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

Stefan Zetzsche[1] and Wojciech Różowski[2]

[1] Amazon Web Services, United Kingdom
stefanze@amazon.co.uk
[2] University College London, United Kingdom
w.rozowski@cs.ucl.ac.uk

Abstract. Regular expressions are commonly 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 deterministic 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 deterministic finite automata. In this paper, we use Dafny, a verification-aware programming language, to formally verify, for the first time, what has been previously established only through proofs-by-hand: the two semantics of regular expressions are well-behaved, in the sense that they are in fact one and the same, up to pointwise bisimilarity. At each step of our formalisation, we propose an interpretation in the language of Coalgebra. We found that Dafny is particularly well suited for the task due to its inductive and coinductive features and hope our approach serves as a blueprint for future generalisations to other theories.

Keywords: Coalgebra · Dafny · Regular Expressions · Semantics

1 Introduction

Regular expressions stand as one of the most ubiquitous formalisms in all of theoretical computer science. Their inception can be traced back all the way to Kleene's seminal paper in 1951 [25]. Today, they play a pivotal role as a foundational element for a wide spectrum of applications [18, 40, 7], encompassing text searching, pattern matching, lexical analysis, and more.

Typically, regular expressions are understood denotationally, through the formal languages, that is, sets of finite words over a fixed alphabet, that they denote. This view is inductive in nature, in the sense that the denotational semantics of regular expressions is constructed from the bottom-up by following the finite inductive structure of an expression.

Alternatively, regular expressions can be understood operationally, through the lenses of deterministic finite automata. This view is coinductive in nature, in

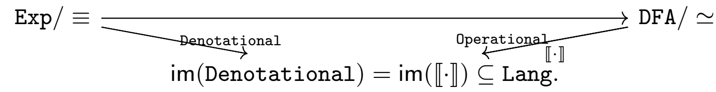

Fig. 1: The triptych of regular expressions, deterministic finite automata, and regular languages.

the sense that the operational semantics of regular expressions is assigned from the top-down, by deconstructing an expression, following the coinductive nature of a potentially infinite language initially observed by Brzozowski [13].

One of Kleene's many contributions was to show that the denotational and operational semantics of regular expressions are two sides of the same coin – they are *well-behaved*. That is, the denotational interpretation of a regular expression matches exactly the observable behaviour of its operational interpretation. Kleene's theorem is of great practical significance, too: for any given regular expression that represents a text pattern, one can derive, in a canonical way, an automaton that gives rise to a deterministic algorithm that decides, in finite time, whether some given string matches the text pattern specified by the expression [22, 21]. The full triptych of regular expressions, deterministic finite automata, and regular languages is depicted in Figure 1: the set of regular expressions modulo the axioms of *Kleene Algebra* [26] ($\equiv$), the set of deterministic finite automata modulo behavioural equivalence ($\simeq$), and the set of regular languages are in bijection. The composition on the right-hand side of the diagram can be seen as a function that assigns the operational semantics to an expression.

In more recent years, a more general approach to automata through the lenses of category theory has become popular: state-based systems are generalised as *coalgebras* over an endofunctor [44, 19, 24]. There are many advantages to using the coalgebraic abstraction of state-based systems. Among others, it allows one to set aside irrelevant specifics of concrete instantiations, and instead work with elegant, universal properties. Of particular interest for us are systems that have both an algebraic (inductive) and a coalgebraic (coinductive) component.

In this paper, we use the built-in inductive and coinductive reasoning capabilities of *Dafny* [3], a programming language and static verifier, to formalise the denotational and operational semantics of regular expressions and formally prove that they are well-behaved, that is, coincide pointwise, up to bisimilarity[3]. Dafny is a statically typed programming language with native support for writing and verifying specifications about programs that was first developed by Leino at Microsoft Research [4, 32]. Dafny combines various paradigms such as imperative, functional, and object-oriented programming and supports common programming concepts such as inductive datatypes, immutable and mutable data structures, lambda functions, and subset types. Dafny can be integrated with common software development IDEs such as VSCode and emacs. It has been used in academia for research and the teaching of program verification [39], as

[3] The full Dafny source code is available at [51].

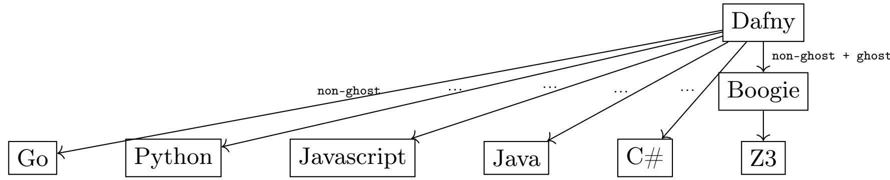

Fig. 2: The compilation pipeline of Dafny.

well as in industry by e.g. Amazon [1], ConsenSys [14], and Intel [50]. Teaching material is available online [31, 48] and in print [36]. A blog covers various aspects of the Dafny ecosystem [2].

One of the features of Dafny is that it allows the clear distinction between an idealised mathematical specification and an efficient implementation thereof. As a first example, consider the following purely functional specification of the Fibonacci sequence:

```
function Fib(n: nat): nat {
  if n <= 1 then n else Fib(n - 1) + Fib(n - 2)
}
```

While elegant in its recursive definition, `Fib` is not particularly efficient. A more realistic implementation is given by the imperative method `ComputeFib` below:

```
method ComputeFib(n: nat) returns (b: nat)
  ensures b == Fib(n)
{
  var c := 1;
  b := 0;
  for i := 0 to n
    invariant b == Fib(i) && c == Fib(i + 1)
  {
    b, c := c, b + c;
  }
}
```

Dafny allows us to extend the method signature with an `ensures` clause, which in this case indicates that the outputs of `Fib` and `ComputeFib` coincide on all possible inputs. To aid Dafny with proving the correctness of the `ensures` clause, we have to identify an appropriate `invariant` of the `for` loop in the body of `ComputeFib`. While not displayed, `ensures` admits a dual, the `requires` clause, which is used to restrict the domain of functions and methods to a subset. The two clauses are best thought of in terms of pre- and postconditions in the spirit of Hoare triples [20].

As illustrated above, Dafny programs contain both so-called *ghost* and *non-ghost* parts. Ghost code is meant for the specification of the behaviour of functions and the proof thereof only, not for compilation. Functions, methods, and variables can be marked ghost with a designated keyword. A method that is ghost and doesn't modify the heap is called a lemma. Pre- and postconditions, assertions, and loop invariants are always considered ghost. The Dafny verifier translates the ghost and non-ghost parts of a Dafny program into a program in

the intermediate verification language Boogie [8], such that the correctness of the output program implies the correctness of the input program. To verify the correctness of a Boogie program, a verification condition is generated from it and passed to the SMT solver Z3 [16] (Figure 2). Besides Dafny, there are other verification-aware languages built on top of Boogie and Z3 (e.g. VCC [15] and Spec# [9]). The non-ghost part of a verified Dafny program can be compiled to C#, Java, Javascript, Python, and Go, making possible the integration of verified code with an existing code base (Figure 2).

We found that Dafny is particularly well suited for the task and would like our approach to serve as a blueprint for future generalisations to other theories. At each step of our formalisation, we propose an interpretation in the language of Coalgebra. We hope that the presentation is accessible both for readers that are familiar with Coalgebra but not so much with Dafny, and for readers unexposed to Coalgebra, but experienced in Dafny.

In detail, the paper makes the following contributions:

- We formalise regular expressions as an inductive datatype (Section 2.1) and formal languages as a coinductive codatatype (Section 2.2). We introduce the concept of bisimilarity of languages (Section 2.5). We equip languages with an algebraic structure (Section 2.3) and in consequence define the denotational semantics of regular expressions as an induced function from regular expressions to formal languages (Section 2.4). Finally, we prove that the latter preserves algebraic structures up to pointwise bisimilarity (Section 2.6). At each step, we propose an interpretation in the language of Coalgebra.
- We equip the set of regular expressions with a coalgebraic structure of the type of unpointed deterministic automata (Section 3.1). We then formalise the operational semantics of regular expressions as an induced function from regular expressions to formal languages (Section 3.2). Finally, we prove that the latter preserves coalgebraic structures (Section 3.3).
- We show that the function that formalises the denotational semantics is also a coalgebra homomorphism (Section 4.1), and that coalgebra homomorphisms are unique up to pointwise bisimilarity (Section 4.2). We deduce that the denotational and operational semantics coincide, up to pointwise bisimilarity (Section 4.3). Finally, we show that the function that formalises the operational semantics is also an algebra homomorphism (Section 4.4).

2 Denotational Semantics

In this section, we define, in Dafny, 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.

2.1 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 [33], which allows us to write `Exp` instead of `Exp<A>`.

The definition above 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. Even more abstractly, `Exp<A>` can be viewed as the initial algebra for the set endofunctor Σ defined on objects by $\Sigma X = 1 + 1 + A + X^2 + X^2 + X$.

2.2 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 Dafny's 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 is available at [35].

To some, our way of modelling formal languages might seem odd at first sight. Typically, a formal language is defined intrinsically, as a set of finite sequences, that is, an element of type $\mathcal{P}(A^*)$ or `iset<seq<A>>` in Dafny. In our approach, we instead treat languages extrinsically, in terms of their universal property: it is well known that `iset<seq<A>>` forms the greatest abstract coalgebraic structure `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 derivative `U.delta(a) == (iset s | [a] + s in U)`. In the language of Coalgebra, `Lang<!A>` can be modelled as the final coalgebra for the set endofunctor B defined on objects by $BX = 2 \times X^A$ [44]. Coalgebras for the functor B correspond precisely to unpointed deterministic automata, and the final object among them provides a universal semantic domain for their behaviour.

We choose the more abstract perspective on formal languages as it hides irrelevant specifics and thus allows us to write more elegant proofs. With this decision, we follow a coalgebraic characterisation of formal languages in Isabelle [47], but depart from e.g. previous formalisations in Coq [38].

2.3 An Algebra of Formal Languages

If one thinks of a formal language as a set of finite sequences, one will soon realise that languages admit quite a bit of algebraic structure. In fact, it becomes clear

that formal languages can be equipped with the same type of algebraic structure as regular expressions.

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

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

Using similar reasoning, we additionally 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* [13]:

```
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 the use of the equality-supporting type parameter `==` in the definition of `Singleton`, which restricts the use of the function to types `A` that are known to support run-time equality comparisons (all types support equality in static contexts). In this case, the restriction is needed to ensure the well-definedness of the expression `a == b` in the definition of `Singleton`.

Also 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 in proofs of the properties of the function. In this case, it is needed to convince Dafny that the corecursive calls in `Comp` and `Star` are logically consistent.

In the language of Coalgebra, the above is best described by us equipping `Lang` with an algebra structure for the functor Σ. To derive a function such as

$$\begin{array}{ccc} \Sigma(\texttt{Exp}) & \xrightarrow{\Sigma(\texttt{Denotational})} & \Sigma(\texttt{Lang}) \\ \downarrow & & \downarrow[\texttt{Zero,One,Singleton,Plus,Comp,Star}] \\ \texttt{Exp} & \xrightarrow{\texttt{Denotational}} & \texttt{Lang} \end{array} \qquad \begin{array}{ccc} \texttt{Exp} & \xrightarrow{\texttt{Operational}} & \texttt{Lang} \\ \downarrow\langle\texttt{Eps,Delta}\rangle & & \downarrow \\ B(\texttt{Exp}) & \xrightarrow{B(\texttt{Operational})} & B(\texttt{Lang}) \end{array}$$

Fig. 3: `Denotational` and `Operational` as induced unique Σ-algebra and B-coalgebra homomorphisms, respectively.

e.g. `Comp` above, one gives the product `(Lang, Lang)` an appropriate B-coalgebra structure and deduces a unique morphism `(Lang, Lang) -> Lang` from the finality of `Lang` as B-coalgebra [44].

2.4 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 in Section 2.3. For the sake of clarity, we encapsulate those in a module named `Languages`:

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

The high-level view through the lenses of Coalgebra is depicted in Figure 3. By the initiality of `Exp` as algebra for Σ, there exists a unique morphism `Denotational: Exp -> Lang` that commutes with the algebraic structures (we formally prove the latter in Dafny in Section 2.6).

2.5 Bisimilarity and Coinduction

Let us briefly recall the notion of bisimilarity of formal languages. A binary relation `R` between languages is called *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. Two languages are called bisimilar if they are related by this greatest bisimulation. In Dafny, we can formalise the latter as follows:

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

Note that we used Dafny's type-parameter mode `!new`, which restricts the use of `Bisimilar` to values of type `A` that are not heap-based. This is necessary since a `forall` expression involved in a `greatest predicate` definition is not allowed to depend on the set of allocated references.

Two languages that are equal are also bisimilar, but the reverse is not necessarily true, since there is no extensional equality for functions in Dafny.

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)>[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)))
}

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

Note the use of the `decreases` clause in the definition of `Bisimilar#[k](L1, L2)`. Dafny requires us to convince it that all functions terminate. A `decreases` clause is used to support the proof of termination of a function in the presence of recursion. At each recursive call to a function, Dafny checks that the `decreases` clause is strictly smaller than the one of its caller with respect to a built-in well-founded order. In this case, Dafny verifies the inequality `k-1 < k` with respect to the natural well-founded order `<` of `nat`.

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 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 `BisimilarityIsReflexive` above to generate i) for any `k: nat`, a *prefix lemma* `BisimilarityIsReflexive#[k](L)` that ensures that the prefix predicate `Bisimilar#[k](L, L)` is satisfied; and ii) a lemma `BisimilarityIsReflexive(L)` that ensures that `Bisimilar(L, L)` is true by calling `BisimilarityIsReflexive#[k](L, L)` for all `k: nat`:

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

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));
    }
  }
}

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

We refer the reader interested in further details about Dafny's take on coinduction, predicates, and ordinals to [37].

2.6 Denotational Semantics as Algebra Homomorphism

In this section, we are interested in homomorphisms of type `f: Exp -> Lang` (more precisely, in `Denotational`), that is, functions which commute, up to bisimilarity, with the algebra structures we encountered in Section 2.1 and Section 2.3, respectively. In Dafny, we call such functions simply algebra homomorphisms. We define pointwise commutativity by comparing languages for bisimilarity:

```
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 => Languages.Zero()
    case One => Languages.One()
    case Char(a) => Languages.Singleton(a)
    case Plus(e1, e2) => Languages.Plus(f(e1), f(e2))
    case Comp(e1, e2) => Languages.Comp(f(e1), f(e2))
    case Star(e1) => Languages.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 implicitly `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));
  }
}
```

3 Operational Semantics

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

3.1 A Coalgebra of Regular Expressions

In Section 2.3 we equipped the set of formal languages with an algebraic structure that resembled the one of regular expressions. Now, we are aiming for the dual: 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 B-coalgebra, that is, 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:

```
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, Delta(e2)(a)))
    case Star(e1) => Comp(Delta(e1)(a), Star(e1))
}
```

3.2 Operational Semantics as Induced Morphism

The operational semantics of regular expressions can now in Dafny be defined via coinduction, as a function `Operational: Exp -> Lang`, by making use of the coalgebraic structure on expressions for the functor B we have just defined in Section 3.1:

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

The high-level view through the lenses of Coalgebra is depicted in Figure 3. By the finality of `Lang` as B-coalgebra, there exists a unique morphism `Operational: Exp -> Lang` that commutes with the B-coalgebra structures (the latter is formally proven in Dafny in Section 3.3).

3.3 Operational Semantics as Coalgebra Homomorphism

In Section 2.6 we defined in Dafny algebra homomorphisms as functions of type `Exp -> Lang` that commute, up to bisimilarity, with the Σ-algebra structures of regular expressions and formal languages, respectively. Analogously, we now define a function of the same type as coalgebra homomorphism, if it commutes, up to pointwise bisimilarity, with the B-coalgebra 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 formally prove that `Operational` is a coalgebra homomorphism in the above sense: 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));
  }
}
```

4 Well-Behaved Semantics

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

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

Next, we show in Dafny that the denotational and operational semantics of regular expressions are *well-behaved* (a term we adapt from [49]): 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 bisimilarity. We then deduce from the former that `Denotational` and `Operational` coincide pointwise, up to bisimilarity. Finally, we show that `Operational` is also an algebra homomorphism.

4.1 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.

4.2 Coalgebra Homomorphisms Are Unique

The aim of this section is to show in Dafny 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))
```

Of course, the perspective of Coalgebra suggests that 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 bisimilarity. 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(!new)>[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, as is illustrated below by the call to `BisimilarityIsTransitivePointwise` in the

proof of `UniqueCoalgebraHomomorphismHelperPointwise`, which in turn is used to prove `UniqueCoalgebraHomomorphism`:

```
lemma UniqueCoalgebraHomomorphismHelperPointwise<A(!new)>
  (k: nat, f: Exp -> Lang, g: Exp -> Lang, L1: Lang, L2: Lang)
  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));
        }
}}}}
```

Note the use of Dafny's let-such-that assignment `:|` in the body of the lemma `UniqueCoalgebraHomomorphismHelperPointwise`. For any predicate `P`, the expression `x :| P(x)` assigns a value to `x` such that `P(x)` is true. The predicate `P` needs to be non-empty, but in a ghost context doesn't have to constrain `x` uniquely: the choice of the latter is non-deterministic. To be compilable, the value of a let-such-that expression must be uniquely determined, however. In this case, the precondition of the lemma guarantees that the predicate is non-empty. For further background, we refer the reader to the Dafny Power User note [34].

4.3 Denotational and Operational Semantics Are Bisimilar

From the previous results, we can immediately deduce our main claim that denotational and operational semantics coincide, 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);
}
```

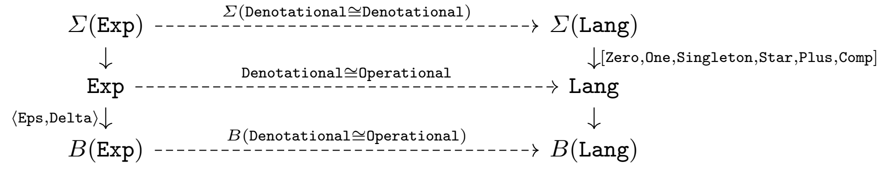

Fig. 4: The `Denotational` and `Operational` semantics of regular expressions are well-behaved.

4.4 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 Section 4.3. 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))
```

The full picture is depicted in Figure 4: `Denotational` is induced by the initiality of `Exp` and `Operational` is induced by the finality of `Lang`. By uniqueness, the two homomorphisms coincide, up to pointwise bisimilarity – the semantics of regular expressions are well-behaved.

5 Discussion and Related Work

We have used Dafny's built-in inductive and coinductive reasoning capabilities to define the denotational and operational semantics of regular expressions and to prove that they are well-behaved. The concept of well-behaved semantics, in the context of bialgebras (which consist of an algebra and a coalgebra over the same carrier that interact with each other in a suitable way), goes back to Turi and Plotkin [49] and was adapted by Jacobs to the case of regular expressions [23]. The bialgebraic perspective on regular expressions can be thought of as a generalisation of the classical automata-theoretic construction from Brzozowski in the 1960s [13]. A more modern presentation can be found in e.g. [45]. The

coalgebraic aspects build on results by Rutten [44], Gumm [19], and others [24]. As our presentation focused on the most important and high-level aspects of the proofs in Dafny, we invite the interested reader to take a look at the full Dafny source code [51].

The work closest to ours in spirit is [47], in which the authors use Isabelle [41], an LCF-style interactive theorem prover, to prove that formal languages represented as a coinductively defined trie satisfy the axioms of *Kleene Algebra* (KA) [26]. As for the differences, the authors of [47] don't touch on the aspect of well-behaved semantics, and we leave a formal proof of the axioms of KA in Dafny as future work. For the latter, because of the interplay between algebraic and coalgebraic structures, we plan to employ so-called up-to-techniques [43], which allow for compact coinductive proofs by making use of the underlying algebraic structure. Further related to this paper and [47] are [11, 12], in which the implementation of corecursion in Isabelle is discussed.

We depart from other work [38, 30, 42], which models formal languages intrinsically, as sets of words, and consequently begins by equipping the set of languages with an appropriate coalgebraic structure, whereas our extrinsic treatment in Dafny essentially axiomatises the latter.

6 Future Work

Besides proving in Dafny the soundness of Kleene Algebra axioms for coalgebraically defined languages, we are mainly interested in adapting our formalisation of the semantics of regular expressions and the proof of their well-behavedness to other theories.

An immediate target is *Kleene Algebra with Tests* (KAT) [27], which extends the theory of regular expressions, Kleene Algebra, with so-called *tests* (elements of a finitely generated Boolean Algebra). KAT can be used to reason about the equivalence of uninterpreted imperative programs, with tests used to model program guards. The theory has been successfully applied to program schematology [6] and has been used to reason about compiler optimizations [29] and cache control [10]. KAT admits both denotational semantics, through so-called guarded string languages, and operational semantics, through so-called automata on guarded strings [28].

There are more natural targets since KAT has been further extended in multiple directions. One such example is *Guarded Kleene Algebra with Tests* (GKAT) [46], an efficiently decidable fragment of KAT that admits operational semantics through strictly deterministic automata on guarded strings [46]. Another example is *NetKAT* [5], which extends KAT with primitives that allow the reasoning about Software Defined Networks. The verification of properties of such networks can be reduced to deciding the equivalence of NetKAT expressions, which in turn relies on their operational semantics [17].

Overall, we hope that the present formalisation both illustrates Dafny's potential for coalgebraic reasoning and serves as a blueprint for further adaption.

Acknowledgments. The authors are thankful to Aaron Tomb and Rustan Leino for their comments on an earlier version of this paper.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Aws encryption sdk for dafny, https://github.com/aws/aws-encryption-sdk-dafny
2. Dafny blog, https://dafny.org/blog/
3. The dafny programming and verification language, https://dafny.org/
4. Microsoft research, https://www.microsoft.com/en-us/research/
5. Anderson, C.J., Foster, N., Guha, A., Jeannin, J., Kozen, D., Schlesinger, C., Walker, D.: Netkat: semantic foundations for networks. In: The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014. pp. 113–126. ACM (2014). https://doi.org/10.1145/2535838.2535862
6. Angus, A., Kozen, D.: Kleene algebra with tests and program schematology. Tech. rep., Cornell University (2002)
7. Ausaf, F., Dyckhoff, R., Urban, C.: POSIX lexing with derivatives of regular expressions (proof pearl). In: Interactive Theorem Proving - 7th International Conference, ITP 2016, Nancy, France, August 22-25, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9807, pp. 69–86. Springer (2016). https://doi.org/10.1007/978-3-319-43144-4_5
8. Barnett, M., Chang, B.Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures 4. pp. 364–387. Springer (2006)
9. Barnett, M., Fähndrich, M., Leino, K.R.M., Müller, P., Schulte, W., Venter, H.: Specification and verification: the spec# experience. Communications of the ACM **54**(6), 81–91 (2011)
10. Barth, A., Kozen, D.: Equational verification of cache blocking in lu decomposition using kleene algebra with tests. Tech. rep., Cornell University (2002)
11. Blanchette, J.C., Bouzy, A., Lochbihler, A., Popescu, A., Traytel, D.: Friends with benefits: Implementing corecursion in foundational proof assistants. In: Programming Languages and Systems: 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22–29, 2017, Proceedings 26. pp. 111–140. Springer (2017)
12. Blanchette, J.C., Popescu, A., Traytel, D.: Foundational extensible corecursion: a proof assistant perspective. In: Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming. pp. 192–204 (2015)
13. Brzozowski, J.A.: Derivatives of regular expressions. Journal of the ACM (JACM) **11**(4), 481–494 (1964)
14. Cassez, F., Fuller, J., Ghale, M.K., Pearce, D.J., Quiles, H.M.: Formal and executable semantics of the ethereum virtual machine in dafny. In: International Symposium on Formal Methods. pp. 571–583. Springer (2023)

15. Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: Vcc: A practical system for verifying concurrent c. In: Theorem Proving in Higher Order Logics: 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings 22. pp. 23–42. Springer (2009)
16. De Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: International conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. Springer (2008)
17. Foster, N., Kozen, D., Milano, M., Silva, A., Thompson, L.: A coalgebraic decision procedure for netkat. In: Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015. pp. 343–355. ACM (2015). https://doi.org/10.1145/2676726.2677011
18. Friedl, J.E.F.: Mastering Regular Expressions. O'Reilly Media, Sebastopol, CA, 3 edn. (Aug 2006)
19. Gumm, H.P.: Elements of the general theory of coalgebras (2000)
20. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969). https://doi.org/10.1145/363235.363259
21. Holzer, M., Kutrib, M.: The complexity of regular(-like) expressions. Int. J. Found. Comput. Sci. **22**(7), 1533–1548 (2011). https://doi.org/10.1142/S0129054111008866
22. Hopcroft, J.E., Karp, R.M.: A linear algorithm for testing equivalence of finite automata. (1971), https://api.semanticscholar.org/CorpusID:120207847
23. Jacobs, B.: A bialgebraic review of deterministic automata, regular expressions and languages. In: Algebra, Meaning, and Computation, Essays Dedicated to Joseph A. Goguen on the Occasion of His 65th Birthday. Lecture Notes in Computer Science, vol. 4060, pp. 375–404. Springer (2006). https://doi.org/10.1007/11780274_20
24. Jacobs, B., Silva, A., Sokolova, A.: Trace semantics via determinization. In: International Workshop on Coalgebraic Methods in Computer Science. pp. 109–129. Springer (2012)
25. Kleene, S.: Representation of events in nerve nets and finite automata. Automata studies **3** (1951)
26. Kozen, D.: A completeness theorem for kleene algebras and the algebra of regular events. Inf. Comput. **110**(2), 366–390 (1994). https://doi.org/10.1006/INCO.1994.1037
27. Kozen, D.: Kleene algebra with tests. ACM Trans. Program. Lang. Syst. **19**(3), 427–443 (1997). https://doi.org/10.1145/256167.256195
28. Kozen, D.: Automata on guarded strings and applications. Tech. rep., Cornell University (2001)
29. Kozen, D., Patron, M.: Certification of compiler optimizations using kleene algebra with tests. In: Computational Logic - CL 2000, First International Conference, London, UK, 24-28 July, 2000, Proceedings. Lecture Notes in Computer Science, vol. 1861, pp. 568–582. Springer (2000). https://doi.org/10.1007/3-540-44957-4_38
30. Krauss, A., Nipkow, T.: Proof pearl: Regular expression equivalence and relation algebra. J. Autom. Reason. **49**(1), 95–106 (2012). https://doi.org/10.1007/S10817-011-9223-4
31. Leino, K.R.M.: Dafny power user, https://leino.science/dafny-power-user/
32. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: International conference on logic for programming artificial intelligence and reasoning. pp. 348–370. Springer (2010)

33. Leino, K.R.M.: Type-parameter completion (June 2019), https://leino.science/papers/krml270.html
34. Leino, K.R.M.: Iterating over a collection (Feb 2020), https://leino.science/papers/krml275.html
35. Leino, K.R.M.: Type-parameter modes: variance and cardinality preservation (Aug 2021), https://leino.science/papers/krml280.html
36. Leino, K.R.M.: Program Proofs. MIT Press (2023)
37. Leino, K.R.M., Tristan, J.B.: Working with coinduction, extreme predicates, and ordinals (Feb 2023), https://leino.science/papers/krml285.html
38. Moreira, N., Pereira, D., de Sousa, S.M.: Deciding kleene algebra terms equivalence in coq. Journal of Logical and Algebraic Methods in Programming **84**(3), 377–401 (2015)
39. Noble, J., Streader, D., Gariano, I.O., Samarakoon, M.: More programming than programming: Teaching formal methods in a software engineering programme. In: NASA Formal Methods Symposium. pp. 431–450. Springer (2022)
40. Owens, S., Reppy, J.H., Turon, A.: Regular-expression derivatives re-examined. J. Funct. Program. **19**(2), 173–190 (2009). https://doi.org/10.1017/S0956796808007090
41. Paulson, L.C.: Isabelle: The next seven hundred theorem provers. In: 9th International Conference on Automated Deduction, Argonne, Illinois, USA, May 23-26, 1988, Proceedings. Lecture Notes in Computer Science, vol. 310, pp. 772–773. Springer (1988). https://doi.org/10.1007/BFB0012891
42. Paulson, L.C.: A formalisation of finite automata using hereditarily finite sets. In: Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings. Lecture Notes in Computer Science, vol. 9195, pp. 231–245. Springer (2015). https://doi.org/10.1007/978-3-319-21401-6_15
43. Rot, J., Bonsangue, M.M., Rutten, J.J.M.M.: Coinductive proof techniques for language equivalence. In: Language and Automata Theory and Applications - 7th International Conference, LATA 2013, Bilbao, Spain, April 2-5, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7810, pp. 480–492. Springer (2013). https://doi.org/10.1007/978-3-642-37064-9_42
44. Rutten, J.J.M.M.: Universal coalgebra: a theory of systems. Theoretical Computer Science **249**(1), 3–80 (2000). https://doi.org/10.1016/S0304-3975(00)00056-6
45. Silva, A.: Kleene coalgebra. Ph.D. thesis, University of Nijmegen (2010)
46. Smolka, S., Foster, N., Hsu, J., Kappé, T., Kozen, D., Silva, A.: Guarded kleene algebra with tests: verification of uninterpreted programs in nearly linear time. Proc. ACM Program. Lang. **4**(POPL), 61:1–61:28 (2020). https://doi.org/10.1145/3371129
47. Traytel, D.: Formal languages, formally and coinductively. Logical Methods in Computer Science **13** (2017)
48. Tristan, J.B., Leino, K.R.M.: Aws dafny training, https://dafny.org/teaching-material/
49. Turi, D., Plotkin, G.D.: Towards a mathematical operational semantics. In: Proceedings, 12th Annual IEEE Symposium on Logic in Computer Science, Warsaw, Poland, June 29 - July 2, 1997. pp. 280–291. IEEE Computer Society (1997). https://doi.org/10.1109/LICS.1997.614955
50. Yang, Z., Wang, W., Casas, J., Cocchini, P., Yang, J.: Towards a correct-by-construction fhe model. Cryptology ePrint Archive (2023)

51. Zetzsche, S., Różowski, W.: Well-behaved (co)algebraic semantics of regular expressions in dafny (Feb 2024), https://dafny.org/blog/assets/src/semantics-of-regular-expressions/Archive.zip