that it is a *reflexive* relation. With the `greatest lemma` construct, Dafny is able to able to derive a proof completely on its own:

```
greatest lemma BisimilarityIsReflexive<A(!new)>[nat](L: Lang)
  ensures Bisimilar(L, L)
{}
```

Once again, it is instructive to think of a `greatest lemma` as pure syntactic sugar. Under the hood, Dafny's compiler uses the body of the `greatest lemma` to generate i) for any `k: nat`, a *prefix lemma* `BisimilarityIsReflexive#[k](L)` that ensures the prefix predicate `Bisimilar#[k](L, L)`; and ii) a lemma `BisimilarityIsReflexive(L)` that ensures `Bisimilar(L, L)` by calling `Bisimilar#[k](L, L)` for all `k: nat`:

```
/* Pseudo code for illustration purposes */

lemma BisimilarityIsReflexive<A(!new)>(L: Lang)
  ensures Bisimilar(L, L)
{
  forall k: nat
    ensures Bisimilar#[k](L, L)
  {
    BisimilarityIsReflexive#[k](L);
  }
}

lemma BisimilarityIsReflexive#<A(!new)>[k: nat](L: Lang)
  ensures Bisimilar#[k](L, L)
  decreases k
{
  if k == 0 {
  } else {
    forall a
      ensures Bisimilar#[k-1](L.delta(a), L.delta(a))
    {
      BisimilarityIsReflexive#[k-1](L.delta(a));
    }
  }
}
```

If you are interested in the full details, we recommend taking a look at this note on coinduction, predicates, and ordinals.

Denotational Semantics as Algebra Homomorphism

For a moment, consider the function `var f: nat -> nat := (n: nat) => n + n` which maps a natural number to twice its value. The function `f` is *structure-preserving*: for any `m: nat` we have `f(m * n) == m * f(n)`, i.e. `f` commutes with the (left-)multiplication of naturals. In this section, we are interested in functions of type `f: Exp -> Lang` (more precisely, in `Denotational: Exp -> Lang`) that commute with the algebraic structures we encountered in Regular Expressions as Datatype and An Algebra of Formal Languages, respectively. We call such structure-preserving functions *algebra homomorphisms*. To define pointwise commutativity in this context, we'll have to be able to compare languages for equality. As you probably guessed, bisimilarity will do the job:

```
ghost predicate IsAlgebraHomomorphism<A(!new)>(f: Exp -> Lang) {
  forall e :: IsAlgebraHomomorphismPointwise(f, e)
}

ghost predicate IsAlgebraHomomorphismPointwise<A(!new)>(f: Exp -> Lang, e: Exp) {
  Bisimilar<A>(
    f(e),
    match e
    case Zero => Languages1.Zero()
    case One => Languages2.One()
    case Char(a) => Languages2.Singleton(a)
    case Plus(e1, e2) => Languages2.Plus(f(e1), f(e2))
    case Comp(e1, e2) => Languages2.Comp(f(e1), f(e2))
    case Star(e1) => Languages2.Star(f(e1))
  )
}
```

Note that we used the `ghost` modifier (which signals that an entity is meant for specification only, not for compilation). A `greatest predicate` is always ghost, so `IsAlgebraHomomorphismPointwise` must be declared ghost to call `Bisimilar`, and `IsAlgebraHomomorphism` must be declared ghost to call `IsAlgebraHomomorphismPointwise`.

The proof that `Denotational` is an algebra homomorphism is straightforward: it essentially follows from bisimilarity being reflexive:

```
lemma DenotationalIsAlgebraHomomorphism<A(!new)>()
  ensures IsAlgebraHomomorphism<A>(Denotational)
{
  forall e
    ensures IsAlgebraHomomorphismPointwise<A>(Denotational, e)
  {
    BisimilarityIsReflexive<A>(Denotational(e));
  }
}
```

Operational Semantics

In this section, we provide an alternative perspective on the semantics of regular expressions. We equip the set of regular expressions with a coalgebraic structure, formalise its *operational* semantics as a function from regular expressions to formal languages, and prove that the latter is a coalgebra homomorphism.

A Coalgebra of Regular Expressions

In An Algebra of Formal Languages we equipped the set of formal languages with an algebraic structure that resembles the one of regular expressions. Now, we are aiming for the reverse: we would like to equip the set of regular expressions with a coalgebraic structure that resembles the one of formal languages. More concretely, we would like to turn the set of regular expressions into a deterministic automaton (without initial state) in which a state `e` is i) accepting iff `Eps(e) == true` and ii) transitions to a state `Delta(e)(a)` if given the input `a: A`. Note how our definitions resemble the Brzozowski derivatives we previously encountered:

```
function Eps<A>(e: Exp): bool {
  match e
  case Zero => false
  case One => true
  case Char(a) => false
  case Plus(e1, e2) => Eps(e1) || Eps(e2)
  case Comp(e1, e2) => Eps(e1) && Eps(e2)
  case Star(e1) => true
}

function Delta<A(==)>(e: Exp): A -> Exp {
  (a: A) =>
    match e
    case Zero => Zero
    case One => Zero
    case Char(b) => if a == b then One else Zero
    case Plus(e1, e2) => Plus(Delta(e1)(a), Delta(e2)(a))
    case Comp(e1, e2) => Plus(Comp(Delta(e1)(a), e2), Comp(if Eps(e1) then One else Zero, De
    case Star(e1) => Comp(Delta(e1)(a), Star(e1))
}
```

Operational Semantics as Induced Morphism

The operational semantics of regular expressions can now be defined via coinduction, as a function `Operational: Exp -> Lang`, by making use of the coalgebraic structure on expressions we have just defined:

```
function Operational<A(==)>(e: Exp): Lang {
  Alpha(Eps(e), (a: A) => Operational(Delta(e)(a)))
}
```

Operational Semantics as Coalgebra Homomorphism

In Denotational Semantics as Algebra Homomorphism we defined algebra homomorphisms as functions `f: Exp -> Lang` that commute with the algebraic structures of regular expressions and formal languages, respectively. Analogously, let us now call a function of the same type a *coalgebra homomorphism*, if it commutes with the *coalgebraic* structures of regular expressions and formal languages, respectively:

```
ghost predicate IsCoalgebraHomomorphism<A(!new)>(f: Exp -> Lang) {
  && (forall e :: f(e).eps == Eps(e))
  && (forall e, a :: Bisimilar(f(e).delta(a), f(Delta(e)(a))))
}
```

It is straightforward to prove that `Operational` is a coalgebra homomorphism: once again, the central argument is that bisimilarity is a reflexive relation.

```
lemma OperationalIsCoalgebraHomomorphism<A(!new)>()
  ensures IsCoalgebraHomomorphism<A>(Operational)
{
  forall e, a
    ensures Bisimilar<A>(Operational(e).delta(a), Operational(Delta(e)(a)))
  {
    BisimilarityIsReflexive(Operational(e).delta(a));
  }
}
```

Well-Behaved Semantics

So far, we have seen two dual approaches for assigning a formal language semantics to regular expressions:

- `Denotational`: an algebra homomorphism obtained via induction
- `Operational`: a coalgebra homomorphism obtained via coinduction

Next, we show that the denotational and operational semantics of regular expressions are *well-behaved*: they constitute two sides of the same coin. First, we show that `Denotational` is also a coalgebra homomorphism, and that coalgebra homomorphisms are unique up to bisimulation. We then deduce from the former that `Denotational` and `Operational` coincide pointwise, up to bisimulation. Finally, we show that `Operational` is also an algebra homomorphism.

Denotational Semantics as Coalgebra Homomorphism

In this section, we establish that `Denotational` not only commutes with the algebraic structures of regular expressions and formal languages, but also with their coalgebraic structures:

```
lemma DenotationalIsCoalgebraHomomorphism<A(!new)>()
  ensures IsCoalgebraHomomorphism<A>(Denotational)
```

The proof of the lemma is a bit more elaborate than the ones we have encountered so far. It can be divided into two subproofs, both of which make use of induction. One of the subproofs is straightforward, the other, more difficult one, again uses the reflexivity of bisimilarity, but also that the latter is a *congruence* relation with respect to `Plus` and `Comp`:

```
greatest lemma PlusCongruence<A(!new)>[nat](L1a: Lang, L1b: Lang, L2a: Lang, L2b: Lang)
  requires Bisimilar(L1a, L1b)
  requires Bisimilar(L2a, L2b)
  ensures Bisimilar(Plus(L1a, L2a), Plus(L1b, L2b))
{}

lemma CompCongruence<A(!new)>(L1a: Lang, L1b: Lang, L2a: Lang, L2b: Lang)
  requires Bisimilar(L1a, L1b)
  requires Bisimilar(L2a, L2b)
  ensures Bisimilar(Comp(L1a, L2a), Comp(L1b, L2b))
```

Dafny is able to prove `PlusCongruence` on its own, as it can take advantage of the syntactic sugaring of the `greatest lemma` construct. For `CompCongruence` we have to put in a bit of manual work ourselves.

Coalgebra Homomorphisms Are Unique

The aim of this section is to show that, up to pointwise bisimilarity, there only exists *one* coalgebra homomorphism of type `Exp -> Lang`:

```
lemma UniqueCoalgebraHomomorphism<A(!new)>(f: Exp -> Lang, g: Exp -> Lang, e: Exp)
  requires IsCoalgebraHomomorphism(f)
  requires IsCoalgebraHomomorphism(g)
  ensures Bisimilar(f(e), g(e))
```

As is well-known, the statement may in fact be strengthened to: for *any* coalgebra `C` there exists exactly one coalgebra homomorphism of type `C -> Lang`, up to pointwise bisimulation. For our purposes, the weaker statement above will be sufficient. At the heart of the proof lies the observation that bisimilarity is transitive:

```
greatest lemma BisimilarityIsTransitive<A>[nat](L1: Lang, L2: Lang, L3: Lang)
  requires Bisimilar(L1, L2) && Bisimilar(L2, L3)
  ensures Bisimilar(L1, L3)
{}
```

In fact, in practice, we actually use a slightly more fine grained formalisation of transitivity, as is illustrated below by the proof of `UniqueCoalgebraHomomorphismHelperPointwise`, which is used in the proof of `UniqueCoalgebraHomomorphism`:

```
lemma UniqueCoalgebraHomomorphismHelperPointwise<A(!new)>(k: nat, f: Exp -> Lang, g: Exp ->
  requires IsCoalgebraHomomorphism(f)
  requires IsCoalgebraHomomorphism(g)
  requires exists e :: Bisimilar#[k](L1, f(e)) && Bisimilar#[k](L2, g(e))
  ensures Bisimilar#[k](L1, L2)
{
  var e :| Bisimilar#[k](L1, f(e)) && Bisimilar#[k](L2, g(e));
  if k != 0 {
    forall a
      ensures Bisimilar#[k-1](L1.delta(a), L2.delta(a))
    {
      BisimilarityIsTransitivePointwise(k-1, L1.delta(a),  f(e).delta(a), f(Delta(e)(a)));
      BisimilarityIsTransitivePointwise(k-1, L2.delta(a),  g(e).delta(a), g(Delta(e)(a)));
      UniqueCoalgebraHomomorphismHelperPointwise(k-1, f, g, L1.delta(a), L2.delta(a));
    }
  }
}

lemma BisimilarityIsTransitivePointwise<A(!new)>(k: nat, L1: Lang, L2: Lang, L3: Lang)
  ensures Bisimilar#[k](L1, L2) && Bisimilar#[k](L2, L3) ==> Bisimilar#[k](L1, L3)
{
  if k != 0 {
    if Bisimilar#[k](L1, L2) && Bisimilar#[k](L2, L3) {
      assert Bisimilar#[k](L1, L3) by {
        forall a
          ensures Bisimilar#[k-1](L1.delta(a), L3.delta(a))
        {
          BisimilarityIsTransitivePointwise(k-1, L1.delta(a), L2.delta(a), L3.delta(a));
        }
      }
    }
  }
}
```

Denotational and Operational Semantics Are Bisimilar

We are done! From the previous results, we can immediately deduce that denotational and operational semantics are the same, up to pointwise bisimilarity:

```
lemma OperationalAndDenotationalAreBisimilar<A(!new)>(e: Exp)
  ensures Bisimilar<A>(Operational(e), Denotational(e))
{
  OperationalIsCoalgebraHomomorphism<A>();
  DenotationalIsCoalgebraHomomorphism<A>();
  UniqueCoalgebraHomomorphism<A>(Operational, Denotational, e);
}
```

Operational Semantics as Algebra Homomorphism

As a bonus, for the sake of symmetry, let us also prove that `Operational` is an algebra homomorphism. (We already know that it is a coalgebra homomorphism, and that `Denotational` is both an algebra and coalgebra homomorphism.)

```
lemma OperationalIsAlgebraHomomorphism<A(!new)>()
  ensures IsAlgebraHomomorphism<A>(Operational)
```

The idea of the proof is to take advantage of `Denotational` being an algebra homomorphism, by translating its properties to `Operational` via the lemma in Denotational and Operational Semantics Are Bisimilar. The relevant new statements capture that bisimilarity is symmetric and a congruence with respect to the `Star` operation:

```
greatest lemma BisimilarityIsSymmetric<A(!new)>[nat](L1: Lang, L2: Lang)
  ensures Bisimilar(L1, L2) ==> Bisimilar(L2, L1)
  ensures Bisimilar(L1, L2) <== Bisimilar(L2, L1)
{}

lemma StarCongruence<A(!new)>(L1: Lang, L2: Lang)
  requires Bisimilar(L1, L2)
  ensures Bisimilar(Star(L1), Star(L2))
```

Conclusion

We have used Dafny's built-in inductive and coinductive reasoning capabilities to define two language semantics for regular expressions: denotational and operational semantics. Through a number of dualities — construction and deconstruction, algebras and coalgebras, and congruence and bisimilarity — we have proven the semantics to be two sides of the same coin. The blogpost is inspired by research in the field of Coalgebra, which was pioneered by Rutten, Gumm, and others. The concept of well-behaved semantics goes back to Turi and Plotkin and was adapted by Jacobs to the case of regular expressions. We heavily used automata theoretic constructions from the 1960s, originally investigated by Brzozowski (a more modern presentation can be found e.g. here). Our presentation focused on the most important intuitive aspects of the proofs. To dive deep, please take a look at the full Dafny source code, which is available here, here, and here.