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

Stefan Zetzsche[1] Wojciech Różowski[2]

[1]Amazon Web Services

[2]University College London

November 27, 2024

Introduction

Semantics of Regular Expressions

$$\text{Exp} \xrightarrow{\texttt{denotational, operational}} \text{Lang}$$

$\text{Exp} ::= 0 \mid 1 \mid a \in A \mid e + f \mid e \cdot f \mid e^*$

$\text{Lang} := \mathcal{P}(A^*)$ with $E(L) := [\varepsilon \in L]$ and $D_a(L) := \{w \mid a \cdot w \in L\}$

Denotational Semantics of Regular Expressions

We can equip Lang with algebraic structure resembling regular expressions:

$$
\begin{aligned}
0 &:= \emptyset \\
1 &:= \{\varepsilon\} \\
L_1 \cdot L_2 &:= \{w_1 \cdot w_2 \mid w_i \in L_i\} \\
L_1 + L_2 &:= L_1 \cup L_2 \\
L^* &:= \bigcup_{n \geq 0} L^n, \quad \text{where} \quad L^{n+1} := L \cdot L^n, \quad L^0 := \{\varepsilon\}
\end{aligned}
$$

Let `denotational` be the unique homomorphism

$$(\mathrm{Exp}, 0, 1, a, +, \cdot, {}^*) \to (\mathrm{Lang}, 0, 1, \{a\}, +, \cdot, {}^*).$$

Operational Semantics of Regular Expressions

We can equip Exp with coalgebraic structure resembling deterministic automata:

$$
\begin{aligned}
E(0) &:= 0 & D_a(0) &:= 0 \\
E(1) &:= 1 & D_a(1) &:= 0 \\
E(b) &:= 0 & D_a(b) &:= [a = b] \\
E(e+f) &:= E(e) + E(f) & D_a(e+f) &:= D_a(e) + D_a(f) \\
E(e \cdot f) &:= E(e) \cdot E(f) & D_a(e \cdot f) &:= D_a(e) \cdot f + E(e) \cdot D_a(f) \\
E(e^*) &:= 1 & D_a(e^*) &:= D_a(e) \cdot e^*.
\end{aligned}
$$

Let `operational` be the unique homomorphism

$$(\mathrm{Exp}, E, D_a) \to (\mathrm{Lang}, E, D_a).$$

Well-Behaved Semantics of Regular Expressions

$$\texttt{denotational} : (\mathsf{Exp}, 0, 1, a, +, \cdot, {}^*) \to (\mathsf{Lang}, 0, 1, \{a\}, +, \cdot, {}^*)$$
$$\texttt{operational} : (\mathsf{Exp}, E, D_a) \to (\mathsf{Lang}, E, D_a)$$

Some immediate questions:

- Are `denotational` and `operational` well-defined?
- Are `denotational` and `operational` also homomorphisms of the respective other type?
- Are `denotational` and `operational` equal?

In this paper, we use *Dafny* to encode `denotational` and `operational` and prove their equivalence.

Dafny: A Verification-Aware Programming Language

Dafny allows a clear distinction between an idealised mathematical specification and an efficient implementation thereof.

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

method ComputeFib(n: nat) returns (fib: nat)
{
  var i, currentFib, nextFib := 0, 0, 1;
  while i < n
  {
    i, currentFib, nextFib := i + 1, nextFib, currentFib + nextFib;
  }
  return currentFib;
}
```

Dafny: A Verification-Aware Programming Language

Dafny allows a clear distinction between an idealised mathematical specification and an efficient implementation thereof.

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

method ComputeFib(n: nat) returns (fib: nat)
  ensures fib == Fib(n)
{
  var i, currentFib, nextFib := 0, 0, 1;
  while i < n
  {
    i, currentFib, nextFib := i + 1, nextFib, currentFib + nextFib;
  }
  return currentFib;
}
```

Dafny: A Verification-Aware Programming Language

Dafny allows a clear distinction between an idealised mathematical specification and an efficient implementation thereof.

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

method ComputeFib(n: nat) returns (fib: nat)
  ensures fib == Fib(n)
{
  var i, currentFib, nextFib := 0, 0, 1;
  while i < n
    invariant i <= n && currentFib == Fib(i) && nextFib == Fib(i + 1)
  {
    i, currentFib, nextFib := i + 1, nextFib, currentFib + nextFib;
  }
  return currentFib;
}
```

Dafny Supports Multiple Paradigms

Polymorphism

Inductive datatypes

- `datatype List<T> = Nil | Cons(T, List)`

Coinductive datatypes

- `codatatype Stream<T> = Cons(T, Stream)`

Lambda expressions

- `var f: int -> int := x => x+1`

Higher-order functions

- `F(f: int -> int, n: int): int { f(n) }`

Classes and traits with mutable states

Some Use Cases of Dafny

Authorization

- AWS Identity and Access Management

Distributed Systems

- IronFleet

Cryptography

- AWS Encryption SDK

Privacy

- DafnyVMC, SampCert

Code Generation

- DafnyBench, DafnyAutopilot, VerMCTS, Laurel, ...

Denotational Semantics

Regular Expressions as Datatype

In Dafny, we define regular expressions as inductive datatype:

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

On a high-level, `Exp` is the *smallest* structure closed under the constructors of regular expressions. It is the *initial* Σ-algebra.

$$\Sigma X = 1 + 1 + A + X^2 + X^2 + X$$

Formal Languages as Codatatype

In Dafny, we define formal languages as coinductive datatype:

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

On a high-level, `Lang` is the *greatest* structure closed under the destructors of deterministic automata. It is the *final* B-coalgebra.

$BX = 2 \times X^A$

Formal Languages as Codatatype

To some, our way of modeling formal languages might seem odd at first sight.

Typically, a formal language is defined intrinsically, as an element of type $\mathcal{P}(A^*)$.

Here, we treat formal languages extrinsically, in terms of their universal property as greatest coalgebraic structure X equipped with functions $E : X \to 2$ and $D : X \to X^A$.

Indeed, for any $U \in \mathcal{P}(A^*)$, we can define $E(U) := [\varepsilon \in U]$ and $D(U)(a) := \{w \mid aw \in U\}$.

An Algebra of Formal Languages

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

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)))
}
```

We equip Lang with a Σ-algebra structure.

Denotational Semantics as Induced Morphism

```
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 Σ-algebra structures of `Exp` and `Lang` allow us to define `Denotational`.

Bisimilarity and Coinduction 1

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

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.

Bisimilarity and Coinduction 2

```
/* 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)
}
```

Under the hood, Dafny transforms the *greatest* predicate `Bisimilar` into the above data.

Bisimilarity and Coinduction 3

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

Dafny automatically proves that bisimilarity is reflexive.

Bisimilarity and Coinduction 4

```
/* 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);
  }
}
```

Under the hood, Dafny transforms the *greatest* lemma `BisimilarityIsReflexive` into the above data.

Denotational Semantics as Algebra Homomorphism

```
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))
  )
}

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

`Denotational` is a Σ-algebra homomorphism (up to pointwise bisimilarity).

Operational Semantics

A Coalgebra of Regular Expressions

```
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))
}
```

Using Brzozowski's derivatives, we equip `Exp` with a B-coalgebra structure.

Operational Semantics as Induced Morphism

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

The B-coalgebra structures of `Exp` and `Lang` allow us to define `Operational`.

Operational Semantics as Coalgebra Homomorphism

```
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))))
}

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));
  }
}
```

`Operational` is a B-coalgebra homomorphism (up to pointwise bisimilarity).

Well-Behaved Semantics

The Status

$$
\begin{tikzcd}
\Sigma(\mathtt{Exp}) \arrow[r, dashed, "\Sigma(\mathtt{Denotational})"] \arrow[d] & \Sigma(\mathtt{Lang}) \arrow[d, "{[\mathtt{Zero},\mathtt{One},\mathtt{Singleton},\mathtt{Plus},\mathtt{Comp},\mathtt{Star}]}"'] \\
\mathtt{Exp} \arrow[r, dashed, "\mathtt{Denotational}"] & \mathtt{Lang}
\end{tikzcd}
$$

$$
\begin{tikzcd}
\mathtt{Exp} \arrow[r, dashed, "\mathtt{Operational}"] \arrow[d, "\langle \mathtt{Eps},\mathtt{Delta} \rangle"] & \mathtt{Lang} \arrow[d] \\
B(\mathtt{Exp}) \arrow[r, dashed, "B(\mathtt{Operational})"] & B(\mathtt{Lang})
\end{tikzcd}
$$

The Goal

$$
\begin{tikzcd}
\Sigma(\texttt{Exp}) \arrow[r, dashed, "\Sigma(\texttt{Denotational}\cong\texttt{Operational})"] \arrow[d] & \Sigma(\texttt{Lang}) \arrow[d, "{[\texttt{Zero},\texttt{One},\texttt{Singleton},\texttt{Star},\texttt{Plus},\texttt{Comp}]}"'] \\
\texttt{Exp} \arrow[r, dashed, "\texttt{Denotational}\cong\texttt{Operational}"] \arrow[d, "\langle\texttt{Eps},\texttt{Delta}\rangle"] & \texttt{Lang} \arrow[d] \\
B(\texttt{Exp}) \arrow[r, dashed, "B(\texttt{Denotational}\cong\texttt{Operational})"] & B(\texttt{Lang})
\end{tikzcd}
$$

The Plan

1. `Denotational` is also a B-coalgebra homomorphism.
2. B-coalgebra homomorphisms are unique up to pointwise bisimilarity.
3. `Denotational` and `Operational` are equal up to pointwise bisimilarity.
4. `Operational` is also a Σ-algebra homomorphism.

(1a/4) Denotational Semantics as Coalgebra Homomorphism

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

The proof is a bit more elaborate than the previous ones. We again use that bisimilarity is reflexive, but also that it 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))
{}
```

(1b/4) Denotational Semantics as Coalgebra Homomorphism

```
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))
{
  ...
}
```

(1b/4) Denotational Semantics as Coalgebra Homomorphism

```
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))
{
  forall k ensures Bisimilar#[k](Comp(L1a, L2a), Comp(L1b, L2b)) {
    ...
  }
}
```

(1b/4) Denotational Semantics as Coalgebra Homomorphism

```
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))
{
  forall k ensures Bisimilar#[k](Comp(L1a, L2a), Comp(L1b, L2b)) {
    if k != 0 {
      ...
    }
  }
}
```

(1b/4) Denotational Semantics as Coalgebra Homomorphism

```
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))
{
  forall k ensures Bisimilar#[k](Comp(L1a, L2a), Comp(L1b, L2b)) {
    if k != 0 {
      var k' :| k' + 1 == k;
      CompCongruenceHelper(k', L1a, L1b, L2a, L2b);
    }
  }
}

lemma CompCongruenceHelper<A(!new)>(k: nat, L1a: Lang, L1b: Lang, L2a: Lang, L2b: Lang)
  requires forall n : nat :: n <= k + 1 ==> Bisimilar#[n](L1a, L1b)
  requires forall n : nat :: n <= k + 1 ==> Bisimilar#[n](L2a, L2b)
  ensures Bisimilar#[k+1](Comp(L1a, L2a), Comp(L1b, L2b))
{
  ...
}
```

(2/4) Coalgebra Homomorphisms Are Unique

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

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)
{}
```

(3/4) Denotational and Operational Semantics Are Bisimilar

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

From the previous results, we can immediately deduce our main claim.

(4/4) Operational Semantics as Algebra Homomorphism

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

The main idea of the proof is to take advantage of that `denotational` is an algebra homomorphism and pointwise bisimilar to `operational`.

We also use that bisimilarity is symmetric and a congruence with respect to `Star`.

Discussion, Related and Future Work

Discussion

$$
\begin{tikzcd}
\Sigma(\mathtt{Exp}) \arrow[r, dashed, "\Sigma(\mathtt{Denotational}\cong\mathtt{Operational})"] \arrow[d] & \Sigma(\mathtt{Lang}) \arrow[d] \\
\mathtt{Exp} \arrow[r, dashed, "\mathtt{Denotational}\cong\mathtt{Operational}"] \arrow[d, "\langle \mathtt{Eps}, \mathtt{Delta} \rangle"] & \mathtt{Lang} \arrow[d] \\
B(\mathtt{Exp}) \arrow[r, dashed, "B(\mathtt{Denotational}\cong\mathtt{Operational})"] & B(\mathtt{Lang})
\end{tikzcd}
$$

$[\mathtt{Zero}, \mathtt{One}, \mathtt{Singleton}, \mathtt{Star}, \mathtt{Plus}, \mathtt{Comp}]$

Some Related Work

Derivatives of Regular Expressions (Brzozowski, 1964)

- Introduces the B-coalgebra structure on `Lang`.

Towards a Mathematical Operational Semantics (Turi; Plotkin, 1997)

- Introduces the concept of well-behaved semantics in the context of bialgebras.

A Bialgebraic Review of Deterministic Automata, Regular Expressions and Languages (Jacobs, 2006)

- Discusses a bialgebraic perspective on regular expressions.

Formal Languages, Formally and Coinductively (Traytel, 2017)

- Represents languages coinductively in Isabelle.

Some Future Work

Kleene Algebra With Tests

- Extends Kleene Algebra with primitives in a Boolean algebra.

Guarded Kleene Algebra with Tests

- Axiomatises an efficiently decidable fragment of KAT.

NetKAT

- Extends Kleene Algebra with Tests with primitives for network verification.