1 Main Result

Want to prove

$$bsimp(bder(c, a)) = bsimp(bder(c, bsimp(a))). \tag{1}$$

For simplicity, we use s to denote $bsimp$ and use $a\backslash c$ or $d\ c\ a$ to denote $bder(c, a)$, then we can write the equation we want to prove in the following manner:

$$s\ d\ c\ a = s\ d\ c\ s\ a$$

Specifically, we are interested in the case where $a = a_1 + a_2$. The inductive hypothesis is that

$$s\ d\ c\ a_1 = s\ d\ c\ s\ a_1 \textbf{ and } s\ d\ c\ a_2 = s\ d\ c\ s\ a_2.$$

We want to prove that the LHS of (1) is equal to the RHS of (1). For better readability the bits are ommitted as they don't inhibit the proof process but just adds to the nuisance of writing. LHS can be manipulated successively as follows:

$$\begin{array}{rcl} LHS & = & s\ (a_1 + a_2)\backslash c \\ & = & s\ (a_1\backslash c + a_2\backslash c) \\ & \overset{Lemma\ 1}{=} & s(s(a_1\backslash c) + s(a_2\backslash c)) \\ & \overset{Lemma\ 2}{=} & s(s(a_1)\backslash c + s(a_2)\backslash c). \end{array}$$

RHS can be manipulated this way:

$$RHS \quad = \quad s\ [(s(a_1 + a_2)]\backslash c$$

If we refer to $s(a_1 + a_2)$ as $core$, then we have

$$RHS \quad = \quad s\ (core\backslash c)$$

and then

$$\begin{array}{rcl} core & = & s\ ALTS(bs, a_1 + a_2) \\ & \overset{bsimp\ def}{=} & Li(ALTS(bs, dB(flats(s(a_1) + s(a_2))))) \end{array}$$

Here we use Li to refer to the operation that opens up the $ALTS$ when it has 1 element, returns 0 when it has 0 elements or does nothing when there are 2 or more elements in the list rs in $ALTS(bs, rs)$(in scala code corresponds to the case clauses).

Now in order to establish that $LHS = RHS$, we need to prove the transformed results we got above for LHS and RHS are equal to each other. That is,

$$s(s(a_1)\backslash c + s(a_2)\backslash c) \quad = \quad Li(ALTS(bs, dB(flats(s(a_1) + s(a_2))))$$

We shall call the two sides of the above equation LHS' and RHS'. To prove this equality we just need to consider what $s(a_1)$ and $s(a_2)$ look like. There are three interesting possibility for each, namely $s(a_i)$ is an alt, a star or a sequence. Combined that is 9 possibilities. We just need to investigate each of these 9 possibilities. Here we only one of the 9 cases. The others are handled in a similar fashion.

When $s(a_1) = ALTS(bs_1, as_1)$ and $s(a_2) = ALTS(bs_2, as_2)$,

$$\begin{array}{c} LHS' \\ = \\ s(ALTS(bs, ALTS(bs_1, as_1)\backslash c + ALTS(bs_2, as_2)\backslash c)) \\ = \\ s(ALTS(bs, ALTS(bs_1, as_1.map\backslash c) + ALTS(bs_2, as_2.map\backslash c))) \\ \overset{Lemma\ 3}{=} \\ s(ALTS(bs, (as_1.map\backslash c).map(fuse(bs_1)) + (as_2.map\backslash c).map(fuse(bs_2)))) \end{array}$$

And then we deal with RHS': RHS'

$\overset{Lemma\ 4}{=}$

$s(ALTS(bs, (as_1.map\backslash c).map(fuse(bs_1)) + +(as_2.map\backslash c).map(fuse(bs_2))))$

and this completes the proof.

Lemma 1. *doing simplification in advance to subparts*
We have that for any annotated regular expressions a_1 a_2 and bitcode bs,
$bsimp(ALTS(bs, a_1, a_2)) = bsimp(ALTS(bs, bsimp(a_1), bsimp(a_2)))$

Lemma 2. *combination of lemma 1 and inductive hypothesis(from now on use simple notation)*
We have that for any annotated regular expressions a_1 a_2 and bitcode bs, $s(s(a_1\backslash c) + s(a_2\backslash c)) = s(s(a_1)\backslash c + s(a_2)\backslash c)$

Lemma 3. *Spilling out ALTS does not affect simplification result*
$s(ALTS(bs, ALTS(bs_1, as_1.map\backslash c) + ALTS(bs_2, as_2.map\backslash c)))$
$\overset{Lemma\ 3}{=}$
$s(ALTS(bs, (as_1.map\backslash c).map(fuse(bs_1)) + (as_2.map\backslash c).map(fuse(bs_2))))$

Lemma 4. *deleting duplicates does not affect simplification result*
$s(ALTS(bs, (as_1.map\backslash c).map(fuse(bs_1)) + (as_2.map\backslash c).map(fuse(bs_2))))$
$=$
$s(ALTS(bs, dB((as_1.map\backslash c).map(fuse(bs_1)) + (as_2.map\backslash c).map(fuse(bs_2)))))$

Lemma 5. *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,_)).

Lemma 6. *What does dB do to two already simplified ALTS*
$dCo(ALTS(bs, dB(bs1 >> rs1 + +bs2 >> rs2))) = dCo(ALTS(bs, bs1 >> rs1 + +((bs2 >> rs2) - -rs1)))$

Proof. We prove that $dB(bs1 >> rs1 + +bs2 >> rs2) = bs1 >> rs1 + +((bs2 >> rs2) - -rs1)$.

□

Lemma 7. *after opening two previously simplified alts up into terms, length must exceed 2*
If sr1, sr2 are of the form ALTS(bs1, rs1), ALTS(bs2, rs2) respectively, then we have that $Co(bs, (bs1 >> rs1) + +(bs2 >> rs2) - -rs1) = ALTS(bs, bs1 >> rs1 + +(bs2 >> rs2) - -rs1)$

Proof. $Co(bs, rs) \sim_{m\epsilon} ALTS(bs, rs)$ if rs is a list of length greater than or equal to 2. As suggested by the title of this lemma, ALTS(bs1, rs1) is a result of simplification, which means that rs1 must be composed of at least 2 distinct regular terms. This alone says that $bs1 >> rs1 + +(bs2 >> rs2) - -rs1$ is a list of length greater than or equal to 2, as the second operand of the concatenation operator $(bs2 >> rs2) - -rs1$ can only contribute a non-negative value to the overall length of the list $bs1 >> rs1 + +(bs2 >> rs2) - -rs1$. □

Lemma 8. *mkepsBC equivalence w.r.t syntactically different regular expressions(2 ALTS+ some deletion after derivatives)*
$dALTS(bs, bs1 >> rs1 + +bs2 >> rs2) \sim_{m\epsilon} dALTS(bs, bs1 >> rs1 + +((bs2 >> rs2) - -rs1))$

Proof. Let's call $bs1 >> rs1\ rs1'$ and $bs2 >> rs2\ rs2'$. Then we need to prove $dALTS(bs, rs1' + +rs2') \sim_{m\epsilon} dALTS(bs, rs1' + +(rs2' - -rs1'))$.
We might as well omit the prime in each rs for simplicty of notation and prove $dALTS(bs, rs1 + +rs2) \sim_{m\epsilon} dALTS(bs, rs1 + +(rs2 - -rs1))$.
We know that the result of derivative is nullable, so there must exist an r in rs1++rs2 s.t. r is nullable.
If $r \in rs1$, then equivalence holds. If $r \in rs2 \wedge r \notin rs1$, equivalence holds as well. This completes the proof. □

Lemma 9. *nullability relation between a regex and its simplified version*
r nullable $\iff$ sr nullable

Lemma 10. *concatenation + simp invariance of mkepsBC*
$mkepsBCr1 \cdot sr2 = mkepsBCr1 \cdot r2$ if both r1 and r2 are nullable.

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 $sdsd...sdr \sim_{m\epsilon} ssddsdsd...sdr \sim_{m\epsilon} ... \sim_{m\epsilon} s....sd....dr$. Using Lemma 1 we have that $s...sd....dr = sd...dr$. By Lemma 2, we have $RHS \sim_{m\epsilon} d...dr$.

Notice that we don't actually need Lemma 1 here. That is because by Lemma 2, we can have that $s...sd....dr \sim_{m\epsilon} sd...dr$. The equality above can be replaced by mkepsBC equivalence without affecting the validity of the whole proof since all we want is mkepsBC equivalence, not equality.

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 Lemma 2.

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

- $r_1 + r_2$
  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 Lemma 3. 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 Lemma 4. On the other hand, $RHS = ds(ALTS(bs, r1, r2)) = dCo(bs, dB(flats(s(r1), s(r2)))) = dCo(bs, dB(bs1 >> rs1 + +bs2 >> rs2))$ by definition of bsimp and flats.
  $dCo(bs, dB(bs1 >> rs1 + +bs2 >> rs2)) = dCo(bs, (bs1 >> rs1 + +((bs2 >> rs2) - -rs1)))$ by Lemma 6.
  $dCo(bs, (bs1 >> rs1 + +((bs2 >> rs2) - -rs1))) = d(ALTS(bs, bs1 >> rs1 + +(bs2 >> rs2) - -rs1))$ by Lemma 7.
  Using Lemma 8, we have $d(ALTS(bs, bs1 >> rs1 + +(bs2 >> rs2) - -rs1)) \sim_{m\epsilon} d(ALTS(bs, bs1 >> rs1 + +bs2 >> rs2)) \sim_{m\epsilon} RHS$.
  This completes the proof.
- $r*$
  $s(r*) = r*$. Our goal is trivially achieved.
- $r1 \cdot r2$
  When r1 is nullable, $dsr1r2 = dsr1 \cdot sr2 + dsr2 \sim_{m\epsilon} dr1 \cdot sr2 + dr2 = dr1 \cdot r2 + dr2$. The last step uses Lemma 10.
  When r1 is not nullable, $dsr1r2 = dsr1 \cdot sr2 \sim_{m\epsilon} dr1 \cdot sr2 \sim_{m\epsilon} dr1 \cdot r2$

- Proof of second part of the theorem: use a similar structure of argument as in the first part.
- This proof has a major flaw: it assumes all dr is nullable along the path of deriving r by s. But it could be the case that $s \in L(r)$ but $\exists s' \in Pref(s)$ *s.t.* $ders(s', r)$ is not nullable (or equivalently, $s' \notin L(r)$). One remedy for this is to replace the mkepsBC equivalence relation into some other transitive relation that entails mkepsBC equivalence.

□

Theorem 2. *This is a very strong claim that has yet to be more carefully examined and proved. However, experiments suggest a very good hope for this.*
Define pushbits as the following:

```
def pushbits(r: ARexp): ARexp = r match {
    case AALTS(bs, rs) => AALTS(Nil, rs.map(r=>fuse(bs, pushbits(r))))
    case ASEQ(bs, r1, r2) => ASEQ(bs, pushbits(r1), pushbits(r2))
    case r => r
  }
```

Then we have $\boldsymbol{pushbits(ders\_simp(ar, s)) == simp(ders(ar, s))}$ *or* $\boldsymbol{ders\_simp(ar, s) == simp(ders(ar, s))}$.
Unfortunately this does not hold. A counterexample is

```
baa
original regex
STA
 └-ALT
    └-STA List(Z)
    |  └-a
    └-ALT List(S)
       └-b List(Z)
       └-a List(S)
```

```
regex after ders simp
ALT List(S, S, Z, C(b))
 └-SEQ
 |  └-STA List(S, Z, S, C(a), S, C(a))
 |  |  └-a
 |  └-STA
 |     └-ALT
 |        └-STA List(Z)
 |        |  └-a
 |        └-ALT List(S)
 |           └-b List(Z)
 |           └-a List(S)
 └-SEQ List(S, Z, S, C(a), Z)
    └-ALT List(S)
    |  └-STA List(Z, S, C(a))
    |  |  └-a
    |  └-ONE List(S, S, C(a))
    └-STA
       └-ALT
          └-STA List(Z)
          |  └-a
          └-ALT List(S)
             └-b List(Z)
             └-a List(S)
regex after ders
ALT
 └-SEQ
 |  └-ALT List(S)
 |  |  └-SEQ List(Z)
 |  |  |  └-ZERO
 |  |  |  └-STA
 |  |  |     └-a
 |  |  └-ALT List(S)
 |  |     └-ZERO
 |  |     └-ZERO
 |  └-STA
 |     └-ALT
 |        └-STA List(Z)
 |        |  └-a
 |        └-ALT List(S)
 |           └-b List(Z)
 |           └-a List(S)
 └-ALT List(S, S, Z, C(b))
    └-SEQ
    |  └-ALT List(S)
    |  |  └-ALT List(Z)
    |  |  |  └-SEQ
    |  |  |  |  └-ZERO
    |  |  |  |  └-STA
    |  |  |  |     └-a
    |  |  |  └-SEQ List(S, C(a))
    |  |  |     └-ONE List(S, C(a))
    |  |  |     └-STA
    |  |  |        └-a
    |  |  └-ALT List(S)
    |  |     └-ZERO
    |  |     └-ZERO
    |  └-STA
    |     └-ALT
    |        └-STA List(Z)
```

```
    |        |  └─a
    |        └─ALT List(S)
    |           └─b List(Z)
    |           └─a List(S)
    └─SEQ List(S, Z, S, C(a), Z)
       └─ALT List(S)
       |  └─SEQ List(Z)
       |  |  └─ONE List(S, C(a))
       |  |  └─STA
       |  |     └─a
       |  └─ALT List(S)
       |     └─ZERO
       |     └─ONE List(S, C(a))
       └─STA
          └─ALT
             └─STA List(Z)
             |  └─a
             └─ALT List(S)
                └─b List(Z)
                └─a List(S)
regex after ders and then a single simp
ALT
 └─SEQ List(S, S, Z, C(b))
 |  └─STA List(S, Z, S, C(a), S, C(a))
 |  |  └─a
 |  └─STA
 |     └─ALT
 |        └─STA List(Z)
 |        |  └─a
 |        └─ALT List(S)
 |           └─b List(Z)
 |           └─a List(S)
 └─SEQ List(S, S, Z, C(b), S, Z, S, C(a), Z)
    └─ALT List(S)
    |  └─STA List(Z, S, C(a))
    |  |  └─a
    |  └─ONE List(S, S, C(a))
    └─STA
       └─ALT
          └─STA List(Z)
          |  └─a
          └─ALT List(S)
             └─b List(Z)
             └─a List(S)
```