

Gale-Stewart Games

Sebastiaan J. C. Joosten

May 16, 2026

Abstract

This is a formalisation of the main result of Gale and Stewart from 1953, showing that closed finite games are determined. This property is now known as the Gale Stewart Theorem. While the original paper shows some additional theorems as well, we only formalize this main result, but do so in a somewhat general way. We formalize games of a fixed arbitrary length, including infinite length, using co-inductive lists, and show that defensive strategies exist unless the other player is winning. For closed games, defensive strategies are winning for the closed player, proving that such games are determined. For finite games, which are a special case in our formalisation, all games are closed.

Contents

1	Introduction	1
2	Alternating lists	2
3	Gale Stewart Games	5
3.1	Basic definitions and their properties.	5
3.2	Winning strategies	10
3.3	Defensive strategies	13
3.4	Determined games	19

1 Introduction

The original paper from Gale and Stewart [2] uses a function to point to a previous position. This encoding of sequences is not followed in this formalization, as it is not the way we think of games these days. Instead, we follow the approach taken in the formalization of Parity Games [1], where co-inductive lists are used to talk about possibly infinite plays. Although we rely on the Parity Games theory for some of the theorems about co-inductive lists, none of the notions about games are shared with that formalization.

We have proven some basic lemmas about prefixes, extended naturals (natural numbers plus infinity), and defined a function 'alternate' alternating lists. We have done this in separate Isabelle theory files, so that they can be reused independently without depending on the formalizations of infinite games presented here. In the same way this formalization is giving a nod to the parity games formalization. In this document, we only present the alternating lists, as this theory file contains new definitions, which are relevant preliminaries to know about. The additional lemmas about prefixes and extended natural numbers are less essential, they only contain 'obvious' properties, so we have left those theory files out of this document.

2 Alternating lists

In lists where even and odd elements play different roles, it helps to define functions to take out the even elements. We defined the function $(l)alternate$ on (coinductive) lists to do exactly this, and define certain properties.

```
theory AlternatingLists
imports MoreCoinductiveList2
begin
```

The functions “alternate” and “lalternate” are our main workhorses: they take every other item, so every item at even indices.

```
fun alternate where
  alternate Nil = Nil |
  alternate (Cons x xs) = Cons x (alternate (tl xs))
```

“lalternate” takes every other item from a co-inductive list.

```
primcorec lalternate :: 'a llist  $\Rightarrow$  'a llist
where
  lalternate xs = (case xs of LNil  $\Rightarrow$  LNil |
                    (LCons x xs)  $\Rightarrow$  LCons x (lalternate (ltl xs)))
```

lemma *lalternate-ltake*:

```
ltake (enat n) (lalternate xs) = lalternate (ltake (2*n) xs)
```

proof(*induct n arbitrary:xs*)

```
case 0
```

```
then show ?case by (metis LNil-eq-ltake-iff enat-defs(1) lalternate.ctr(1) lnull-def mult-zero-right)
```

```
next
```

```
case (Suc n)
```

```
hence lt:ltake (enat n) (lalternate (ltl (ltl xs))) = lalternate (ltake (enat (2 * n)) (ltl (ltl xs))).
```

```
show ?case
```

```
proof(cases lalternate xs)
```

```
case LNil
```

```
then show ?thesis by(metis lalternate.disc(2) lnull-def ltake-LNil)
```

```

next
  case (LCons x21 x22)
  thus ?thesis unfolding ltake-ltl mult-Suc-right add-2-eq-Suc
    using eSuc-enat lalternate.code lalternate.ctr(1) lhd-LCons-ltl llist.sel(1)
    by (smt (verit) lt ltake-ltl llist.simps(3) llist.simps(5) ltake-eSuc-LCons)
qed
qed

lemma lalternate-llist-of[simp]:
  lalternate (llist-of xs) = llist-of (alternate xs)
proof(induct alternate xs arbitrary:xs)
  case Nil
  then show ?case
    by (metis alternate.elims lalternate.ctr(1) list.simps(3) llist-of.simps(1) lnull-llist-of)
next
  case (Cons a xs)
  then show ?case by(cases xs,auto simp: lalternate.ctr)
qed

lemma lalternate-finite-helper:
  assumes lfinite (lalternate xs)
  shows lfinite xs
using assms proof(induct lalternate xs arbitrary:xs rule:lfinite-induct)
  case LNil
  then show ?case unfolding lalternate.code[of xs] by(cases xs;auto)
next
  case (LCons xs)
  then show ?case unfolding lalternate.code[of xs] by(cases xs;cases ltl xs;auto)
qed

lemma alternate-list-of:
  assumes lfinite xs
  shows alternate (list-of xs) = list-of (lalternate xs)
  using assms by (metis lalternate-llist-of list-of-llist-of llist-of-list-of)

lemma alternate-length:
  length (alternate xs) = (1+length xs) div 2
  by (induct xs rule:induct-list012;simp)

lemma lalternate-llength:
  llength (lalternate xs) * 2 = (1+llength xs) ∨ llength (lalternate xs) * 2 = llength
  xs
proof(cases lfinite xs)
  case True
  let ?xs = list-of xs
  have length (alternate ?xs) = (1+length ?xs) div 2 using alternate-length by
  auto
  hence length (alternate ?xs) * 2 = (1+length ?xs) ∨ length (alternate ?xs) * 2
  = length ?xs

```

```

    by auto
  then show ?thesis using alternate-list-of[OF True] lalternate-llist-of True
    length-list-of-conv-the-enat[OF True] llist-of-list-of[OF True]
    by (metis llength-llist-of numeral-One of-nat-eq-enat of-nat-mult of-nat-numeral
plus-enat-simps(1))
next
  case False
  have ¬ lfinite (lalternate xs) using False lalternate-finite-helper by auto
  hence l1:length (lalternate xs) = ∞ by (rule not-lfinite-llength)
  from False have l2:length xs = ∞ using not-lfinite-llength by auto
  show ?thesis using l1 l2 by (simp add: mult-2-right)
qed

```

```

lemma lalternate-finite[simp]:
  shows lfinite (lalternate xs) = lfinite xs
proof(cases lfinite xs)
  case True
  then show ?thesis
  proof(cases lfinite (lalternate xs))
    case False
    hence False using not-lfinite-llength[OF False] True[unfolded lfinite-conv-llength-enat]
      lalternate-llength[of xs]
      by (auto simp: one-enat-def numeral-eq-enat)
    thus ?thesis by metis
  qed auto
next
  case False
  then show ?thesis using lalternate-finite-helper by blast
qed

```

```

lemma nth-alternate:
  assumes 2*n < length xs
  shows alternate xs ! n = xs ! (2 * n)
  using assms proof (induct xs arbitrary: n rule: induct-list012)
  case (3 x y zs)
  then show ?case proof(cases n)
    case (Suc nat)
    show ?thesis using 3.hyps(1) 3.prem1 Suc by force
  qed simp
qed auto

```

```

lemma lnth-lalternate:
  assumes 2*n < llength xs
  shows lalternate xs $ n = xs $ (2 * n)
proof -
  let ?xs = ltake (2*Suc n) xs
  have lalternate ?xs $ n = ?xs $ (2 * n)
    using assms alternate-list-of[of ltake (2*Suc n) xs] nth-alternate[of n list-of ?xs]
  by (smt (verit) Suc-1 Suc-mult-less-cancel1 enat-ord-simps(2) infinite-small-llength

```

*lalternate-ltake length-list-of lessI llength-eq-enat-lfiniteD llength-ltake' ltake-all not-less
nth-list-of numeral-eq-enat the-enat.simps times-enat-simps(1)*

thus *?thesis*

by (*metis Suc-1 Suc-mult-less-cancel1 enat-ord-simps(2) lalternate-ltake lessI
lnth-ltake*)

qed

lemma *lnth-lalternate2[simp]*:

assumes $n < \text{length } (lalternate \ xs)$

shows $lalternate \ xs \ \$ \ n = xs \ \$ \ (2 * n)$

proof –

from *assms* **have** $2 * enat \ n < \text{length } xs$

by (*metis enat-numeral lalternate-ltake leI linorder-neq-iff llength-ltake' ltake-all
times-enat-simps(1)*)

from *lnth-lalternate[OF this]* **show** *?thesis*.

qed

end

3 Gale Stewart Games

Gale Stewart Games are infinite two player games.

theory *GaleStewartGames*

imports *AlternatingLists MorePrefix MoreENat*

begin

3.1 Basic definitions and their properties.

A GSgame $G(A)$ is defined by a set of sequences that denote the winning games for the first player. Our notion of GSgames generalizes both finite and infinite games by setting a game length. Note that the type of n is 'enat' (extended nat): either a nonnegative integer or infinity. Our only requirement on GSgames is that the winning games must have the length as specified as the length of the game. This helps certain theorems about winning look a bit more natural.

locale *GSgame* =

fixes $A \ N$

assumes $\text{length} : \forall e \in A. \text{length } e = 2 * N$

begin

A position is a finite sequence of valid moves.

definition *position* **where**

position ($e :: 'a \text{ list}$) $\equiv \text{length } e \leq 2 * N$

lemma *position-maxlength-cannotbe-augmented*:

assumes $\text{length } p = 2 * N$

shows \neg *position* ($p @ [m]$)
by (*auto simp:position-def assms[symmetric]*)

A play is a sequence of valid moves of the right length.

definition *play where*
play ($e::'a$ *llist*) \equiv *length* $e = 2*N$

lemma *plays-are-positions-conv:*
shows *play* (*llist-of* p) \longleftrightarrow *position* $p \wedge$ *length* $p = 2*N$
unfolding *play-def position-def* **by** *auto*

lemma *finite-plays-are-positions:*
assumes *play* p *lfinite* p
shows *position* (*list-of* p)
using *assms*
unfolding *play-def position-def* **by** (*cases lfinite p;auto simp:length-list-of*)

end

We call our players Even and Odd, where Even makes the first move. This means that Even is to make moves on plays of even length, and Odd on the others. This corresponds nicely to Even making all the moves in an even position, as the 'nth' and 'lnth' functions as predefined in Isabelle's library count from 0. In literature the players are sometimes called I and II.

A strategy for Even/Odd is simply a function that takes a position of even/odd length and returns a move. We use total functions for strategies. This means that their Isabelle-type determines that it is a strategy. Consequently, we do not have a definition of 'strategy'. Nevertheless, we will use σ as a letter to indicate when something is a strategy. We can combine two strategies into one function, which gives a collective strategy that we will refer to as the joint strategy.

definition *joint-strategy* $:: ('b$ *list* $\Rightarrow 'a) \Rightarrow ('b$ *list* $\Rightarrow 'a) \Rightarrow ('b$ *list* $\Rightarrow 'a)$ **where**
joint-strategy $\sigma_e \sigma_o p =$ (*if even* (*length* p) *then* $\sigma_e p$ *else* $\sigma_o p$)

Following a strategy leads to an infinite sequence of moves. Note that we are not in the context of 'GSGame' where 'N' determines the length of our plays: we just let sequences go on ad infinitum here. Rather than reasoning about our own recursive definitions, we build this infinite sequence by reusing definitions that are already in place. We do this by first defining all prefixes of the infinite sequence we are interested in. This gives an infinite list such that the nth element is of length n. Note that this definition allows us to talk about how a strategy would continue if it were played from an arbitrary position (not necessarily one that is reached via that strategy).

definition *strategy-progression where*
strategy-progression $\sigma p =$ *lappend* (*llist-of* (*prefixes* p)) (*ltl* (*iterates* (*augment-list* σ) p))

lemma *induced-play-infinite*:
 \neg *lfinite* (*strategy-progression* σ p)
unfolding *strategy-progression-def* **by** *auto*

lemma *plays-from-strategy-lengths*[*simp*]:
 $\text{length } (\text{strategy-progression } \sigma \ p \ \$ \ i) = i$
proof(*induct* i)
case 0
then show *?case* **by**(*cases* p ; *auto simp: strategy-progression-def lnth-lappend take-map ltake-lappend*)
next
case (*Suc* i)
then show *?case*
by (*cases* $i < \text{length } p$) (*auto simp: strategy-progression-def lnth-lappend length-augment-list tl-prefixes-idx*)
qed

lemma *length-plays-from-strategy*[*simp*]:
 $\text{length } (\text{strategy-progression } \sigma \ p) = \infty$
unfolding *strategy-progression-def* **by** *auto*

lemma *length-ltl-plays-from-strategy*[*simp*]:
 $\text{length } (\text{ltl } (\text{strategy-progression } \sigma \ p)) = \infty$
unfolding *strategy-progression-def* **by** *auto*

lemma *plays-from-strategy-chain-Suc*:
shows *prefix* (*strategy-progression* σ $p \ \$ \ n$) (*strategy-progression* σ $p \ \$ \ \text{Suc } n$)
unfolding *strategy-progression-def*
by (*auto simp: take-Suc-prefix nth-prefixes lnth-lappend nth-prefixes-is-prefix-tl augment-list-prefix*)

lemma *plays-from-strategy-chain*:
shows $n \leq m \implies \text{prefix } (\text{strategy-progression } \sigma \ p \ \$ \ n) (\text{strategy-progression } \sigma \ p \ \$ \ m)$
proof (*induct* $m - n$ *arbitrary: m n*)
case (*Suc* x)
hence [*simp*]: *Suc* ($x + n$) = m **by** *auto*
from *Suc.hyps*(2)
 $\text{prefix-order.trans}[OF \ \text{Suc.hyps}(1)[of \ x + n \ n] \ \text{plays-from-strategy-chain-Suc}[of \ - \ x + n]]$
show *?case* **by** *auto*
qed *auto*

lemma *plays-from-strategy-remains-const*:
assumes $n \leq i$
shows $\text{take } n \ (\text{strategy-progression } \sigma \ p \ \$ \ i) = \text{strategy-progression } \sigma \ p \ \$ \ n$
apply(*rule sym, subst prefix-same-length-eq[symmetric]*)
using *assms plays-from-strategy-chain[OF assms]*
by (*auto intro!: prefix-takeI*)

```

lemma infplays-augment-one[simp]:
  strategy-progression  $\sigma$  ( $p @ [\sigma p]$ ) = strategy-progression  $\sigma$   $p$ 
proof(induct  $p$ )
  note defs = strategy-progression-def
  {
    case Nil
    then show ?case
      by (auto simp: defs iterates.code[of - [ $\sigma$  []]])
  }
  next
  case (Cons  $a$   $p$ )
  then show ?case
    by (auto simp: defs iterates.code[of -  $a \# p @ [\sigma (a \# p)]$ ] lappend-llist-of-LCons)
  }
qed

```

```

lemma infplays-augment-many[simp]:
  strategy-progression  $\sigma$  ((augment-list  $\sigma$   $\sim n$ )  $p$ ) = strategy-progression  $\sigma$   $p$ 
by(induct  $n$ ,auto)

```

```

lemma infplays-augment-one-joint[simp]:
  even (length  $p$ )  $\implies$  strategy-progression (joint-strategy  $\sigma_e$   $\sigma_o$ ) (augment-list  $\sigma_e$ 
 $p$ )
  = strategy-progression (joint-strategy  $\sigma_e$   $\sigma_o$ )  $p$ 
  odd (length  $p$ )  $\implies$  strategy-progression (joint-strategy  $\sigma_e$   $\sigma_o$ ) (augment-list  $\sigma_o$   $p$ )
  = strategy-progression (joint-strategy  $\sigma_e$   $\sigma_o$ )  $p$ 
using infplays-augment-one[of joint-strategy  $\sigma_e$   $\sigma_o$   $p$ ]
unfolding joint-strategy-def by auto

```

Following two different strategies from a single position will lead to the same plays if the strategies agree on moves played after that position. This lemma allows us to ignore the behavior of strategies for moves that are already played.

```

lemma infplays-eq:
  assumes  $\bigwedge p'. \text{prefix } p p' \implies \text{augment-list } s1 p' = \text{augment-list } s2 p'$ 
  shows strategy-progression  $s1$   $p = \text{strategy-progression } s2$   $p$ 
proof –
  from assms[of  $p$ ] have [intro]: $s1 p = s2 p$  by auto
  have (augment-list  $s1$   $\sim n$ ) (augment-list  $s1$   $p$ ) =
    (augment-list  $s2$   $\sim n$ ) (augment-list  $s2$   $p$ ) for  $n$ 
  proof(induct  $n$ )
  case (Suc  $n$ )
  with assms[OF prefix-order.trans[OF - prefix-augment]]
  show ?case by (auto)
  qed auto
  hence strategy-progression  $s1$   $p \$ n = \text{strategy-progression } s2$   $p \$ n$ 
  for  $n$  unfolding strategy-progression-def lnth-lappend by auto
  thus ?thesis by(intro coinductive-eq-I,auto)
qed

```

context *GSgame*
begin

By looking at the last elements of the infinite progression, we can get a single sequence, which we trim down to the right length. Since it has the right length, this always forms a play. We therefore name this the 'induced play'.

definition *induced-play* **where**

induced-play $\sigma \equiv \text{ltake } (2*N) \circ \text{lmap last} \circ \text{ltl} \circ \text{strategy-progression } \sigma$

lemma *induced-play-infinite-le[simp]*:

enat $x < \text{llength } (\text{strategy-progression } \sigma \ p)$

enat $x < \text{llength } (\text{lmap } f \ (\text{strategy-progression } \sigma \ p))$

enat $x < \text{llength } (\text{ltake } (2*N) \ (\text{lmap } f \ (\text{strategy-progression } \sigma \ p))) \longleftrightarrow x < 2*N$

using *induced-play-infinite* **by** *auto*

lemma *induced-play-is-lprefix*:

assumes *position* p

shows *lprefix* (*l*list-of p) (*induced-play* $\sigma \ p$)

proof –

have *l*:*llength* (*l*list-of p) $\leq 2 * N$ **using** *assms* **unfolding** *position-def* **by** *auto*

have *lprefix* (*l*list-of p) (*lmap last* (*l*tl (*l*list-of (*prefixes* p)))) **by** *auto*

hence *lprefix* (*l*list-of p) ((*lmap last* \circ *l*tl \circ *strategy-progression* σ) p)

unfolding *strategy-progression-def* **by**(*auto simp add: lmap-lappend-distrib lprefix-lappend*)

thus *?thesis* **unfolding** *induced-play-def o-def*

using *lprefix-ltakeI[OF - l]* **by** *blast*

qed

lemma *length-induced-play[simp]*:

llength (*induced-play* $s \ p$) = $2 * N$

unfolding *induced-play-def* **by** *auto*

lemma *induced-play-lprefix-non-positions*:

assumes *length* ($p::'a \ \text{list}$) $\geq 2 * N$

shows *induced-play* $\sigma \ p = \text{ltake } (2 * N) \ (\text{llist-of p)$

proof(*cases* N)

case (*enat* nat)

let $?p = \text{take } (2 * \text{nat}) \ p$

from *assms* **have** [*intro*]: $2 * N \leq \text{enat } (\text{length } p)$ **by** *auto*

have [*intro*]: $2 * N \leq \text{enat } (\text{min } (\text{length } p) \ (2 * \text{nat}))$ **unfolding** *enat*

by (*metis* *assms enat min.orderI min-def numeral-eq-enat times-enat-simps(1)*)

have [*intro*]:*enat* ($\text{min } (\text{length } p) \ (2 * \text{nat})) = 2 * N$

by (*metis* (*mono-tags, lifting*) *assms enat min.absorb2 min-enat-simps(1)*)

numeral-eq-enat times-enat-simps(1))

have $n:2 * N \leq \text{llength } (\text{llist-of p) $2 * N \leq \text{llength } (\text{llist-of ($\text{take } (2 * \text{nat}) \ p$))$$

by *auto*

have *pp:position* $?p$

apply(*subst position-def*)
by (*metis (no-types, lifting) assms dual-order.order-iff-strict enat llength-llist-of llength-ltake' ltake-llist-of numeral-eq-enat take-all-times-enat-simps(1)*)
have *lp:lprefix (llist-of ?p) (induced-play σ ?p)* **by**(*rule induced-play-is-lprefix[OF pp]*)

have *ltake (2 * N) (llist-of p) = ltake (2 * N) (llist-of (take (2 * nat) p))*
unfolding *ltake-llist-of[symmetric] enat ltake-ltake numeral-eq-enat* **by** *auto*
hence *eq:induced-play σ p = induced-play σ ?p*
unfolding *induced-play-def strategy-progression-def*
by(*auto simp add: lmap-lappend-distrib n[THEN ltake-lappend1]*)
have *llist-of (take (2 * nat) p) = induced-play σ p*
by(*rule lprefix-llength-eq-imp-eq[OF lp[folded eq]],auto*)
then show *?thesis*
unfolding *enat ltake-llist-of[symmetric]*
numeral-eq-enat times-enat-simps(1) **by** *metis*

next
case *infinity*
hence $2 * N = \infty$ **by** (*simp add: imult-is-infinity*)
then show *?thesis using assms* **by** *auto*
qed

lemma *infplays-augment-many-lprefix[simp]:*
shows *lprefix (llist-of ((augment-list σ \sim n) p)) (induced-play σ p)*
= position ((augment-list σ \sim n) p) (is ?lhs = ?rhs)

proof
assume *?lhs*
from *lprefix-llength-le[OF this]* **show** *?rhs* **unfolding** *induced-play-def*
by (*auto simp:position-def length-augment-list*) **next**
assume *assm: ?rhs*
from *induced-play-is-lprefix[OF this, of σ]*
show *?lhs* **unfolding** *induced-play-def* **by** *simp*
qed

3.2 Winning strategies

A strategy is winning (in position p) if, no matter the moves by the other player, it leads to a sequence in the winning set.

definition *strategy-winning-by-Even* **where**
strategy-winning-by-Even $\sigma_e p \equiv (\forall \sigma_o. \text{induced-play (joint-strategy } \sigma_e \sigma_o) p \in A)$

definition *strategy-winning-by-Odd* **where**
strategy-winning-by-Odd $\sigma_o p \equiv (\forall \sigma_e. \text{induced-play (joint-strategy } \sigma_e \sigma_o) p \notin A)$

It immediately follows that not both players can have a winning strategy.

lemma *at-most-one-player-winning:*
shows $\neg (\exists \sigma_e. \text{strategy-winning-by-Even } \sigma_e p) \vee \neg (\exists \sigma_o. \text{strategy-winning-by-Odd } \sigma_o p)$

unfolding *strategy-winning-by-Even-def strategy-winning-by-Odd-def* **by** *auto*

If a player whose turn it is not makes any move, winning strategies remain winning. All of the following proofs are duplicated for Even and Odd, as the game is entirely symmetrical. These 'dual' theorems can be obtained by considering a game in which an additional first and final move are played yet ignored, but it is quite convenient to have both theorems at hand regardless, and the proofs are quite small, so we accept the code duplication.

lemma *any-moves-remain-winning-Even:*

assumes *odd (length p) strategy-winning-by-Even σ p*

shows *strategy-winning-by-Even σ (p @ [m])*

unfolding *strategy-winning-by-Even-def* **proof**

fix σ_o

let $?s = \sigma_o(p:=m)$

have *prfx:prefix (p @ [m]) p' \implies*

p' @ [joint-strategy σ σ_o p'] = p' @ [joint-strategy σ ?s p']

for p' **by** *(auto simp: joint-strategy-def)*

from *assms(2)[unfolded strategy-winning-by-Even-def,rule-format,of ?s]*

infpays-augment-one-joint(2)[OF assms(1)]

have *induced-play (joint-strategy σ ?s) (augment-list ?s p) $\in A$*

by *(metis (mono-tags, lifting) induced-play-def comp-apply)*

thus *induced-play (joint-strategy σ σ_o) (p @ [m]) $\in A$*

unfolding *induced-play-def o-def*

using *infpays-eq[OF prfx]* **by** *auto*

qed

lemma *any-moves-remain-winning-Odd:*

assumes *even (length p) strategy-winning-by-Odd σ p*

shows *strategy-winning-by-Odd σ (p @ [m])*

unfolding *strategy-winning-by-Odd-def* **proof**

fix σ_e

let $?s = \sigma_e(p:=m)$

have *prfx:prefix (p @ [m]) p' \implies*

p' @ [joint-strategy σ_e σ p'] = p' @ [joint-strategy ?s σ p']

for p' **by** *(auto simp: joint-strategy-def)*

from *assms(2)[unfolded strategy-winning-by-Odd-def,rule-format,of ?s]*

infpays-augment-one-joint(1)[OF assms(1)]

have *induced-play (joint-strategy ?s σ) (augment-list ?s p) $\notin A$*

by *(metis (mono-tags, lifting) induced-play-def comp-apply)*

thus *induced-play (joint-strategy σ_e σ) (p @ [m]) $\notin A$*

unfolding *induced-play-def o-def*

using *infpays-eq[OF prfx]* **by** *auto*

qed

If a player does not have a winning strategy, a move by that player will not give it one.

lemma *non-winning-moves-remains-non-winning-Even:*

assumes *even (length p) $\forall \sigma. \neg$ strategy-winning-by-Even σ p*

shows \neg *strategy-winning-by-Even* σ (p @ [m])
proof(*rule contrapos-nn*[*of* $\exists \sigma$. *strategy-winning-by-Even* σ p])
assume a :*strategy-winning-by-Even* σ (p @ [m])
let $?s = \sigma(p:=m)$
have *prfx:prefix* (p @ [m]) $p' \implies$
 $p' @ [joint-strategy \sigma \sigma_o p'] = p' @ [joint-strategy ?s \sigma_o p']$
for $p' \sigma_o$ **by** (*auto simp:joint-strategy-def*)
from a *infplays-eq*[*OF prfx*]
have *strategy-winning-by-Even* $?s$ (p @ [m])
unfolding *strategy-winning-by-Even-def induced-play-def* **by** *simp*
hence *strategy-winning-by-Even* $?s$ p
using *infplays-augment-one-joint(1)*[*OF assms(1)*]
unfolding *strategy-winning-by-Even-def induced-play-def o-def*
by (*metis fun-upd-same*)
thus $\exists \sigma$. *strategy-winning-by-Even* σ p **by** *blast next*
from *assms(2)* **show** \neg ($\exists \sigma$. *strategy-winning-by-Even* σ p) **by** *meson*
qed

lemma *non-winning-moves-remains-non-winning-Odd*:
assumes *odd* (*length* p) $\forall \sigma$. \neg *strategy-winning-by-Odd* σ p
shows \neg *strategy-winning-by-Odd* σ (p @ [m])
proof(*rule contrapos-nn*[*of* $\exists \sigma$. *strategy-winning-by-Odd* σ p])
assume a :*strategy-winning-by-Odd* σ (p @ [m])
let $?s = \sigma(p:=m)$
have *prfx:prefix* (p @ [m]) $p' \implies$
 $p' @ [joint-strategy \sigma_e \sigma p'] = p' @ [joint-strategy \sigma_e ?s p']$
for $p' \sigma_e$ **by** (*auto simp:joint-strategy-def*)
from a *infplays-eq*[*OF prfx*]
have *strategy-winning-by-Odd* $?s$ (p @ [m])
unfolding *strategy-winning-by-Odd-def induced-play-def* **by** *simp*
hence *strategy-winning-by-Odd* $?s$ p
using *infplays-augment-one-joint(2)*[*OF assms(1)*]
unfolding *strategy-winning-by-Odd-def induced-play-def o-def*
by (*metis fun-upd-same*)
thus $\exists \sigma$. *strategy-winning-by-Odd* σ p **by** *blast next*
from *assms(2)* **show** \neg ($\exists \sigma$. *strategy-winning-by-Odd* σ p) **by** *meson*
qed

If a player whose turn it is makes a move according to its strategy, the new position will remain winning.

lemma *winning-moves-remain-winning-Even*:
assumes *even* (*length* p) *strategy-winning-by-Even* σ p
shows *strategy-winning-by-Even* σ (p @ [σ p])
using *assms infplays-augment-one*
unfolding *induced-play-def strategy-winning-by-Even-def* **by** *auto*

lemma *winning-moves-remain-winning-Odd*:
assumes *odd* (*length* p) *strategy-winning-by-Odd* σ p
shows *strategy-winning-by-Odd* σ (p @ [σ p])

using *assms infplays-augment-one*
unfolding *induced-play-def strategy-winning-by-Odd-def* **by** *auto*

We speak of winning positions as those positions in which the player has a winning strategy. This is mainly for presentation purposes.

abbreviation *winning-position-Even* **where**
winning-position-Even $p \equiv \text{position } p \wedge (\exists \sigma. \text{strategy-winning-by-Even } \sigma \ p)$

abbreviation *winning-position-Odd* **where**
winning-position-Odd $p \equiv \text{position } p \wedge (\exists \sigma. \text{strategy-winning-by-Odd } \sigma \ p)$

lemma *winning-position-can-remain-winning-Even*:
assumes *even* (*length* p) $\forall m. \text{position } (p \ @ \ [m]) \text{ winning-position-Even } p$
shows $\exists m. \text{winning-position-Even } (p \ @ \ [m])$
using *assms winning-moves-remain-winning-Even*[*OF assms(1)*] **by** *auto*

lemma *winning-position-can-remain-winning-Odd*:
assumes *odd* (*length* p) $\forall m. \text{position } (p \ @ \ [m]) \text{ winning-position-Odd } p$
shows $\exists m. \text{winning-position-Odd } (p \ @ \ [m])$
using *assms winning-moves-remain-winning-Odd*[*OF assms(1)*] **by** *auto*

lemma *winning-position-will-remain-winning-Even*:
assumes *odd* (*length* p) *position* ($p \ @ \ [m]$) *winning-position-Even* p
shows *winning-position-Even* ($p \ @ \ [m]$)
using *assms any-moves-remain-winning-Even*[*OF assms(1)*] **by** *auto*

lemma *winning-position-will-remain-winning-Odd*:
assumes *even* (*length* p) *position* ($p \ @ \ [m]$) *winning-position-Odd* p
shows *winning-position-Odd* ($p \ @ \ [m]$)
using *assms any-moves-remain-winning-Odd*[*OF assms(1)*] **by** *auto*

lemma *induced-play-eq*:
assumes $\forall p'. \text{prefix } p \ p' \longrightarrow (\text{augment-list } s1) \ p' = (\text{augment-list } s2) \ p'$
shows *induced-play* $s1 \ p = \text{induced-play } s2 \ p$
unfolding *induced-play-def* **by** (*auto simp:infplays-eq*[*OF assms[rule-format]*])

end

end

3.3 Defensive strategies

A strategy is defensive if a player can avoid reaching winning positions. If the opponent is not already in a winning position, such defensive strategies exist. In closed games, a defensive strategy is winning for the closed player, so these strategies are a crucial step towards proving that such games are determined.

theory *GaleStewartDefensiveStrategies*
imports *GaleStewartGames*

```

begin

context GGame
begin

definition move-defensive-by-Even where
  move-defensive-by-Even  $m\ p \equiv \text{even } (\text{length } p) \longrightarrow \neg \text{winning-position-Odd } (p @ [m])$ 
definition move-defensive-by-Odd where
  move-defensive-by-Odd  $m\ p \equiv \text{odd } (\text{length } p) \longrightarrow \neg \text{winning-position-Even } (p @ [m])$ 

lemma defensive-move-exists-for-Even:
assumes [intro]:position  $p$ 
shows winning-position-Odd  $p \vee (\exists m. \text{move-defensive-by-Even } m\ p)$  (is  $?w \vee ?d$ )
proof(cases length  $p = 2*N \vee \text{odd } (\text{length } p)$ )
  case False
  hence pl[intro]:length  $p < 2*N$ 
    and ev[intro]:even (length  $p$ ) using asms[unfolded position-def] by auto
  show ?thesis proof(rule impI[of  $\neg ?d \longrightarrow \neg ?w \longrightarrow \text{False}$ , rule-format], force)
    assume not-def: $\neg ?d$ 
    from not-def[unfolded move-defensive-by-Even-def]
    have  $\forall m. \exists \sigma. \text{strategy-winning-by-Odd } \sigma (p @ [m])$  by blast
    from choice[OF this] obtain  $\sigma_o$  where
      str-Odd: $\wedge m. \text{strategy-winning-by-Odd } (\sigma_o\ m) (p @ [m])$  by blast
    define  $\sigma$  where  $\sigma\ p' = \sigma_o (p' ! \text{length } p)\ p'$  for  $p'$ 
    assume not-win: $\neg ?w$ 
    from not-win[unfolded move-defensive-by-Even-def strategy-winning-by-Odd-def]
    obtain  $\sigma_e$  where
      str-Even:induced-play (joint-strategy  $\sigma_e\ \sigma$ )  $p \in A$ 
      (is  $?pe\ p \in A$ )
      by blast
    let  $?pnext = (p @ [\text{joint-strategy } \sigma_e\ \sigma\ p])$ 
    { fix  $p'\ m$ 
      assume prefix (p @ [m])  $p'$ 
      hence  $(p' ! \text{length } p) = m$ 
        unfolding prefix-def by auto
    }
    hence eq-a: $\forall p'. \text{prefix } ?pnext\ p' \longrightarrow p' @ [\text{joint-strategy } \sigma_e\ \sigma\ p'] =$ 
       $p' @ [\text{joint-strategy } \sigma_e (\sigma_o (\text{joint-strategy } \sigma_e\ \sigma\ p))\ p']$ 
      unfolding joint-strategy-def  $\sigma$ -def by auto
    have simps: $?pe\ p = ?pe (p @ [\text{joint-strategy } \sigma_e\ \sigma\ p])$ 
      unfolding induced-play-def by auto
    from str-Even str-Odd[of joint-strategy  $\sigma_e\ \sigma\ p$ , unfolded strategy-winning-by-Odd-def,
rule-format, of  $\sigma_e$ ]
      induced-play-eq[OF eq-a]
    show False unfolding simps by auto
  qed

```

qed (*auto simp: move-defensive-by-Even-def strategy-winning-by-Even-def position-maxlength-cannotbe-augment*)

lemma *defensive-move-exists-for-Odd*:

assumes [*intro*]:*position p*

shows *winning-position-Even p* \vee ($\exists m$. *move-defensive-by-Odd m p*) (**is** *?w* \vee *?d*)

proof(*cases length p = 2*N* \vee *even (length p)*)

case *False*

hence *pl[intro]:length p < 2*N*

and *ev[intro]:odd (length p)* **using** *assms[unfolded position-def]* **by** *auto*

show *?thesis* **proof**(*rule impI[of* $\neg ?d \longrightarrow \neg ?w \longrightarrow$ *False, rule-format]*, *force*)

assume *not-def:* $\neg ?d$

from *not-def[unfolded move-defensive-by-Odd-def]*

have $\forall m$. $\exists \sigma$. *strategy-winning-by-Even* σ (*p @ [m]*) **by** *blast*

from *choice[OF this]* **obtain** σ_e **where**

str-Even: $\bigwedge m$. *strategy-winning-by-Even* ($\sigma_e m$) (*p @ [m]*) **by** *blast*

define σ **where** $\sigma p' = \sigma_e (p' ! \text{length } p)$ *p' for p'*

assume *not-win:* $\neg ?w$

from *not-win[unfolded move-defensive-by-Odd-def strategy-winning-by-Even-def]*

obtain σ_o **where**

str-Odd:induced-play (joint-strategy σ σ_o) *p* $\notin A$

(**is** *?pe* *p* $\notin A$)

by *blast*

let *?strat* = *joint-strategy* σ σ_o

let *?pnext* = (*p @ [?strat p]*)

{ **fix** *p' m*

assume *prefix (p @ [m]) p'*

hence (*p' ! length p*) = *m*

unfolding *prefix-def* **by** *auto*

}

hence *eq-a:* $\forall p'$. *prefix ?pnext p' \longrightarrow p' @ [?strat p'] =*

p' @ [joint-strategy (σ_e (?strat p)) σ_o *p']*

unfolding *joint-strategy-def* *σ -def* **by** *auto*

have *simps:* *?pe p = ?pe (p @ [?strat p])*

unfolding *induced-play-def* **by** *auto*

from *str-Odd str-Even[of ?strat p, unfolded strategy-winning-by-Even-def,*

rule-format]

induced-play-eq[OF eq-a]

show *False* **unfolding** *simps* **by** *auto*

qed

qed (*auto simp: move-defensive-by-Odd-def strategy-winning-by-Odd-def position-maxlength-cannotbe-augment*)

definition *defensive-strategy-Even* **where**

defensive-strategy-Even p \equiv *SOME m. move-defensive-by-Even m p*

definition *defensive-strategy-Odd* **where**

defensive-strategy-Odd p \equiv *SOME m. move-defensive-by-Odd m p*

lemma *position-augment*:

assumes *position ((augment-list f \sim n) p)*

```

shows position p
using assms length-augment-list[of n f p] unfolding position-def
by fastforce

lemma defensive-strategy-Odd:
  assumes  $\neg$  winning-position-Even p
  shows  $\neg$  winning-position-Even (((augment-list (joint-strategy  $\sigma_e$  defensive-strategy-Odd))
 $\sim$  n) p)
proof –
  show ?thesis using assms proof(induct n arbitrary:p)
    case (Suc n)
    show ?case proof(cases position p)
      case True
      from Suc.prems defensive-move-exists-for-Odd[OF True] defensive-strategy-Odd-def
someI
      have move-defensive-by-Odd (defensive-strategy-Odd p) p by metis
      from this[unfolded move-defensive-by-Odd-def] Suc.prems
        non-winning-moves-remains-non-winning-Even[of p]
      have  $\neg$  winning-position-Even (p @ [joint-strategy  $\sigma_e$  defensive-strategy-Odd
p])
        by (simp add: joint-strategy-def True)
        with Suc.hyps[of p @ [joint-strategy  $\sigma_e$  defensive-strategy-Odd p]]
        show ?thesis unfolding funpow-Suc-right comp-def by fastforce
      qed (insert position-augment,blast)
    qed auto
  qed

lemma defensive-strategy-Even:
  assumes  $\neg$  winning-position-Odd p
  shows  $\neg$  winning-position-Odd (((augment-list (joint-strategy defensive-strategy-Even
 $\sigma_o$ ))  $\sim$  n) p)
proof –
  show ?thesis using assms proof(induct n arbitrary:p)
    case (Suc n)
    show ?case proof(cases position p)
      case True
      from Suc.prems defensive-move-exists-for-Even[OF True] defensive-strategy-Even-def
someI
      have move-defensive-by-Even (defensive-strategy-Even p) p by metis
      from this[unfolded move-defensive-by-Even-def] Suc.prems
        non-winning-moves-remains-non-winning-Odd[of p]
      have  $\neg$  winning-position-Odd (p @ [joint-strategy defensive-strategy-Even  $\sigma_o$ 
p])
        by (simp add: joint-strategy-def True)
        with Suc.hyps[of p @ [joint-strategy defensive-strategy-Even  $\sigma_o p$ ]]
        show ?thesis unfolding funpow-Suc-right comp-def by fastforce
      qed (insert position-augment,blast)
    qed auto
  qed

```

end

locale *closed-GSgame* = *GSgame* +
assumes *closed*: $e \in A \implies \exists p. \text{lprefix} (\text{l-list-of } p) e \wedge (\forall e'. \text{lprefix} (\text{l-list-of } p) e' \implies \text{l-length } e' = 2 * N \implies e' \in A)$

locale *finite-GSgame* = *GSgame* +
assumes *fin*: $N \neq \infty$
begin

Finite games are closed games. As a corollary to the GS theorem, this lets us conclude that finite games are determined.

sublocale *closed-GSgame*

proof

fix *e* **assume** *eA*: $e \in A$
let *?p* = *list-of e*
from *eA* **have** *len*: $\text{l-length } e = 2 * N$ **using** *length* **by** *blast*
with *fin* **have** *p*: *l-list-of ?p = e*
by (*metis l-list-of-list-of mult-2 not-lfinite-l-length plus-eq-infty-iff-enat*)
hence *pfx*: $\text{lprefix} (\text{l-list-of } ?p) e$ **by** *auto*
{ fix *e'*
assume $\text{lprefix} (\text{l-list-of } ?p) e' \text{ l-length } e' = 2 * N$
hence $e' = e$ **using** *len* **by** (*metis lprefix-l-length-eq-imp-eq p*)
with *eA* **have** $e' \in A$ **by** *simp*
}
with *pfx* **show** $\exists p. \text{lprefix} (\text{l-list-of } p) e \wedge (\forall e'. \text{lprefix} (\text{l-list-of } p) e' \implies \text{l-length } e' = 2 * N \implies e' \in A)$
by *blast*
qed
end

context *closed-GSgame* **begin**

lemma *never-winning-is-losing-even*:

assumes *position* $p \forall n. \neg \text{winning-position-Even } (((\text{augment-list } \sigma) \sim n) p)$

shows *induced-play* $\sigma p \notin A$

proof

assume *induced-play* $\sigma p \in A$

from *closed*[*OF this*] **obtain** *p'* **where**

p': $\text{lprefix} (\text{l-list-of } p') (\text{induced-play } \sigma p)$

$\wedge e. \text{lprefix} (\text{l-list-of } p') e \implies \text{l-length } e = 2 * N \implies e \in A$ **by** *blast*

from *lprefix-l-length-le*[*OF p'(1)*] **have** *lp'*: $\text{l-length} (\text{l-list-of } p') \leq 2 * N$ **by** *auto*

show *False* **proof** (*cases length p' ≤ length p*)

case *True*

hence $\text{l-length} (\text{l-list-of } p') \leq \text{l-length} (\text{l-list-of } p)$ **by** *auto*

from *lprefix-l-length-lprefix*[*OF p'(1) - this*]

induced-play-is-lprefix[*OF assms(1)*]

```

    lprefix-trans
  have pref:lprefix (llist-of p') (induced-play strat p) for strat by blast
  from assms(2)[rule-format,of 0] assms(1) have  $\neg$  strategy-winning-by-Even  $\sigma$ 
p for  $\sigma$  by auto
  from this[unfolded strategy-winning-by-Even-def] obtain strat where
    strat:induced-play strat p  $\notin$  A by auto
  from strat p'(2)[OF pref] show False by simp
next
  case False
  let ?n = length p' - length p
  let ?pos = (augment-list  $\sigma$   $\sim$  ?n) p
  from False have length p'  $\geq$  length p by auto
  hence [simp]:length ?pos = length p'
    by (auto simp:length-augment-list)
  hence pos[intro]:position ?pos
    using False lp'(1) unfolding position-def by auto
  have llist-of p' = llist-of ?pos
    using p'(1)
    by (intro lprefix-antisym[OF lprefix-llength-lprefix lprefix-llength-lprefix],auto)
  hence p'-pos:p' = ?pos by simp
  from assms(2)[rule-format,of ?n] assms(1) have  $\neg$  strategy-winning-by-Even  $\sigma$ 
?pos for  $\sigma$  by auto
  from this[unfolded strategy-winning-by-Even-def] obtain strat where
    strat:induced-play strat ?pos  $\notin$  A by auto
  from p'-pos induced-play-is-lprefix[OF pos, of strat]
  have pref:lprefix (llist-of p') (induced-play strat ?pos) by simp
  with p'(2)[OF pref] strat show False by simp
qed
qed

lemma every-position-is-determined:
  assumes position p
  shows winning-position-Even p  $\vee$  winning-position-Odd p (is ?we  $\vee$  ?wo)
proof(rule impI[of  $\neg$  ?we  $\longrightarrow$   $\neg$  ?wo  $\longrightarrow$  False,rule-format],force)
  assume  $\neg$  ?we
  from defensive-strategy-Odd[OF this] never-winning-is-losing-even[OF assms]
  have js-no:induced-play
    (joint-strategy s defensive-strategy-Odd) p  $\notin$  A for s
    by auto
  assume  $\neg$  ?wo
  from this[unfolded strategy-winning-by-Odd-def] assms
  have  $\exists$  s. induced-play
    (joint-strategy s defensive-strategy-Odd) p  $\in$  A by simp
  thus False using js-no by simp
qed

end

end

```

3.4 Determined games

```
theory GaleStewartDeterminedGames
  imports GaleStewartDefensiveStrategies
begin
```

```
locale closed-GSgame = GSgame +
  assumes closed:e ∈ A ⇒ ∃ p. lprefix (llist-of p) e ∧ (∀ e'. lprefix (llist-of p) e'
  → llength e' = 2*N → e' ∈ A)
```

```
locale finite-GSgame = GSgame +
  assumes fin:N ≠ ∞
begin
```

Finite games are closed games. As a corollary to the GS theorem, this lets us conclude that finite games are determined.

```
sublocale closed-GSgame
```

```
proof
```

```
  fix e assume eA:e ∈ A
  let ?p = llist-of e
  from eA have len:llength e = 2*N using length by blast
  with fin have p:llist-of ?p = e
    by (metis llist-of-list-of mult-2 not-lfinite-llength plus-eq-infty-iff-enat)
  hence pfx:lprefix (llist-of ?p) e by auto
  { fix e'
    assume lprefix (llist-of ?p) e' llength e' = 2 * N
    hence e' = e using len by (metis lprefix-llength-eq-imp-eq p)
    with eA have e' ∈ A by simp
  }
  with pfx show ∃ p. lprefix (llist-of p) e ∧ (∀ e'. lprefix (llist-of p) e' → llength
  e' = 2 * N → e' ∈ A)
    by blast
qed
end
```

```
context closed-GSgame begin
```

```
lemma never-winning-is-losing-even:
```

```
  assumes position p ∀ n. ¬ winning-position-Even (((augment-list σ) ~ n) p)
  shows induced-play σ p ∉ A
```

```
proof
```

```
  assume induced-play σ p ∈ A
```

```
  from closed[OF this] obtain p' where
```

```
    p':lprefix (llist-of p') (induced-play σ p)
```

```
    ∧ e. lprefix (llist-of p') e ⇒ llength e = 2 * N ⇒ e ∈ A by blast
```

```
  from lprefix-llength-le[OF p'(1)] have lp':llength (llist-of p') ≤ 2 * N by auto
```

```
  show False proof (cases length p' ≤ length p)
```

```
    case True
```

```
    hence llength (llist-of p') ≤ llength (llist-of p) by auto
```

```
    from lprefix-llength-lprefix[OF p'(1) - this]
```

```

    induced-play-is-lprefix[OF assms(1)]
    lprefix-trans
  have pref:lprefix (llist-of p') (induced-play strat p) for strat by blast
  from assms(2)[rule-format,of 0] assms(1) have  $\neg$  strategy-winning-by-Even  $\sigma$ 
  p for  $\sigma$  by auto
  from this[unfolded strategy-winning-by-Even-def] obtain strat where
    strat:induced-play strat p  $\notin$  A by auto
  from strat p'(2)[OF pref] show False by simp
next
case False
let ?n = length p' - length p
let ?pos = (augment-list  $\sigma$   $\sim$  ?n) p
from False have length p'  $\geq$  length p by auto
hence [simp]:length ?pos = length p'
  by (auto simp:length-augment-list)
hence pos[intro]:position ?pos
  using False lp'(1) unfolding position-def by auto
have llist-of p' = llist-of ?pos
  using p'(1)
  by(intro lprefix-antisym[OF lprefix-llength-lprefix lprefix-llength-lprefix],auto)
hence p'-pos:p' = ?pos by simp
from assms(2)[rule-format,of ?n] assms(1) have  $\neg$  strategy-winning-by-Even  $\sigma$ 
?pos for  $\sigma$  by auto
from this[unfolded strategy-winning-by-Even-def] obtain strat where
  strat:induced-play strat ?pos  $\notin$  A by auto
from p'-pos induced-play-is-lprefix[OF pos, of strat]
have pref:lprefix (llist-of p') (induced-play strat ?pos) by simp
with p'(2)[OF pref] strat show False by simp
qed
qed

```

By proving that every position is determined, this proves that every game is determined (since a game is determined if its initial position \square is)

lemma *every-position-is-determined:*

```

  assumes position p
  shows winning-position-Even p  $\vee$  winning-position-Odd p (is ?we  $\vee$  ?wo)
proof(rule impI[of  $\neg$  ?we  $\longrightarrow$   $\neg$  ?wo  $\longrightarrow$  False,rule-format],force)
  assume  $\neg$  ?we
  from defensive-strategy-Odd[OF this] never-winning-is-losing-even[OF assms]
  have js-no:induced-play
    (joint-strategy s defensive-strategy-Odd) p  $\notin$  A for s
  by auto
  assume  $\neg$  ?wo
  from this[unfolded strategy-winning-by-Odd-def] assms
  have  $\exists$  s. induced-play
    (joint-strategy s defensive-strategy-Odd) p  $\in$  A by simp
  thus False using js-no by simp
qed
lemma empty-position: position  $\square$  using zero-enat-def position-def by auto

```

lemmas *every-game-is-determined* = *every-position-is-determined*[*OF empty-position*]

We expect that this theorem can be easier to apply without the 'position p' requirement, so we present that theorem as well.

lemma *every-position-has-winning-strategy*:

shows $(\exists \sigma. \textit{strategy-winning-by-Even } \sigma \ p) \vee (\exists \sigma. \textit{strategy-winning-by-Odd } \sigma \ p)$ (**is** *?we* \vee *?wo*)

proof(*cases position p*)

case *True*

then show *?thesis using every-position-is-determined by blast*

next

case *False*

hence $2 * N \leq \textit{enat } (\textit{length } p)$ **unfolding** *position-def* **by** *force*

from *induced-play-lprefix-non-positions*[*OF this*]

show *?thesis unfolding strategy-winning-by-Even-def strategy-winning-by-Odd-def*

by *simp*

qed

end

end

References

- [1] C. Dittmann. Positional determinacy of parity games. *Archive of Formal Proofs*, Nov. 2015. https://isa-afp.org/entries/Parity_Game.html, Formal proof development.
- [2] D. Gale and F. M. Stewart. Infinite games with perfect information. *Contributions to the Theory of Games*, 2(245-266):2–16, 1953.