Dyck Language

Tobias Nipkow and Moritz Roos

September 1, 2025

Abstract

The Dyck language over a pair of brackets, e.g. (and), is the set of balanced strings/words/lists of brackets. That is, the set of words with the same number of (and), where every prefix of the word contains no more) than (. In general, a Dyck language is defined over a whole set of matching pairs of brackets.

Contents

1 Dyck Languages **1**

1.1 Balanced, Inductive and Recursive 1

1.2 Equivalence of *bal* and *bal_stk* 2

1.3 More properties of *bal*, using *bal_stk* 3

1.4 Dyck Language over an Alphabet 5

1 Dyck Languages

theory *Dyck_Language*
imports *Main*
begin

Dyck languages are sets of words/lists of balanced brackets. A bracket is a pair of type *bool* × *'a* where *True* is an opening and *False* a closing bracket. That is, brackets are tagged with elements of type *'a*.

type_synonym *'a bracket* = *bool* × *'a*

abbreviation *Open a* ≡ (*True*,*a*)
abbreviation *Close a* ≡ (*False*,*a*)

1.1 Balanced, Inductive and Recursive

Definition of what it means to be a *balanced* list of brackets:

inductive *bal* :: *'a bracket list* ⇒ *bool* **where**
  bal [] |
  bal xs ⟹ *bal ys* ⟹ *bal* (*xs* @ *ys*) |

```
  bal xs ⟹ bal (Open a # xs @ [Close a])

declare bal.intros(1)[iff] bal.intros(2)[intro,simp] bal.intros(3)[intro,simp]

lemma bal2[iff]: bal [Open a, Close a]
  using bal.intros(3)[of []] by simp
```

The inductive definition of balanced is complemented with a functional version that uses a stack to remember which opening brackets need to be closed:

```
fun bal_stk :: 'a list ⇒ 'a bracket list ⇒ 'a list * 'a bracket list where
  bal_stk s [] = (s,[]) |
  bal_stk s (Open a # bs) = bal_stk (a # s) bs |
  bal_stk (a' # s) (Close a # bs) =
    (if a = a' then bal_stk s bs else (a' # s, Close a # bs)) |
  bal_stk bs stk = (bs,stk)

lemma bal_stk_more_stk: bal_stk s1 xs = (s1',[]) ⟹ bal_stk (s1@s2) xs =
(s1'@s2,[])
by(induction s1 xs arbitrary: s2 rule: bal_stk.induct) (auto split: if_splits)

lemma bal_stk_if_Nils[simp]: ASSUMPTION(bal_stk [] bs = ([], [])) ⟹ bal_stk
s bs = (s, [])
unfolding ASSUMPTION_def using bal_stk_more_stk[of [] _ []] by simp

lemma bal_stk_append:
  bal_stk s (xs @ ys)
   = (let (s',xs') = bal_stk s xs in if xs' = [] then bal_stk s' ys else (s', xs' @ ys))
by(induction s xs rule:bal_stk.induct) (auto split: if_splits)

lemma bal_stk_append_if:
  bal_stk s1 xs = (s2,[]) ⟹ bal_stk s1 (xs @ ys) = bal_stk s2 ys
by(simp add: bal_stk_append[of _ xs])

lemma bal_stk_split:
  bal_stk s xs = (s',xs') ⟹ ∃ us. xs = us@xs' ∧ bal_stk s us = (s',[])
by(induction s xs rule:bal_stk.induct) (auto split: if_splits)
```

1.2 Equivalence of *bal* and *bal_stk*

```
lemma bal_stk_if_bal:  bal xs ⟹ bal_stk s xs = (s,[])
by(induction arbitrary: s rule: bal.induct)(auto simp: bal_stk_append_if split: if_splits)

lemma bal_insert_AB:
  bal (v @ w) ⟹ bal (v @ (Open a # Close a # w))
proof(induction v @ w arbitrary: v w rule: bal.induct)
  case 1 thus ?case by blast
next
  case (3 u b)
```

```
  then show ?case
  proof (cases v)
    case Nil
    hence w = Open b # u @ [Close b]
      using 3.hyps(3) by fastforce
    hence bal w using 3.hyps
      by blast
    hence bal ([Open a, Close a] @ w)
      by blast
    thus ?thesis using Nil by simp
  next
    case [simp]: (Cons x v')
    show ?thesis
    proof (cases w rule:rev_cases)
      case Nil
      from 3.hyps have bal ((Open a # u @ [Close a]) @ [Open a, Close a])
        using bal.intros(2) by blast
      thus ?thesis using Nil Cons 3
        by (metis append_Nil append_Nil2 bal.simps)
    next
      case (snoc w' y)
      thus ?thesis
        using 3.hyps(2,3) bal.intros(3) by force
    qed
  qed
next
  case (2 v' w')
  then obtain r where v'=v@r ∧ r@w'=w ∨ v'@r=v ∧ w'=r@w
    by (meson append_eq_append_conv2)
  thus ?case
    using 2.hyps bal.intros(2) by force
qed

lemma bal_if_bal_stk: bal_stk s w = ([],[]) ⟹ bal (rev(map (λx. Open x) s) @
w)
proof(induction s w rule: bal_stk.induct)
  case 2
  then show ?case by simp
next
  case 3
  then show ?case by (auto simp add: bal_insert_AB split: if_splits)
qed (auto)

corollary bal_iff_bal_stk: bal w ⟷ bal_stk [] w = ([],[])
using bal_if_bal_stk[of []] bal_stk_if_bal by auto
```

1.3 More properties of *bal*, using *bal_stk*

```
theorem bal_append_inv: bal (u @ v) ⟹ bal u ⟹ bal v
```

```
  using bal_stk_append_if bal_iff_bal_stk by metis

lemma bal_insert_bal_iff[simp]:
  bal b ⟹ bal (v @ b @ w) = bal (v@w)
unfolding bal_iff_bal_stk by(auto simp add: bal_stk_append split: prod.splits
if_splits)

lemma bal_start_Open: ‹bal (x#xs) ⟹∃ a. x = Open a›
  using bal_stk.elims bal_iff_bal_stk by blast

lemma bal_Open_split: assumes ‹bal (x # xs)›
  shows ‹∃ y r a. bal y ∧ bal r ∧ x = Open a ∧ xs = y @ Close a # r›
proof−
  from assms obtain a where ‹x = Open a›
    using bal_start_Open by blast
  have ‹bal (Open a # xs) ⟹ ∃ y r. bal y ∧ bal r ∧ xs = y @ Close a # r›
  proof(induction ‹length xs› arbitrary: xs rule: less_induct)
    case less
    have IH: ‹⋀w. ⟦length w < length xs; bal (Open a # w)⟧ ⟹ ∃ y r. bal y ∧ bal
r ∧ w = y @ Close a # r›
      using less by blast
    have ‹bal (Open a # xs)›
      using less by blast
    from less(2) show ?case
    proof(induction ‹Open a # xs› rule: bal.induct)
      case (2 as bs)
       consider (as_empty) ‹as = []› | (bs_empty) ‹bs = []› | (both_not_empty)
‹as ≠ [] ∧ bs ≠ []› by blast
      then show ?case
      proof(cases)
        case as_empty
        then show ?thesis using 2 by (metis append_Nil)
      next
        case bs_empty
        then show ?thesis using 2 by (metis append_self_conv)
      next
        case both_not_empty
        then obtain as' where as'_def: ‹Open a # as' = as›
          using 2 by (metis append_eq_Cons_conv)
        then have ‹length as' < length xs›
          using 2.hyps(5) both_not_empty by fastforce
        with IH ‹bal as› obtain y r where yr: ‹bal y ∧ bal r ∧ as' = y @ Close a
# r›
          using as'_def by meson
        then have ‹xs = y @ Close a # r @ bs›
          using 2.hyps(5) as'_def by fastforce
        moreover have ‹bal y›
          using yr by blast
        moreover have ‹bal (r@bs)›
```

```
        using yr by (simp add: 2.hyps(3))
      ultimately show ?thesis by blast
    qed
  next
    case (3 xs)
    then show ?case by blast
  qed
 qed
 then show ?thesis using assms ‹x = _› by blast
qed
```

1.4 Dyck Language over an Alphabet

The Dyck/bracket language over a set Γ is the set of balanced words over Γ:

```
definition Dyck_lang :: 'a set ⇒ 'a bracket list set where
Dyck_lang Γ = {w. bal w ∧ snd ` (set w) ⊆ Γ}

lemma Dyck_langI[intro]:
  assumes ‹bal w›
    and ‹snd ` (set w) ⊆ Γ›
  shows ‹w ∈ Dyck_lang Γ›
  using assms unfolding Dyck_lang_def by blast

lemma Dyck_langD[dest]:
  assumes ‹w ∈ Dyck_lang Γ›
  shows ‹bal w›
    and ‹snd ` (set w) ⊆ Γ›
  using assms unfolding Dyck_lang_def by auto
```

Balanced subwords are again in the Dyck Language.

```
lemma Dyck_lang_substring:
  ‹bal w ⟹ u @ w @ v ∈ Dyck_lang Γ ⟹ w ∈ Dyck_lang Γ›
unfolding Dyck_lang_def by auto

end
```