

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

### Definition 3. bnullable

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

### Definition 5. bder

```
def bder(c: Char, r: AExp) : AExp = 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: AExp): AExp = 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, with a slight modification to suit later reasoning.

```

def Co(bs1, rs): ARexp = {
  rs 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 equivalence

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  $\text{bsimp}(r)$ .

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  $\text{ders\_simp}(c, r)$  can be written as  $s(d_c(r))$  or even  $\text{sdr}$  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.**  $\text{distinctBy}$  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 each term appears in  $rs1 - -rs2$  is preserved as in the original list.

## 2 Main Result

**Lemma 1.** *simplification function does not simplify an already simplified regex*

$\text{bsimp}(r) == \text{bsimp}(\text{bsimp}(r))$  holds for any annotated regular expression  $r$ .

**Lemma 2.** *simp and mkeys*

When  $r$  is nullable, we have that  $\text{mkeys}(\text{bsimp}(r)) == \text{mkeys}(r)$

**Lemma 3.** *mkeys 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 of the form  $\text{ALTS}(bs1, rs1)$ , we have that  $ds(\text{ALTS}(bs, r1, r2)) \sim_{m\epsilon} d(\text{ALTS}(bs, sr_1, sr_2))$

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

Details:  $ds(\text{ALTS}(bs, r1, r2)) = dCo(bs, dB(flats(sr1, sr2)))$  □

**Lemma 4.** *mkeysBC invariant manipulation of bits and notation*

$\text{ALTS}(bs, \text{ALTS}(bs1, rs1), \text{ALTS}(bs2, rs2)) \sim_{m\epsilon} \text{ALTS}(bs, rs1.\text{map}(\text{fuse}(bs1, -)) ++ rs2.\text{map}(\text{fuse}(bs2, -)))$ .

We also use  $bs2 \gg rs2$  as a shorthand notation for  $rs2.\text{map}(\text{fuse}(bs2, -))$ .

**Lemma 5.** *mkeysBC equivalence w.r.t syntactically different regular expressions(2 ALTS)*

$sr_1 = \text{ALTS}(bs1, rs1)$  and  $sr_2 = \text{ALTS}(bs2, rs2)$  we have  $d(sr_1 + sr_2) \sim_{m\epsilon} d(\text{ALTS}(bs, bs1 \gg rs1 ++ bs2 \gg rs2))$

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

**Lemma 6.** *What does dB do to two already simplified ALTS*

$dCo(\text{ALTS}(bs, dB(bs1 \gg rs1 ++ bs2 \gg rs2))) = dCo(\text{ALTS}(bs, bs1 \gg rs1 ++ ((bs2 \gg rs2) - -rs1)))$

*Proof.* We prove that  $dB(bs1 \gg rs1 ++ bs2 \gg rs2) = bs1 \gg rs1 ++ ((bs2 \gg rs2) - -rs1)$ . □

**Lemma 7.** *after opening two previously simplified alts up into terms, length must exceed 2*

If  $sr1, sr2$  are of the form  $\text{ALTS}(bs1, rs1), \text{ALTS}(bs2, rs2)$  respectively, then we have that  $Co(bs, (bs1 \gg rs1) ++ (bs2 \gg rs2) - -rs1) = \text{ALTS}(bs, bs1 \gg rs1 ++ (bs2 \gg rs2) - -rs1)$

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

**Lemma 8.** *mkeysBC equivalence w.r.t syntactically different regular expressions(2 ALTS+ some deletion after derivatives)*

$d\text{ALTS}(bs, bs1 \gg rs1 ++ bs2 \gg rs2) \sim_{m\epsilon} d\text{ALTS}(bs, bs1 \gg rs1 ++ ((bs2 \gg rs2) - -rs1))$

*Proof.* Let's call  $bs1 \gg rs1$   $rs1'$  and  $bs2 \gg rs2$   $rs2'$ . Then we need to prove  $d\text{ALTS}(bs, rs1' ++ rs2') \sim_{m\epsilon} d\text{ALTS}(bs, rs1' ++ (rs2' - -rs1'))$ .

We might as well omit the prime in each  $rs$  for simplicity of notation and prove  $d\text{ALTS}(bs, rs1 ++ rs2) \sim_{m\epsilon} d\text{ALTS}(bs, rs1 ++ (rs2 - -rs1))$ .

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 mkeysBC*

$\text{mkeysBC}r1 \cdot sr2 = \text{mkeysBC}r1 \cdot r2$  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  $s = c_1 \dots c_n$  ( $n \geq 1$ ) where each of the  $c_i$  are characters. Then  $ders\_simp(ar, s) = s(d_{c_n}(\dots s(d_{c_1}(r))\dots)) = sdsd\dots sdr$ . 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  $sdsd\dots sdr \sim_{m\epsilon} ssddsdsd\dots sdr \sim_{m\epsilon} \dots \sim_{m\epsilon} s\dots sd\dots dr$ . Using Lemma 1 we have that  $s\dots sd\dots dr = sd\dots dr$ . By Lemma 2, we have  $RHS \sim_{m\epsilon} d\dots dr$ .

Notice that we don't actually need Lemma 1 here. That is because by Lemma 2, we can have that  $s\dots sd\dots dr \sim_{m\epsilon} sd\dots dr$ . 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  $sr_1$  and  $sr_2$  are both ALTS, so that they will be opened up in the flats function and some terms in  $sr_2$  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, namely,  $dsr_1 + dsr_2 \sim_{m\epsilon} ds(r_1 + r_2)$ .

We have that  $LHS = dsr_1 + dsr_2 = d(sr_1 + sr_2)$ . Since  $sr_1 = ALTS(bs_1, rs_1)$  and  $sr_2 = ALTS(bs_2, rs_2)$  we have  $d(sr_1 + sr_2) \sim_{m\epsilon} d(ALTS(bs, bs_1 \gg rs_1 + bs_2 \gg rs_2))$  by Lemma 5. On the other hand,  $RHS = ds(ALTS(bs, r_1, r_2)) = dCo(bs, dB(flats(s(r_1), s(r_2)))) = dCo(bs, dB(bs_1 \gg rs_1 + bs_2 \gg rs_2))$  by definition of  $bsimp$  and  $flats$ .

$dCo(bs, dB(bs_1 \gg rs_1 + bs_2 \gg rs_2)) = dCo(bs, (bs_1 \gg rs_1 + ((bs_2 \gg rs_2) - -rs_1)))$  by Lemma 6.

$dCo(bs, (bs_1 \gg rs_1 + ((bs_2 \gg rs_2) - -rs_1))) = d(ALTS(bs, bs_1 \gg rs_1 + (bs_2 \gg rs_2) - -rs_1))$  by Lemma 7.

Using Lemma 8, we have  $d(ALTS(bs, bs_1 \gg rs_1 + (bs_2 \gg rs_2) - -rs_1)) \sim_{m\epsilon} d(ALTS(bs, bs_1 \gg rs_1 + bs_2 \gg rs_2)) \sim_{m\epsilon} RHS$ .

This completes the proof.

–  $r^*$

$s(r^*) = r^*$ . Our goal is trivially achieved.

–  $r_1 \cdot r_2$

When  $r_1$  is nullable,  $dsr_1r_2 = dsr_1 \cdot sr_2 + dsr_2 \sim_{m\epsilon} dr_1 \cdot sr_2 + dr_2 = dr_1 \cdot r_2 + dr_2$ . The last step uses Lemma 10.

When  $r_1$  is not nullable,  $dsr_1r_2 = dsr_1 \cdot sr_2 \sim_{m\epsilon} dr_1 \cdot sr_2 \sim_{m\epsilon} dr_1 \cdot r_2$

- 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  $\text{pushbits}(\text{ders\_simp}(ar, s)) == \text{simp}(\text{ders}(ar, s))$  or  $\text{ders\_simp}(ar, s) == \text{simp}(\text{ders}(ar, s))$ . Unfortunately this does not hold. A counterexample is

```

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

```

```

|          L-a List(S)
L-ALT List(S, S, Z, C(b))
  L-SEQ
  | L-ALT List(S)
  | | L-ALT List(Z)
  | | | L-SEQ
  | | | | L-ZERO
  | | | | L-STA
  | | | | L-a
  | | | L-SEQ List(S, C(a))
  | | | L-ONE List(S, C(a))
  | | | L-STA
  | | | L-a
  | | L-ALT List(S)
  | | L-ZERO
  | | L-ZERO
  | L-STA
  | L-ALT
  | L-STA List(Z)
  | | L-a
  | | L-ALT List(S)
  | | | L-b List(Z)
  | | | L-a List(S)
L-SEQ List(S, Z, S, C(a), Z)
  L-ALT List(S)
  | L-SEQ List(Z)
  | | L-ONE List(S, C(a))
  | | L-STA
  | | L-a
  | L-ALT List(S)
  | L-ZERO
  | L-ONE List(S, C(a))
  L-STA
  L-ALT
  L-STA List(Z)
  | L-a
  L-ALT List(S)
  L-b List(Z)
  L-a List(S)

```

regex after ders and then a single simp

ALT

```

L-SEQ List(S, S, Z, C(b))
| L-STA List(S, Z, S, C(a), S, C(a))
| | L-a
| L-STA
| L-ALT
| L-STA List(Z)
| | L-a
| L-ALT List(S)
| L-b List(Z)
| L-a List(S)
L-SEQ List(S, S, Z, C(b), S, Z, S, C(a), Z)
  L-ALT List(S)
  | L-STA List(Z, S, C(a))
  | | L-a
  | L-ONE List(S, S, C(a))
  L-STA
  L-ALT
  L-STA List(Z)
  | L-a

```

L-ALT List(S)  
L-b List(Z)  
L-a List(S)