This is a sketch proof for the correctness of the algorithm ders_simp.

1 Function Definitions

Definition 1. Bits

```
abstract class Bit
case object Z extends Bit
case object S extends Bit
case class C(c: Char) extends Bit

type Bits = List[Bit]
```

Definition 2. Annotated Regular Expressions

```
abstract class ARexp
case object AZERO extends ARexp
case class AONE(bs: Bits) extends ARexp
case class ACHAR(bs: Bits, f: Char) extends ARexp
case class AALTS(bs: Bits, rs: List[ARexp]) extends ARexp
case class ASEQ(bs: Bits, r1: ARexp, r2: ARexp) extends ARexp
case class ASTAR(bs: Bits, r: ARexp) extends ARexp
```

Definition 3. bnullable

```
  def bnullable (r: ARexp) : Boolean = r match {
    case AZERO => false
    case AONE(_) => true
    case ACHAR(_,_) => false
    case AALTS(_, rs) => rs.exists(bnullable)
    case ASEQ(_, r1, r2) => bnullable(r1) && bnullable(r2)
    case ASTAR(_, _) => true
  }
```

Definition 4. ders_simp

```
def ders_simp(r: ARexp, s: List[Char]): ARexp = {
 s match {
   case Nil => r
   case c::cs => ders_simp(bsimp(bder(c, r)), cs)
 }
}
```

Definition 5. bder

```
def bder(c: Char, r: ARexp) : ARexp = r match {
 case AZERO => AZERO
 case AONE(_) => AZERO
 case ACHAR(bs, f) => if (c == f) AONE(bs:::List(C(c))) else AZERO
 case AALTS(bs, rs) => AALTS(bs, rs.map(bder(c, _)))
 case ASEQ(bs, r1, r2) => {
  if (bnullable(r1)) AALT(bs, ASEQ(Nil, bder(c, r1), r2), fuse(mkepsBC(r1), bder(c, r2)))
  else ASEQ(bs, bder(c, r1), r2)
  }
 case ASTAR(bs, r) => ASEQ(bs, fuse(List(S), bder(c, r)), ASTAR(Nil, r))
}
```

Definition 6. bsimp

```
  def bsimp(r: ARexp): ARexp = r match {
    case ASEQ(bs1, r1, r2) => (bsimp(r1), bsimp(r2)) match {
        case (AZERO, _) => AZERO
        case (_, AZERO) => AZERO
        case (AONE(bs2), r2s) => fuse(bs1 ++ bs2, r2s)
```

```
      case (r1s, r2s) => ASEQ(bs1, r1s, r2s)
    }
    case AALTS(bs1, rs) => {
      val rs_simp = rs.map(bsimp)
      val flat_res = flats(rs_simp)
      val dist_res = distinctBy(flat_res, erase)
      dist_res match {
        case Nil => AZERO
        case s :: Nil => fuse(bs1, s)
        case rs => AALTS(bs1, rs)
      }
    }
    //case ASTAR(bs, r) => ASTAR(bs, bsimp(r))
    case r => r
  }
```

Definition 7. sub-parts of bsimp

- flats
  flattens the list.
- dB
  means distinctBy
- Co
  The last matching clause of the function bsimp, namely dist_res match case Nil =¿ AZERO case s :: Nil =¿ fuse(bs1, s) case rs =¿ AALTS(bs1, rs)

Definition 8. fuse

```
def fuse(bs: Bits, r: ARexp) : ARexp = r match {
  case AZERO => AZERO
  case AONE(cs) => AONE(bs ++ cs)
  case ACHAR(cs, f) => ACHAR(bs ++ cs, f)
  case AALTS(cs, rs) => AALTS(bs ++ cs, rs)
  case ASEQ(cs, r1, r2) => ASEQ(bs ++ cs, r1, r2)
  case ASTAR(cs, r) => ASTAR(bs ++ cs, r)
}
```

Definition 9. mkepsBC

```
def mkepsBC(r: ARexp) : Bits = r match {
  case AONE(bs) => bs
  case AALTS(bs, rs) => {
    val n = rs.indexWhere(bnullable)
    bs ++ mkepsBC(rs(n))
  }
  case ASEQ(bs, r1, r2) => bs ++ mkepsBC(r1) ++ mkepsBC(r2)
  case ASTAR(bs, r) => bs ++ List(Z)
}
```

Definition 10. mkepsBC equicalence
Given 2 nullable annotated regular expressions r1, r2, if mkepsBC(r1) == mkepsBC(r2) then r1 and r2 are mkepsBC equivalent, denoted as r1 $\sim_{m\epsilon}$ r2

Definition 11. shorthand notation for ders
For the sake of reducing verbosity, we sometimes use the shorthand notation $d_c(r)$ for the function application bder(c, r) and $s(r)$(s here stands for simplification) for the function application bsimp(r) .
We omit the subscript when it is clear from context what that character is and write $d(r)$ instead of $d_c(r)$.
And we omit the parentheses when no confusion can be caused. For example ders_simp(c, r) can be written as $s(d_c(r))$ or even sdr as we know the derivative operation is w.r.t the character c. Here the s and d are more like operators that take an annotated regular expression as an input and return an annotated regular expression as an output

Definition 12. mkepsBC invariant manipulation of bits and notation
ALTS(bs, ALTS(bs1, rs1), ALTS(bs2, rs2)) $\sim_{m\epsilon}$ ALTS(bs, rs1.map(fuse(bs1, _)) ++ rs2.map(fuse(bs2, _))).
We also use $bs2 >> rs2$ as a shorthand notation for rs2.map(fuse(bs2,_)).

Definition 13. distinctBy operation expressed in a different way–how it transforms the list
Given two lists rs1 and rs2, we define the operation $--$:
$rs1 - -rs2 := [r \in rs1 | r \notin rs2]$ Note that the order is preserved as in the original list.

2 Main Result

Lemma 1. *simplification function does not simplify an already simplified regex*
bsimp(r) == bsimp(bsimp(r)) holds for any annotated regular expression r.

Lemma 2. *simp and mkeps*
When r is nullable, we have that mkeps(bsimp(r)) == mkeps(r)

Lemma 3. *mkeps equivalence w.r.t some syntactically different regular expressions(1 ALTS)*
When one of the 2 regular expressions $s(r_1)$ and $s(r_2)$ is ALTS(bs1, rs1), we have that $ds(ALTS(bs, r1, r2)) \sim_{m\epsilon} d(ALTS(bs, sr_1, sr_2))$

Proof. By opening up one of the alts and show no additional changes are made. ☐

Lemma 4. *mkepsBC equivalence w.r.t syntactically different regular expressions(2 ALTS)*
$sr_1 = ALTS(bs1, rs1)$ *and* $sr_2 = ALTS(bs2, rs2)$ *we have* $d(sr_1 + sr_2) \sim_{m\epsilon} d(ALTS(bs, bs1 >> rs1 + +bs2 >> rs2))$

Proof. We are just fusing bits inside here, there is no other structural change. ☐

Lemma 5. *mkepsBC equivalence w.r.t syntactically different regular expressions(2 ALTS+ some deletion)*
$dCo(ALTS(bs, dB(bs1 >> rs1 + +bs2 >> rs2))) \sim_{m\epsilon} dCo(ALTS(bs, dB(bs1 >> rs1 + +((bs2 >> rs2) - -rs1))))$

Proof. The removed parts have already appeared before in rs_1, so if any of them is truly nullable and is chosen as the mkeps path, it will have been traversed through in its previous counterpart.
(We probably need to switch the position of lemma5 and lemma6) ☐

Lemma 6. *after opening two previously simplified alts up into terms, length must exceed 2*
$dCo(ALTS(bs, rs)) \sim_{m\epsilon} d(ALTS(bs, rs))$ *if rs is a list of length greater than or equal to 2.*

Proof. As suggested by the title of this lemma ☐

Theorem 1. *Correctness Result*

- *When s is a string in the language L(ar),*
  ders_simp(ar, s) $\sim_{m\epsilon}$ ders(ar, s),

- *when s is not a string of the language L(ar) ders_simp(ar, s) is not nullable*

Proof. Split into 2 parts.

- When we have an annotated regular expression ar and a string s that matches ar, by the correctness of the algorithm ders, we have that ders(ar, s) is nullable, and that mkepsBC will extract the desired bits for decoding the correct value v for the matching, and v is a POSIX value. Now we prove that mkepsBC(ders_simp(ar, s)) yields the same bitsequence. We first open up the ders_simp function into nested alternating sequences of ders and simp. Assume that $s = c_1...c_n (n \geq 1)$ where each of the c_i are characters. Then $ders\_simp(ar, s) = s(d_{c_n}(...s(d_{c_1}(r))...)) = sdsd......sdr$. If we can prove that $sdr \sim_{m\epsilon} dsr$ holds for any regular expression and any character, then we are done. This is because then we can push ders operation inside and move simp operation outside and have that $sdsd...sdr \sim_{m\epsilon} ssddsdsd...sdr \sim_{m\epsilon} ... \sim_{m\epsilon} s....sd....dr$ and using lemma1 we have that $s...sd....dr = sd...dr$. By lemma2, we have $RHS \sim_{m\epsilon} d...dr$.
  Now we proceed to prove that $sdr \sim_{m\epsilon} dsr$. This can be reduced to proving $dr \sim_{m\epsilon} dsr$ as we know that $dr \sim_{m\epsilon} sdr$ by lemma2.

  we use an induction proof. Base cases are omitted. Here are the 3 inductive cases.

  - $r_1 + r_2$ $r_1 + r_2$
    The most difficult case is when $sr1$ and $sr2$ are both ALTS, so that they will be opened up in the flats function and some terms in sr2 might be deleted. Or otherwise we can use the argument that $d(r_1 + r_2) = dr_1 + dr_2 \sim_{m\epsilon} dsr_1 + dsr_2 \sim_{m\epsilon} ds(r_1 + r_2)$, the last equivalence being established by lemma3. When $s(r_1), s(r_2)$ are both ALTS, we have to be more careful for the last equivalence step, namelly, $dsr_1 + dsr_2 \sim_{m\epsilon} ds(r_1 + r_2)$.
    We have that $LHS = dsr_1 + dsr_2 = d(sr_1 + sr_2)$. Since $sr_1 = ALTS(bs1, rs1)$ and $sr_2 = ALTS(bs2, rs2)$ we have $d(sr_1 + sr_2) \sim_{m\epsilon} d(ALTS(bs, bs1 >> rs1 + +bs2 >> rs2))$ by lemma4. On the other hand, $RHS =$

$ds(ALTS(bs, r1, r2)) \sim_{m\epsilon} dCo(ALTS(bs, dB(flats(s(r1), s(r2))))) == dCo(ALTS(bs, dB(bs1 >> rs1{+}{+}bs2 >> rs2)))$.

By definition of bsimp and flats, $dCo(ALTS(bs, dB(bs1 >> rs1 + +bs2 >> rs2))) \sim_{m\epsilon} dCo(ALTS(bs, (bs1 >> rs1 + +((bs2 >> rs2) - -rs1))))$ by lemma5.

$dCo(ALTS(bs, (bs1 >> rs1 + +((bs2 >> rs2) - -rs1)))) \sim_{m\epsilon} d(ALTS(bs, bs1 >> rs1 + +(bs2 >> rs2) - -rs1))$ by lemma6.

Using lemma5 again, we have $d(ALTS(bs, bs1 >> rs1 + +(bs2 >> rs2) - -rs1)) \sim_{m\epsilon} d(ALTS(bs, bs1 >> rs1 + +bs2 >> rs2))$.

This completes the proof.

- $r*$

  s(r*) = s(r).

- $r1.r2$

  using previous.

- Proof of second part of the theorem: use a similar structure of argument as in the first part.

□