We present a formalization of parity games (a two-player game on directed graphs) and a proof of their positional determinacy in Isabelle/HOL. This proof works for both finite and infinite games. We follow the proof in [2], which is based on [5].

Contents

1 Introduction 4
  1.1 Formal Introduction .............................................. 4
  1.2 Overview .......................................................... 4
  1.3 Technical Aspects ................................................ 5

2 Auxiliary Lemmas for Coinductive Lists 5
  2.1 lset ................................................................. 5
  2.2 llenlength .......................................................... 6
  2.3 ltake ................................................................. 6
  2.4 ldropn ............................................................... 7
  2.5 lfinite ............................................................... 7
  2.6 lmap ................................................................. 8
  2.7 Notation ............................................................ 8

3 Parity Games 8
  3.1 Basic definitions .................................................. 8
  3.2 Graphs .............................................................. 9
  3.3 Valid Paths ........................................................ 9
  3.4 Maximal Paths ...................................................... 11
  3.5 Parity Games ....................................................... 12
  3.6 Sets of Deadends ................................................... 13
  3.7 Subgames ........................................................... 13
  3.8 Priorities Occurring Infinitely Often .......................... 15
  3.9 Winning Condition ................................................ 16
  3.10 Valid Maximal Paths ............................................. 18
## 4 Positional Strategies

- **4.1 Definitions** ................................................. 20
- **4.2 Strategy-Conforming Paths** ................................. 20
- **4.3 An Arbitrary Strategy** .................................... 20
- **4.4 Valid Strategies** ........................................... 21
- **4.5 Conforming Strategies** .................................... 22
- **4.6 Greedy Conforming Path** .................................. 25
- **4.7 Valid Maximal Conforming Paths** .......................... 28
- **4.8 Valid Maximal Conforming Paths with One Edge** .......... 29
- **4.9 Induction Schemas for Paths** ............................... 30

## 5 Attracting Strategies

- **5.1 Paths Visiting a Set** ....................................... 31
- **5.2 Attracting Strategy from a Single Node** ................... 32
- **5.3 Attracting strategy from a set of nodes** .................... 35

## 6 Attractor Sets

- **6.1 directly-attracted** ........................................... 38
- **6.2 attractor-step** ............................................. 38
- **6.3 Basic Properties of an Attractor** ............................ 39
- **6.4 Attractor Set Extensions** ................................ 39
- **6.5 Removing an Attractor** ..................................... 39
- **6.6 Attractor Set Induction** .................................... 40

## 7 Winning Strategies

- **7.1 Deadends** .................................................. 42
- **7.2 Extension Theorems** ....................................... 43

## 8 Well-Ordered Strategy

- **8.1 Strategies on a Path** ...................................... 46
- **8.2 Eventually One Strategy** ................................... 48

## 9 Winning Regions

- **9.1 Paths in Winning Regions** .................................. 50
- **9.2 Irrelevant Updates** ......................................... 51
- **9.3 Extending Winning Regions** ................................ 51

## 10 Uniform Strategies

- **10.1 A Uniform Attractor Strategy** ............................. 52
- **10.2 A Uniform Winning Strategy** ............................... 54
- **10.3 Extending Winning Regions** ............................... 56

## 11 Attractor Strategies

- **11.1 Existence** .................................................. 58

## 12 Positional Determinacy of Parity Games

- **12.1 Induction Step** ............................................. 59
1 Introduction

Parity games are games played by two players, called Even and Odd, on labelled directed graphs. Each node is labelled with their player and with a natural number, called its priority.

To call this a parity game, we only need to assume that the number of different priorities is finite. Of course, this condition is only relevant on infinite graphs.

One reason parity games are important is that determining the winner is polynomial-time equivalent to the model-checking problem of the modal $\mu$-calculus, a logic able to express LTL and CTL* properties ([1]).

1.1 Formal Introduction

Formally, a parity game is $G = (V, E, V_0, \omega)$, where $(V, E)$ is a directed graph, $V_0 \subseteq V$ is the set of Even nodes, and $\omega : V \to \mathbb{N}$ is a function with $|f(V)| < \infty$.

A play is a maximal path in $G$. A finite play is winning for Even iff the last node is not in $V_0$. An infinite play is winning for Even iff the minimum priority occurring infinitely often on the path is even. On an infinite path at least one priority occurs infinitely often because there is only a finite number of different priorities.

A node $v$ is winning for a player $p$ iff all plays starting from $v$ are winning for $p$. It is well-known that parity games are determined, that is, every node is winning for some player.

A more surprising property is that parity games are also positionally determined. This means that for every node $v$ winning for Even, there is a function $\sigma : V_0 \to V$ such that all Even needs to do in order to win from $v$ is to consult this function whenever it is his turn (similarly if $v$ is winning for Odd). This is also called a positional strategy for the winning player.

We define the winning region of player $p$ as the set of nodes from which player $p$ has positional winning strategies. Positional determinacy then says that the winning regions of Even and of Odd partition the graph.

See [3] for a modern survey on positional determinacy of parity games. Their proof is based on a proof by Zielonka [5].

1.2 Overview

Here we formalize the proof from [2] in Isabelle/HOL. This proof is similar to the proof in [3], but we do not explicitly define so-called “$\sigma$-traps”. Using $\sigma$-traps could be worth exploring, because it has the potential to simplify our formalization.

Our proof has no assumptions except those required by every parity game. In particular the parity game

- may have arbitrary cardinality,
- may have loops,
- may have deadends, that is, nodes with no successors.

The main theorem is in section 12.4.
1.3 Technical Aspects

We use a coinductive list of nodes to represent paths in a graph because this gives us a uniform representation for finite and infinite paths. We can then express properties such as that a path is maximal or conforms to a given strategy directly as coinductive properties. We use the coinductive list developed by Lochbihler in [4].

We also explored representing paths as functions \( \text{n} = a \Rightarrow \text{option} \) with the property that the domain is an initial segment of \( \text{n} \) (and where \( a \) is the node type). However, it turned out that coinductive lists give simpler proofs.

It is possible to represent a graph as a function \( a \Rightarrow a \Rightarrow \text{bool} \), see for example in the proof of König’s lemma in [4]. However, we instead go for a record which contains a set of nodes and a set of edges explicitly. By not requiring that the set of nodes is \( \text{UNIV} :: a \text{ set} \) but rather a subset of \( \text{UNIV} :: a \text{ set} \), it becomes easier to reason about subgraphs.

Another point is that we make extensive use of locales, in particular to represent maximal paths conforming to a specific strategy. Thus proofs often start with interpret \( \text{vmc-path G P v_0 p s} \) to say that \( P \) is a valid maximal path in the graph \( G \) starting in \( v_0 \) and conforming to the strategy \( s \) for player \( p \).

2 Auxiliary Lemmas for Coinductive Lists

Some lemmas to allow better reasoning with coinductive lists.

```plaintext
theory MoreCoinductiveList
imports
  Main
  Coinductive.Coinductive-List
begin

2.1 lset

lemma lset-lnth: \( x \in lset \text{ xs} \Rightarrow \exists n. \text{ lnth xs n} = x \)
  by (induct rule: llist.set-induct, meson lnth-0, meson lnth-Suc-LCons)

lemma lset-lnth-member: \( lset \text{ xs} \subseteq A; \text{ enat n} < \text{ llength} \text{ xs} \) = \( \text{ lnth} \text{ xs n} \in A \)
  using contra-subsetD[of lset xs A] in-lset-conv-lnth[of - xs] by blast

lemma lset-nth-member-inf: \( \neg \text{ lfinite} \text{ xs}; lset \text{ xs} \subseteq A \) = \( \text{ lnth} \text{ xs n} \in A \)
  by (metis contra-subsetD inf-llist-lnth lset-inf-llist rangeI)

lemma lset-intersect-lnth: \( lset \text{ xs} \cap A \neq \{\} \Rightarrow \exists n. \text{ enat n} < \text{ llength} \text{ xs} \wedge \text{ lnth} \text{ xs n} \in A \)
  by (metis disjoint-iff-not-equal in-lset-conv-lnth)

lemma lset-ltake-Suc:
  assumes \( \neg \text{ lnull} \text{ xs} \text{ lnth} \text{ xs} 0 = x \text{ set} \text{ ltake (enat n) (lll xs)} \subseteq A \)
  shows \( lset \text{ ltake (enat (Suc n)) xs} \subseteq insert x A \)
  proof
    have \( lset \text{ ltake (eSuc (enat n)) (LCons x (lll xs))} \subseteq insert x A \)
      using assms(3) by auto
    moreover from assms(1,2) have \( LCons x (lll xs) = xs \)
      by (metis lnth-0 llt-simps(2) not-lnull-conv)
  ```
ultimately show \( \text{thesis} \) by (simp add: eSuc-enat)
qed

lemma lfinite-lset: \( \text{lfinite } xs \implies \neg \text{lnull } xs \implies \text{llast } xs \in \text{lset } xs \)
proof (induct rule: lfinite-induct)
  case (LCons xs)
  show \( \text{thesis} \) proof
    cases
    assume \( *: \neg \text{lnull } (\text{ltl } xs) \)
    hence \( \text{llast } (\text{ltl } xs) \in \text{lset } (\text{ltl } xs) \) using LCons.hyps(3) by blast
    hence \( \text{llast } (\text{ltl } xs) \in \text{lset } xs \) by (simp add: in-lset-ltlD)
    thus \( \text{thesis} \) by (metis * LCons.prems lhd-LCons-ltl llast-LCons2)
  qed (metis LCons.prems lhd-LCons-ltl llast-LCons llist.set-sel(1))
qed simp

lemma lset-subset: \( \neg (\text{lset } xs \subseteq A) \implies \exists n. \text{enat } n < \text{llength } xs \land \text{lnth } xs n \notin A \)
by (metis in-lset-conv-lnth subsetI)

2.2 llength

lemma enat-Suc-ltl:
  assumes \( \text{enat } (\text{Suc } n) < \text{llength } xs \)
  shows \( \text{enat } n < \text{llength } (\text{ltl } xs) \)
proof
  from assms have \( \text{eSuc } (\text{enat } n) < \text{llength } xs \) by (simp add: eSuc-enat)
  hence \( \text{enat } n < \text{epred } (\text{llength } xs) \) using eSuc-le-iff ileI1 by fastforce
  thus \( \text{thesis} \) by (simp add: epred-llength)
qed

lemma enat-ltl-Suc:
  \( \text{enat } n < \text{llength } (\text{ltl } xs) \implies \text{enat } (\text{Suc } n) < \text{llength } xs \)
by (metis eSuc-enat ldrop-ltl leD leI lnull-ldrop)

lemma infinite-small-llength [intro]: \( \neg \text{lfinite } xs \implies \text{enat } n < \text{llength } xs \)
using enat-less lfinite-conv-llength-enat neq-iff by blast

lemma lnull-0-llength: \( \neg \text{lnull } xs \implies \text{enat } 0 < \text{llength } xs \)
using zero-enat-def by auto

lemma Suc-llength: \( \text{enat } (\text{Suc } n) < \text{llength } xs \implies \text{enat } n < \text{llength } xs \)
using dual-order.strict-trans enat-ord-simps(2) by blast

2.3 ltake

lemma ltake-lnth: \( \text{ltake } n xs = \text{ltake } n ys \implies \text{enat } m < n \implies \text{lnth } xs m = \text{lnth } ys m \)
by (metis lnth-ltake)

lemma lset-ltake-prefix [simp]: \( n \leq m \implies \text{lset } (\text{ltake } n xs) \subseteq \text{lset } (\text{ltake } m xs) \)
by (simp add: lprefix-lsetD)

lemma lset-ltake: \( (\forall m. m < n \implies \text{lnth } xs m \in A) \implies \text{lset } (\text{ltake } (\text{enat } n) xs) \subseteq A \)
proof (induct n arbitrary: xs)
  case 0
  have \( \text{ltake } (\text{enat } 0) xs = \text{LNil} \) by (simp add: zero-enat-def)
thus ?case by simp

next
case (Suc n)
show ?case proof (cases)
  assume xs ≠ LN Nil
  then obtain x xs' where xs = LCons x xs' by (meson neq-LNil-conv)
  { fix m assume m < n
      hence Suc m < Suc n by simp
      hence lnth xs (Suc m) ∈ A using Suc.prems by presburger
      hence lnth xs' m ∈ A using xs by simp
  }
  hence lset (ltake (enat n) xs') ⊆ A using Suc.hyps by blast
moreover have ltake (enat (Suc n)) xs = LCons x (ltake (enat n) xs')
  using xs ltake-eSuc-LCons[af - x xs'] by (metis (no-types) eSuc-enat)
moreover have x ∈ A using Suc.prems xs by force
ultimately show ?thesis by simp
qed simp

qed

lemma llength-ltake': enat n < llength xs ⇒ llength (ltake (enat n) xs) = enat n
by (metis llength-ltake min.strict-order-iff)

lemma llast-ltake:
assumes enat (Suc n) < llength xs
shows llast (ltake (enat (Suc n)) xs) = lnth xs n
using llast-def unfolding llength-ltake'[OF assms] by (auto simp add: lnth-ltake)

lemma lset-ltake-ltl:
lset (ltake (enat n) (ltl xs)) ⊆ lset (ltake (enat (Suc n)) xs)
proof (cases)
  assume ¬ lnull xs
  then obtain v0 where xs = LCons v0 (ltl xs) by (metis lhd-LCons-ltl)
  hence ltake (eSuc (enat n)) xs = LCons v0 (ltake (enat n) (ltl xs))
      by (metis ltake-eSuc-LCons)
  hence lset (ltake (enat (Suc n)) xs) = lset (LCons v0 (ltake (enat n) (ltl xs)))
      by (simp add: eSuc-enat)
  thus ?thesis using lset-LCons[af v0 ltake (enat n) (ltl xs)] by blast
qed (simp add: lnull-def)

2.4 ldropn

lemma ltl-ldrop: [∀ xs. P xs ⇒ P (ltl xs); P xs ] ⇒ P (ldropn n xs)
  unfolding ldropn-def by (induct n) simp-all

2.5 lfinite

lemma lfinite-drop-set: lfinite xs ⇒ ∃ n. v ∉ lset (ldrop n xs)
by (metis ldrop-inf lmember-code(1) lset-lmember)

lemma index-infinite-set:
[¬ lfinite x; lnth x m = y; ∃ i. lnth x i = y ⇒ (∃ m > i. lnth x m = y) ] ⇒ y ∈ lset (ldropn n x)
proof (induct n arbitrary: x m)
case 0 thus \(?case using \(\text{set-nth-member-inf}\) by auto\)
next
  case \((\text{Suc} \, n)\)
  obtain \(a\) \(xs\) where \(x = \text{LCons} \, a\) \(xs\) by \((\text{meson Suc.prems(1)}\) \(\text{lnull-imp-\text{finite}\) \(\text{not-lnull-cons}\))\)
  obtain \(j\) \(where j > m\) \(\text{lnth} \, x \, j = y\) \(using\) \(\text{Suc.prems(2,3)}\) \(by\) \(\text{blast}\)
have \(\text{lnth} \, xs \, (j - 1) = y\) \(by\) \((\text{metis lnth-LCons' \(j; 2\) not-less\(0\) \(x\))}\)
moreover \{  
  fix \(i\) \(assume\) \(\text{lnth} \, xs \, i = y\)
  hence \(\text{lnth} \, x \, (\text{Suc} \, i) = y\) \(by\) \((\text{simp add: } x)\)
  hence \(\exists j > i.\) \(\text{lnth} \, xs \, j = y\) \(by\) \((\text{metis Suc.prems(3)}\) \(\text{Suc-lessE\) lnth-Suc-LCons \(x\))}\)
\}  
ultimately show ?case \(using\) \(\text{Suc.hyps\) Suc.prems(1)}\) \(x\) \(by\) \(\text{auto}\)
qed

2.6 \text{lmap}

\text{lemma lnth-lmap-ldropn:}
\(\text{enat} \, n < \text{llength} \, xs \implies \text{lnth} \, (\text{lmap} \, f \, (\text{ldropn} \, n \, xs))\) \(0 = \text{lnth} \, (\text{lmap} \, f \, xs) \, n\)
\(by\) \((\text{simp add: lhd-ldropn lnth-0-conv-lhd})\)

\text{lemma lnth-lmap-ldropn-Suc:}
\(\text{enat} \, (\text{Suc} \, n) < \text{llength} \, xs \implies \text{lnth} \, (\text{lmap} \, f \, (\text{ldropn} \, n \, xs))\) \((\text{Suc} \, 0) = \text{lnth} \, (\text{lmap} \, f \, xs) \, (\text{Suc} \, n)\)
\(by\) \((\text{metis no-types, lifting) Suc-length ldropn-ltl leD llist.map-disc-iff lnth-lmap-ldropn lnth-ltl lnull-ldropn ltl-ldropn ltl-lmap})\)

2.7 Notation

We introduce the notation \$\ to denote \text{lnth}.

\text{notation lnth (infix \(\$\) 61)}

end

3 Parity Games

theory ParityGame
imports  
  \text{Main}
  \text{MoreCoinductiveList}
begin

3.1 Basic definitions

'a' is the node type. Edges are pairs of nodes.

\text{type-synonym} 'a \text{Edge} = 'a \times 'a

A path is a possibly infinite list of nodes.

\text{type-synonym} 'a \text{Path} = 'a \text{llist}
3.2 Graphs

We define graphs as a locale over a record. The record contains nodes (AKA vertices) and edges. The locale adds the assumption that the edges are pairs of nodes.

\[
\text{record } 'a \text{ Graph } = \\
\text{ verts } :: 'a \text{ set } (V_1) \\
\text{ arcs } :: 'a \text{ Edge set } (E_1)
\]

\[
\text{abbreviation is-arc } :: (',a,,'b) \text{ Graph-scheme } \Rightarrow 'a \Rightarrow 'a \Rightarrow \text{ bool } \quad \text{(infixl } \rightarrow \text{ 60}) \quad \text{where}
\]

\[
v \rightarrow_G w \equiv (v,w) \in E_G
\]

locale Digraph =
  fixes G (structure)
  assumes valid-edge-set: \( E \subseteq V \times V \)
begin

lemma edges-are-in-V [intro]: \( v \rightarrow w \Rightarrow v \in V \quad v \rightarrow w \Rightarrow w \in V \) using valid-edge-set by blast+

A node without successors is a deadend.

abbreviation deadend :: 'a \Rightarrow bool where deadend v \equiv \neg (\exists w \in V. \ v \rightarrow w)

3.3 Valid Paths

We say that a path is valid if it is empty or if it starts in \( V \) and walks along edges.

\[
\text{coinductive valid-path } :: 'a \text{ Path } \Rightarrow \text{ bool } \quad \text{where}
\]

\[
\text{valid-path-base: valid-path } LNil \\
\text{ valid-path-base': } v \in V \quad \Rightarrow \quad \text{valid-path } (LCons v LNil) \\
\text{ valid-path-cons: } \left\{ \begin{array}{l}
\forall v \in V; w \in V; v \rightarrow w; \quad \text{valid-path } Ps; \quad \neg \text{lnull } Ps; \quad \text{lhd } Ps = w \\
\Rightarrow \quad \text{valid-path } (LCons v Ps)
\end{array} \right.
\]

\text{inductive-simps} valid-path-cons-simp: valid-path \( (LCons x xs) \)

\text{lemma valid-path-ltl': valid-path } (LCons v Ps) \Rightarrow \text{valid-path } Ps
\text{ using valid-path-simps by blast}

\text{lemma valid-path-ltl: valid-path } P \Rightarrow \text{valid-path } (\text{lTL } P)
\text{ by (metis llist.exhaust-set ltl-simps(1) valid-path-ltl')}

\text{lemma valid-path-drop: valid-path } P \Rightarrow \text{valid-path } (\text{lDROPn } n P)
\text{ by (simp add: valid-path-ltl ltl-ldrop)}

\text{lemma valid-path-in-V: assumes valid-path } P \text{ shows } \text{lset } P \subseteq V
\text{ proof}
\text{ fix } x \text{ assume } x \in \text{lset } P \text{ thus } x \in V
\text{ using assms by (induct rule: llist.set-induct) (auto intro: valid-path.cases)}
\text{ qed}

\text{lemma valid-path-finite-in-V: } \left\{ \begin{array}{l}
\text{valid-path } P; \quad \text{enat } n < \text{llength } P \\
\Rightarrow \quad P \$ n \in V
\end{array} \right.
\text{ using valid-path-in-V lset-lnth-member by blast}

\text{lemma valid-path-edges': valid-path } (LCons v (LCons w Ps)) \Rightarrow \text{v } \rightarrow w
\text{ using valid-path.cases by fastforce}

\text{lemma valid-path-edges:}
\text{ assumes valid-path } P \text{ enat } (\text{Suc } n) < \text{llength } P
\text{ shows } P \$ n \rightarrow P \$ \text{Suc } n
\text{ proof}
define $P'$ where $P' = ldropn n P$

have $enat n \leq llength P$ using assms(2) enat-ord-simps(2) less-trans by blast

hence $P' \leq n = P \leq n$ by (simp add: $P'$-def)

moreover have $P' \leq n = P \leq n$ by (metis One-nat-def $P'$-def Suc-eq-plus1 add.commute assms(2) lnth-ldropn)

ultimately have $\exists Ps. P' = LCons (P \leq n) (LCons (P \leq n) Ps)$

by (metis $P'$-def $enat n \leq llength P$, assms(2) ldropn-Suc-conv-ldropn)

moreover have valid-path $P'$ by (simp add: $P'$-def assms(1) valid-path-drop)

ultimately show $\thesis$ using valid-path-edges' by blast

qed

lemma valid-path-coinduct [consumes 1, case_names base step, coinduct pred: valid-path]:
  assumes major: $Q P$
  and base: $\forall v P. Q (LCons v LNil) \implies v \in V$
  and step: $\forall v w P. Q (LCons v (LCons w P)) \implies v \rightarrow w \land (Q (LCons w P) \lor valid-path (LCons w P))$
  shows valid-path $P$

using major proof (coinduction arbitrary: $P$)

  case valid-path
  
  { assume $P \neq LNil \neg(\exists v. P = LCons v LNil \land v \in V)$
    then obtain $v w P'$ where $P = LCons v (LCons w P')$
      using neq-LNil-conv base valid-path by metis
    hence $\thesis$ using step valid-path by auto
  }

  thus $\thesis$ by blast

qed

lemma valid-path-no-deadends:
  $\thesis$ valid-path $P$, $enat (Suc i) \leq llength P \implies \neg deadend (P \leq i)$
  using valid-path-edges by blast

lemma valid-path-ends-on-deadend:
  $\thesis$ valid-path $P$, $enat i \leq llength P$, deadend ($P \leq i$) $\implies enat (Suc i) = llength P$
  using valid-path-no-deadends by (metis enat-iless enat-ord-simps(2) neq_iff not-less-eq)

lemma valid-path-prefix: $\thesis$ valid-path $P$, $lprefix P' P$ $\implies$ valid-path $P'$

proof (coinduction arbitrary: $P'$ $P$)
  case (step $v w P'' P' P$)
  then obtain $Ps$ where $Ps: LCons v (LCons w Ps)$ $P$ by (metis LCons-lprefix-conv)
  hence valid-path ($LCons w Ps$) using valid-path-ltl step(2) by blast
  moreover have $lprefix (LCons w P') (LCons w Ps)$ using $Ps$ step(1,3) by auto
  ultimately show $\thesis$ using $Ps$ step(2) valid-path-edges' by blast

qed (metis LCons-lprefix-conv valid-path-cons-simp)

lemma valid-path-lappend:
  assumes valid-path $P$ valid-path $P' [\neg lnull P; \neg lnull P'] \implies llast P \rightarrow lhd P'$
  shows valid-path (lappend $P P'$)

proof (cases, cases)
  assume $\neg lnull P \neg lnull P'$
  thus $\thesis$ using assms proof (coinduction arbitrary: $P' P$)
  case (step $v w P'' P' P$)
  show $\thesis$ proof (cases)
assume \( \text{lnull} (\text{ltl} P) \)
thus \(?\text{case using} \ \text{step}(1,2,3,5,6) \)
by (metis \text{ldh-LCons} \text{ldh-LCons-ltl} \text{ldh-lappend} \text{llast-singleton} \
\text{llist.collapse}(1) \text{ltl-lappend} \text{ltl-simps}(2))

next
assume \( \neg \text{lnull} (\text{ltl} P) \)
moreover have \( \text{ltl} \ (\text{ltappend} \ P \ P') = \text{ltappend} \ (\text{ltl} P) \ P' \) using \( \text{step}(2) \) by simp
ultimately show \(?\text{case using} \ \text{step} \)
by (metis \text{no-types, lifting} \
\text{ldh-LCons} \text{ldh-LCons-ltl} \text{ldh-lappend} \text{ltl-LCons} \text{ltl-simps}(2) \
\text{valid-path-edges'} \text{valid-path-ltl})

qed

A valid path is still valid in a supergame.

lemma \text{valid-path-supergame}:
assumes \( \text{valid-path} \ P \) and \( G' : \text{Digraph} \ G' \ V \subseteq V_G, E \subseteq E_G \)
shows \( \text{Digraph} . \text{valid-path} \ G' \ P \)
using \( \text{valid-path} P \)
proof (coinduction arbitrary: \( P \)
rule: \( \text{Digraph} . \text{valid-path-coinduct}(\text{OF} \ G'(1), \text{case-names base step}) \)
\text{case base} thus \(?\text{case using} \ G'(2) \text{valid-path-cons-simp} \) by auto
qed (meson \( G'(3) \subsetseteq \text{valid-path-edges'} \text{valid-path-ltl'} \))

3.4 Maximal Paths

We say that a path is \textit{maximal} if it is empty or if it ends in a deadend.

\textbf{coinductive} \textit{maximal-path} where
\begin{align*}
\text{maximal-path-base} & : \text{maximal-path} \ \text{LNil} \\
\text{maximal-path-base'} & : \text{deadend} \ v \Rightarrow \text{maximal-path} \ (\text{LCons} \ v \ \text{LNil}) \\
\text{maximal-path-cons} & : \neg \text{lnull} \ Ps \Rightarrow \text{maximal-path} \ Ps \Rightarrow \text{maximal-path} \ (\text{LCons} \ v \ Ps)
\end{align*}

\textbf{lemma} \text{maximal-no-deadend}:
\begin{align*}
\text{maximal-path} \ (\text{LCons} \ v \ Ps) \Rightarrow \neg \text{deadend} \ v \Rightarrow \neg \text{lnull} \ Ps
\end{align*}
by (metis \text{ldh-LCons} \text{llist.distinct}(1) \text{ltl-simps}(2) \text{maximal-path.simps})

\textbf{lemma} \text{maximal-ltl}:
\begin{align*}
\text{maximal-path} \ P \Rightarrow \text{maximal-path} \ (\text{ltl} P)
\end{align*}
by (metis \text{ltl.simps}(1) \text{ltl-simps}(2) \text{maximal-path.simps})

\textbf{lemma} \text{maximal-drop}:
\begin{align*}
\text{maximal-path} \ P \Rightarrow \text{maximal-path} \ (\text{ldropn} \ n \ P)
\end{align*}
by (simp add: \text{maximal-ltl} \text{ltl-ldrop})

\textbf{lemma} \text{maximal-path-lappend}:
assumes \( \neg \text{lnull} \ P' \ \text{maximal-path} \ P' \)
shows \( \text{maximal-path} \ (\text{ltappend} \ P \ P') \)
proof (cases)
assume \( \neg \text{lnull} \ P \)
thus \(?\text{thesis using} \ \text{assms} \text{ proof} \) (coinduction arbitrary: \( P' \ P \) rule: \( \text{maximal-path.coinduct} \)
\text{case} \( \text{maximal-path} \ P' P' \)
let \( ?P = \text{ltappend} \ P \ P' \)
show \(?\text{case proof} \) (cases \( ?P = \text{LNil} \lor (\exists v. \ ?P = \text{LCons} v \ \text{LNil} \land \text{deadend} v) \))
\text{case} False
then obtain \( Ps \ v \) where \( P : \ ?P = \text{LCons} v \ Ps \) by (meson \text{neg-LNil-conv})
hence \( Ps = \text{ltappend} \ (\text{ltl} P) \ P' \) by (simp add: \text{ltappend-ltl maximal-path}(1))
hence $\exists Ps1 P'. Ps = \text{lappend} Ps1 P' \land \neg\text{lnull} P' \land \text{maximal-path} P'$

using maximal-path(2) maximal-path(3) by auto

thus $\square$thesis using $P \triangleright \text{lappend-lnull1}$ by fastforce

qed blast

qed (simp add: assms(2) lappend-lnull1[of P P'])

lemma maximal-ends-on-deadend:

assumes maximal-path $P$ lfinite $P$ $\neg$null $P$

shows deadend (llast $P$)

proof

from $\langle$finite $P$, $\neg$null $P\rangle$ obtain $n$ where $n$: llength $P$ = enat (Suc $n$)

by (metis enat-ord-simps(2) gr0-implies-Suc lfinite-length-enat lnull-0-length)

define $P'$ where $P' = \text{ldropn} n P$

hence maximal-path $P$ using assms(1) maximal-drop by simp

thus $\square$thesis proof (cases rule: maximal-path.cases)

  case (maximal-path-base $v$)

    hence deadend (last $P'$) unfolding $P'$-def by simp

    thus $\square$thesis unfolding $P'$-def using llast-ldropn[of $n$ $P$] $n$

    by (metis $P'$-def ldropn-eq-LConsD local.maximal-path-base(1))

next

  case (maximal-path-cons $P'$ $v$)

    hence ldropn (Suc $n$) $P = P'$ unfolding $P'$-def by (metis ldrop-eSuc-llt ltl-ldropn ltl-simps(2))

    thus $\square$thesis using $n$ maximal-path-cons(2) by auto

qed (simp add: $P'$-def $n$ ldropn-eq-LNil)

qed

lemma maximal-ends-on-deadend': $[\langle$finite $P$, deadend (llast $P$) $\rangle] \implies \text{maximal-path} P$

proof (coinduction arbitrary: $P$ rule: maximal-path.coinduct)

  case (maximal-path $P$)

  show $\langle$case proof (cases)

    assume $P \neq \text{LNil}$

    then obtain $v$ $P'$ where $P' = \text{LCons} v$ $P'$ by (meson neq-LNil-conv)

    show $\langle$thesis proof (cases)

      assume $P' = \text{LNil}$ thus $\square$thesis using $P'$ maximal-path(2) by auto

      qed (metis $P'$ lfinite-LCons llast-LCons llist.collapse(1) maximal-path(1,2))

    qed simp

  qed

lemma infinite-path-is-maximal: $[\langle$valid-path $P$, $\neg$finite $P$ $\rangle] \implies \text{maximal-path} P$

by (coinduction arbitrary: $P$ rule: maximal-path.coinduct)

(cases rule: valid-path.cases, auto)

end — locale Digraph

3.5 Parity Games

Parity games are games played by two players, called Even and Odd.

datatype Player = Even | Odd

abbreviation other-player $p \equiv (if p = \text{Even} \ then \ \text{Odd} \ else \ \text{Even})$

12
A parity game is tuple \((V, E, V_0, \omega)\), where \((V, E)\) is a graph, \(V_0 \subseteq V\) and \(\omega\) is a function from \(V \to \mathbb{N}\) with finite image.

### 3.6 Sets of Deadends

**Definition**

\[ \text{deadends } p \equiv \{ v \in VV p. \text{deadend } v\} \]

**Lemma**

\[ \text{deadends-in-V: deadends } p \subseteq V \text{ unfolding deadends-def by blast} \]

### 3.7 Subgames

We define a subgame by restricting the set of nodes to a given subset.

**Definition**

\[ \text{subgame where} \]

\[ \text{subgame } V' \equiv G (\]

\[ \text{verts} := V \cap V', \]

\[ \text{arcs} := E \cap (V' \times V'), \]

\[ \text{player0} := V_0 \cap V' \] \]

**Lemma**

\[ \text{subgame-V [simp]: } V_{\text{subgame } V'} \subseteq V \]

\[ \text{and subgame-E [simp]: } E_{\text{subgame } V'} \subseteq E \]

\[ \text{and subgame-\omega: } \omega_{\text{subgame } V'} = \omega \]

**Unfolding** subgame-def by simp-all

**Lemma**

\[ \text{assumes } V' \subseteq V \]

\[ \text{shows subgame-\text{V'} [simp]: } V_{\text{subgame } V'} = V' \]

\[ \text{and subgame-\text{E} [simp]: } E_{\text{subgame } V'} = E \cap (V_{\text{subgame } V'} \times V_{\text{subgame } V'}) \]

**Unfolding** subgame-def using assms by auto

**Lemma**

\[ \text{subgame-\text{V} [simp]: } \text{ParityGame} \cdot \text{VV (subgame } V') = V' \cap \text{VV } p \text{ proof—} \]

\[ \text{have ParityGame} \cdot \text{VV (subgame } V') \text{ Even } = V' \cap \text{VV Even unfolding subgame-def by auto} \]

\[ \text{moreover have ParityGame} \cdot \text{VV (subgame } V') \text{ Odd } = V' \cap \text{VV Odd proof—} \]
have $V' \cap V - (V_0 \cap V') = V' \cap V \cap (V - V_0)$ by blast
thus $\text{thesis}$ unfolding subgame-def by auto
qed
ultimately show $\text{thesis}$ by simp
qed
corollary subgame-VV-subset [simp]: $\text{ParityGame}_{VV}(\text{subgame } V') \ p \subseteq VV \ p$ by simp

lemma subgame-finite [simp]: finite $(\omega_{\text{subgame } V'}) \ V_{\text{subgame } V'}$ proof –
  have $\text{finite} (\omega ' V_{\text{subgame } V'})$ using subgame-V priorities-finite
  by (meson finite-subset image-mono)
  thus $\text{thesis}$ by (simp add: subgame-def)
qed

lemma subgame-\omega-subset [simp]: $\omega_{\text{subgame } V'} \ V_{\text{subgame } V'} \subseteq \omega ' V$
  by (simp add: image-mono subgame-\omega)

lemma subgame-Digraph: Digraph (subgame $V'$)
  by (unfold-locales) (auto simp add: subgame-def)

lemma subgame-ParityGame:
  shows $\text{ParityGame}_{(\text{subgame } V')}$
proof (unfold-locales)
  show $E_{\text{subgame } V'} \subseteq V_{\text{subgame } V'} \ V_{\text{subgame } V'}$
    using subgame-Digraph[unfolded Digraph-def].
  show $V_0_{\text{subgame } V'} \subseteq V_{\text{subgame } V'}$ unfolding subgame-def using valid-player0-set by auto
  show finite $(\omega_{\text{subgame } V'} \ V_{\text{subgame } V'})$ by simp
qed

lemma subgame-valid-path:
  assumes $P$: valid-path $P$ lset $P \subseteq V'$
  shows Digraph.valid-path (subgame $V'$) $P$
proof –
  have lset $P \subseteq V$ using $P(1)$ valid-path-in-V by blast
  hence lset $P \subseteq V_{\text{subgame } V'}$ unfolding subgame-def using $P(2)$ by auto
  with $P(1)$ show $\text{thesis}$
proof (coinduction arbitrary: $P$
  rule: Digraph.valid-path.coinduct[OF subgame-Digraph, case-names IH])
case IH
  thus $\text{case}$ proof (cases rule: valid-path.cases)
case (valid-path-cons $v \ w$ $Ps$)
  moreover hence $v \in V_{\text{subgame } V'} \ w \in V_{\text{subgame } V'}$ using IH(2) by auto
  moreover hence $v \rightarrow_{\text{subgame } V'} \ w$ using local.valid-path-cons(4) subgame-def by auto
  moreover have valid-path $Ps$ using IH(1) valid-path-ltl' local.valid-path-cons(1) by blast
  ultimately show $\text{thesis}$ using IH(2) by auto
qed auto
qed

lemma subgame-maximal-path:
  assumes $V'$: $V' \subseteq V$ and $P$: maximal-path $P$ lset $P \subseteq V'$
shows Digraph.maximal-path (subgame V') P
proof -
  have lset P ⊆ V_{subgame V'} unfolding subgame-def using P(2) V' by auto
  with P(1) V' show ?thesis
    by (coinduction arbitrary; P rule: Digraph.maximal-path.coinduct[OF subgame-Digraph])
      (cases rule: maximal-path.cases, auto)
qed

3.8 Priorities Occurring Infinitely Often

The set of priorities that occur infinitely often on a given path. We need this to define the
winning condition of parity games.

definition path-inf-priorities :: 'a Path ⇒ nat set where
  path-inf-priorities P ≡ {k. ∀ n. k ∈ lset (ldropn n (lmap ω P))}

Because ω is image-finite, by the pigeon-hole principle every infinite path has at least one
priority that occurs infinitely often.

lemma path-inf-priorities-is-nonempty:
  assumes P: valid-path P ¬ lfinite P
  shows ∃ k. k ∈ path-inf-priorities P
proof -
  Define a map from indices to priorities on the path.
  define f where f i = ω (P $ i) for i
  have range f ⊆ ω ' V unfolding f-def
    using valid-path-in-V[OF P(1)] lset-nth-member-inf[OF P(2)]
    by blast
  hence finite (range f)
    using priorities-finite finite-subset by blast
  then obtain n0 where n0: ¬(finite {n. f n = f n0})
    using pigeonhole-infinite[of UNIV f] by auto
  define k where k = f n0

  The priority k occurs infinitely often.
  have lmap ω P $ n0 = k unfolding f-def k-def
    using assms(2) by (simp add: infinite-small-llength)
  moreover {
    fix n assume lmap ω P $ n = k
    have ∃ n' > n. f n' = k unfolding k-def using n0 infinite-nat-iff-unbounded by auto
    hence ∃ n' > n. lmap ω P $ n' = k unfolding f-def
      using assms(2) by (simp add: infinite-small-llength)
  }
  ultimately have ∀ n. k ∈ lset (ldropn n (lmap ω P))
    using index-infinite-set[of lmap ω P n0 k] P(2) lfinite-lmap
    by blast
  thus ?thesis unfolding path-inf-priorities-def by blast
qed

lemma path-inf-priorities-at-least-min-prio:
  assumes P: valid-path P and a: a ∈ path-inf-priorities P
shows \( \min (\omega \cdot V) \leq a \)

proof
  have \( a \in \text{lset } (\text{idropn } 0 \text{ (lmap } \omega \text{ } P)) \) using a unfolding path-inf-priorities-def by blast
  hence \( a \in \omega \cdot \text{lset } P \) by simp
  thus \( \text{thesis using } P \text{ valid-path-in-V priorities-finite } \min \text{-le by blast} \)
qed

lemma path-inf-priorities-LCons:
path-inf-priorities \( P = \) path-inf-priorities \((\text{LCons } v \text{ P})\) (is \( ?A = ?B \))
proof
  show \( ?A \subseteq ?B \)
  proof
    fix \( a \) assume \( a \in ?A \)
    hence \( \forall n. \ a \in \text{lset } (\text{idropn } n \text{ (lmap } \omega \text{ } (\text{LCons } v \text{ P}))) \)
    unfolding path-inf-priorities-def
    using in-lset-ltlD[of a] by (simp add: ltl-idropn)
    thus \( a \in ?B \) unfolding path-inf-priorities-def by blast
  qed
next
  show \( ?B \subseteq ?A \)
  proof
    fix \( a \) assume \( a \in ?B \)
    hence \( \forall n. \ a \in \text{lset } (\text{idropn } (\text{Suc } n) \text{ (lmap } \omega \text{ } (\text{LCons } v \text{ P}))) \)
    unfolding path-inf-priorities-def by blast
    thus \( a \in ?A \) unfolding path-inf-priorities-def by simp
  qed
qed

corollary path-inf-priorities-ltl: path-inf-priorities \( P = \) path-inf-priorities \((\text{ltl } P)\)
by (metis llist.exhaust ltl-simps path-inf-priorities-LCons)

3.9 Winning Condition

Let \( G = (V, E, V_0, \omega) \) be a parity game. An infinite path \( v_0, v_1, \ldots \) in \( G \) is winning for player \( \text{Even} \) \((\text{Odd})\) if the minimum priority occurring infinitely often is even (odd). A finite path is winning for player \( p \) iff the last node on the path belongs to the other player.

Empty paths are irrelevant, but it is useful to assign a fixed winner to them in order to get simpler lemmas.

abbreviation winning-priority \( p \equiv (\text{if } p = \text{Even} \text{ then even else odd}) \)

definition winning-path :: \( \text{Player } \Rightarrow \text{'Path } \Rightarrow \text{bool } \)
where
winning-path \( p \ P \equiv \)
(\( \neg lfinite \ P \land (\exists a \in \text{path-inf-priorities } P) \).
  \( (\forall b \in \text{path-inf-priorities } P. \ a \leq b) \land \text{winning-priority } p \ a) \)
\lor (\( \neg lnull \ P \land lfinite \ P \land \text{llast } P \in VV \ p**) \)
\lor (\( lnull \ P \land p = \text{Even} )

Every path has a unique winner.

lemma paths-are-winning-for-one-player:
  assumes valid-path \( P \)
  shows winning-path \( p \ P \leftrightarrow \neg \text{winning-path } p** \ P \)
proof (cases)
  assume \( \neg lnull \ P \)
show \( ?\text{thesis} \) proof (cases)
  assume \( !\text{finite} \ P \)
  thus \( ?\text{thesis} \)
    using assms \( !\text{finite-lset} \) valid-path-in-V
    unfolding winning-path-def
    by auto
next
  assume \( !\text{finite} \ P \)
  then obtain \( a \) where \( a \in \text{path-inf-priorities} \ P \land \ b < a \implies b \notin \text{path-inf-priorities} \ P \)
    using assms ex-least-nat-le[of \( \lambda a. a \in \text{path-inf-priorities} \ P \)] path-inf-priorities-is-nonempty
    by blast
  hence \( \forall q. \text{winning-priority} \ q \ a \leftrightarrow \text{winning-path} \ q \ P \)
    unfolding winning-path-def using \( \neg \text{lnull} \ P \) \( \neg \text{finite} \ P \) by (metis le-antisym not-le)
  moreover have \( \forall q. \text{winning-priority} \ p \ q \leftrightarrow \neg \text{winning-priority} \ p \ast \ast \ q \)
    by simp
  ultimately show \( ?\text{thesis} \)
    by blast
qed

qed (simp add: winning-path-def)

lemma winning-path-ltl:
  assumes \( P: \text{winning-path} \ p \ P \ \neg \text{lnull} \ P \ \neg \text{lnull} \ (\text{ltl} \ P) \)
  shows \( \text{winning-path} \ p \ (\text{ltl} \ P) \)
proof (cases)
  assume \( !\text{finite} \ P \)
  moreover have \( \text{last} \ P = \text{last} \ (\text{ltl} \ P) \)
    using \( P(2,3) \) by (metis last-LCons2 ltl-simps(2) not-lnull-conv)
  ultimately show \( ?\text{thesis} \)
    using \( P \) by (simp add: winning-path-def)
next
  assume \( !\text{finite} \ P \)
  thus \( ?\text{thesis} \)
    using winning-path-def path-inf-priorities-ltl \( P(1,2) \) by auto
qed

corollary winning-path-drop:
  assumes \( \text{valid-path} \ P \ \text{winning-path} \ p \ (\text{ldropn} \ n \ P) \)
  shows \( \text{winning-path} \ p \ (\text{ldropn} \ n \ P) \)
using assms proof (induct \( n \))
  case \( \text{Suc} \ n \)
  hence \( \text{winning-path} \ p \ (\text{ldropn} \ n \ P) \)
    using dual-order.strict-trans enat-ord-simps(2) by blast
  moreover have \( \text{ltl} \ (\text{ldropn} \ n \ P) = \text{ldropn} \ (\text{Suc} \ n) \ P \)
    by (simp add: ldrop-cSuc-ltl ltl-ldropn)
  moreover hence \( \neg \text{lnull} \ (\text{ldropn} \ n \ P) \)
    using Suc.prems(2) by (metis leD lnull-ldropn lnull-ltlI)
  ultimately show \( ?\text{case} \)
    using winning-path-ltl[of \( p \ \text{ldropn} \ n \ P \)] Suc.prems(2) by auto
qed simp

corollary winning-path-drop-add:
  assumes \( \text{valid-path-drop} \ P \ \text{winning-path} \ p \ (\text{ldropn} \ n \ P) \)
  shows \( \text{winning-path} \ p \ (\text{ldropn} \ n \ P) \)
using assms proof
  case \( \text{Suc} \ n \)
  hence \( \text{winning-path} \ p \ (\text{ldropn} \ n \ P) \)
    using dual-order.strict-trans enat-ord-simps(2) by blast
  moreover have \( \text{ltl} \ (\text{ldropn} \ n \ P) = \text{ldropn} \ (\text{Suc} \ n) \ P \)
    by (simp add: ldrop-cSuc-ltl ltl-ldropn)
  moreover hence \( \neg \text{lnull} \ (\text{ldropn} \ n \ P) \)
    using Suc.prems(2) by (metis leD lnull-ldropn lnull-ltlI)
  ultimately show \( ?\text{case} \)
    using winning-path-ltl[of \( p \ \text{ldropn} \ n \ P \)] Suc.prems(2) by auto
qed simp

lemma winning-path-LCons:
  assumes \( P: \text{winning-path} \ p \ (\text{LCons} \ v \ P) \)
  shows \( \text{winning-path} \ p \ (\text{LCons} \ v \ P) \)
proof (cases)
  assume \( !\text{finite} \ P \)

17
moreover have \( \text{llast } P = \text{llast } (\text{LCons } v P) \)
using \( P(2) \) by (metis \text{llast-LCons2 not-lnull-conv})
ultimately show \( \exists \text{thesis using } P \) unfolding winning-path-def by simp
next
assume \( \neg \text{llfinite } P \)
thus \( \exists \text{thesis using } P \) path-inf-priorities-LCons unfolding winning-path-def by simp
qed

lemma winning-path-supergame:
assumes \( P \) \( \vdash \exists \text{thesis using } P \) unfolding winning-path-def
and \( G': \text{ParityGame } G' VV p** \subseteq \text{ParityGame}.VV G' p** \omega = \omega_G' \)
shows \( \vdash \exists \text{thesis using } P \) path-inf-priorities-LCons unfolding winning-path-def by simp
proof
interpret \( G': \text{ParityGame } G' \) using \( G'(1) \).
have \( \exists \text{thesis using } P \) unfolding winning-path-def using \( G'(2) \) by auto
thus \( \exists \text{thesis using } P \) path-inf-priorities-def \( G'.path-inf-priorities-def G'(3) \)
by blast
qed

end — locale ParityGame

3.10 Valid Maximal Paths

Define a locale for valid maximal paths, because we need them often.
locale um-path = ParityGame +
fixes \( P v0 \)
assumes \( P \)-not-null [simp]: \( \neg \text{lnull } P \)
and \( P \)-valid [simp]: valid-path \( P \)
and \( P \)-maximal [simp]: maximal-path \( P \)
and \( P-v0 \) [simp]: \( \text{lhd } P = v0 \)
begin
lemma \( P-LCons \) : \( P = \text{LCons } v0 \) (ltl \( P \)) using \( \text{lhd-LCons-ltl}[OF \neg\text{not-null}] \) by simp

lemma \( P-len \) [simp]: \( \text{enat } 0 < \text{llength } P \) by (simp add: \( \text{lnull-0-llength} \))
lemma \( P-0 \) [simp]: \( P \$ 0 = v0 \) by (simp add: \( \text{lnth-0-cone-lhd} \))
lemma \( P-Lnth-Suc \) : \( P \$ \text{Suc } n = \text{ltl } P \$ n \) by (simp add: \( \text{lnth-ltl} \))
lemma \( P-no-deadends \) : \( \text{enat } (\text{Suc } n) < \text{llength } P \) \( \Longrightarrow \) \( \neg \text{deadend } (P \$ n) \)
using valid-path-no-deadends by simp
lemma \( P-no-deadend-v0 \) : \( \neg \text{lnull } (\text{ltl } P) \) \( \Longrightarrow \) \( \neg \text{deadend } v0 \)
by (metis \( P-LCons \) valid-edges-are-in-V (2) not-lnull-conv valid-path-edges)
lemma \( P-no-deadend-v0\)-length: \( \text{enat } (\text{Suc } n) < \text{llength } P \) \( \Longrightarrow \) \( \neg \text{deadend } v0 \)
by (metis \( P-0 \) \( P-len \) \( P\)-valid \( \text{enat-ord-simps} \) (2) not-less-eq valid-path-ends-on-deadend zero-less-Suc)
lemma \( P ends-on-deadend \) : \( \exists \) \( n \) \( \text{enat } n < \text{llength } P ; \text{deadend } (P \$ n) \) \( \Longrightarrow \) \( \text{enat } (\text{Suc } n) = \text{llength } P \)
using \( P \)-valid path-ends-on-deadend by blast

lemma \( P-lnull-ltl-deadend-v0 \) : \( \neg \text{lnull } (\text{ltl } P) \) \( \Longrightarrow \) \( \text{deadend } v0 \)
using \( P-LCons \) maximal-no-deadend by force
lemma \( P-lnull-ltl-LCons \) : \( \neg \text{lnull } (\text{ltl } P) \) \( \Longrightarrow \) \( P = \text{LCons } v0 \) \( \text{LNil} \)
using P-LCons lnull-def by metis
lemma P-deadend-v0-LCons: deadend v0 =⇒ P = LCons v0 LNil
  using P-null-ltl-LCons P-no-deadend-v0 by blast

lemma Ptl-valid [simp]: valid-path (ltl P) using valid-path-ltl by auto
lemma Ptl-maximal [simp]: maximal-path (ltl P) using maximal-ltl by auto

lemma Pdrop-valid [simp]: valid-path (ldrop n P) using valid-path-drop by auto
lemma Pdrop-maximal [simp]: maximal-path (ldrop n P) using maximal-drop by auto

lemma prefix-valid [simp]: valid-path (ltake n P) using valid-path-prefix by auto

lemma extension-valid [simp]: v0 =⇒ valid-path (LCons v P)
  using P-not-null P-v0 P-valid valid-path-cons by blast

lemma extension-maximal [simp]: maximal-path (LCons v P)
  by (simp add: maximal-path-cons)
lemma lappend-maximal [simp]: maximal-path (lappend P P')
  by (simp add: maximal-path-lappend)

lemma lappend-maximal [simp]: maximal-path (lappend P P')
  by (simp add: maximal-path-lappend)

lemma finite-ltl-deadend [simp]: lfinite P =⇒ deadend (llast P)
  using P-maximal P-not-null maximal-ends-on-deadend by blast
lemma finite-ltl-V [simp]: lfinite P =⇒ llast P ∈ V
  using P-not-null lfinite-lset lset-P-V by blast

If a path visits a deadend, it is winning for the other player.

lemma visits-deadend:
  assumes lset P ∩ deadends p ≠ {}
  shows winning-path p** P
proof –
  obtain n where n: enat n < llength P P $ n ∈ deadends p
    using assms by (meson lset-intersect-lnth)
  hence *: enat (Suc n) = llength P using P-ends-on-deadend unfolding deadends-def by blast
  hence llast P = P $ n by (simp add: eSuc-enat llast-conv-lnth)
  hence llast P ∈ deadends p using n(2) by simp
  moreover have lfinite P using * llength-eq-enat-lfiniteD by force
  ultimately show ?thesis unfolding winning-path-def deadends-def by auto
qed

end

end
4 Positional Strategies

theory Strategy
imports
  Main
  ParityGame
begin

4.1 Definitions

A strategy is simply a function from nodes to nodes. We only consider positional strategies.

type-synonym 'a Strategy = 'a ⇒ 'a

A valid strategy for player \( p \) is a function assigning a successor to each node in \( VV_p \).

definition (in ParityGame) strategy :: Player ⇒ 'a Strategy ⇒ bool where
  strategy p σ ≡ ∀ v ∈ VV_p. ¬deadend v → vσ v

lemma (in ParityGame) strategyI [intro]:
  (∀ v. v ∈ VV_p; ¬deadend v) ⇒ vσ v) ⇒ strategy p σ
unfolding strategy-def by blast

4.2 Strategy-Conforming Paths

If path-conforms-with-strategy \( p \ P \ σ \) holds, then we call \( P \) a \( σ \)-path. This means that \( P \) follows \( σ \) on all nodes of player \( p \) except maybe the last node on the path.

coinductive (in ParityGame) path-conforms-with-strategy
  :: Player ⇒ 'a Path ⇒ 'a Strategy ⇒ bool where
    path-conforms-LNil: path-conforms-with-strategy p LNil σ
  | path-conforms-LCons-LNil: path-conforms-with-strategy p (LCons v LNil) σ
  | path-conforms-VVp: [ v ∈ VV_p; w = σ v; path-conforms-with-strategy p (LCons w Ps) σ ]
    ⇒ path-conforms-with-strategy p (LCons v (LCons w Ps)) σ
  | path-conforms-VVps: [ v /∈ VV_p; path-conforms-with-strategy p Ps σ ]
    ⇒ path-conforms-with-strategy p (LCons v Ps) σ

Define a locale for valid maximal paths that conform to a given strategy, because we need this concept quite often. However, we are not yet able to add interesting lemmas to this locale. We will do this at the end of this section, where we have more lemmas available.

locale vmc-path = vm-path +
  fixes p σ assumes P-conforms [simp]: path-conforms-with-strategy p P σ

Similary, define a locale for valid maximal paths that conform to given strategies for both players.

locale vmc2-path = comp?: vmc-path G P v0 p** σ' + vmc-path G P v0 p σ
  for G P v0 p σ σ'

4.3 An Arbitrary Strategy

context ParityGame begin
Define an arbitrary strategy. This is useful to define other strategies by overriding part of this strategy.

**definition** \( \sigma \text{-arbitrary} \equiv \lambda v. \text{SOME } w. v \to w \)

**lemma** valid-arbitrary-strategy [simp]: strategy \( p \sigma \text{-arbitrary} \) proof

- fix \( v \) assume \( \neg \text{deadend } v \)
- thus \( v \to \sigma \text{-arbitrary } v \) unfolding \( \sigma \text{-arbitrary-def} \) using \text{someI-ex[of } \lambda w. v \to w] \) by blast

**4.4 Valid Strategies**

**lemma** valid-strategy-updates: [ strategy \( p \sigma; v0 \to w0 ] \implies \text{strategy } p (\sigma(v0 := w0))]

unfolding strategy-def by auto

**lemma** valid-strategy-updates-set:  
assumes strategy \( p \sigma \land v \in A; v \in VV p; \neg \text{deadend } v \) \implies \( v \to \sigma' v \)  
shows strategy \( p \) (override-on \( \sigma \sigma' A) \)  
unfolding strategy-def by (metis assms override-on-def strategy-def)

**lemma** subgame-strategy-stays-in-subgame:  
assumes \( \sigma \in \text{ParityGame.strategy } (\text{subgame } V'); p \sigma \)  
and \( v \in \text{ParityGame.VV } (\text{subgame } V'); \neg \text{Digraph.deadend } (\text{subgame } V') v \)  
shows \( \sigma v \in V' \)  
proof–

interpret \( G'\text{: ParityGame subgame } V' \text{ using } \text{subgame-ParityGame} \).

have \( \sigma v \in V'\text{subgame } V' \text{ using assms unfolding } G'.\text{strategy-def } G'.\text{edges-are-in-V(2)} \) by blast

thus \( \sigma v \in V' \) by (metis Diff-iff IntE subgame-VV Player.distinct(2))

qed

**lemma** valid-strategy-supergame:  
assumes \( \sigma \in \text{strategy } p \sigma \)  
and \( \sigma' \in \text{ParityGame.strategy } (\text{subgame } V'); p \sigma' \)  
and \( G'\text{-no-deadends}: \land v \in V' \implies \neg \text{Digraph.deadend } (\text{subgame } V') v \)  
shows strategy \( p \) (override-on \( \sigma \sigma' V' \)) (is strategy \( p ?\sigma) \)

Proof

interpret \( G'\text{: ParityGame subgame } V' \text{ using } \text{subgame-ParityGame} \).

fix \( v \) assume \( v \in V V p \neg \text{deadend } v \)

show \( v \to ?\sigma v \) proof (cases)

- assume \( v \in V' \)
  hence \( v \in G'\text{VV } p \) using subgame-VV \( v \in V V p \) by blast
  moreover have \( \neg G'.\text{deadend } v \) using \( G'\text{-no-deadends } v \in V' \) by blast
  ultimately have \( v \to \text{subgame } V' \sigma' v \) using \( \sigma' \text{ unfolding } G'.\text{strategy-def} \) by blast
  moreover have \( \sigma' v = ?\sigma v \) using \( v \in V' \) by simp
  ultimately show \( ?\text{thesis} \) by (metis subgame-E subsetCE)

next

assume \( v \notin V' \)
thus \( \text{thesis using } v \sigma \) unfolding strategy-def by simp

qed

lemma valid-strategy-in-V: \[ \text{strategy } p \sigma; \forall v \in V \Rightarrow \sigma v \in V \]
unfolding valid-edge-set by auto

lemma valid-strategy-only-in-V: \[ \text{strategy } p \sigma; \forall v \in V \Rightarrow \sigma v = \sigma' v \] \Rightarrow \text{strategy } p \sigma'

unfolding strategy-def using edges-are-in-V(1) by auto

4.5 Conforming Strategies

lemma path-conforms-with-strategy-ltl [intro]:
  path-conforms-with-strategy p P \sigma \Rightarrow \text{path-conforms-with-strategy } p \text{(ltl P) } \sigma
  by (drule path-conforms-with-strategy.cases) (simp-all add: path-conforms-with-strategy.intros(1))

lemma path-conforms-with-strategy-drop:
  path-conforms-with-strategy p P \sigma \Rightarrow \text{path-conforms-with-strategy } p \text{(ldropn n P) } \sigma
  by (simp add: path-conforms-with-strategy-ltl ltl-ldrop[\lambda P. path-conforms-with-strategy p P \sigma])

lemma path-conforms-with-strategy-prefix:
  path-conforms-with-strategy p P \sigma \Rightarrow \text{lprefix } P' P \Rightarrow \text{path-conforms-with-strategy } p P' \sigma
proof (coinduction arbitrary: P P')
case (path-conforms-with-strategy P P')
thus ?thesis
proof
  cases rule: path-conforms-with-strategy.cases
  case path-conforms-LNil
  thus ?thesis using local.path-conforms-VVp v w)
next
case (path-conforms-VVp v w)
thus ?thesis
proof
  cases
  assume P' \neq LNil \land P' \neq LCons v LNil
  hence \exists Q. P' = LCons v (LCons w Q)
  by (metis local.path-conforms-VVp(1) lprefix-LCons-conv path-conforms-with-strategy(2))
  thus ?thesis using local.path-conforms-VVp(1,3,4) path-conforms-with-strategy(2) by force
qed auto
next
case (path-conforms-VVpstar v)
thus ?thesis
proof
  cases
  assume P' \neq LNil
  hence \exists Q. P' = LCons v Q
  using local.path-conforms-VVpstar(1) lprefix-LCons-conv path-conforms-with-strategy(2) by fastforce
  thus ?thesis using local.path-conforms-VVpstar path-conforms-with-strategy(2) by auto
  qed simp
  qed
next
lemma path-conforms-with-strategy-irrelevant:
  assumes path-conforms-with-strategy p P \sigma v /\in lset P
shows \( \text{path-conforms-with-strategy } p \ P \ (\sigma(v := w)) \)
using \( \text{assms apply } \) (coinduction arbitrary: \( P \)) by (drule \( \text{path-conforms-with-strategy.cases} \) auto)

lemma \( \text{path-conforms-with-strategy-irrelevant-deadend} \):
assumes \( \text{path-conforms-with-strategy } p \ P \ \sigma \ \text{deadend } v \lor v \notin VV \ P \)
shows \( \text{path-conforms-with-strategy } p \ P \ (\sigma(v := w)) \)
using \( \text{assms proof } \) (coinduction arbitrary: \( P \))
let \( \sigma = \sigma(v := w) \)
\( \text{case } \) (\( \text{path-conforms-with-strategy } P \))
thus \( \text{?case proof } \) (cases rule: \( \text{path-conforms-with-strategy.cases} \))
\( \text{case } \) (\( \text{path-conforms-VVp } v' w Ps \))
have \( w = \sigma v' \)
proof
from \( \text{valid-path } P \)
have \( \neg \text{deadend } v' \)
using \( \text{local.path-conforms- VVp} \)
with \( \text{assms} \)
have \( v' \neq v \)
using \( \text{local.path-conforms-VVp} \)
thus \( w = \sigma v' \)
by (simp add: local.path-conforms-VVp)
qed
moreover
have \( \exists P. \ LCons w Ps = P \land \text{path-conforms-with-strategy } p \ P \ \sigma \land (\text{deadend } v \lor v \notin VV \ P) \land \text{valid-path } P \)
proof
have \( \text{valid-path } (LCons w Ps) \)
using \( \text{local.path-conforms- VVp} \)
path-conforms-with-strategy \( \) valid-path-ltl' by blast
thus \( \text{thesis using } \) local.path-conforms-VVp \( \)
path-conforms-with-strategy \( \) valid-path-ltl' by blast
qed
ultimately show \( \text{thesis using } \) local.path-conforms-VVp \( \)
next
\( \text{case } \) (\( \text{path-conforms-VVpstar } v' Ps \))
have \( \exists P. \ \text{path-conforms-with-strategy } p \ Ps \ \sigma \land (\text{deadend } v \lor v \notin VV \ P) \land \text{valid-path } Ps \)
using \( \text{local.path-conforms-VVpstar} \)
path-conforms-with-strategy \( \) valid-path-ltl' by blast
thus \( \text{thesis by } \) (simp add: local.path-conforms-VVpstar)
qed simp-all
qed

lemma \( \text{path-conforms-with-strategy-irrelevant-updates} \):
assumes \( \text{path-conforms-with-strategy } p \ P \ \sigma \land v \in \text{lset } P \implies \sigma v = \sigma' v \)
shows \( \text{path-conforms-with-strategy } p \ P \ \sigma' \)
using \( \text{assms proof } \) (coinduction arbitrary: \( P \))
\( \text{case } \) (\( \text{path-conforms-with-strategy } P \))
\( \text{case } \) (\( \text{path-conforms-VVp } v' w Ps \))
have \( w = \sigma' v' \)
using \( \text{local.path-conforms-VVp} \)
path-conforms-with-strategy \( \) path-conforms-with-strategy by auto
thus \( \text{thesis using } \) local.path-conforms-VVp \( \)
path-conforms-with-strategy by auto
qed simp-all
qed

lemma \( \text{path-conforms-with-strategy-irrelevant} \):
assumes \( \text{path-conforms-with-strategy } p \ P \ (\sigma(v := w)) \ v \notin \text{lset } P \)
shows \( \text{path-conforms-with-strategy } p \ P \ \sigma \)
by (metis \( \text{assms fun-upd-triv fun-upd-upd} \) \( \text{path-conforms-with-strategy-irrelevant} \))
lemma path-conforms-with-strategy-irrelevant-deadend':
assumes path-conforms-with-strategy p P (σ(v := w)) deadend v ∨ v ∉ VV p valid-path P
shows path-conforms-with-strategy p P σ
by (metis assms fun-upd-triv fun-upd-upd path-conforms-with-strategy-irrelevant-deadend)

lemma path-conforms-with-strategy-start:
path-conforms-with-strategy p (LCons v (LCons w P)) σ ⇒ v ∈ VV p ⇒ σ v = w
by (drule path-conforms-with-strategy.cases) simp-all

lemma path-conforms-with-strategy-lappend:
assumes P: lfinite P ¬lnull P path-conforms-with-strategy p P σ
and P': ¬lnull P' path-conforms-with-strategy p P' σ
and conforms; llast P ∈ VV p ⇒ σ (llast P) = lhd P'
shows path-conforms-with-strategy p (lappend P P') σ
using assms proof (induct P rule: lfinite-induct)
case (LCons P)
show ?case proof (cases)
  assume lnull (lhd P)
  then obtain v0 where v0: P = LCons v0 LNil
    by (metis LCons.prems(1) lhd-(LCons-ltl) llist.collapse(1))
  have path-conforms-with-strategy p (LCons (lhd P) P') σ proof (cases)
    assume lhd P ∈ VV p
    moreover with v0 have lhd P' = σ (lhd P)
      using LCons.prems(5) by auto
    ultimately show ?thesis
      using path-conforms-VVp[of lhd P p lhd P' σ]
      by (metis (no-types) LCons.prems(4) ¬lnull P' lhd-LCons-ltl)
  next
    assume lhd P \notin VV p
    thus ?thesis using path-conforms-VVpstar using LCons.prems(4) v0 by blast
  qed
  thus ?thesis by (simp add: v0)
next
  assume ¬lnull (lhd P)
  hence *: path-conforms-with-strategy p (lappend (lhd P) P') σ
    by (metis LCons.prems(3) LCons.prems(1) LCons.prems(2) LCons.prems(5) LCons.prems(5)
      assms(4) assms(5) lhd-LCons-ltl path-conforms-with-strategy-ltl)
  have path-conforms-with-strategy p (LCons (lhd P) (lappend (lhd P) P')) σ proof (cases)
    assume lhd P ∈ VV p
    moreover hence lhd (lhd P) = σ (lhd P)
      by (metis LCons.prems(1) LCons.prems(2) ¬lnull (lhd P)
        lhd-LCons-ltl path-conforms-with-strategy-start)
    ultimately show ?thesis
      using path-conforms-VVp[of lhd P p lhd (lhd P) σ] * ¬lnull (lhd P)
      by (metis lappend-code(2) lhd-LCons-ltl)
  next
    assume lhd P \notin VV p
    thus ?thesis by (simp add: * path-conforms-VVpstar)
  qed
with ¬lnull P' show path-conforms-with-strategy p (lappend P P') σ
  by (metis lappend-code(2) lhd-LCons-ltl)
lemma path-conforms-with-strategy-VVpstar: 
  assumes lset P ⊆ V V p
  shows path-conforms-with-strategy p P σ
using assms proof (coinduction arbitrary: P)
case (path-conforms-with-strategy P)
moreover have ∀ v Ps. P = LCons v Ps ⇒ ?case using path-conforms-with-strategy by auto
ultimately show ?case by (cases P = LNil, simp) (metis lnull-def not-lnull-conv)
qed

lemma subgame-path-conforms-with-strategy:
  assumes V': V' ⊆ V and P: path-conforms-with-strategy p P σ lset P ⊆ V'
  shows ParityGame.path-conforms-with-strategy (subgame V') p P σ
proof-
  have lset P ⊆ V subgame V' unfolding subgame-def using P(2) V' by auto
with P(1) show ?thesis
  by (coinduction arbitrary: P rule: ParityGame.path-conforms-with-strategy.coinduct[OF subgame-ParityGame])
  (cases rule: path-conforms-with-strategy.cases, auto)
qed

lemma (in vmc-path) subgame-path-vmc-path:
  assumes V': V' ⊆ V and P: lset P ⊆ V'
  shows vmc-path (subgame V') P v0 p σ
proof-
  interpret G': ParityGame subgame V' using subgame-ParityGame by blast
  show ?thesis proof
    show G'.valid-path P using subgame-valid-path P-valid P by blast
    show G'.maximal-path P using subgame-maximal-path V' P-maximal P by blast
    show G'.path-conforms-with-strategy p P σ
      using subgame-path-conforms-with-strategy V' P-conforms P by blast
  qed simp-all
qed

4.6 Greedy Conforming Path

Given a starting point and two strategies, there exists a path conforming to both strategies. Here we define this path. Incidentally, this also shows that the assumptions of the locales vmc-path and vmc2-path are satisfiable.

We are only interested in proving the existence of such a path, so the definition (i.e., the implementation) and most lemmas are private.

context begin

private primcorec greedy-conforming-path :: Player ⇒ 'a Strategy ⇒ 'a Strategy ⇒ 'a ⇒ 'a Path
where
  greedy-conforming-path p σ σ' v0 =
  LCons v0 (if deadend v0
    then LNil

qed simp
else if $v_0 \in \text{VV } p$
then greedy-conforming-path $p \sigma \sigma' (\sigma v_0)$
else greedy-conforming-path $p \sigma \sigma' (\sigma' v_0)$

private lemma greedy-path-LNil: greedy-conforming-path $p \sigma \sigma' v_0 \neq \text{LNil}$
using greedy-conforming-path-disc-iff llist-discI(1) by blast

private lemma greedy-path-lhd: greedy-conforming-path $p \sigma \sigma' v_0 = \text{LCons } v \ P \implies v = v_0$
using greedy-conforming-path-code by auto

private lemma greedy-path-deadend-v0: greedy-conforming-path $p \sigma \sigma' v_0 = \text{LCons } v \ P \implies P = \text{LNil } \iff \text{deadend } v$
by (metis (no-types, lifting) greedy-conforming-path-disc-iff
  greedy-conforming-path-simps(3) llist-disc(1) ltl-simps(2))

corollary greedy-path-deadend-v: greedy-conforming-path $p \sigma \sigma' v_0 = \text{LCons } v \ P \implies P = \text{LNil } \iff \text{deadend } v$
using greedy-path-deadend-v0 greedy-path-lhd by metis

corollary greedy-path-deadend-v*: greedy-conforming-path $p \sigma \sigma' v_0 = \text{LCons } v \ \text{LNil} \implies \text{deadend } v$
using greedy-path-deadend-v by blast

private lemma greedy-path-ltl:
assumes greedy-conforming-path $p \sigma \sigma' v_0 = \text{LCons } v \ P$
shows $P = \text{LNil } \lor P = \text{greedy-conforming-path } p \sigma \sigma' (\sigma v_0) \lor P = \text{greedy-conforming-path } p \sigma \sigma' (\sigma' v_0)$
apply (insert assms, frule greedy-path-lhd)
apply (cases deadend v0, simp add: greedy-conforming-path-code)
by (metis (no-types, lifting) greedy-conforming-path-simps(3) llist-disc(1) ltl-simps(2))

private lemma greedy-path-ltl-ex:
assumes greedy-conforming-path $p \sigma \sigma' v_0 = \text{LCons } v \ P$
shows $P = \text{LNil } \lor (\exists v. P = \text{greedy-conforming-path } p \sigma \sigma' v)$
using assms greedy-path-ltl by blast

private lemma greedy-path-ltl-VVp:
assumes greedy-conforming-path $p \sigma \sigma' v_0 = \text{LCons } v_0 \ P \ v_0 \in \text{VV } p \neg \text{deadend } v_0$
shows $\sigma v_0 = \text{lhd } P$
using assms greedy-conforming-path-code by auto

private lemma greedy-path-ltl-VVpstar:
assumes greedy-conforming-path $p \sigma \sigma' v_0 = \text{LCons } v_0 \ P \ v_0 \in \text{VV } p^{*} \neg \text{deadend } v_0$
shows $\sigma' v_0 = \text{lhd } P$
using assms greedy-conforming-path-code by auto

private lemma greedy-conforming-path-properties:
assumes $v_0 \in \text{V strategy } p \sigma \text{ strategy } p^{*} \sigma'$
shows greedy-path-not-null: $\neg \text{lnull } (\text{greedy-conforming-path } p \sigma \sigma' v_0)$
and greedy-path-v0: greedy-conforming-path $p \sigma \sigma' v_0 \ \text{&& } v_0$
and greedy-path-valid: valid-path (greedy-conforming-path $p \sigma \sigma' v_0$)
and greedy-path-maximal: maximal-path (greedy-conforming-path $p \sigma \sigma' v_0$)
and greedy-path-conforms: path-conforms-with-strategy p (greedy-conforming-path p σ σ' v0) σ
and greedy-path-conforms': path-conforms-with-strategy p** (greedy-conforming-path p σ σ' v0) σ'

proof –

define P where [simp]: P = greedy-conforming-path p σ σ' v0

show ¬lnull P P $ 0 = v0 by (simp-all add: lnth-0-conv-lhd)

{ fix v0 assume v0 ∈ V
let ?P = greedy-conforming-path p σ σ' v0
assume asm: ¬(∃ v. ?P = LCons v LNil)
obtain P' where P': ?P = LCons v0 P' by (metis greedy-path-LNil greedy-path-lhd neq-LNil-cone)
hence ¬deadend v0 using asm greedy-path-deadend-v0 by blast
from P' have 1: ¬lnull P' using asm llist.collapse(1) ⟨v0 ∈ V⟩ greedy-path-deadend-v0 by blast
moreover have 1: ¬lnull P' using asm llist.collapse(1) ⟨v0 ∈ V⟩ greedy-path-deadend-v0 by blast
moreover hence lhd P' ∈ V by blast
moreover hence ⟨∃ v. P' = greedy-conforming-path p σ σ' v ∧ v ∈ V⟩ by (metis P' calculation(1) greedy-conforming-path.simps(2) greedy-path-ltl-ex lnull-def)

The conjunction of all the above.

ultimately have ⟨∃ P'. ?P = LCons v0 P' ∧ ¬lnull P' ∧ v0→lhd P' ∧ lhd P' ∈ V ⟨∃ v. P' = greedy-conforming-path p σ σ' v ∧ v ∈ V⟩ using P' by blast
} note coinduction-helper = this

show valid-path P using assms unfolding P-def
proof (coinduction arbitrary: v0 rule: valid-path.coinduct)
case (valid-path v0)
from ⟨v0 ∈ V⟩ assms(2,3) show ?case
  using coinduction-helper[of v0] greedy-path-lhd by blast
qed

show maximal-path P using assms unfolding P-def
proof (coinduction arbitrary: v0)
case (maximal-path v0)
from ⟨v0 ∈ V⟩ assms(2,3) show ?case
  using coinduction-helper[of v0] greedy-path-deadend-v' by blast
qed

{ fix p'' σ'' assume p'': (p'' = p ∧ σ'' = σ) ∨ (p'' = p** ∧ σ'' = σ')
moreover have assms have strategy p'' σ'' by blast
hence path-conforms-with-strategy p'' P σ'' using ⟨v0 ∈ V⟩ unfold P-def
proof (coinduction arbitrary: v0)
case (path-conforms-with-strategy v0)
show ?case proof (cases v0 ∈ VV p'')
case True
{ assume \( \neg (\exists v. \text{greedy-conforming-path } p \sigma \sigma' v_0 = LCons v LNil) \)
with \( v_0 \in V \) obtain \( P' \) where
\( P' \) : greedy-conforming-path \( p \sigma \sigma' v_0 = LCons v LNil \)
\( \text{lhd } P' \in V \exists v. P' = \text{greedy-conforming-path } p \sigma \sigma' v \land v \in V \)
using coinduction-helper by blast
with \( v_0 \in V \) have \( \sigma'' v_0 = \text{lhd } P' \)
using greedy-path-ltl-VVp greedy-path-ltl-VVpstar by blast
with \( v_0 \in V \) \( P'(1,2,5) \) have \( \text{path-conforms-VVp} \)
using greedy-conforming-path.code path-conforms-with-strategy by fastforce
}
thus \( \neg \)thesis by auto
next
case False
thus \( \neg \)thesis using coinduction-helper[of v0] path-conforms-with-strategy by auto
qed
qed
}
thus path-conforms-with-strategy \( p P \sigma \) path-conforms-with-strategy \( p^{++} P \sigma' \) by blast
qed

corollary strategy-conforming-path-exists:
assumes \( v_0 \in V \) strategy \( p \sigma \) strategy \( p^{++} \sigma' \)
obtains \( P \) where \( \text{vmc2-path } G P v_0 p \sigma \sigma' \)
proof
show \( \text{vmc2-path } G (\text{greedy-conforming-path } p \sigma \sigma' v_0) v_0 p \sigma \sigma' \)
using assms by unfold-locales (simp-all add: greedy-conforming-path-properties)
qed

corollary strategy-conforming-path-exists-single:
assumes \( v_0 \in V \) strategy \( p \sigma \)
obtains \( P \) where \( \text{vmc-path } G P v_0 p \sigma \)
proof
show \( \text{vmc-path } G (\text{greedy-conforming-path } p \sigma \sigma\text{-arbitrary } v_0) v_0 p \sigma \)
using assms by unfold-locales (simp-all add: greedy-conforming-path-properties)
qed

4.7 Valid Maximal Conforming Paths

Now is the time to add some lemmas to the locale \textit{vmc-path}.

context \textit{vmc-path} begin

lemma Ptl-conforms [simp]: path-conforms-with-strategy \( p \sigma \) \( (\text{tl} P) \sigma \)
using P-conforms path-conforms-with-strategy-ttl by blast

lemma Pdrop-conforms [simp]: path-conforms-with-strategy \( p \sigma \) \( (\text{ldropn } n P) \sigma \)
using P-conforms path-conforms-with-strategy-drop by blast

lemma prefix-conforms [simp]: path-conforms-with-strategy \( p \sigma \) \( (\text{ltake } n P) \sigma \)
using P-conforms path-conforms-with-strategy-prefix by blast

end
lemma extension-conforms [simp]:
\((v' \in \mathbb{V} \P p \Rightarrow \sigma v' = v0) \Rightarrow \text{path-conforms-with-strategy } p \ (LCons \ v' \ P) \ \sigma\)
by (metis P-LCons P-conforms path-conforms-VVp path-conforms-VVpstar)

lemma extension-valid-maximal-conforming:
assumes \(v' \rightarrow v0 \ v' \in \mathbb{V} \P p \Rightarrow \sigma v' = v0\)
shows \(\text{vmc-path } G \ (LCons \ v' \ P) \ v' \ P \ \sigma\)
using assms by unfold-locales simp-all

lemma vmc-path-ldropn:
assumes \(\text{enat } n < \text{llength } P\)
shows \(\text{vmc-path } G \ (\text{ldropn } n \ P) \ (P \$ n) \ P \ \sigma\)
using assms by unfold-locales (simp-all add: lhd-ldropn)

lemma conforms-to-another-strategy:
\(\text{path-conforms-with-strategy } p \ P \ \sigma' \Rightarrow \text{vmc-path } G \ P \ v0 \ P \ \sigma'\)
using P-not-null P-valid P-maximal P-v0 by unfold-locales blast+

end

lemma (in ParityGame) valid-maximal-conforming-path-0:
assumes \(\neg lnull \ P \ \text{valid-path } P \ \text{maximal-path } P \ \text{path-conforms-with-strategy } p \ P \ \sigma\)
shows \(\text{vmc-path } G \ P \ v0 \ P \ \sigma\)
using assms by unfold-locales (simp-all add: lnth-0-conv-lhd)

4.8 Valid Maximal Conforming Paths with One Edge

We define a locale for valid maximal conforming paths that contain at least one edge. This is equivalent to the first node being no deadend. This assumption allows us to prove much stronger lemmas about \(\text{ltl } P\) compared to \(\text{vmc-path}\).

locale vmc-path-no-deadend =

begin

definition \(w0 \equiv \text{lhd } (\text{ltl } P)\)

lemma Ptl-not-null [simp]: \(\neg lnull \ (\text{ltl } P)\)
using P-LCons P-maximal maximoinal-no-deadend v0-no-deadend by metis

lemma Ptl-LCons: \(\text{ltl } P = LCons \ w0 \ (\text{ltl } (\text{ltl } P))\)
using unfolding w0-def by simp

lemma P-LCons': \(P = LCons \ w0 \ (LCons \ w0 \ (\text{ltl } (\text{ltl } P)))\)
using P-LCons Ptl-LCons by simp

lemma v0-edge-w0 [simp]: \(v0 \rightarrow w0\)
using P-valid P-LCons' by (metis valid-path-edges')

lemma Ptl-0: \(\text{ltl } P \$ 0 = \text{lhd } (\text{ltl } P)\)
by (simp add: lhd-conv-lnth)

lemma P-Suc-0: \(P \$ \text{Suc } 0 = w0\)
by (simp add: P-lnth-Suc Ptl-0 w0-def)

lemma Ptl-edge [simp]: \(\text{ltl } P \rightarrow \text{lhd } (\text{ltl } P)\)
by (metis P-LCons' P-valid valid-path-edges' w0-def)

lemma v0l-conforms: \(v0 \in \mathbb{V} \P p \Rightarrow \sigma v0 = w0\)
using path-conforms-with-strategy-start by (metis P-LCons' P-conforms)

lemma w0-l [simp]: \(w0 \in \mathbb{V}\)
by (metis Ptl-LCons Ptl-valid valid-path-cons-simp)

lemma w0-lset-P [simp]: \(w0 \in \text{lset } P\)
by (metis P-LCons' lset-intros(1) lset-intros(2))

lemma vmc-path-ltl [simp]: \(\text{vmc-path } G \ (\text{ltl } P) \ w0 \ P \ \sigma\)
by (unfold-locales) (simp-all add: w0-def)

29
context vmc-path begin

lemma vmc-path-lnull-ltl-no-deadend:
\( \neg \text{lnull (ltl } P) \Rightarrow \text{vmc-path-no-deadend } G P v0 p \sigma \)
using P-0 P-no-deadends by (unfold-locales) (metis enat-ltl-Suc lnull-0-llength)

lemma vmc-path-conforms:
assumes enat (Suc n) < llength P P $ n \in VV p
shows \( \sigma (P $ n) = P $ Suc n \)
proof
  define \( P' \) where \( P' = \text{ldropn } n P \)
  then interpret \( \text{vmc-path } G P' P $ n p \sigma \) using vmc-path-ldropn assms
  have \( \neg \text{deadend } (P $ n) \) using assms
  then interpret \( \text{vmc-path-no-deadend } G P' P $ n p \sigma \) by unfold-locales
  thus \( ?\text{thesis} \) using \( P' \)-def P'-Suc-0 assms by simp
qed

4.9 lset Induction Schemas for Paths

Let us define an induction schema useful for proving \( \text{lset } P \subseteq S \).

lemma vmc-path-lset-induction [consumes 1, case-names base step]:
assumes Q P and base: v0 \in S
and step-assumption: \( \forall P v0. [ \text{vmc-path-no-deadend } G P v0 p \sigma; v0 \in S; Q P ] \)
shows \( \text{lset } P \subseteq S \)
proof
  fix v assume v \in lset P
  thus v \in S using vmc-path-axioms assms(1,2) proof (induct arbitrary: v0 rule: llist-set-induct)
  case (find P)
  then interpret \( \text{vmc-path } G P v0 p \sigma \) by blast
  show ?case by (simp add: find.prems(3))
next
  case (step P v)
  then interpret \( \text{vmc-path } G P v0 p \sigma \) by blast
  show ?case proof (cases)
    assume lnull (ltl P)
    hence P = LCons v LNil by (metis llist.disc(2) lset-cases step.hyps(2))
    thus ?thesis using step.prems(3) P-LCons by blast
next
  assume \( \neg \text{lnull } (\text{ltl } P) \)
  then interpret \( \text{vmc-path-no-deadend } G P v0 p \sigma \)
  using vmc-path-lnull-ltl-no-deadend by blast
  show v \in S
    using step.hyps(3)
      step-assumption[OF vmc-path-no-deadend-axioms \( v0 \in S \), \( Q P \)]
    vmc-path-ltl

end
by blast
qed
qed
qed

\[ [\exists Q P; v0 \in ?S; \chi P v0. \chi \Box \chi P \chi v0 P \chi v0 \in ?S; ?Q P] \Rightarrow ?Q \chi (ltl P) \land \Box \chi \chi P \chi v0 P \chi v0 \in ?S] \Rightarrow lset P \subseteq ?S \text{ without the Q predicate.} \]

corollary vmc-path-lset-induction-simple [case-names base step]:
assumes base: \( v0 \in S \)
and step: \( \chi P v0. \chi \Box \chi P \chi v0 P \chi v0 \in S \)
shows lset P \subseteq S
using assms vmc-path-lset-induction[of \( \lambda P. \chi \)] by blast

Another induction schema for proving \( lset P \subseteq S \) based on closure properties.

lemma vmc-path-lset-induction-closed-subset [case-names VVp VVpstar v0 disjoint]:
assumes VVp: \( \chi v. \chi v \in S; \neg \text{deadend } v; v \in VV p \) \( \Rightarrow \sigma v \in S \cup T \)
and VVpstar: \( \chi v w. \chi v \in S; \neg \text{deadend } v; v \in VV p^*; v \rightarrow w \) \( \Rightarrow w \in S \cup T \)
and v0: \( v0 \in S \)
and disjoint: lset P \cap T = \{\}
shows lset P \subseteq S
using disjoint proof (induct rule: vmc-path-lset-induction)
case (step P v0)
interpret vmc-path-no-deadend G P v0 p \sigma using step.hyps(1).
have lset (ltl P) \cap T = \{\} using step.hyps(3)
  by (meson disjoint-eq-subset-Compl lset-ltl order.trans)
moreover have w0 \in S \cup T
  using assms(1,2)[of v0] step.hyps(2) v0-no-deadend v0-conforms
  by (cases v0 \in VV p) simp-all
ultimately show ?case using step.hyps(3) w0-lset-P by blast
qed (insert v0)

end
end

5 Attracting Strategies

theory AttractingStrategy
imports Main Strategy
begin

Here we introduce the concept of attracting strategies.

context ParityGame begin

5.1 Paths Visiting a Set

A path that stays in \( A \) until eventually it visits \( W \).
\textbf{definition} \( \text{visits-via } P A W \equiv \exists n. \text{enat } n < \text{llength } P \wedge P \subseteq n \in W \wedge \text{lset (ltake (enat } n) P) \subseteq A \)

\textbf{lemma} \( \text{visits-via-monotone: } [\text{visits-via } P A W; A \subseteq A'] \implies \text{visits-via } P A' W \)
\textbf{unfolding} \( \text{visits-via-def by blast} \)

\textbf{lemma} \( \text{visits-via-visits: } \text{visits-via } P A W \implies \text{lset } P \cap W \neq \emptyset \)
\textbf{unfolding} \( \text{visits-via-def by } \text{meson disjoint-iff-not-equal in-lset-conv-lnth} \)

\textbf{lemma} \( \text{(in vmc-path) visits-via-trivial: } v0 \in W \implies \text{visits-via } P A W \)
\textbf{unfolding} \( \text{visits-via-def apply (rule exI[af - 0]) using zero-enat-def by auto} \)

\textbf{lemma} \( \text{visits-via-LCons:} \)
\textbf{assumes} \( \text{visits-via } P A W \)
\textbf{shows} \( \text{visits-via } (\text{LCons } v0 P) \text{ (insert } v0 A) W \)
\textbf{proof} –
\textbf{obtain} \( n \text{ where } n. \text{enat } n < \text{llength } P \subseteq n \in W \text{lset (ltake (enat } n) P) \subseteq A \)
\textbf{using} \( \text{assms unfolding visits-via-def by blast} \)
\textbf{define} \( P' \text{ where } P' = \text{LCons } v0 P \)
\textbf{have} \( \text{enat } (\text{Suc } n) < \text{llength } P' \text{ unfolding } P'-\text{def} \)
\textbf{by} \( \text{metis n(1) ldropn-Suc-LCons ldropn-Suc-nat-Suc-Suc Conv-LCons-LConsD}) \)
\textbf{moreover have} \( P' \subseteq n \in W \text{ unfolding } P'-\text{def by } (\text{simp add: } n(2)) \)
\textbf{moreover have} \( \text{lset (ltake (enat } (\text{Suc } n) P') \subseteq \text{insert } v0 A \)
\textbf{using} \( \text{lset-ltake-Suc of } P' v0 A \text{ unfolding } P'-\text{def by } (\text{simp add: } n(3)) \)
\textbf{ultimately show} \( \text{?thesis unfolding visits-via-def } P'-\text{def by blast} \)
\textbf{qed}

\textbf{lemma} \( \text{(in vmc-path-no-deadend) visits-via-ltl:} \)
\textbf{assumes} \( \text{visits-via } P A W \)
\textbf{and} \( v0: v0 \notin W \)
\textbf{shows} \( \text{visits-via } (\text{ltl } P) A W \)
\textbf{proof} –
\textbf{obtain} \( n \text{ where } n. \text{enat } n < \text{llength } P \subseteq n \in W \text{lset (ltake (enat } n) P) \subseteq A \)
\textbf{using} \( \text{assms unfolded visits-via-def by blast} \)
\textbf{have} \( n \neq 0 \text{ using } v0 n(2) \text{ DiffE by force} \)
\textbf{then obtain} \( n' \text{ where } n'. \text{Suc } n' = n \text{ using nat.exhaust by metis} \)
\textbf{have} \( \exists n. \text{enat } n < \text{llength } (\text{ltl } P) \wedge (\text{ltl } P) \subseteq n \in W \wedge \text{lset (ltake (enat } n) (\text{ltl } P)) \subseteq A \)
\textbf{apply} \( \text{rule exI[af - n']} \)
\textbf{using} \( n n' \text{enat-Suc-ltl[of } n' P] \text{ P-ltl-Suc lset-ltake-ltl[of } n' P] \text{ by auto} \)
\textbf{thus} \( \text{?thesis using visits-via-def by blast} \)
\textbf{qed}

\textbf{lemma} \( \text{(in vm-path) visits-via-deadend:} \)
\textbf{assumes} \( \text{visits-via } P A \text{ (deadends } p) \)
\textbf{shows} \( \text{winning-path } p^+ P \)
\textbf{using} \( \text{assms visits-via-visits visits-deadend by blast} \)

\subsection*{5.2 Attracting Strategy from a Single Node}

All \( \sigma \)-paths starting from \( v0 \) visit \( W \) and until then they stay in \( A \).

\textbf{definition} \( \text{strategy-attracts-via : Player } \Rightarrow 'a \text{ Strategy } \Rightarrow 'a \Rightarrow 'a \text{ set } \Rightarrow 'a \text{ set } \Rightarrow \text{bool} \) \text{where} \( \text{strategy-attracts-via } p \sigma v0 A W \equiv \forall P. \text{vmc-path } G P v0 p \sigma \implies \text{visits-via } P A W \)
lemma (in vmc-path) strategy-attracts-viaE:
  assumes strategy-attracts-via p σ v0 A W
  shows visits-via P A W
  using strategy-attracts-via-def assms vmc-path-axioms by blast

lemma (in vmc-path) strategy-attracts-via-SucE:
  assumes strategy-attracts-via p σ v0 A W v0 ∈ W
  shows ∃n. enat (Suc n) < llength P ∧ P $ Suc n ∈ W ∧ lset (ltake (enat (Suc n)) P) ⊆ A
  proof
    obtain n where n: enat n < llength P ∧ P $ n ∈ W ∧ lset (ltake (enat n) P) ⊆ A
    using strategy-attracts-viaE [unfolded visits-via-def assms(1)] by blast
    have n ≠ 0 using assms(2) n(2) by (metis P-0)
    thus ?thesis using n not0-implies-Suc by blast
  qed

lemma (in vmc-path) strategy-attracts-via-lset:
  assumes strategy-attracts-via p σ v0 A W
  shows lset P ∩ W ≠ {}
  using assms [THEN strategy-attracts-viaE, unfolded visits-via-def]
  by (meson disjoint-iff-not-equal lset-lnth-member subset-refl)

lemma strategy-attracts-via-v0:
  assumes σ: strategy p σ strategy-attracts-via p σ v0 A W
  and v0: v0 ∈ V
  shows v0 ∈ A ∪ W
  proof
    obtain P where vmc-path G P v0 p using strategy-conforming-path-exists-single assms by blast
    then interpret vmc-path G P v0 p .
    obtain n where n: enat n < llength P ∧ P $ n ∈ W ∧ lset (ltake (enat n) P) ⊆ A
    using σ(2)[unfolded strategy-attracts-via-def visits-via-def] vmc-path-axioms by blast
    show ?thesis proof (cases n = 0)
      case True thus ?thesis using n(2) by simp
    next
      case False
      hence lhd (ltake (enat n) P) = lhd P by (simp add: enat-0-iff(1))
      hence v0 ∈ lset (ltake (enat n) P)
      by (metis (n ≠ 0) P-not-null P-v0 enat-0-iff(1) llist.set- sel(1) ltake. disc(2))
      thus ?thesis using n(3) by blast
    qed
  qed

corollary strategy-attracts-not-outside:
  [ v0 ∈ V − A − W; strategy p σ ] ⇒ ¬strategy-attracts-via p σ v0 A W
  using strategy-attracts-via-v0 by blast

lemma strategy-attracts-viaI [intro]:
  assumes P. vmc-path G P v0 p ⇒ visits-via P A W
  shows strategy-attracts-via p σ v0 A W
  unfolding strategy-attracts-via-def using assms by blast

lemma strategy-attracts-via-no-deadends:
assumes $v \in V \land v \in A - W$ strategy-attracts-via $p \sigma v A W$
shows $\neg \text{deadend } v$
proof
assume deadend $v$
define $P$ where [simp]: $P = LCons v LNil$
interpret vmc-path $G P v p \sigma$
proof
show valid-path $P$ using $\langle v \in A - W \rangle \langle v \in V \rangle$ valid-path-base' by auto
show maximal-path $P$ using deadend $v$ by (simp add: maximal-path.intros(2))
show path-conforms-with-strategy $p \sigma P$ by (simp add: path-conforms-LCons-LNil)
qed simp-all
have visits-via $P A W$ using assms(3) strategy-attracts-viaE by blast
moreover have length $P = eSuc 0$ by simp
ultimately have $P \not\in 0 \in W$ by (simp add: enat-0-iff(1) visits-via-def)
with $\langle v \in A - W \rangle$ show False by auto
qed simp-all

lemma attractor-strategy-on-extends:
\[
\begin{array}{c}
\left[ \text{strategy-attracts-via } p \sigma v0 A W; A \subseteq A' \right] \\
\Rightarrow \text{strategy-attracts-via } p \sigma v0 A' W
\end{array}
\]
unfolding strategy-attracts-via-def using visits-via-monotone by blast

lemma strategy-attracts-via-trivial: $v0 \in W \Rightarrow \text{strategy-attracts-via } p \sigma v0 A W$
proof
fix $P$ assume $v0 \in W$ vmc-path $G P v0 p \sigma$
then interpret vmc-path $G P v0 p \sigma$ by blast
show visits-via $P A W$ using visits-via-trivial using $\langle v0 \in W \rangle$ by blast
qed

lemma strategy-attracts-via-successor:
assumes $\sigma$: strategy $p \sigma$ strategy-attracts-via $p \sigma v0 A W$
and $v0$: $v0 \in A - W$
and $w0$: $v0 \mapsto w0 \in VV p \Rightarrow \sigma v0 = w0$
shows strategy-attracts-via $p \sigma w0 A W$
proof
fix $P$ assume vmc-path $G P w0 p \sigma$
then interpret vmc-path $G P w0 p \sigma$ .
define $P'$ where [simp]: $P' = LCons v0 P$
then interpret $P'$: vmc-path $G P' v0 p \sigma$
using extension-valid-maximal-conforming $w0$ by blast
interpret $P'$: vmc-path-no-deadend $G P' v0 p \sigma$ using $\langle v0 \mapsto w0 \rangle$ by unfold-locales blast
have visits-via $P' A W$ using $\langle v0 \mapsto w0 \rangle$ by blast
thus visits-via $P A W$ using $P'.visits-via-ltl v0$ by simp
qed

lemma strategy-attracts-VVp:
assumes $\sigma$: strategy $p \sigma$ strategy-attracts-via $p \sigma v0 A W$
and $v$: $v0 \in A - W v0 \in VV p \neg \text{deadend } v0$
shows $\sigma v0 \in A \cup W$
proof
have $v0 \mapsto v0$ using $\langle v0 \mapsto v0 \rangle$ by blast
hence strategy-attracts-via $p \sigma (\sigma v0) A W$
using strategy-attracts-via-successor $\sigma v0$ by blast
thus thesis using strategy-attracts-via-v0 $\langle v0 \mapsto \sigma v0 \rangle$ by blast

34
5.3 Attracting strategy from a set of nodes

All $\sigma$-paths starting from $A$ visit $W$ and until then they stay in $A$.

**Definition** strategy-attracts :: Player $\Rightarrow$ 'a Strategy $\Rightarrow$ 'a set $\Rightarrow$ bool where
strategy-attracts $p \sigma A W$ $\equiv$ $\forall v0 \in A$. strategy-attracts-via $p \sigma v0 A W$

**Lemma** (in vmc-path) strategy-attractsE:
assumes strategy-attracts $p \sigma A W v0 A$
shows visits-via $P A W$
using asms(1)[unfolded strategy-attracts-def] asms(2) strategy-attracts-viaE by blast

**Lemma** (in vmc-path) strategy-attracts-lset:
assumes strategy-attracts $p \sigma A W v0 A$
shows $\text{lset} P \cap W \neq \{\}$
using asms(1)[unfolded strategy-attracts-def] asms(2) strategy-attracts-via-lset(1)[of $A W$]
by blast

**Lemma** strategy-attracts-empty [simp]: strategy-attracts $p \sigma \{\}$ $W$ by blast

**Lemma** strategy-attracts-invalid-path:
assumes $P : P = \text{LCons} v (\text{LCons} w P)$, $v \in A − W \setminus \{A \cup W\}$
shows $\neg$visits-via $P A W$ (is $\neg\forall A$
proof
assume $?A$
then obtain $n$ where $n : \text{enat} n < \text{llength} P P \$ n \in W \text{lset} (\text{ltake} (\text{enat} n) P) \subseteq A$
unfolding visits-via-def by blast
have $n \neq 0$ using $v \in A − W\setminus n(2)$ $P(1)$ DiffD2 by force
moreover have $n \neq \text{Suc} 0$ using $v \notin A \cup W\setminus n(2)$ $P(1)$ by auto
ultimately have $\text{Suc} (\text{Suc} 0) \leq n$ by presburger
hence $\text{lset} (\text{ltake} (\text{enat} (\text{Suc} (\text{Suc} 0))) P) \subseteq A$ using $n(3)$
by (meson contra-subsetD enat-ord-simps(1) lset-ltake-prefix lset-lnth-member lset-subset)
moreover have $\text{enat} (\text{Suc} 0) < \text{llength} (\text{ltake} (\text{eSuc} (\text{Suc} 0))) P$ proof—
have $*$: $\text{enat} (\text{Suc} (\text{Suc} 0)) < \text{llength} P$
using $\text{Suc} (\text{Suc} 0) \leq n \cdot n(1)$ by (meson enat-ord-simps(2) le-less-linear less-le-trans neg-iff)
have $\text{llength} (\text{ltake} (\text{enat} (\text{Suc} (\text{Suc} 0)))) P = \min (\text{enat} (\text{Suc} (\text{Suc} 0))) (\text{llength} P)$ by simp
hence $\text{llength} (\text{ltake} (\text{enat} (\text{Suc} (\text{Suc} 0)))) P = \text{enat} (\text{Suc} (\text{Suc} 0))$
using $*$ by (simp add: min-absorb1)
thus $?\text{thesis}$ by (simp add: eSuc-enat zero-enat-def)
ultimately have \( ltake (enat (Suc (Suc 0))) \) \( P \subseteq Suc 0 \subseteq A \) by \( (simp add: lset-lnth-member) \)

hence \( P \subseteq Suc 0 \subseteq A \) by \( (simp add: lnth-ltake) \)

thus \( False \) using \( P(1,3) \) by auto

qed

If \( A \) is an attractor set of \( W \) and an edge leaves \( A \) without going through \( W \), then \( v \) belongs to \( VV p \) and the attractor strategy \( \sigma \) avoids this edge. All other cases give a contradiction.

**lemma** strategy-attracts-does-not-leave:

assumes \( \sigma : \text{strategy-attracts} \ p \ \sigma \ A \ W \ \text{strategy} \ p \ \sigma \)

and \( v : v \rightarrow w \ v \in A - W \ w \notin A \cup W \)

shows \( v \in VV p \land \sigma v \neq w \)

**proof** (rule ccontr)

assume contra: \( \neg (v \in VV p \land \sigma v \neq w) \)

define \( \sigma' \) where \( \sigma' = \sigma\text{-arbitrary}(v := w) \)

hence strategy \( p^{**} \sigma' \) using \( (v \rightarrow w) \) by \( (simp add: valid-strategy-updates) \)

then interpret \( vmc2\text{-path} G P v p \ \sigma' \).

interpret \( vmc\text{-path-no-deadend} G P v p \ \sigma \) using \( (v \rightarrow w) \) by unfold-locales blast

interpret \( \text{comp} : \text{vmc\text{-path-no-deadend}} G P v^{**} \sigma' \) using \( (v \rightarrow w) \) by unfold-locales blast

have \( w = w0 \) using \( \sigma'\text{-def} \ v0\text{-conforms} \ \text{comp} v0\text{-conforms} \) by \( (cases v \in VV p) \) auto

hence \( \neg \text{visits-via} P A W \)

using strategy-attracts-invalid-path[of \( P v w ltl \ (ltl P) \)] \( v(2,3) \) \( P-LCons' \) by simp

thus \( False \) by \( (meson DiffI \ \sigma(1) \ \text{strategy-attractsE} \ v(2)) \)

qed

Given an attracting strategy \( \sigma \), we can turn every strategy \( \sigma' \) into an attracting strategy by overriding \( \sigma' \) on a suitable subset of the nodes. This also means that an attracting strategy is still attracting if we override it outside of \( A - W \).

**lemma** strategy-attracts-irrelevant-override:

assumes \( \text{strategy-attracts} \ p \ \sigma \ A \ W \ \text{strategy} \ p \ \sigma \)

shows \( \text{strategy-attracts} \ p (\text{override-on} \ \sigma' \ \sigma (A - W)) \ A \ W \)

**proof** (rule strategy-attractsI, rule ccontr)

fix \( v \)

let \( ?\sigma = \text{override-on} \ \sigma' \ \sigma (A - W) \)

assume \( \text{vmc}\text{-path} G P v p \ ?\sigma \)

then interpret \( \text{vmc}\text{-path} G P v p \ ?\sigma \).

assume \( v \in A \)

hence \( P \subseteq 0 \subseteq A \) using \( (v \in A) \) by simp

moreover assume contra: \( \neg \text{visits-via} P A W \)

ultimately have \( P \subseteq 0 \subseteq A - W \) unfolding visits-via-def by \( (meson DiffI \ P\text{-len} \ not-less0 lset-ltake) \)

have \( \neg \text{lset} P \subseteq A - W \) proof

assume \( lset P \subseteq A - W \)

hence \( \forall v. v \in lset P \implies \text{override-on} \ \sigma' \ \sigma (A - W) \ v = \sigma v \) by auto

hence \( \text{path-conforms-with-strategy} \ P \ \sigma \)

using \( \text{path-conforms-with-strategy-irrelevant-updates}[OF \ \text{P-conforms}] \) by blast

hence \( \text{vmc}\text{-path} G P (P 0) \ P \ \sigma \)

using \( \text{conforms-to-another-strategy} \ P 0 \) by blast
thus False
  using contra \( P \not\in A \) \( \text{assms(1)} \)
  by (meson \( \text{vmc-path.strategy-attractsE} \))

qed

hence \( \exists n. \text{enat } n < \text{lenght } P \land P \not\in A - W \) by (meson \( \text{lset-subset} \))

then obtain \( n \) where \( n: \text{enat } n < \text{lenght } P \land P \not\in A - W \)
  \( \land i. i < n \implies \neg(\text{enat } i < \text{lenght } P \land P \not\in A - W) \)
  using \( \text{ex-least-nat-let}[\lambda n. \text{enat } n < \text{lenght } P \land P \not\in A - W] \) by blast

hence \( n \)-\( \min: \land i. i < n \implies P \not\in A - W \)
  using \( \text{dual-order.strict-trans } \text{enat-ord-simps(2)} \) by blast

have \( n \not= 0 \) using \( \langle P \not\in A - W \rangle \) \( \text{assms(1)} \) by meson

then obtain \( n' \) where \( n': \text{Suc } n' = n \) using \( \text{not0-implies-Suc} \) by blast

hence \( P \not\in A - W \) using \( \text{n-min} \) by blast

moreover have \( P \not\in A - W \) using \( \text{n-min} \) by blast

moreover have \( P \not\in A - W \) proof-
  have \( P \not\in A - W \) using contra \( \text{assms(1)} \) \( n\)-\( \min \) \( \text{unfolding } \text{visits-via-def} \)
  by (meson \( \text{Diff-subset lset-ldropn subsetCE} \))

thus \( \text{thesis using } \langle P \not\in A - W \rangle \) \( \text{by } \) (meson \( \text{override-on-apply-in} \))

qed

lemma \( \text{strategy-attracts-trivial } \) [simp]: \( \text{strategy-attracts } p \sigma W W \)
  by (simp add: \( \text{strategy-attracts-def } \text{strategy-attracts-trivial} \))

If a \( \sigma \)-conforming path \( P \) hits an attractor \( A \), it will visit \( W \).

lemma \( \text{(in vmc-path) attracted-path:} \)
  assumes \( W \subseteq V \)
  and \( \sigma: \text{strategy-attracts } p \sigma A W \)
  and \( P\text{-hits-A: } \text{lset } P \cap A \not= \{\} \)
  shows \( \text{let } P \cap W \not= \{\} \)

proof-
  obtain \( n \) where \( n: \text{enat } n < \text{lenght } P \) \( P \not\in A \) using \( \text{P-hits-A} \) by (meson \( \text{lset-intersect-lnth} \))
  define \( P' \) where \( P' = \text{ldropn } n P \)
  interpret \( \text{vmc-path } G P' P \not\in p \sigma \) \( P'\)-\( \text{def using } \text{vmc-path-ldropn n(1)} \) by blast
  have \( \text{visits-via } P' A W \) using \( \sigma \not\in(2) \) \( \text{strategy-attractsE} \) by blast
  thus \( \text{thesis unfolding } \langle P' \not\in A - W \rangle \) \( \text{by } \) (meson \( \text{override-on-apply-in} \))

qed

lemma \( \text{attracted-strategy-step:} \)
  assumes \( \sigma: \text{strategy } p \sigma A W \)
  and \( v0: \neg \text{deadend } v0 v0 \in A - W \)
  shows \( \sigma v0 \in A \cup W \)
  by (meson \( \text{DiffD1 strategy-attracts-VVp assms strategy-attracts-def} \))

lemma \( \text{(in vmc-path-no-deadend) attracted-path-step:} \)
  assumes \( \sigma: \text{strategy } p \sigma A W \)
  and \( v0: v0 \in A - W \)

37
\[ w_0 \in A \cup W \]

by (metis (no-types) DiffD1 P-LCons' \sigma strategy-attractsE strategy-attracts-invalid-path v0)

end — context ParityGame

end

6 Attractor Sets

theory Attractor
imports
  Main
  AttractingStrategy
begin

Here we define the \( p \)-attractor of a set of nodes.

context ParityGame begin

We define the conditions for a node to be directly attracted from a given set.

definition directly-attracted :: Player ⇒ 'a set ⇒ 'a set where
  directly-attracted \( p \) \( S \) \( \equiv \{ v \in V - S . \neg \text{deadend } v \land \)
  \( (v \in V V \ p \rightarrow (\exists w. v \rightarrow w \land w \in S)) \land (v \in V V \ p** \rightarrow (\forall w. v \rightarrow w \rightarrow w \in S)) \}\)

abbreviation attractor-step \( p \) \( W \) \( S \) \( \equiv \) \( W \cup S \cup \) directly-attracted \( p \) \( S \)

The \( p \)-attractor set of \( W \), defined as a least fixed point.

definition attractor :: Player ⇒ 'a set ⇒ 'a set where
  attractor \( p \) \( W \) = lfp (attractor-step \( p \) \( W \))

6.1 directly-attracted

Show a few basic properties of directly-attracted.

lemma directly-attracted-disjoint \[ \text{[simp]}: \text{directly-attracted } p \ W \cap W = \{ \} \]
and directly-attracted-empty \[ \text{[simp]}: \text{directly-attracted } p \ \{ \} = \{ \} \]
and directly-attracted-V-empty \[ \text{[simp]}: \text{directly-attracted } p \ V = \{ \} \]
and directly-attracted-bounded-by-V \[ \text{[simp]}: \text{directly-attracted } p \ W \subseteq V \]
and directly-attracted-contains-no-deadends \[ \text{[elim]}: v \in \text{directly-attracted } p \ W \implies \neg \text{deadend } v \]

unfolding directly-attracted-def by blast+

6.2 attractor-step

lemma attractor-step-empty; attractor-step p \{ \} \{ \} = \{ \}
and attractor-step-bounded-by-V \[ \{ W \subseteq V; S \subseteq V \} \implies \text{attractor-step } p \ W \ S \subseteq V \]
by simp-all

The definition of attractor uses lfp. For this to be well-defined, we need show that attractor-step is monotone.

lemma attractor-step-mono; mono (attractor-step p \( W \))
unfolding directly-attracted-def by (rule mono1) auto
6.3 Basic Properties of an Attractor

**Lemma** `attractor-unfolding`: `attractor p W = attractor-step p W`  
**Unfolding** `attractor-def using attractor-step-mono lfp-unfold` by `blast`

**Lemma** `attractor-lowerbound`: `attractor-step p W S ⊆ S ⇒ attractor p W ⊆ S`  
**Unfolding** `attractor-def using attractor-step-mono lfp-unfold` by `blast`

**Lemma** `attractor-set-non-empty`: `W \neq {} ⇒ attractor p W \neq {}`  
**Using** `attractor-unfolding` by `auto`

6.4 Attractor Set Extensions

**Lemma** `attractor-set-VVp`:
- **Assumes** `v ∈ VV p v → w w ∈ attractor p W`
- **Shows** `v ∈ attractor p W`
  - **Apply** `(subst attractor-unfolding)` **Unfolding** `directly-attracted-def using assms by auto`

**Lemma** `attractor-set-VVpstar`:
- **Assumes** `¬deadend v \land \forall w. v \rightarrow w ⇒ w ∈ attractor p W`
- **Shows** `v ∈ attractor p W`
  - **Apply** `(subst attractor-unfolding)` **Unfolding** `directly-attracted-def using assms by auto`

6.5 Removing an Attractor

**Lemma** `removing-attractor-induces-no-deadends`:
- **Assumes** `v ∈ S \setminus attractor p W v \rightarrow w w ∈ S \setminus attractor p W`  
- **Shows** `∃ w ∈ S \setminus attractor p W. v \rightarrow w`
  - **Proof**
    - Have `v ∈ V` using `{ v : w : v \rightarrow w` **Unfolding** `V′-def subgame-def by simp`
    - **Assume** `v ∈ VV p` **Hence** `v ∈ attractor p W`  
      - **Using** `deadend v` **Attractor-set-base`  
    - **Next**
      - **Assume** `v ∈ VV p**`  
        - **Using** `attractor-set-VVpstar assms by auto` (**metis Diff-iff edges-are-in-V(2))`  
    - Qed

Removing the attractor sets of deadends leaves a subgame without deadends.

**Lemma** `subgame-without-deadends`:
- **Assumes** `V′-def: V′ = V ∖ attractor p (deadends p**) ∖ attractor p** (deadends p****)`  
  (is `V′ = V ∖ ?A ∖ ?B`)  
- **And** `v : v ∈ V` **Subgame** `V′ v`
- **Shows** `¬Digraph.deadend (subgame V′) v`
  - **Cases**
    - **Assume** `deadend v`  
      - **Have** `v : v ∈ V ∖ ?A ∖ ?B` **Using** `V′-def subgame-def by simp`
      - **Fix** `p′` **Assume** `v ∈ VV p′**`  
        - **Hence** `v ∈ attractor p′` (deadends p**)
        - **Using** `(deadend v)` **Attractor-set-base`[of deadends p** p′]`  

39
unfolding deadends-def by blast
hence False using v by (cases p'; cases p) auto
}
thus ?thesis using v by blast
next
assume ¬deadend v
have v: v ∈ V ⇒ ?A ∧ ?B using v unfolding V'-def subgame-def by simp
define G' where G' = subgame V'
interpret G': ParityGame G' unfolding G'-def using subgame-ParityGame.
show ?thesis proof
assume Digraph.deadend (subgame V') v
hence G'.deadend v unfolding G'-def.
have all-in-attractor: ∀w. v→w ⇒ w ∈ ?A ∨ w ∈ ?B proof (rule ccontr)
  fix w
  assume v→w ¬(w ∈ ?A ∨ w ∈ ?B)
  hence w ∈ V' unfolding V'-def by blast
  hence v ∈ V"G" unfolding G''-def subgame-def using (v→w) by auto
  hence v→G' w using (v→w) assms(2) unfolding G''-def subgame-def by auto
thus False using G'.deadend v using w ∈ V"G" by blast
qed

{ fix p' assume v ∈ VV p'
  { assume ∃w. v→w ∧ w ∈ attractor p' (deadends p'**)
    hence v ∈ attractor p' (deadends p'***) using v ∈ VV p' attractor-set-VVp by blast
    hence False using v by (cases p'; cases p) auto
  }
  hence ∃w. v→w ⇒ w ∈ attractor p'*** (deadends p'****)
    using all-in-attractor by (cases p'; cases p) auto
  hence v ∈ attractor p'*** (deadends p'****)
    using ¬deadend v. v ∈ VV p' attractor-set-VVp by auto
  hence False using v by (cases p'; cases p) auto
}
thus False using v by blast
qed

6.6 Attractor Set Induction

lemma mono-restriction-is-mono: mono f ⇒ mono (λS. f (S ∩ V))
unfolding mono-def by (meson inf-mono monoD subset-refl)

Here we prove a powerful induction schema for attractor. Being able to prove this is the only reason why we do not use inductive_set to define the attractor set.

See also https://lists.cam.ac.uk/pipermail/cl-isabelle-users/2015-October/msg00123.html

lemma attractor-set-induction [consumes 1, case-names step union]:
assumes W ⊆ V
  and step: ∀S. S ⊆ V ⇒ P S ⇒ P (attractor-step p W S)
  and union: ∀M. ∀S ∈ M. S ⊆ V ∧ P S ⇒ P (∪ M)
sows P (attractor p W)
proof
  let ?P = λS. P (S ∩ V)
  let ?f = λS. attractor-step p W (S ∩ V)
let \( ?A = \text{lfp } ?f \)
let \( ?B = \text{lfp } (\text{attractor-step } p \ W) \)

have f-mono: mono \( ?f \)
  using mono-restriction-is-mono[of \text{attractor-step } p \ W] \text{attractor-step-mono by simp}

have P-A: \( ?P \implies ?A \)
  proof (rule lfp-ordinal-induct-set)
    show \( \forall S \in M. \ ?P S \implies ?P (\bigcup M) \)
      proof -
      fix \( M \)
      let \( ?M = \{ S \cap V | S. S \in M \} \)
      assume \( \forall S \in M. \ ?P S \)
      hence \( \forall S \in ?M. \ S \subseteq V \wedge P S \) by auto
      hence \( \ast: P (\bigcup ?M) \) by (simp add: union)
      hence \( \forall S \in ?M. S \subseteq V \wedge P S \)
      hence \( ?P (\bigcup ?M) \) using \( \ast \) by auto
      qed
    qed (insert f-mono)

  have \( ?A \subseteq V \ ?B \subseteq V \)
    using \( W \subseteq V \) \text{attractor-step-bounded-by-V by auto}
  have \( ?A \subseteq V \ ?B \subseteq V \)
    using \( W \subseteq V \) \text{attractor-step-bounded-by-V by auto}

  have \( ?A = ?f ?A \) using f-mono lfp-unfold by blast
  hence \( ?A = W \cup (\bigcup ?M) \)
    using \( \subseteq V \) \text{attractor-step-bounded-by-V by auto}
  hence \( \ast: \bigcup ?M \subseteq V \)
    using \( ?A \subseteq V \) by simp

  have \( ?B = \text{attractor-step } p \ W \ ?B \)
    using \( \text{attractor-step-mono } \)
    lfp-unfold by blast
  hence \( ?f ?B \subseteq ?B \) using \( \subseteq V \)
    by (metis \( \text{no-types, lifting equalityD2 le-iff-inf} \))

  have \( ?A \subseteq ?B \)
    proof -
    show \( ?A \subseteq ?B \) using \( ?f ?B \subseteq ?B \)
      by (simp add: lfp-lowerbound)
    show \( ?B \subseteq ?A \)
      using \( ?A \subseteq V \)
      by (simp add: lfp-lowerbound)
    qed
  hence \( ?P ?B \)
    using \( P-A \)
    by (simp add: attractor-def)
  thus \( \text{thesis } ?B \subseteq V \)
    by (simp add: attractor-def le-iff-inf)
  qed

end — context ParityGame

end

7 Winning Strategies

theory WinningStrategy
imports
  Main
  Strategy
begin

context ParityGame begin
Here we define winning strategies.

A strategy is winning for player $p$ from $v_0$ if every maximal $\sigma$-path starting in $v_0$ is winning.

**definition** winning-strategy :: Player $\Rightarrow$ 'a $\Rightarrow$ bool where

\[
\text{winning-strategy } p \in V \equiv \forall P. \text{vmc-path } G P v_0 p \in \sigma \implies \text{winning-path } p \in P
\]

**lemma** winning-strategyI [intro]:

\[
\begin{align*}
\text{assumes } & \forall P. \text{vmc-path } G P v_0 p \in \sigma \implies \text{winning-path } p \in P \\
\text{shows } & \text{winning-strategy } p \in \sigma v_0
\end{align*}
\]

**unfolding** winning-strategy-def using assms by blast

**lemma** (in vmc-path) paths-hits-winning-strategy-is-winning:

\[
\begin{align*}
\text{assumes } & \sigma \in \text{strategy } p \in \sigma v \\
\text{and } & v \in \text{lset } P \\
\text{shows } & \text{winning-path } p \in P
\end{align*}
\]

**proof**

\[
\begin{align*}
\text{obtain } & n \text{ where } n : \text{enat } n < \text{llength } P P \not\subseteq n \equiv v \text{ using } v \text{ by (meson in-lset-cone-lnth)} \\
\text{interpret } & P' \text{ using } n \text{ vmc-path-ldropn } P v \in \sigma \text{ using } n \text{ vmc-path-ldropn } \text{by blast} \\
\text{have } & \text{winning-path } p \in (\text{ldropn } n P) \text{ using } \sigma \text{ by (simp add: winning-strategy-def P'.vmc-path-axioms)} \\
\text{thus } & \exists \text{thesis using } \text{winning-path-drop-add } P \text{-valid } n(1) \text{ by blast}
\end{align*}
\]

**qed**

There cannot exist winning strategies for both players for the same node.

**lemma** winning-strategy-only-for-one-player:

\[
\begin{align*}
\text{assumes } & \sigma \in \text{strategy } p \in \sigma v \\
\text{and } & \sigma' \in \text{strategy } p' \in \sigma' v \\
\text{and } & v \in V \\
\text{shows } & \text{False}
\end{align*}
\]

**proof**

\[
\begin{align*}
\text{obtain } & P \text{ where } \text{vmc2-path } G P v \in \sigma \in \sigma' \text{ using } \text{assms strategy-conforming-path-exists } \text{by blast} \\
\text{then interpret } & \text{vmc2-path } G P v \in \sigma \in \sigma' \\
\text{have } & \text{winning-path } p \in P \\
\text{using } & \text{paths-hits-winning-strategy-is-winning } \sigma(2) \text{ v0-lset-P by blast} \\
\text{moreover have } & \text{winning-path } p' \in P \\
\text{using } & \text{comp.paths-hits-winning-strategy-is-winning } \sigma'(2) \text{ v0-lset-P by blast} \\
\text{ultimately show } & \text{False using } P \text{-valid paths-are-winning-for-one-player } \text{by blast}
\end{align*}
\]

**qed**

7.1 Deadends

**lemma** no-winning-strategy-on-deadends:

\[
\begin{align*}
\text{assumes } & v \in V \text{ v } \text{deadend } v \text{ strategy } p \in \sigma \\
\text{shows } & \text{~winning-strategy } p \in \sigma v
\end{align*}
\]

**proof**

\[
\begin{align*}
\text{obtain } & P \text{ where } \text{vmc-path } G P v \in \sigma \in \sigma' \text{ using } \text{assms strategy-conforming-path-exists-single } \text{by blast} \\
\text{then interpret } & \text{vmc-path } G P v \in \sigma \\
\text{have } & P = LCons v LNil \text{ using } P\text{-deadend-v0-LCons } \text{deadend } v \text{ by blast} \\
\text{hence } & \text{~winning-path } p \in P \text{ unfolding } \text{winning-path-def } \text{using } v \in V \text{ v } \text{by auto} \\
\text{thus } & \exists \text{thesis using } \text{winning-strategy-def } \text{vmc-path-axioms } \text{by blast}
\end{align*}
\]

**qed**

**lemma** winning-strategy-on-deadends:
assumes $v \in VV \ p \ \text{deadend} \ v \ \text{strategy} \ p \ \sigma$
shows winning-strategy $p** \ \sigma \ v$
proof
fix $P$ assume vmc-path $G \ P \ v \ p** \ \sigma$
then interpret vmc-path $G \ P \ v \ p** \ \sigma$.
have $P = L\text{Cons} \ v \ L\text{Nil}$ using $P$-deadend-$v$-LCons (deadend $v$) by blast
thus winning-path $p** \ P$ unfolding winning-path-def
using $(v \in VV \ p) \ P$-valid paths-are-winning-for-one-player by auto
qed

7.2 Extension Theorems

lemma strategy-extends-VVp:
assumes $v0: v0 \in VV \ p \ \neg$deadend $v0$
and $\sigma$: strategy $p \ \sigma \ \text{winning-strategy} \ p \ \sigma \ v0$
shows winning-strategy $p \ \sigma$ ($\sigma \ v0$)
proof
fix $P$ assume vmc-path $G \ P \ (\sigma \ v0) \ p \ \sigma$
then interpret vmc-path $G \ P \ \sigma \ v0 \ p \ \sigma$.
have $v0 \rightarrow v0$ using $v0$ $\sigma(1) \ \text{strategy-def}$ by blast
hence winning-path $p$ ($L\text{Cons} \ v0 \ P$)
using $\sigma(2) \ \text{extension-valid-maximal-conforming} \ \text{winning-strategy-def}$ by blast
thus winning-path $p \ P$ using winning-path-ltl$[\text{of} \ p \ L\text{Cons} \ v0 \ P]$ by auto
qed

lemma strategy-extends-VVpstar:
assumes $v0: v0 \in VV \ p** \ v0 \rightarrow w0$
and $\sigma$: winning-strategy $p \ \sigma \ v0$
shows winning-strategy $p \ \sigma \ w0$
proof
fix $P$ assume vmc-path $G \ P \ w0 \ p \ \sigma$
then interpret vmc-path $G \ P \ w0 \ p \ \sigma$.
have winning-path $p$ ($L\text{Cons} \ v0 \ P$)
using $\text{extension-valid-maximal-conforming} \ \text{VV-impl1} \ \sigma \ v0 \ \text{winning-strategy-def}$ by auto
thus winning-path $p \ P$ using winning-path-ltl$[\text{of} \ p \ L\text{Cons} \ v0 \ P]$ by auto
qed

lemma strategy-extends-backwards-VVpstar:
assumes $v0: v0 \in VV \ p**$
and $\sigma$: strategy $p \ \sigma \ \wedge \ w. \ v0 \rightarrow w \implies \ \text{winning-strategy} \ p \ \sigma \ w$
shows winning-strategy $p \ \sigma \ v0$
proof
fix $P$ assume vmc-path $G \ P \ v0 \ p \ \sigma$
then interpret vmc-path $G \ P \ v0 \ p \ \sigma$.
show winning-path $p \ P$ proof (cases)
assume deadend $v0$
thus $\sigma$thesis using $P$-deadend-$v0$-LCons winning-path-def $v0$ by auto
next
assume $\neg$deadend $v0$
then interpret vmc-path-no-deadend $G \ P \ v0 \ p \ \sigma$ by unfold-locale$s$
interpret ltlP: vmc-path $G \ ltl \ P \ w0 \ p \ \sigma$ using vmc-path-ltl .

43
have winning-path p (ltl P)
using σ(2) v₀-edge-w₀ vmc-path-ltl winning-strategy-def by blast
thus winning-path p P
using winning-path-LCons by (metis P-LCons¹ ltlP P-LCons ltlP P-not-null)
qed

lemma strategy-extends-backwards-VVp:
assumes v₀: v₀ ∈ VV p σ v₀ = w v₀→w
and σ: strategy p σ winning-strategy p σ w
shows winning-strategy p σ v₀
proof
fix P assume vmc-path G P v₀ p σ
then interpret vmc-path G P v₀ p σ.
have ¬deadend v₀ using v₀→w by blast
then interpret vmc-path-no-deadend G P v₀ p σ by unfold-locale
have winning-path p (ltl P)
using σ(2)|unfolded winning-strategy-def| v₀(1,2) v₀-conforms vmc-path-ltl by presburger
thus winning-path p P using winning-path-LCons by (metis P-LCons Ptl-not-null)
qed

end — context ParityGame
end

8 Well-Ordered Strategy

theory WellOrderedStrategy
imports
  Main
  Strategy
begin

Constructing a uniform strategy from a set of strategies on a set of nodes often works by
well-ordering the strategies and then choosing the minimal strategy on each node. Then
every path eventually follows one strategy because we choose the strategies along the path
to be non-increasing in the well-ordering.

The following locale formalizes this idea.

We will use this to construct uniform attractor and winning strategies.

locale WellOrderedStrategies = ParityGame +
fixes S :: 'a set
and p :: Player
— The set of good strategies on a node v
and good :: 'a ⇒ 'a Strategy set
and r :: ('a Strategy × 'a Strategy) set
assumes S-V: S ⊆ V
— r is a wellorder on the set of all strategies which are good somewhere.
and r-wo: well-order-on {σ, ∃ v ∈ S. σ ∈ good v} r
— Every node has a good strategy.
and good-ex: ∀ v ∈ S ⇒ ∃ σ. σ ∈ good v
— good strategies are well-formed strategies.

**and good-strategies:** \( \forall v \sigma. \sigma \in \text{good } v \Rightarrow \text{strategy } p \sigma \)

— A good strategy on \( v \) is also good on possible successors of \( v \).

**and strategies-continue:** \( \forall v w \sigma. [ v \in S; v \rightarrow w; v \in VV p \Rightarrow \sigma v = w; \sigma \in \text{good } v ] \Rightarrow \sigma \in \text{good } w \)

**begin**

The set of all strategies which are good somewhere.

**abbreviation** \( \text{Strategies} \equiv \{ \sigma. \exists v \in S. \sigma \in \text{good } v \} \)

**definition** minimal-good-strategy where

\[
\text{minimal-good-strategy } v \sigma \equiv \sigma \in \text{good } v \land (\forall \sigma'. (\sigma', \sigma) \in r - \text{Id} \Rightarrow \sigma' \not\in \text{good } v)
\]

**no-notation** binomial (infixl choose 65)

Among the good strategies on \( v \), choose the minimum.

**definition** choose where

\[
\text{choose } v \equiv \text{THE } \sigma. \text{minimal-good-strategy } v \sigma
\]

Define a strategy which uses the minimum strategy on all nodes of \( S \). Of course, we need to prove that this is a well-formed strategy.

**definition** well-ordered-strategy where

\[
\text{well-ordered-strategy} \equiv \text{override-on } \sigma-\text{arbitrary } (\lambda v. \text{choose } v v) S
\]

Show some simple properties of the binary relation \( r \) on the set \( \text{Strategies} \).

**lemma** r-refl [simp]: refl-on \( \text{Strategies} \ r \)

**using** r-wo unfolding well-order-on-def linear-order-on-def partial-order-on-def preorder-on-def

**by** blast

**lemma** r-total [simp]: total-on \( \text{Strategies} \ r \)

**using** r-wo unfolding well-order-on-def linear-order-on-def

**by** blast

**lemma** r-trans [simp]: trans \( r \)

**using** r-wo unfolding well-order-on-def linear-order-on-def partial-order-on-def preorder-on-def

**by** blast

**lemma** r-wf [simp]: \( \text{wf } (r - \text{Id}) \)

**using** well-order-on-def r-wo

**by** blast

**choose** always chooses a minimal good strategy on \( S \).

**lemma** choose-works:

assumes \( v \in S \)

shows minimal-good-strategy \( v \) (choose \( v \))

**proof**

**have** wf; \( \text{wf } (r - \text{Id}) \) using well-order-on-def r-wo by blast

**obtain** \( \sigma \) where \( \sigma.1: \text{minimal-good-strategy } v \sigma \)

**unfolding** minimal-good-strategy-def by (meson good-ex [OF \( \forall v \in S \) \( \text{wf } \text{wf-eq-minimal} \))

hence \( \sigma: \sigma \in \text{good } v \land (\sigma', \sigma) \in r - \text{Id} \Rightarrow \sigma' \not\in \text{good } v \)

**unfolding** minimal-good-strategy-def by auto

\{ **fix** \( \sigma' \) assume minimal-good-strategy \( v \sigma' \)

hence \( \sigma': \sigma' \in \text{good } v \land (\sigma, \sigma') \in r - \text{Id} \Rightarrow \sigma \not\in \text{good } v \)

**unfolding** minimal-good-strategy-def by auto

**have** \( (\sigma, \sigma') \not\in r - \text{Id} \) using \( \sigma(1) \sigma'(2) \) by blast

45
moreover have \((\sigma', \sigma) \notin r \setminus Id\) using \(\sigma(2) \sigma'(1)\) by auto
moreover have \(\sigma \in \text{Strategies}\) using \(\sigma(1) \langle v \in S \rangle\) by auto
moreover have \(\sigma' \in \text{Strategies}\) using \(\sigma'(1) \langle v \in S \rangle\) by auto
ultimately have \(\sigma' = \sigma\)
  using \(\text{r-wo Linear-order-in-diff-Id well-order-on-Field well-order-on-def}\) by fastforce
}
with \(\sigma\) have \(\exists!\sigma. \text{minimal-good-strategy } v \sigma\) by blast
thus \(?\text{thesis using the}\{\text{of minimal-good-strategy } v, \text{folded choose-def}\}\) by blast
qed
corollary
assumes \(v \in S\)
shows \(\text{choose-good: choose } v \in \text{good } v\)
  and \(\text{choose-minimal: } \land \sigma'. (\sigma', \text{choose } v) \in r - Id \implies \sigma' \notin \text{good } v\)
  and \(\text{choose-strategy: strategy } p (\text{choose } v)\)
  using \(\text{choose-works[OF assms, unfolded minimal-good-strategy-def]}\) \text{good-strategies}\ by blast+
corollary \(\text{choose-in-Strategies: } v \in S \implies \text{choose } v \in \text{Strategies}\) using \(\text{choose-good}\) by blast
lemma \(\text{well-ordered-strategy-valid: strategy } p \text{ well-ordered-strategy}\)
proof–
\{
  fix \(v\) assume \(v \in S\) \(v \in V V p \neg \text{deadend } v\)
  moreover have \(\text{strategy } p (\text{choose } v)\)
  using \(\text{choose-works[OF } v \in S, \text{unfolded minimal-good-strategy-def, THEN conjunct1]}\) \text{good-strategies}\ by blast
  ultimately have \(v \rightarrow (\lambda v. \text{choose } v \ v) \ v\) using \(\text{strategy-def}\) by blast
\}
thus \(?\text{thesis unfolding well-ordered-strategy-def using valid-strategy-updates-set\} by force}\)
qed

8.1 Strategies on a Path
Maps a path to its strategies.
definition \(\text{path-strategies } \equiv \text{lmap choose}\)
lemma \(\text{path-strategies-in-Strategies}:\)
  assumes \(\text{bset } P \subseteq S\)
  shows \(\text{bset (path-strategies } P) \subseteq \text{Strategies}\)
  using \(\text{path-strategies-def assms choose-in-Strategies}\) by auto
lemma \(\text{path-strategies-good}:\)
  assumes \(\text{bset } P \subseteq S\) \(\text{enat } n \text{ < } \text{llength } P\)
  shows \(\text{path-strategies } P \ $ n \ \text{in } \text{good } (P \ $ n)\)
  by \((\text{simp add: path-strategies-def assms choose-good bset-lnth-member})\)
lemma \(\text{path-strategies-strategy}:\)
  assumes \(\text{bset } P \subseteq S\) \(\text{enat } n \text{ < } \text{llength } P\)
  shows \(\text{strategy } p (\text{path-strategies } P \ $ n)\)
  using \(\text{path-strategies-good assms good-strategies}\) by blast
lemma path-strategies-monotone-Suc:

assumes P: lset P ⊆ S valid-path P path-conforms-with-strategy p P well-ordered-strategy
enat (Suc n) < length P

shows (path-strategies P $ Suc n, path-strategies P $ n) ∈ r

proof

define P' where P' = ldropn n P
hence enat (Suc 0) < length P' using P(4)

by (metis enat-ltl-Suc ldrop-Suc ltl ldrop-Suc-conv-ldrop lnul-ltl ltl-ldrop)
then obtain v w Ps where vw: P' = LCons v (LCons w Ps)
by (metis ldrop-0 ltl ldrop-0-nul ltl-ldrop)
moreover have lset P' ⊆ S unfolding P'-def using P(1) ltl ldropn-subset[of n P] by blast
ultimately have v ∈ S w ∈ S by auto
moreover have v→w using valid-path-edges[of v w Ps, folded vw] valid-path-drop[of P(2)]
P'-def by blast
moreover have choose v ∈ good v using choose-good[of v ∈ S] by blast
moreover have v ∈ V V p ⇒ choose v v = w proof

assume v ∈ V V p
moreover have path-conforms-with-strategy p P' well-ordered-strategy
unfolding P'-def using path-conforms-with-strategy-drop P(3) by blast
ultimately have well-ordered-strategy v = w using vw path-conforms-with-strategy-start by blast
thus choose v v = w unfolding well-ordered-strategy-def using v ∈ S by auto
qed

ultimately have choose v ∈ good w using strategies-continue by blast
hence *:(choose v, choose w) $ r = Id using choose-minimal[of w ∈ S] by blast

have (choose w, choose v) ∈ r proof (cases)

assume choose v choose w
thus ?thesis using r-refl reflD choose-in-Strategies[OFF v ∈ S] by fastforce

next
assume choose v ≠ choose w
by (metis (lifting) Linear-order-in-diff Id r-wo well-order-on-Field well-order-on-def)
qed

hence (path-strategies P $ Suc 0, path-strategies P' $ 0) ∈ r
unfolding path-strategies-def using vw by simp
thus ?thesis unfolding path-strategies-def P'-def
using ltl lmap ldrop[of Suc ltl lmap ldrop P(4)], of choose

ltl lmap ldrop Suc P(4), of choose
by simp

qed

lemma path-strategies-monotone:

assumes P: lset P ⊆ S valid-path P path-conforms-with-strategy p P well-ordered-strategy
enat m < length P

shows (path-strategies P $ m, path-strategies P $ n) ∈ r

using assms proof (induct m − n arbitrary: n m)
case (Suc d)
show ?case proof (cases)

assume d = 0
thus ?thesis using path-strategies-monotone-Suc[OFF P(1,2,3)]
by (metis (no-types) Suc.hyps(2) Suc.prems(4,5) Suc-diff-Suc Suc-inject Suc-le1 diff-is-0-eq diff0-imp-equal)

next

assume d ≠ 0

have m ≠ 0 using Suc.hyps(2) by linarith

then obtain m' where m' = m using not0-implies-Suc by blast

hence d = m' - n using Suc.hyps(2) by presburger

moreover hence n < m using d ≠ 0 by presburger

ultimately have (path-strategies P $ m', path-strategies P $ n) ∈ r

using Suc.hyps(1) [of m' n, OF - P(1,2,3)] Suc.prems(5) dual-order.strict-trans enat-ord-simps(2)

m'

by blast

thus ?thesis

using m' path-strategies-monotone-Suc[OF P(1,2,3)] by (metis (no-types) Suc.prems(5) r-trans trans-def)

qed

qed simp

lemma path-strategies-eventually-constant:

assumes "finite P lset P ⊆ S valid-path P path-conforms-with-strategy p P well-ordered-strategy"

shows "∃ n. ∀ m ≥ n. path-strategies P $ n = path-strategies P $ m"

proof -

define σ-set where σ-set = lset (path-strategies P)

have "∃ σ. σ ∈ σ-set unfolding σ-set-def path-strategies-def"

using assms(1) finite-lmap lset-nth-member-inf by blast

then obtain σ' where σ' ∈ σ-set \τ. (τ, σ') ∈ r - Id \τ ∈ σ-set

using wfe-min[of r - Id - σ-set] by auto

obtain n where n: path-strategies P $ n = σ'

using σ'(1) lset-nth[of σ'] unfolding σ-set-def by blast

{ fix m assume n ≤ m

have path-strategies P $ n = path-strategies P $ m proof (rule ccontr)

  assume "*: path-strategies P $ n ≠ path-strategies P $ m"

  with "n ≤ m" have n < m using le-imp-less-or-eq by blast

  with path-strategies-monotone have (path-strategies P $ m, path-strategies P $ n) ∈ r

  using assms by (simp add: infinite-small-length)

  with "*: have (path-strategies P $ m, path-strategies P $ n) ∈ r - Id by simp

  with σ'(2) have path-strategies P $ m ∉ σ-set by blast

  thus False unfolding σ-set-def path-strategies-def

  using assms(1) finite-lmap lset-nth-member-inf by blast

  qed

  } thus ?thesis by blast

qed

8.2 Eventually One Strategy

The key lemma: Every path that stays in S and follows well-ordered-strategy eventually follows one strategy because the strategies are well-ordered and non-increasing along the path.

lemma path-eventually-conforms-to-σ-map-n:
assumes $lset P \subseteq S$ valid-path $P$ path-conforms-with-strategy $p$ $P$ well-ordered-strategy
shows $\exists n. \ path-conforms-with-strategy p (ldropn n P) (path-strategies P \mathcal{\$} n)$
proof (cases)
assume $\infinite P$
then obtain $n$ where $llength P = enat n$ using $\infinite$-llength-enat by blast
hence $ldropn n P = LNil$ by simp
thus $\exists$thesis by (metis path-conforms-LNil)
next
assume $\lnot \infinite P$
then obtain $n$ where $\forall m. n \leq m \implies path-strategies P \mathcal{\$} n = path-strategies P \mathcal{\$} m$
using path-strategies-eventually-constant assms by blast
let $\sigma = well-ordered-strategy$
define $P'$ where $P' = ldropn n P$
{ fix $v$ assume $v \in lset P'$
  hence $v \in S$ using $lset P \subseteq S$ $P'$-def in-lset-ldropnD by fastforce
    from $v \in lset P'$ obtain $m$ where $m: enat m < llength P' P' \mathcal{\$} m = v$ by (meson
      in-lset-conv-lnth)
  hence $P \mathcal{\$} m + n = v$ unfolding $P'$-def by (simp add: $\lnot \infinite P$, infinite-small-llength)
  moreover have $\sigma v = choose v v$ unfolding well-ordered-strategy-def using $v \in S$ by auto
  ultimately have $\sigma v = \mbox{choose } v v$
    unfolding path-strategies-def using infinite-small-llength[OF $\lnot \infinite P$] by simp
  hence $\sigma v = (\mbox{path-strategies } P \mathcal{\$} n) v$ using $n[of m + n]$ by simp
}
moreover have path-conforms-with-strategy $p P'$ well-ordered-strategy
  unfolding $P'$-def by (simp add: assms(3) path-conforms-with-strategy-drop)
ultimately show $\exists$thesis
  using path-conforms-with-strategy-irrelevant-updates $P'$-def by blast
qed
end — WellOrderedStrategies

end

9 Winning Regions

theory WinningRegion
imports
  Main
  WinningStrategy
begin
Here we define winning regions of parity games. The winning region for player $p$ is the set
of nodes from which $p$ has a positional winning strategy.

context ParityGame begin

definition winning-region $p \equiv \{ v \in V. \exists \sigma. \ strategy p \sigma \land winning-strategy p \sigma v \}$

lemma winning-regionI [intro]:
  assumes $v \in V$ strategy $p \sigma$ winning-strategy $p \sigma v$
  shows $v \in \mbox{winning-region } p$
  using assms unfolding winning-region-def by blast
lemma winning-region-in-V [simp]: winning-region p ⊆ V unfolding winning-region-def by blast

lemma winning-region-deadends:
  assumes v ∈ VV p deadend v
  shows v ∈ winning-region p**
proof
  show v ∈ V using v ∈ VV p by blast
  show winning-strategy p** σ-arbitrary v using assms winning-strategy-on-deadends by simp
qed simp

9.1 Paths in Winning Regions

lemma (in vmc-path) paths-stay-in-winning-region:
  assumes σ': strategy p σ' winning-strategy p σ' v0
  and σ: ∀v. v ∈ winning-region p =⇒ σ' v = σ v
  shows set P ⊆ winning-region p
proof
  fix x assume x ∈ lset P
  thus x ∈ winning-region p using assms vmc-path-axioms
proof (induct arbitrary: v0 rule: llist-set-induct)
  case (find P v0)
  interpret vmc-path G P v0 p σ using find.prems(4) .
  show ?case using P-v0 σ' (1) find.prems(2) v0-V unfolding winning-region-def by blast
next
  case (step P x v0)
  interpret vmc-path G P v0 p σ using step.prems(4) .
  show ?case (cases)
    assume lnull (ltl P)
    thus ?thesis using P-lnull-ltl-LCons step.hyps(2) by auto
next
  assume ¬lnull (ltl P)
  then interpret vmc-path-no-deadend G P v0 p σ using P-no-deadend-v0 by unfold-locales
  have winning-strategy p σ′ w0 proof (cases)
    assume w0 ∈ VV p
    hence winning-strategy p σ′ (σ′ v0)
      using strategy-extends-VVp local.step(4) step.prems(2) v0-no-deadend by blast
  moreover have σ v0 = w0 using v0-conforms (v0 ∈ VV p) by blast
  moreover have σ′ v0 = σ v0
    using σ assms(1) step.prems(2) v0-V unfolding winning-region-def by blast
  ultimately show ?thesis by simp
next
  assume v0 /∈ VV p
  thus ?thesis using v0-V strategy-extends-VVpstar step(4) step.prems(2) by simp
  qed
  thus ?thesis using step.hyps(3) step(4) σ vmc-path-ltl by blast
  qed
  qed
  qed

lemma (in vmc-path) path-hits-winning-region-is-winning:
  assumes σ': strategy p σ' ∀v. v ∈ winning-region p =⇒ winning-strategy p σ' v
and $\sigma$: $\forall v. \ v \in \text{winning-region } p \Rightarrow \sigma' v = \sigma v$
and $P$: $\text{lset } P \cap \text{winning-region } p \neq \emptyset$
shows $\text{winning-path } p \ P$

proof

obtain $n$ where $n: \text{enat } n < \text{llength } P \ P \ S \ n \in \text{winning-region } p$
using $P$ by (meson lset-intersect-lnth)
define $P'$ where $P' = \text{idropn } n \ P$
then interpret $P'$: $\text{vmc-path } G \ P' \ P \ S \ n \ p \ \sigma$
unfolding $P'$-def using $\text{vmc-path-idropn } n(1) \ \text{by blast}$
have $\text{winning-strategy } p \sigma'(P \ S \ n) \ \text{using } \sigma'(2) \ n(2) \ \text{by blast}$
hence $\text{lset } P' \subseteq \text{winning-region } p$
using $\text{lset P' \ s \ \text{paths-stay-in-winning-region}[OF } \sigma'(1) - \sigma]$
by blast

hence $\forall v. \ v \in \text{lset } P' \Rightarrow \sigma v = \sigma' v \ \text{using } \sigma$ by auto
hence $v \notin \text{lset } P$ using $\text{paths-stay-in-winning-region}$
unfolding $\text{winning-region-def}$
by blast

then interpret $P'$: $\text{vmc-path } G \ P' \ P \ S \ n \ p \ \sigma'$ using $\text{conforms-to-another-strategy}$ by blast
have $\text{winning-path } p \ P' \ \text{using } \sigma'(2) \ n(2) \ P' \ \text{vmc-path-axioms}$
winning-strategy-def by blast
thus $\text{winning-path } p \ P \ \text{using } \sigma'$ using $\text{conforms-to-another-strategy}$ by blast
qed

9.2 Irrelevant Updates

Updating a winning strategy outside of the winning region is irrelevant.

lemma winning-strategy-updates:
assumes $\sigma$: strategy $p \ \sigma$ winning-strategy $p \ \sigma v0$
and $v$: $v \notin \text{winning-region } p \ v \rightarrow w$
shows $\text{winning-strategy } p \ (\sigma(\nu := w)) \ v0$

proof
fix $P$ assume $\text{vmc-path } G \ P \ v0 \ p \ (\sigma(\nu := w))$
then interpret $\text{vmc-path } G \ P \ v0 \ p \ (\sigma(\nu := w))$

have $\forall v'. \ v' \in \text{winning-region } p \Rightarrow \sigma v' = (\sigma(\nu := w)) \ v'$ using $v$ by auto
hence $v \notin \text{lset } P$ using $v$ paths-stay-in-winning-region
unfolding $\text{winning-region-def}$ by blast
hence $\text{path-conforms-with-strategy } p \ \sigma$
using $\text{path-conforms-with-strategy-irrelevant-updates } P'$.P-conforms
by blast

then interpret $P'$: $\text{vmc-path } G \ P' \ P \ S \ n \ p \ \sigma'$ using $\text{conforms-to-another-strategy}$ by blast
have $\text{winning-path } p \ P' \ \text{using } \sigma'(2) \ n(2) \ P' \ \text{vmc-path-axioms}$
winning-strategy-def by blast
thus $\text{winning-path } p \ P$ unfolding $P'$-def using $\text{winning-path-drop-add } n(1) \ P$-valid by blast
qed

9.3 Extending Winning Regions

lemma winning-region-extends-VVp:
assumes $v$: $v \in \text{VV } p \rightarrow w$ and $w$: $w \in \text{winning-region } p$
shows $v \in \text{winning-region } p$

proof (rule ccontr)
obtain $\sigma$ where $\sigma$: strategy $p \ \sigma$ winning-strategy $p \ \sigma w$
using $w$ unfolding $\text{winning-region-def}$ by blast
let $\exists \sigma = \sigma(\nu := w)$
assume contra: $v \notin \text{winning-region } p$
moreover have strategy $p \ ?\sigma$ using $\text{valid-strategy-updates } \sigma(1) \ \rightarrow w \ \text{by blast}$
moreover hence $\text{winning-strategy } p \ ?\sigma \ v$

qed
using winning-strategy-updates σ contra v strategy-extends-backwards-VVp
by auto
ultimately show False using \( v \rightarrow w \) unfolding winning-region-def by auto
qed

Unfortunately, we cannot prove the corresponding theorem winning-region-extends-VVpstar for VV p∗∗-nodes yet. First, we need to show that there exists a uniform winning strategy on winning-region p. We will prove winning-region-extends-VVpstar as soon as we have this.

end — context ParityGame

10 Uniform Strategies

Theorems about how to get a uniform strategy given strategies for each node.

theory UniformStrategy
imports
  Main
  AttractingStrategy WinningStrategy WellOrderedStrategy WinningRegion
begin

context ParityGame begin

10.1 A Uniform Attractor Strategy

lemma merge-attractor-strategies:
  assumes \( S \subseteq V \)
  and strategies-ex: \( \forall v. v \in S \implies \exists \sigma. \text{strategy } p \sigma \land \text{strategy-attracts-via } p \sigma v S W \)
  shows \( \exists \sigma. \text{strategy } p \sigma \land \text{strategy-attracts } p \sigma S W \)
proof −
  define good where good = \{ \sigma. \text{strategy } p \sigma \land \text{strategy-attracts-via } p \sigma v S W \} for v
  let ?G = \{ \sigma. \exists v \in S - W. \sigma \in \text{good } v \}
  obtain r where r = well-order-on ?G r using well-order-on by blast

interpret WellOrderedStrategies G S = W p good r proof
  show S = W \subseteq V using \( \langle S \subseteq V \rangle \) by blast
next
  show \( \forall v. v \in S - W \implies \exists \sigma. \sigma \in \text{good } v \) unfolding good-def using strategies-ex by blast
next
  show \( \forall v. \sigma \in \text{good } v \implies \text{strategy } p \sigma \) unfolding good-def by blast
next
  fix v w σ assume v: v \in S - W v\rightarrow w v \in VV p \implies σ v = w σ \in \text{good } v
  hence σ: strategy p σ strategy-attracts-via p σ v S W unfolding good-def by simp-all
  hence strategy-attracts-via p σ w S W using strategy-attracts-via-successor v by blast
  thus σ \in \text{good } w unfolding good-def using σ(1) by blast
qed (insert r)

have S-W-no-deadends: \( \forall v. v \in S - W \implies \neg \text{deadend } v \)
  using strategy-attracts-via-no-deadends[of S W] strategies-ex
  by (metis (no-types) Diff-iff S-V rev-subsetD)

end

52
\{ 
  fix v0 assume v0 ∈ S 
  fix P assume P: vmc-path G P v0 p well-ordered-strategy 
  then interpret vmc-path G P v0 p well-ordered-strategy 
  have visits-via P S W proof (rule ccontr)
    assume contra: ¬visits-via P S W 
    have visits-via P S W proof (induct rule: \(\text{vmc-path-lset-induction}\))
      case base
        show v0 ∈ S − W using \(v0 ∈ S\) contra visits-via-trivial by blast
      next
        case (step P v0)
        interpret \(\text{vmc-path-no-deadend}\) G P v0 p well-ordered-strategy using step.hyps(1) .
        have insert v0 S = S using step.hyps(2) by blast
        hence ∗: ¬visits-via (ltl P) S W 
          using visits-via-LCons[of ltl P S W v0, folded P-LCons] step.hyps(3) by auto
        hence ∗∗: w0 /∈ W using \(\text{vmc-path.visits-via-trivial}[OF \text{vmc-path-ltl}]\) by blast
        have w0 ∈ S ∪ W proof (cases)
          assume v0 ∈ V V p
          hence well-ordered-strategy v0 = w0 using v0-conforms by blast
          hence choose v0 v0 = w0 using step.hyps(2) well-ordered-strategy-def by auto
          moreover have strategy-attracts-via p (choose v0) v0 S W 
            using choose-good good-def step.hyps(2) by blast
          ultimately show ?thesis
            by (metis strategy-attracts-via-successor strategy-attracts-via-v0 
               choose-strategy step.hyps(2) v0-edge-w0 w0-V)
        qed (metis DiffD1 assms(2) step.hyps(2) strategy-attracts-via-successor 
               strategy-attracts-via-v0 v0-edge-w0 w0-V)
      with ∗ ∗∗ show ?case by blast
    qed

  have ¬lfinite P proof 
    assume lfinite P
    hence deadend (llast P) using P-maximal P-not-null maximal-ends-on-deadend by blast
    moreover have llast P ∈ S − W using \(\text{lset P ⊆ S − W}\) \(\text{P-not-null \langle lfinite P \rangle} \text{lfinite-lset}\) by blast
    by blast
    ultimately show False using S-W-no-deadends by blast
    qed

  obtain n where n: path-conforms-with-strategy p (ldropn n P) (path-strategies P $ n) 
    using path-eventually-conforms-to-σ-map-n[OF \(\text{lset P ⊆ S − W}\) P-valid P-conforms] 
    by blast
  define σ' where [simp]: \(\text{σ'} = \text{path-strategies P} \$ \text{n}\) 
  define P' where [simp]: \(\text{P'} = \text{ldropn n P}\) 
  interpret vmc-path G P' lhd P' p σ' 
  proof
    show ¬lnull P' unfolding P'-def 
      using ¬lfinite P' lfinite-ldropn lnull-imp-lfinite by blast
    qed (simp-all add: n)
  have strategy p σ' unfolding σ'-def 
    using path-strategies-strategy \(\text{lset P ⊆ S − W}\) ¬lfinite P infinite-small-length
by blast
moreover have strategy-attracts-via \( \sigma' \) \((lhd P')\) \(S W\) proof—
have \(P \subseteq S - W\) using \(\{\text{set } P \subseteq S - W\}\) \(\langle \text{set } P \subseteq S - W\rangle\) \(\langle \text{set } P \subseteq S - W\rangle\) by blast

hence \(\sigma' \in \text{good } (P \subseteq S - W)\)

using path-strategies-good \(\sigma'\)-def \((\neg \text{finite } P)\) \(\{\text{set } P \subseteq S - W\}\) by blast

hence \(\text{strategy-attracts-via } P \sigma' \subseteq S W\) unfolding good-def by blast

thus \(\exists \sigma \text{ unfolding } P'\)-def using \(P-0\) by \((\text{simp add: } (\neg \text{finite } P) \langle \text{finite-small-length} \rangle)\)

qed
moreover from \(\{\text{set } P \subseteq S - W\}\) have \(\text{set } P' \subseteq S - W\)

ultimately show False using strategy-attracts-via-lset by blast

qed

}\)

thus \(\exists \sigma\) thesis unfolding well-ordered-strategy-valid by blast

qed

10.2 A Uniform Winning Strategy

Let \(S\) be the winning region of player \(p\). Then there exists a uniform winning strategy on \(S\).

lemma merge-winning-strategies:

shows \(\exists \sigma. \text{ strategy } p \sigma \land \forall v \in \text{winning-region } p. \text{ winning-strategy } p \sigma v\)

proof—

define good where good \(v = \{\sigma. \text{ strategy } p \sigma \land \text{winning-strategy } p \sigma v\}\) for \(v\)

let \(?G = \{\sigma. \exists v \in \text{winning-region } p. \sigma \in \text{good } v\}\)

obtain \(r\) where \(r\) : well-order-on \(?G\) \(r\) using well-order-on by blast

have no-VVp-deadends: \(\forall v. [v \in \text{winning-region } p; v \in VV p] \implies \neg \text{deadend } v\)

using no-winning-strategy-on-deadends unfolding winning-region-def by blast

interpret WellOrderedStrategies \(G\) \(\text{winning-region } p p \text{ good } r\) proof

show \(\forall v. v \in \text{winning-region } p \implies \exists \sigma. \sigma \in \text{good } v\)

unfolding good-def winning-region-def by blast

next

show \(\forall v. \sigma. \sigma \in \text{good } v \implies \text{strategy } p \sigma \text{ unfolding } \text{good-def by blast}\)

next

fix \(v w \sigma\) assume \(v. v \in \text{winning-region } p v \rightarrow w v \in VV p \implies \sigma v = w \sigma \in \text{good } v\)

hence \(\text{strategy } p \sigma \text{ winning-strategy } p \sigma v \text{ unfolding } \text{good-def by simp-all}\)

hence winning-strategy \(p \sigma w\) proof (cases)

assume \(*: v \in VV p\)

hence \(*: \sigma v = w\) using \(v(3)\) by blast

have \(\neg \text{deadend } v\) using no-VVp-deadends \(v \in VV p\) \(v(1)\) by blast

with \(* *\) show \(\exists \sigma \text{ unfolding } \text{strategy-extends-VVp } \sigma \) by blast

next

assume \(v \notin VV p\)

thus \(\exists \sigma \text{ unfolding } \text{strategy-extends-VVp } \sigma \) \(v \rightarrow w\) by blast

qed

thus \(\sigma \in \text{good } w\) unfolding good-def using \(\sigma(1)\) by blast

qed (insert winning-region-in-V \(r\))
fix v0 assume v0 ∈ \text{winning-region} p
fix P assume P: \text{vmc-path} G P v0 p \text{ well-ordered-strategy}
then interpret \text{vmc-path} G P v0 p \text{ well-ordered-strategy}.

have lset P ⊆ \text{winning-region} p \text{ proof (induct rule: vmc-path-lset-induction-simple)}
case (step P v0)
interpret \text{vmc-path-no-deadend} G P v0 p \text{ well-ordered-strategy using step.hyps(1)}.
\{ assume v0 ∈ \text{VV} p
  hence \text{well-ordered-strategy} v0 = w0 using v0-conforms by blast
  hence choose v0 v0 = w0 by \text{(simp add: step.hyps(2) well-ordered-strategy-def)}
\}
hence choose v0 ∈ \text{good} w0 using \text{strategies-continue} choose-good step.hyps.(2) by simp
thus \text{case unfolding good-def winning-region-def using w0-V by blast}
qed (insert \text{v0 ∈ \text{winning-region} p})

have \text{winning-path} p P \text{ proof (rule ccontr)}
assume contra: ~\text{winning-path} p P
have ~\text{\text{\text{lfinite}}} P \text{ proof}
assume \text{\text{lfinite}} P
hence \text{deadend} (\text{llast} P) using \text{maximal-ends-on-deadend} by simp
moreover have \text{llast} P ∈ \text{winning-region} p
  using \text{lset P ⊆ \text{winning-region} p} \text{ P-notch} \text{ lfinite \text{\text{-lset}} by blast}
moreover have \text{llast} P ∈ \text{VV} p
  using contra \text{paths-are-winning-for-one-player \text{lfinite} P}
  unfolding \text{winning-path-def by simp}
ultimately show \text{False using no-VVp-deadends by blast}
qed

obtain n where n: \text{path-conforms-with-strategy} p (ldropn n P) \text{ (path-strategies P $ n)}
using \text{path-eventually-conforms-to-\text{σ}\text{-map}-n[\text{OF \text{lset P ⊆ \text{winning-region} p}} \text{ P-valid P-conforms]}
by blast
define \text{σ'} where [simp]: \text{σ'} = \text{path-strategies} P \text{ $ n}
define P' where [simp]: P' = ldropn n P
interpret P': \text{vmc-path} G P' lhd P' p \text{ σ' proof}
  show ~\text{\text{lnull}} P' using ~\text{lfinite} P' unfolding P'\text{-def}
    using \text{\text{lfinite-ldropn lnull-imp-lfinite by blast}
qed (simp-all add: n)
have \text{strategy} p \text{ σ' unfolding σ'\text{-def}
  using \text{path-strategies-strategy} \text{lset P ⊆ \text{winning-region} p} \text{ ~\text{lfinite} P} by blast
moreover have \text{winning-strategy} p \text{ σ'} (lhd P') \text{ proof--
  have P \text{ $ n} ∈ \text{winning-region} p
    using \text{lset P ⊆ \text{winning-region} p} \text{ ~\text{lfinite} P} \text{ lset-nth-member-inf by blast}
    hence σ' ∈ \text{good} (P \text{ $ n})
    using \text{path-strategies-good choose-good σ'\text{-def ~\text{lfinite} P} \text{ lset P ⊆ \text{winning-region} p}
      by blast
    hence \text{winning-strategy} p \text{ σ'} (P \text{ $ n}) \text{ unfolding good-def by blast}
    thus \text{thesis}
      unfolding P'\text{-def using P-0 ~\text{lfinite} P} by (simp add: infinite-small-length lhd-ldropn)
qed
ultimately have \text{winning-path} p P' unfolding \text{winning-strategy-def}
  using P'.vmc-path-axioms by blast

55
moreover have ¬finite P' using ¬finite P P'-def by simp
ultimately show False using contra winning-path-drop-add[OF P-valid] by auto
qed
}
thus ?thesis unfolding winning-strategy-def using well-ordered-strategy-valid by auto
qed

10.3 Extending Winning Regions

Now we are finally able to prove the complement of winning-region-extends-VVp for VV p** nodes, which was still missing.

lemma winning-region-extends-VVpstar:
assumes v: v ∈ VV p** and w: ∀w. v→w ⇒ w ∈ winning-region p
shows v ∈ winning-region p
proof −
  obtain σ where σ: strategy p σ ∃v. v ∈ winning-region p ⇒ winning-strategy p σ v using merge-winning-strategies by blast
  have winning-strategy p σ v using strategy-extends-backwards-VVpstar[OF v σ(1)] σ(2) w by blast
  thus ?thesis unfolding winning-region-def using v σ(1) by blast
qed

It immediately follows that removing a winning region cannot create new deadends.

lemma removing-winning-region-induces-no-deadends:
assumes v ∈ V − winning-region p ¬deadend v
shows ∃w ∈ V − winning-region p. v→w
using assms winning-region-extends-VVp winning-region-extends-VVpstar by blast

end — context ParityGame

end

11 Attractor Strategies

theory AttractorStrategy
imports
  Main
  Attractor UniformStrategy
begin

This section proves that every attractor set has an attractor strategy.

context ParityGame begin

lemma strategy-attracts-extends-VVp:
assumes σ: strategy p σ strategy-attracts p σ S W
and v0: v0 ∈ VV p v0 ∈ directly-attracted p S v0 /∈ S
shows ∃σ. strategy p σ ∧ strategy-attracts-via p σ v0 (insert v0 S) W
proof −
  from v0(1.2) obtain w where v0→w w ∈ S using directly-attracted-def by blast
  from ∀w ∈ S σ(2) have strategy-attracts-via p σ w S W unfolding strategy-attracts-def by blast

56
let $\sigma = \sigma(v_0 := w)$ — Extend $\sigma$ to the new node.

have strategy $p \sigma$ using $\sigma(1) \langle v_0 \rightarrow w \rangle$ valid-strategy-updates by blast

moreover have strategy-attracts-via $p \sigma v_0$ (insert $v_0 S$) $W$ proof
  fix $P$
  assume vmc-path $G P v_0 p \sigma$
  then interpret vmc-path $G P v_0 p \sigma$.
  have $\neg$deadend $v_0$ using $\langle v_0 \rightarrow w \rangle$ by blast
  then interpret vmc-path-no-deadend $G P v_0 p \sigma$ by unfold-locales

define $P''$ where $\langle simp \rangle$: $P'' = \text{ltl} P$

have $\text{lhd} P'' = w$ using $v_0(1)$ $v_0$-conforms $w_0$-def by auto

hence vmc-path $G P'' w p \sigma$ using vmc-path-ltl by (simp add: $w_0$-def)

have $\ast$: $v_0 \notin S \rightarrow W$ using $\langle v_0 \notin S \rangle$ by blast

have override-on $\langle \sigma(v_0 := w) \rangle \sigma (S - W) = \sigma$ by (rule ext) (metis fun-upd-def override-on-def)

hence strategy-attracts $p \sigma S W$
  by (simp add: strategy-attracts-irrelevant-ignore[OF $\sigma(2,1)$ strategy $p \sigma$])

hence strategy-attracts-via $p \sigma w S W$ unfolding strategy-attracts-def
  using $\langle w \in S \rangle$ by blast

hence visits-via $P'' S W$ unfolding strategy-attracts-via-def
  using vmc-path $G P'' w p \sigma$ by blast

thus visits-via $P (\text{insert} v_0 S) W$
  by (simp add: visits-via-LCons[of ltl $P S W v_0$] P-LCons)

qed

ultimately show ?thesis by blast

qed

lemma strategy-attracts-extends-VVpstar:
  assumes $\sigma$: strategy-attracts $p \sigma S W$
    and $v_0$: $v_0 \notin V'$ $v_0 \in \text{directly-attracted} p S$
  shows strategy-attracts-via $p \sigma v_0 (\text{insert} v_0 S) W$
proof
  fix $P$
  assume vmc-path $G P v_0 p \sigma$
  then interpret vmc-path $G P v_0 p \sigma$.
  have $\neg$deadend $v_0$ using $v_0(2)$ directly-attracted-contains-no-deadends by blast
  then interpret vmc-path-no-deadend $G P v_0 p \sigma$ by unfold-locales
  have visits-via (lhd $P$) $S W$
    using vmc-path.strategy-attractsE[OF vmc-path-ltl $\sigma$] $v_0$ directly-attracted-def by simp
  thus visits-via $P (\text{insert} v_0 S) W$ using visits-via-LCons[of ltl $P S W v_0$] P-LCons by simp

qed

lemma attractor-has-strategy-single:
  assumes $W \subseteq V$
    and $v_0$-def: $v_0 \in \text{attractor} p W$ ($\text{is} - \in \text{?A}$)
  shows $\exists \sigma$. strategy $p \sigma$ $\land$ strategy-attracts-via $p \sigma v_0 ?A W$
using assms proof (induct arbitrary: $v_0$ rule: attractor-set-induction)
  case (step $S$)
  have $v_0 \in W \Rightarrow \exists \sigma$. strategy $p \sigma$ $\land$ strategy-attracts-via $p \sigma v_0 \{ \}$ $W$
    using strategy-attracts-via-trivial valid-arbitrary-strategy by blast

moreover 

57
\textbf{11.1 Existence}

Prove that every attractor set has an attractor strategy.

\textbf{theorem} attractor-has-strategy:
\begin{itemize}
  \item \textbf{assumes} $W \subseteq V$
  \item \textbf{shows} $\exists \sigma. \text{strategy } p \sigma \land \text{strategy-attracts } p \sigma (\text{attractor } p W) W$
\end{itemize}
\textbf{proof—}
\begin{itemize}
  \item let $?A = \text{attractor } p W$
  \item have $?A \subseteq V$ by (simp add: $\langle W \subseteq V \rangle$, attractor-in-V)
  \item moreover
    \begin{itemize}
      \item have $\ \land v, v \in ?A \implies \exists \sigma. \text{strategy } p \sigma \land \text{strategy-attracts-via } p \sigma v ?A W$
      \item using $\langle W \subseteq V \rangle$, attractor-has-strategy-single by blast
    \end{itemize}
  \item ultimately show $?\text{thesis using merge-attractor-strategies } \langle W \subseteq V \rangle$ by blast
\end{itemize}
\textbf{qed}

\begin{flushright}
end — context ParityGame
\end{flushright}

\textbf{12 Positional Determinacy of Parity Games}

\textbf{theory} PositionalDeterminacy
\textbf{imports}
  \begin{itemize}
    \item Main
    \item AttractorStrategy
  \end{itemize}
\textbf{begin}
\begin{flushright}
context ParityGame begin
\end{flushright}

\textbf{58}
12.1 Induction Step

The proof of positional determinacy is by induction over the size of the finite set \( \omega \cdot V \), the set of priorities. The following lemma is the induction step.

For now, we assume there are no deadends in the graph. Later we will get rid of this assumption.

**Lemma** positional-strategy-induction-step:

**Assumes** \( v \in V \)

**And** no-deadends: \( \forall v \in V \Rightarrow \neg \text{deadend } v \)

**And** \( \text{IH}: \bigwedge (G :: (a, 'b) ParityGame-scheme) v. \)

[ \text{card} (\omega_G \cdot V_G) < \text{card} (\omega \cdot V); v \in V_G; ParityGame G; \]

\[ \forall v, v \in V_G \Rightarrow \neg \text{Digraph.deadend } G \ v \]

\[ \Rightarrow \exists p. v \in \text{ParityGame}.\text{winning-region } G \ p \]

**Shows** \( \exists p. v \in \text{winning-region } p \)

**Proof**

First, we determine the minimum priority and the player who likes it.

**Define** min-prio where \( \text{min-prio} = \text{Min} (\omega \cdot V) \)

**Have** \( \exists p. \text{winning-priority } p \ \text{min-prio} \)

**Then obtain** \( p \) where \( p: \text{winning-priority } p \ \text{min-prio} \)

Then we define the tentative winning region of player \( p^{**} \). The rest of the proof is to show that this is the complete winning region.

**Define** \( W1 \) where \( W1 = \text{winning-region } p^{**} \)

For this, we define several more sets of nodes. First, \( U \) is the tentative winning region of player \( p \).

**Define** \( U \) where \( U = V - W1 \)

**Define** \( K \) where \( K = U \cap (\omega - \{ \text{min-prio} \}) \)

**Define** \( V' \) where \( V' = U - \text{attractor } p \ K \)

**Define** \( G' \) where [simp]: \( G' = \text{subgame } V' \)

**Interpret** \( G': \text{ParityGame } G' \)

**Using** subgame-ParityGame by simp

**Have** \( U\text{-equiv}: \forall v, v \in V \Rightarrow v \in U \leftrightarrow v \notin \text{winning-region } p^{**} \)

**Unfolding** U-def W1-def by blast

**Have** \( V' \subseteq V \)

**Unfolding** U-def V'-def by blast

**Hence** [simp]: \( V' = V' \)

**Unfolding** G'-def by simp

**Have** \( G', V' \ p = V' \cap VV \ p \)

**Unfolding** G'-def using subgame-VV by simp

**Have** \( V \text{-decomp}: V = \text{attractor } p \ K \cup V' \cup W1 \)

**Proof**

**Have** \( V \subseteq \text{attractor } p \ K \cup V' \cup W1 \)

**Unfolding** V'-def U-def by blast

**Moreover have** \( \text{attractor } p \ K \subseteq V \)

**Using** attractor-in-V'[of K] unfolding K-def U-def by blast

**Ultimately show** \( ?\text{thesis} \)

59
It turns out the winning region of player $p$ is empty, so we have a strategy for player $p$.

have $v \in G'.winning-region$ \textbf{proof} (rule ccontr)
  assume $\neg \text{thesis}$
  moreover obtain $p' \sigma$ where $p', G'.strategy p' \sigma G'.winning-strategy p' \sigma v$
  using $G'.winning-strategy unfolding G'.winning-region-def$ \textbf{by blast}
  ultimately have $p' \neq p$ \textbf{proof} $v \in V_{G'}$ \textbf{unfolding} $G'.winning-region-def$ \textbf{by blast}
  hence $p' = p**$ (cases $p$; cases $p'$) \textbf{auto}
  with $p'$ have $\sigma: G'.strategy p** \sigma G'.winning-strategy p** \sigma v$ \textbf{by simp-all}

By definition of $W1$, we obtain a winning strategy on $W1$ for player $p**$.

obtain $\sigma W1$ where $\sigma W1$
  strategy $p** \sigma W1 \bigvee v \in W1 \implies \text{winning-strategy } p** \sigma W1 v$

unfolding $W1-def$ \textbf{using} merge-winning-strategies \textbf{by blast}

$\{ \begin{align*}
\text{fix } v & \text{ assume } v \in V_{G'} \\
\text{have } & G'-\text{no-deadends;} \bigvee v, v \in V_{G'} \implies \neg G'.\text{deadend } v \textbf{proof} - \\
& \text{fix } v \text{ assume } v \in V_{G'} \\
& \text{hence } \ast: v \in U - \text{attractor } p K \textbf{proof using} \langle V_{G'} = V' \rangle, V'-\text{def by blast} \\
& \text{moreover hence } \exists w \in U. v \to w \\
& \text{using removing-winning-region-induces-no-deadends[of } v \text{ } p**] \textbf{by blast} \\
& \text{moreover have } \bigwedge \text{ w, } [ v \in V_{P**}; v \to w ] \implies w \in U \\
& \text{using } U-equiv winning-region-extends-VVp \textbf{by blast} \\
& \text{ultimately have } \exists w \in V'. v \to w \\
& \text{using } U-equiv winning-region-extends-VVp \textbf{by blast} \\
& \text{thus } \neg G'.\text{deadend } v \textbf{ proof } (v \in V_{G'}) \langle V' \subseteq V \rangle \textbf{ by simp} \\
\textbf{qed} \\
\end{align*} \}$

Apply the induction hypothesis to get the winning strategy for $v$ in $G'$.

have $G'-\text{winning-strategy;} \exists p, v \in G'.\text{winning-region } p$ \textbf{proof} - \\
have $\text{card } (\omega_{G'}' \setminus V_{G'}) < \text{card } (\omega \setminus V)$ \textbf{proof} - \\
  $\{ \begin{align*}
& \text{assume } \text{min-prio } \in \omega_{G'}' \setminus V_{G'} \\
& \text{then obtain } v \text{ where } v, v \in V_{G'}, \omega_{G'}, v = \text{min-prio } \textbf{by blast} \\
& \text{hence } v \in \omega' \{ \text{min-prio} \} \textbf{ by simp} \\
& \text{hence } False \textbf{ proof using } V'-\text{def } K'-\text{def } \text{attractor-set-base } (V_{G'} = V' \setminus v(1)) \\
& \textbf{by (metis DiffD1 DiffD2 IntI contra-subsetD)} \\
\} \\
& \text{hence } \text{min-prio } \notin \omega_{G'}' \setminus V_{G'} \textbf{ by blast} \\
& \text{moreover have } \text{min-prio } \in \omega \setminus V \\
& \text{unfolding } \text{min-prio-def using priorities-finite Min-in assms(1)} \textbf{ by blast} \\
& \text{moreover have } \omega_{G'}' \setminus V_{G'} \subseteq \omega \setminus V \textbf{ unfolding } G'-\text{def by simp} \\
& \text{ultimately show } \text{thesis by (metis priorities-finite psubsetI psubset-card-mono)} \\
& \textbf{qed} \\
& \text{thus } \text{thesis using IH[of } G' \rangle \langle v \in V_{G'} \rangle \text{ G'-no-deadends } G'.\text{PurityGame-axioms by blast} \\
& \textbf{qed} \\

unfolding } W1-def \textbf{ using } \langle V' \subseteq V \rangle \textbf{ by blast} \\
\textbf{qed} 

have $v \in \text{winning-region } p**$ proof

show $v \in V$ using $(v \in V_G') \land V_G' \subseteq V'$ by blast

define $\sigma'$ where $\sigma' = \text{override-on } (\text{override-on } \sigma \text{-arbitrary } \sigma W1 W1) \sigma' V'$

thus strategy $p** \sigma'$

using $\text{valid-strategy-updates-set-strong } \sigma W1(1)$

valid-strategy-supergame $\sigma(1)$ $G'$-no-deadends $(V_G' = V')$

unfolding $G'$-def by blast

show winning-strategy $p** \sigma' v$

proof (rule winning-strategyI, rule ccontr)

fix $P$ assume $\text{vmc-path } G P v p** \sigma'$

then interpret $\text{vmc-path } G P v p** \sigma'$.

assume $\neg \text{winning-path } p** P$

First we show that $P$ stays in $V'$, because if it stays in $V'$, then it conforms to $\sigma$, so it must be winning for $p**$.

have $\text{lset } P \subseteq V'$ proof (induct rule: $\text{vmc-path-lset-induction-closed-subset}$)

fix $v$ assume $v \in V' \neg \text{deadend } v \in V V p**$

hence $v \in \text{ParityGame } VV$ (subgame $V'$) $p**$ by auto

moreover have $\neg G', \text{deadend } v$ using $G'$-no-deadends $(V_G' = V') \land \forall v \in V'$ by blast

ultimately have $\sigma v \in V'$

using $\text{subgame-strategy-stays-in-subgame } p'(1) \land p' = p**$

unfolding $G'$-def by blast

thus $\sigma' v \in V' \cup W1$ unfolding $\sigma'$-def using $(v \in V')$ by simp

next

fix $v w$ assume $v \in V' \neg \text{deadend } v \in V V p***** v \rightarrow w$

show $w \in V' \cup W1$ proof (rule ccontr)

assume $w \notin V' \cup W1$

hence $w \in \text{attractor } p K$ using $V-decomp \ (v \rightarrow w)$ by blast

hence $v \in \text{attractor } p K$ using $(v \in V V p***** \Rightarrow \text{attractor-set } VV p \ (v \rightarrow w))$ by auto

thus False using $(v \in V') \land V'-\text{def}$ by blast

qed

next

have $\forall v. v \in W1 \Rightarrow \sigma W1 v = \sigma' v$ unfolding $\sigma'$-def $V'-\text{def}$ $U-\text{def}$ by simp

thus $\text{lset } P \cap W1 = \{\}$

using $\text{path-hits-winning-region-is-winning } \sigma W1 \ (\neg \text{winning-path } p** P)$

unfolding $W1-\text{def}$

by blast

next

show $v \in V'$ using $(V_G' = V') \land v \in V G'$ by blast

qed

This concludes the proof of $\text{lset } P \subseteq V'$.

hence $G', \text{valid-path } P$ using subgame-valid-path by simp

moreover have $G', \text{maximal-path } P$

using $\langle \text{lset } P \subseteq V' \rangle \land \text{subgame-maximal-path } (V' \subseteq V)$ by simp

moreover have $G', \text{path-conforms-with-strategy } p** P \sigma$ proof—

have $G', \text{path-conforms-with-strategy } p** P \sigma'$

using $\text{subgame-path-conforms-with-strategy } (V' \subseteq V) \langle \text{lset } P \subseteq V' \rangle$

by simp

moreover have $\forall v. v \in \text{lset } P \Rightarrow \sigma' v = \sigma v$ using $(\text{lset } P \subseteq V') \land v \in V V p***** v \rightarrow w$

by auto

61
ultimately show thesis  
using $G'.path$-conforms-with-strategy-irrelevant-updates by blast  

qed 
ultimately have $G'.winning-path p** P$  
using $\sigma(2) G'.winning-strategy-def G'.valid-maximal-conforming-path-0 P-0 P-not-null by blast  

moreover have $G'.VV p**** \subseteq VV p**** using subgame-VV-subset G'-def by blast  
ultimately show False  
using $G'.winning-path-supergame[of p**] \omega_{G'} = \omega$  
\text{\textcircled{\smallsym{\omega}} winning-path} \ \text{\textcircled{\smallsym{\omega}} ParityGame-axioms}

by blast  

qed 

moreover have $v \in V$ using \text{\textcircled{\smallsym{V}}} $G' \subseteq V$ by blast  
ultimately have $v \in W1 unfolding W1-def winning-region-def by blast  
thus False using $v \in V G' using U-def V'-def V' = V \Leftrightarrow v \in V G'$ by blast  

qed 

}\ text{\textcircled{\smallsym{V}}} note recursion = this

We compose a winning strategy for player $p$ on $V-W1$ out of three pieces.

First, if we happen to land in the attractor region of $K$, we follow the attractor strategy. This is good because the priority of the nodes in $K$ is good for player $p$, so he likes to go there.

\text{obtain $\sigma1$  
where $\sigma1$: strategy $p \sigma1$  
strategy-attracts $p \sigma1$ (attractor $p K$) $K$  
using $\text{\textcircled{\smallsym{\omega}}}$ attractor-has-strategy[of $K p$] $K-def U-def$ by auto  

Next, on $G'$ we follow the winning strategy whose existence we proved earlier.

\text{have $G'.winning-region p = V G'$ using recursion unfolding $G'.winning-region-def$ by blast  
then obtain $\sigma2$  
where $\sigma2$: $\forall v. v \in V G' \Rightarrow G'.strategy p \sigma2$  
$\forall v. v \in V G' \Rightarrow G'.winning-strategy p \sigma2 v$  
using $G'.merge-winning-strategies$ by blast  

As a last option we choose an arbitrary successor but avoid entering $W1$. In particular, this defines the strategy on the set $K$.

\text{define succ where succ $v = (\text{\textcircled{\smallsym{\omega}}}$ SOME $w. v \rightarrow w \land (v \in W1 \lor w \notin W1))$ for $v$  

Compose the three pieces.

\text{define $\sigma$ where $\sigma = \text{\textcircled{\smallsym{\omega}}}$ override-on (override-on succ $\sigma2 V'$ $\sigma1$ (attractor $p K - K$)  

have attractor $p K \cap W1 = \{\}$ proof (rule ccontr)  
assume attractor $p K \cap W1 \neq \{\}$  
then obtain $v$ where $v \in$ attractor $p K v \in W1$ by blast  

hence $v \in V$ using $\text{\textcircled{\smallsym{\omega}}}$ $W1-def winning-region-def$ by blast  

obtain $P$ where vmc2-path $G p v p \sigma1 \sigma W1$  
using strategy-conforming-path-exists $\sigma W1(1) \sigma1(1) v \in V$ by blast  
then interpret $\text{\textcircled{\smallsym{\omega}}}$ vmc2-path $G p v p \sigma1 \sigma W1$.
have strategy-attracts-via \( p \sigma 1 \) \( v \) \( \langle \text{attractor } p \ K \rangle \ K \) using \( v(1) \) \( \sigma 1(2) \) strategy-attracts-def by blast

hence \( \exists \) \( \cdot \) \( v \) \( \in \) \( \text{attractor } p \ K \) \(- K \) \( \Rightarrow \) \( v \) \( = \) \( \sigma 1 \) \( v \) unfolding \( \sigma \)-def by simp

hence \( \neg \exists \cdot \) \( \neg v \in W1 \) unfolding \( K\)-def \( U\)-def by blast

thus False unfolding \( W1\)-def using comp.paths-stay-in-winning-region \( \sigma \) \( W1 \) \( v(2) \) by auto

qed

On specific sets, \( \sigma \) behaves like one of the three pieces.

have \( \sigma\cdot\sigma 1\cdot \langle \forall \cdot \) \( v \) \( \in \) \( \text{attractor } p \ K \) \(- K \) \( \Rightarrow \) \( v \) \( = \) \( \sigma 1 \) \( v \) unfolding \( \sigma\)-def by simp

have \( \sigma\cdot\sigma 2\cdot \langle \forall \cdot \) \( v \in V' \) \( \Rightarrow \) \( v \) \( = \) \( \sigma 2 \) \( v \) unfolding \( \sigma\)-def \( V'\)-def by auto

have \( \sigma\cdot K\cdot \langle \forall \cdot \) \( v \) \( \in \) \( K \cup W1 \) \( \Rightarrow \) \( v \) \( = \) \( \sigma \) \( v \) succ by proof-

fix \( v \) assume \( v \) \( \in \) \( K \cup W1 \)

hence \( \exists \cdot \) \( \exists v \not \in V' \) unfolding \( V'\)-def \( U\)-def using attrator-set-base by auto

with \( v \) show \( \sigma \) \( v \) succ unfolding \( \sigma\)-def \( U\)-def using \( \langle \text{attractor } p \ K \cap W1 = \{\} \rangle \)

by (metis (mono-tags, lifting) Diff-iff IntI UnE override-on-def override-on-emptyset)

qed

Show that \( \sigma \) succeeds in avoiding entering \( W1 \).

\{ fix \( v \) assume \( v \) \( \in \) \( \forall \cdot \) \( VV \) \( p \)

hence \( \neg \cdot \) \( \neg \) \( \cdot \) \( \text{deadend } v \) unfolding \( \text{no-deadends} \) by blast

have \( \exists \cdot \) \( \forall w \cdot \neg \cdot \) \( (v \in W1 \lor w \not \in W1) \) proof (cases)

assume \( v \) \( \in \) \( W1 \)

thus ?thesis using \( \text{no-deadends} \) \( \langle \text{deadend } v \rangle \) by blast

next

assume \( v \not \in W1 \)

show ?thesis proof (rule ccontra)

assume \( \neg (\exists \cdot \) \( \exists w \cdot \neg \cdot \) \( (v \in W1 \lor w \not \in W1)) \)

hence \( \forall w \cdot \neg \cdot \) \( (v \in W1 \lor w \not \in W1) \) unfolding \( \text{no-deadends} \) by blast

hence winning-strategy \( \forall w \cdot \neg \cdot \) \( (v \in W1(2)) \) by blast

hence winning-strategy \( \forall w \cdot \neg \cdot \) \( (v \in W1) \) by auto

thus \( \neg \cdot \) \( \neg \) \( \cdot \) \( \text{deadend } v \) unfolding \( \text{winning-region-def} \) using \( \sigma W1(1) \) \( \langle \text{deadend } v \rangle \) by blast

thus False using \( v \not \in W1 \) by blast

qed

show ?thesis unfolding succ-def

using somel-ex [of \( \lambda w . v \rightarrow w \land (v \in W1 \lor w \not \in W1) \)] by blast+

\} note succ-works = \( \langle \cdot \rangle \)

have strategy \( p \) \( \sigma \)

proof

fix \( v \) assume \( v \) \( \in \) \( \forall \cdot \) \( VV \) \( p \) \( \neg \cdot \) \( \text{deadend } v \)

hence \( v \in \text{attractor } p \ K \) \(- K \) \( \Rightarrow \) \( v \rightarrow \sigma \) \( v \) unfolding \( \sigma\)-attractor-1 \( v \) unfolding strategy-def by auto

moreover have \( v \in V' \) \( \Rightarrow \) \( v \rightarrow \sigma \) \( v \) unfolding by auto-

assume \( v \in V' \)

moreover have \( v \in V \) \( G' \) \( VV \) \( p \) unfolding \( \langle G' . \forall v \in V(1) \rangle \) \( (v \in V') \) \( (v \in VV \) \( p \) \) by blast

moreover have \( \neg \) \( \langle \langle \text{Digraph.attractor } G \rangle \) \( v \) using \( \langle G' . \forall v \in V \rangle \) \( G' \)-no-deadends \( \langle v \in V \) \( G' \rangle \) by blast

ultimately have \( v \rightarrow G' \) \( \sigma 2 \) \( v \) unfolding \( \sigma 2(1) \) \( G' \)-strategy-def[of \( p \) \( \sigma 2 \) ] by blast

with \( v \in V' \) show \( v \rightarrow \sigma \) \( v \) unfolding \( \langle E' \subseteq E \rangle \) \( \sigma\)-attractor-2 by (metis subsetCE)

qed

moreover have \( v \in \overline{K} \cup W1 \) \( \Rightarrow \) \( v \rightarrow \sigma \) \( v \) unfolding succ-works \( \langle \cdot \rangle \) \( v \) \( \sigma-K \) by auto
moreover have $v \in V$ using $(v \in V \land p \land q)$ by blast
ultimately show $v \to \sigma \land q$ using $V$-decomp by blast
qed

have $\sigma$-attracts: strategy-attracts $p \sigma$ (attractor $p \land K$) $K$ proof
have strategy-attracts $p \sigma$ (attractor $p \land K$) $K$ proof
using strategy-attracts-irrelevant-override $\sigma_1$ (strategy $p \sigma$) by blast
moreover have $\sigma = \text{override-on } \sigma \land \sigma_1$ (attractor $p \land K$)
by (rule ext) (simp add: override-on-def $\sigma \land \sigma_1$)
ultimately show $\text{thesis by simp}$
qed

Show that $\sigma$ is a winning strategy on $V - W1$.

have $\forall v \in V - W1$. winning-strategy $p \sigma \land q$ proof (intro ballI winning-strategyI)
fix $v \in V$ assume $P : v \in V - W1$ vmc-path $G \land p \sigma$
interpret vmc-path $G \land p \sigma$ using $P(2)$

have lset $P \subseteq V - W1$
proof (induct rule: vmc-path-lset-induction-closed-subset)
fix $v$ assume $v \in V - W1 \land \neg \text{deadend } v \land v \in V \land p$
show $\sigma \land v \in V - W1 \cup \{\}$ proof (rule ccontr)
assume $\neg \text{thesis}$
hence $\sigma \land v \in W1$
using (strategy $p \land \sigma$) $\neg \text{deadend } v \land v \in V \land p$
unfolding strategy-def by blast
hence $v \notin K$ using succ-works(2)[OF $v \in V \land p$] $v \in V - W1 \land \sigma \land K$ by auto
moreover have $v \notin K$ proof
assume $v \in \text{attractor } p \land K - K$
proof
hence $\sigma \land v \in \text{attractor } p \land K$
using attracted-strategy-step (strategy $p \land \sigma$) $\sigma$-attracts $\neg \text{deadend } v \land v \in V \land p$
attractor-set-base
by blast
thus False using $(\sigma \land v \in W1) \land (\text{attractor } p \land K \land W1 = \{\})$ by blast
qed
moreover have $v \notin V'$ proof
assume $v \in V'$
have $\sigma \land v \in V'$ proof (rule $G'$.valid-strategy-in-$V[\sigma \land p \land \sigma 2 \land v]$)
have $v \in V'$ using $V' = v \land v \in V'$ by simp
thus $\neg G'.\text{deadend } v \text{ using } G'.\text{no-deadends by blast}$
proof
hence $\sigma \land v \in V'$ using $(v \in V' \land p \land \sigma \land 2 \land v)'$ by blast
proof
show $v \in G'.VV \land p \land q \land (v \in V' \land p \land \sigma \land 2 \land v)'$ by simp
hence $\sigma \land v \in V'$ using $(v \in V' \land p \land \sigma \land 2 \land v)'$ by simp
thus False using $V' = v \land (\sigma \land v \in W1) \land V'\land U\land \text{def by blast}$
qed
ultimately show False using $(v \in V - W1) \land V\land \text{decomp by blast}$
qed
next
fix $v \land w$ assume $v \in V - W1 \land \neg \text{deadend } v \land v \in V \land p \land \sigma 2 \land v' \land w$
proof (rule ccontr)
assume $\neg \text{thesis}$
hence \( w \in W_1 \) using \( \langle v \to w \rangle \) by blast
let \( \exists \alpha = \sigma W_1 (v := w) \)
have winning-strategy \( p^* \sigma W_1 w \) using \( \langle w \in W_1 \rangle \sigma W_1 (2) \) by blast
moreover have \( \neg (\exists \alpha. \text{strategy } p^* \bigwedge \text{winning-strategy } p^* \sigma v) \)
using \( \langle v \in V - W_1 \rangle \) unfolding \( W_1\)-def winning-region-def by blast
ultimately have winning-strategy \( p^* \sigma W_1 w \)
unfolding \( \text{winning-region-def} \) by blast
moreover have strategy \( p^* \sigma \) using \( \langle v \to w \rangle \) by blast
ultimately have winning-strategy \( p^* \sigma \)
valid-strategy-updates \[ \bigwedge \text{of } v \bigwedge \text{strategy } p^* \sigma \]
by auto
hence \( v \in W_1 \) unfolding \( W_1\)-def winning-region-def
using \( \langle v \to w \rangle \) by blast
thus False using \( \langle v \in V - W_1 \rangle \) by blast
qed

This concludes the proof of \( \text{lset } P \subseteq V - W_1 \).

hence \( \text{lset } P \subseteq \text{attractor } p K \cup V' \) using \( V\)-decomp by blast

have \( \neg \text{finite } P \)
using \( \text{no-deadends } \text{lfinite-lset maximal-ends-on-deadend}[\text{of } P] \) \( P\)-maximal \( P\)-not-null \( \text{lset-P-V} \)
by blast

Every \( \sigma\)-conforming path starting in \( V - W_1 \) is winning. We distinguish two cases:

1. \( P \) eventually stays in \( V' \). Then \( P \) is winning because \( \sigma 2 \) is winning.
2. \( P \) visits \( K \) infinitely often. Then \( P \) is winning because of the priority of the nodes in \( K \).

show winning-path \( p P \)
proof (cases)
assume \( \exists n. \text{lset } (\text{ldropn } n P) \subseteq V' \)

The first case: \( P \) eventually stays in \( V' \).

then obtain \( n \) where \( n: \text{lset } (\text{ldropn } n P) \subseteq V' \) by blast
define \( P' \) where \( P' = \text{ldropn } n P \)
hence \( \text{lset } P' \subseteq V' \) using \( n \) by blast
interpret \( \text{vmc-path}': \text{vmc-path } G' P' \) \( \text{lhs } P' p \sigma 2 \) proof
show \( \neg \text{null } P' \) unfolding \( P'\)-def
using \( \neg \text{finite } P' \) \( \text{finite-ldropn null-imp-finite} \) by blast
show \( G'.\text{valid-path } P' \) proof–
  have valid-path \( P' \) unfolding \( P'\)-def by simp
thus \( \?\text{thesis} \) using \( \text{subgame-valid-path } \langle \text{lset } P' \subseteq V' \rangle G'\)-def by blast
qed
show \( G'.\text{maximal-path } P' \) proof–
  have maximal-path \( P' \) unfolding \( P'\)-def by simp
thus \( \?\text{thesis} \) using \( \text{subgame-maximal-path } \langle \text{lset } P' \subseteq V' \rangle (V' \subseteq V) G'\)-def by blast
qed
show \( G'.\text{path-conforms-with-strategy } p P' \sigma 2 \) proof–
have path-conforms-with-strategy p P' σ unfolding P'-def by simp
hence path-conforms-with-strategy p P' σ2 by blast
  using path-conforms-with-strategy-irrelevant-updates \lset P' ⊆ V' \ σ-σ2
  by blast
thus ?thesis
  using subgame-path-conforms-with-strategy \lset P' ⊆ V' \ \ lset V' ⊆ V' \ G'-def
  by blast
qed
qed simp
have G'.winning-strategy p σ2 \ lhd P'
  using \lset P' ⊆ V' \ vmc-path'.P-not-null σ2(2)[of \ lhd P'] \ ω G' = V' \ llset.set.sel(1)
  by blast
hence G'.winning-path p P' using G'.winning-strategy-def vmc-path'.vmc-path-axioms by blast
moreover have G'.VV p** ⊆ VV p** unfolding G'-def using subgame-VV by simp
ultimately have winning-path p P'
  using G'.winning-path-supergame[of p P' G] \ ω G' = ω ParityGame-axioms by blast
thus ?thesis
unfolding P'-def
  using infinite-small-l-length[OF (~finite P)]
  winning-path-drop-add[of P p n] P-valid
  by blast
next
assume asm: ~(∃n. \ lset (ldropn n P) ⊆ V')
The second case: P visits K infinitely often. Then min-prio occurs infinitely often on P.

have min-prio ∈ path-inf-priorities P
unfolding path-inf-priorities-def proof (intro CollectI allI)
  fix n
  obtain k1 where k1: ldropn n P $ k1 \notin V' using asm by (metis \ lset-lnth subsetI)
  define k2 where k2 = k1 + n
  interpret vmc-path G ldropn k2 P P $ k2 p σ
  using vmc-path-ldropn infinite-small-l-length (~finite P) by blast
have P $ k2 ∉ V' unfolding k2-def
  using k1 \nth-ldropn infinite-small-l-length[OF (~finite P)] by simp
hence P $ k2 ∈ \ attractor p K using (~finite P) \ \ lset P ⊆ V − W1
    by (metis DiffI U-def V'-def \ lset-nth-member-inf)
then obtain k3 where k3: ldropn k2 P $ k3 ∈ K
  using σ-attracts strategy-attractsE unfolding G'.visits-via-def by blast
  define k4 where k4 = k3 + k2
hence P \ lset (ldropn n P) ∩ K ≠ \{\}
  using (~finite P) \ \ lset-nth-member-inf[of ldropn n P $ k4 − n]
  by blast
thus min-prio ∈ \ lset (ldropn n (lmap ω P)) unfolding K-def by auto
qed
thus ?thesis unfolding winning-path-def
using \text{path-inf-priorities-at-least-min-prio}\ OF\ \text{P-valid,}\ folded\ \text{min-prio-def}\ \\
\langle\text{winning-priority}\ p\ \text{min-prio}\ \langle\text{finite}\ P\rangle\ \\
\text{by}\ \text{blast}\ \\
\text{qed}\ \\
\text{qed}\ \\
\text{hence}\ \forall v\in V,\ \exists p\ \sigma.\ \text{strategy}\ p\ \sigma\ \land\ \text{winning-strategy}\ p\ \sigma\ v\ \\
\text{unfolding}\ W1\text{-def}\ \text{winning-region-def}\ \text{using}\ \langle\text{strategy}\ p\ \sigma\rangle\ \text{by}\ \text{blast}\ \\
\text{hence}\ \exists p\ \sigma.\ \text{strategy}\ p\ \sigma\ \land\ \text{winning-strategy}\ p\ \sigma\ v\ \text{using}\ \langle v\in V\rangle\ \text{by}\ \text{simp}\ \\
\text{thus}\ \text{thesis}\ \text{unfolding}\ \text{winning-region-def}\ \text{using}\ \langle v\in V\rangle\ \text{by}\ \text{blast}\ \\
\text{qed}\ \\

12.2 Positional Determinacy without Deadends

\text{theorem}\ \text{positional-strategy-exists-without-deadends}:\ \\
\text{assumes} v\in V\ \land v, v\in V\ \Rightarrow\ \neg \text{deadend}\ v\ \\
\text{shows} \exists p.\ v\in \text{winning-region}\ p\ \\
\text{using}\ \text{assms}\ ParityGame\text{-axioms}\ \\
\text{by}\ \text{(induct card} (\omega' V)\text{ arbitrary; }G v\ \text{rule: nat-less-induct})\ \\
\quad \text{(rule ParityGame.positional-strategy-induction-step, simp-all)}\ \\

12.3 Positional Determinacy with Deadends

Prove a stronger version of the previous theorem: Allow deadends.

\text{theorem}\ \text{positional-strategy-exists}:\ \\
\text{assumes}\ v0\in V\ \\
\text{shows} \exists p.\ v0\in \text{winning-region}\ p\ \\
\text{proof}\ \\
\quad \{\text{fix } p\ \\
\quad \text{define } A\ \text{where } A = \text{attractor}\ p\ \text{(deadends p**)}\ \\
\quad \text{assume } v0\in \text{attractor}\ p\ \text{(deadends p**)}\ \\
\quad \text{then obtain } \sigma\ \text{where } \sigma: \text{strategy}\ p\ \sigma\ \text{strategy-attracts}\ p\ \sigma\ A\ \text{(deadends p**)}\ \\
\quad \text{using}\ \text{attractor-has-strategy}\ [\text{of deadends p**} p] A\text{-def}\ \text{deadends-in-V}\ \text{by}\ \text{blast}\ \\
\quad \text{have } A\subseteq V\ \text{using}\ A\text{-def}\ \text{using}\ \text{attractor-in-V}\ \text{deadends-in-V}\ \text{by}\ \text{blast}\ \\
\quad \text{hence } A - \text{deadends p**} \subseteq V\ \text{by}\\text{auto}\ \\
\quad \text{have } \text{winning-strategy}\ p\ \sigma\ v0\ \text{proof}\ (\text{unfold}\ \text{winning-strategy-def, intro allI impI})\ \\
\quad \text{fix } P\ \text{assume } v0\text{-in-attractor}\ G P\ v0\ p\ \sigma\ \\
\quad \text{then interpret } v0\text{-in-attractor}\ G P\ v0\ p\ \sigma.\ \\
\quad \text{show } \text{winning-path}\ p\ P\ \\
\quad \quad \text{using}\ \text{visits-deadend}\ [\text{of p**}]\ \sigma(2)\ \text{strategy-attracts-lset}\ v0\text{-in-attractor}\ \\
\quad \quad \text{unfolding}\ A\text{-def}\ \text{by}\ \text{simp}\ \\
\quad \quad \text{qed}\ \\
\quad \text{hence}\ \exists p\ \sigma.\ \text{strategy}\ p\ \sigma\ \land\ \text{winning-strategy}\ p\ \sigma\ v0\ \text{using}\ \sigma\ \text{by}\ \text{blast}\ \\
\quad \}\ \text{note}\ \text{lemma-path-to-deadend = this}\ \\
\quad \text{define}\ A\ \text{where}\ A = \text{attractor}\ p\ \text{(deadends p**)}\ \text{for}\ p\ \\

Remove the attractor sets of the sets of deadends.

\text{define}\ V'\ \text{where}\ V' = V - A\ \text{Even} - A\ Odd\ \\
\text{hence}\ V' \subseteq V\ \text{by}\ \text{blast}\ \\
\text{show}\ \text{thesis}\ \text{proof}\ (\text{cases})
assume \( v0 \in V' \)

define \( G' \) where \( G' = \) subgame \( V' \)

interpret \( G' \): ParityGame \( G' \) unfolding \( G'-\text{def} \) using subgame-ParityGame.

have \( V G' = V' \) unfolding \( G'-\text{def} \) using \( \forall V' \subseteq V, \text{by simp} \)
hence \( v0 \in V G' \) unfolding \( v0 \subseteq V', \text{by simp} \)

moreover have \( V'-\text{no-deadends}: \forall v. v \in V G' \implies \neg G'.\text{deadend} v \) proof

fix \( v \) assume \( v \in V G' \)

moreover have \( V' = V - A \text{ Even} - A \text{ Even}^{**} \) using \( V'-\text{def} \) by simp

ultimately show \( \neg G'.\text{deadend} v \) proof

using subgame-without-deadends \( v \in V G' \) unfolding \( A\text{-def} G'-\text{def} \) by blast

qed

ultimately obtain \( p \sigma \) where \( \sigma: G'.\text{strategy} p \sigma G'.\text{winning-strategy} p \sigma v0 \)

using \( G'.\text{positional-strategy-exists-without-deadends} \)

unfolding \( G'.\text{winning-region-def} \) by blast

have \( V'-\text{no-deadends}: \forall v. v \in V' \implies \neg \text{deadend} v \) proof

fix \( v \) assume \( v \in V' \)

hence \( \neg G'.\text{deadend} v \) using \( V'-\text{no-deadends} \) unfolding \( G'-\text{def} \) by auto

thus \( \neg \text{deadend} v \) unfolding \( G'-\text{def} \) using \( v \subseteq V \) by auto

qed

obtain \( \sigma^-\text{attr} \)

where \( \sigma^-\text{attr}: \text{strategy} p \sigma^-\text{attr} \sigma \text{attr} \) (A \( p \)) \( \text{deend} p^{**} \)

using attractor-has-strategy[of \( p \text{deend} s \text{in}\ V] unfolded \( A\text{-def} G' \) by blast

define \( \sigma' \) where \( \sigma' = \) override-on \( \sigma \) \( \sigma^-\text{attr} \) (A\( \text{Even} \cup A \text{Odd} \))

have \( \sigma'^{-\text{is-}\sigma^-\text{on}\ V'}: \forall v. v \in V' \implies \sigma' v = \sigma v \)

unfolding \( V'-\text{def} \) \( \sigma'^{-\text{def}} A\text{-def} \) by (cases \( p \)) simp-all

have strategy \( p \sigma' \) proof

have \( \sigma' = \) override-on \( \sigma^-\text{attr} \sigma \) (A\( \text{Univ} - A \text{ Even} - A \text{ Odd} \))

unfolding \( \sigma'^{-\text{def}} \) override-on-\( \text{def} \) by (rule ext) simp

moreover have strategy \( p (\) override-on \( \sigma^-\text{attr} \sigma \ V'))

using valid-strategy-supergame \( \sigma^-\text{attr}(1) \sigma(1) \) \( V'-\text{no-deadends} \) \( \forall G' = V' \)

unfolding \( G'-\text{def} \) by blast

ultimately show \( \text{thesis} \) by (simp add: valid-strategy-only-in-V \( V'-\text{def} \) override-on-\( \text{def} \))

qed

moreover have winning-strategy \( p \sigma' v0 \) proof (rule winning-strategyI, rule ccontr)

fix \( P \) assume \( \text{vmc-path} G P v0 p \sigma' \)

then interpret \( \text{vmc-path} G P v0 p \sigma' \)

interpre \( \text{vmc-path-no-deadend} G P v0 p \sigma' \)

using \( V'-\text{no-deadends} \) \( v0 \in V' \) by unfold-locals

assume contra: \( \neg \) winning-path \( p \) \( P \)

have lset \( P \subseteq V' \) proof (induct rule: \( \text{vmc-path-lset-induction-closed-subset} \))

fix \( v \) assume \( v \in V' \) \( \neg \text{deadend} v \) \( v \in V V \)

hence \( v \in G', V V \) unfolding \( G'-\text{def} \) by (simp add: \( v \subseteq V' \))

moreover have \( \neg G'.\text{deadend} v \) unfolding \( V'-\text{no-deadends} \) \( v \subseteq V' \) \( \forall G' = V' \) by blast

moreover have \( G'.\text{strategy} p \sigma' \)

using \( G'.\text{valid-strategy-only-in-V} \) \( \sigma'^{-\text{def}} \) \( \sigma'^{-\text{is-}\sigma^-\text{on}\ V'} \) \( \sigma(1) \) \( \forall G' = V' \) by auto

ultimately show \( \sigma' v \in V' \cup A p \) using subgame-strategy-stays-in-subgame

unfolding \( G'-\text{def} \) by blast
next

fix v w assume v ∈ V′ ∧ deadend v v ∈ VV p→ v→ w

have w ⊈ A p** proof

assume w ∈ A p**
hence v ∈ A p** unfolding A-def

using ⟨v ∈ VV p**⟩ ⟨v→ w⟩ attractor-set-VVp by blast

thus False using ⟨v ∈ V′⟩ unfolding V′-def by (cases p) auto

qed

thus w ∈ V′ ∪ A p unfolding V′-def using ⟨v→ w⟩ by (cases p) auto

next

show lset P ∩ A p = { } proof (rule ccontr)

assume lset P ∩ A p ≠ { }

have strategy-attracts p (override-on σ′ σ-attr (A p − deadends p**))

( A p )

( deadends p** )

using strategy-attracts-irrelevant-override[OF σ-attr(2) σ-attr(1) (strategy p σ′)]

by blast

moreover have override-on σ′ σ-attr (A p − deadends p**) = σ′

by (rule ext, unfold σ′-def, cases p) (simp-all add: override-on-def)

ultimately have strategy-attracts p σ′ (A p) (deadends p**) by simp

hence lset P ∩ deadends p** ≠ { }

using ⟨lset P ∩ A p ≠ { }⟩, attracted-path[OF deadends-in-V] by simp

thus False using contra visits-deadend[of p**] by simp

qed

qed (insert ⟨v0 ∈ V′⟩)

then interpret vmc-path G′ P v0 p σ′ unfolding G′-def using subgame-path-vmc-path[OF v′ ⊆ V′] by blast

have G′.path-conforms-with-strategy p P σ proof-

have ∀ v. v ∈ lset P → σ′ v = σ v

using σ′-is σ-on V′ ⟨ V′ , = , V′ ⟩ lset-P-V by blast

thus G′.path-conforms-with-strategy p P σ using P-conforms G′.path-conforms-with-strategy-irrelevant-updates by blast

qed

then interpret vmc-path G′ P v0 p σ using conforms-to-another-strategy by blast

have G′.winning-path p P using σ(2)[unfolded G′.winning-strategy-def] vmc-path-axioms by blast

from (∼winning-path p P):

G′.winning-path-supergame[OF this ParityGame-axioms, unfolded G′-def]

subgame-VV-subset[of p** V′]

subgame-ω[of V′]

show False by blast

qed

ultimately show ?thesis unfolding winning-region-def using ⟨v0 ∈ V⟩ by blast

next

assume v0 ∉ V′

then obtain p where v0 ∈ attractor p (deadends p**)

unfolding V′-def A-def using ⟨v0 ∈ V⟩ by blast

thus ?thesis unfolding winning-region-def using lemma-path-to-deadend ⟨v0 ∈ V⟩ by blast

qed
12.4 The Main Theorem: Positional Determinacy

Prove the main theorem: The winning regions of player EVEN and ODD are a partition of the set of nodes $V$.

theorem partition-into-winning-regions:
  shows $V = \text{winning-region Even} \cup \text{winning-region Odd}$
  and $\text{winning-region Even} \cap \text{winning-region Odd} = \{}$
proof
  show $V \subseteq \text{winning-region Even} \cup \text{winning-region Odd}$
  by (rule subsetI) (metis (full-types) Un-iff other-other-player positional-strategy-exists)
next
  show $\text{winning-region Even} \cup \text{winning-region Odd} \subseteq V$
  by (rule subsetI) (meson Un-iff subsetCE winning-region-in-V)
next
  show $\text{winning-region Even} \cap \text{winning-region Odd} = \{}$
  using winning-strategy-only-for-one-player[of Even]
  unfolding winning-region-def by auto
qed

end — context ParityGame

end

13 Defining the Attractor with inductive_set

theory AttractorInductive
imports
  Main
  Attractor
begin

context ParityGame begin

In section 6 we defined attractor manually via lfp. We can also define it with inductive_set. In this section, we do exactly this and prove that the new definition yields the same set as the old definition.

13.1 attractor-inductive

The attractor set of a given set of nodes, defined inductively.

inductive-set attractor-inductive :: Player => 'a set => 'a set
  for p :: Player and W :: 'a set where
  Base [intro!]: $v \in W \Rightarrow v \in \text{attractor-inductive } p W$
  | | VVp: [ $v \in VV p$; $\exists w. v \rightarrow w \land w \in \text{attractor-inductive } p W$ ]
  | | | $\Rightarrow v \in \text{attractor-inductive } p W$
  | | VVpstar: [ $v \in VV p**$; $\neg \text{deadend } v$; $\forall w. v \rightarrow w \Rightarrow w \in \text{attractor-inductive } p W$ ]
  | | | $\Rightarrow v \in \text{attractor-inductive } p W$

We show that the inductive definition and the definition via least fixed point are the same.
lemma attractor-inductive-is-attractor:
assumes $W \subseteq V$
shows attractor-inductive $p \ W = \text{attractor } p \ W$
proof
  show attractor-inductive $p \ W \subseteq \text{attractor } p \ W$
    proof
      fix $v$ assume $v \in \text{attractor-inductive } p \ W$
      thus $v \in \text{attractor } p \ W$ proof (induct rule: attractor-inductive.induct)
        case (Base $v$) thus ?case using attractor-set-base by auto
      next
case (VVp $v$) thus ?case using attractor-set-VVm by auto
next
case (VVpstar $v$) thus ?case using attractor-set-VVmstar by auto
qed
qed
  show \text{attractor } p \ W \subseteq \text{attractor-inductive } p \ W
    proof
    define $P$ where $P(S) \leftrightarrow S \subseteq \text{attractor-inductive } p \ W$ for $S$
    from $\langle W \subseteq V \rangle$ have $P(\text{attractor } p \ W)$ proof (induct rule: attractor-set-induction)
      case (step $S$)
hence $S \subseteq \text{attractor-inductive } p \ W$ using $P$-def by simp
    have $W \cup S \cup \text{directly-attracted } p S \subseteq \text{attractor-inductive } p \ W$
      proof
fix $v$ assume $v \in W \cup S \cup \text{directly-attracted } p S$
      moreover \{ assume $v \in W$ hence $v \in \text{attractor-inductive } p \ W$ by blast \}
      moreover \{ assume $v \in S$ hence $v \in \text{attractor-inductive } p \ W$
        by (meson $\langle S \subseteq \text{attractor-inductive } p \ W \rangle$ rev-subsetD) \}
      moreover \{ assume $v \in \text{v-attracted}; v \in \text{directly-attracted } p S$
        hence $v \in V$ using $\langle S \subseteq V \rangle$ attractor-step-bounded-by-$V$ by blast
        hence $v \in \text{attractor-inductive } p \ W$ proof (cases rule: VV-cases)
        assume $v \in VVp$
        hence $\exists w. v \rightarrow w \land w \in S$ using $v \text{-attracted}$ directly-attracted-def by blast
        hence $\exists w. v \rightarrow w \land w \in \text{attractor-inductive } p \ W$
          using $\langle S \subseteq \text{attractor-inductive } p \ W \rangle$ by blast
        thus ?thesis by (simp add: $\langle v \in VVp \rangle$ attractor-inductive.VVm)
      next
        assume $v \in VVpstar$
        hence $\forall w. v \rightarrow w \land w \notin S$ using $v \text{-attracted}$ directly-attracted-def by blast
        have $\neg \text{deadend } v$ using $v \text{-attracted}$ directly-attracted-def by blast
        show ?thesis proof (rule ccontr)
          assume $v \notin \text{attractor-inductive } p \ W$
          hence $\exists w. v \rightarrow w \land w \notin \text{attractor-inductive } p \ W$
            by (metis attractor-inductive.VVmstar $\langle v \in VVpstar \rangle$ $\langle \neg \text{deadend } v \rangle$)
          hence $\exists w. v \rightarrow w \land w \notin S$ using $\langle S \subseteq \text{attractor-inductive } p \ W \rangle$ by (meson subsetCE)
            thus False using $\neg \text{deadend } v$ by blast
        qed
      qed
    } ultimately show $v \in \text{attractor-inductive } p \ W$ by (meson UnE)
    qed
  thus $P(\langle W \cup S \cup \text{directly-attracted } p S \rangle$ using $P$-def by simp

71
14 Compatibility with the Graph Theory Package

theory Graph-TheoryCompatibility
imports
  ParityGame
  Graph-Theory.Digraph
  Graph-Theory.Digraph-Isomorphism
begin

In this section, we show that our Digraph locale is compatible to the nomulti-digraph locale from the graph theory package from the Archive of Formal Proofs.

For this, we will define two functions converting between the different types and show that with these conversion functions the locales interpret each other. Together, this indicates that our definition of digraph is reasonable.

14.1 To Graph Theory

We can easily convert our graphs into pre-digraph objects.

definition to-pre-digraph :: ('a, 'b) Graph-scheme ⇒ ('a, 'a × 'a) pre-digraph
where
to-pre-digraph G ≡ (|
  |pre-digraph verts = Graph.verts G,
  |pre-digraph arcs = Graph.arcs G,
  |tail = fst,
  |head = snd |)

With this conversion function, our Digraph locale contains the locale nomulti-digraph from the graph theory package.

context Digraph begin
interpretation is-nomulti-digraph: nomulti-digraph to-pre-digraph G proof
  fix e assume *: e ∈ pre-digraph.arcs (to-pre-digraph G)
  show tail (to-pre-digraph G) e ∈ pre-digraphverts (to-pre-digraph G)
  by (metis * edges-are-in-V(1) pre-digraph.ext-inject pre-digraph.surjective prod.collapse to-pre-digraph-def)
  show head (to-pre-digraph G) e ∈ pre-digraphverts (to-pre-digraph G)
  by (metis * edges-are-in-V(2) pre-digraph.ext-inject pre-digraph.surjective prod.collapse to-pre-digraph-def)
qed (simp add: arc-to-ends-def to-pre-digraph-def)
end

14.2 From Graph Theory

We can also convert in the other direction.
**definition** from-pre-digraph :: (a, b) pre-digraph ⇒ a Graph

**where** from-pre-digraph G ≡ |
Graph.verts = pre-digraph.verts G,
Graph.arcs = arcs-ends G
|

**context** nomulti-digraph begin

**interpretation** is-Digraph: Digraph from-pre-digraph G

**proof**

{ 
  fix v w assume (v, w) ∈ E from-pre-digraph G
  then obtain e where e: e ∈ pre-digraph.arcs G tail G e = v head G e = w
  unfolding from-pre-digraph-def by auto
  hence (v, w) ∈ V from-pre-digraph G × V from-pre-digraph G
  unfolding from-pre-digraph-def by auto
}

thus Digraph (from-pre-digraph G) by (simp add: Digraph.intro subrelI)

qed

end

14.3 Isomorphisms

We also show that our conversion functions make sense. That is, we show that they are nearly inverses of each other. Unfortunately, from-pre-digraph irretrievably loses information about the arcs, and only keeps tail/head intact, so the best we can get for this case is that the back-and-forth converted graphs are isomorphic.

**lemma** graph-conversion-bij: G = from-pre-digraph (to-pre-digraph G)

**unfolding** to-pre-digraph-def from-pre-digraph-def arcs-ends-def arc-to-ends-def by auto

**lemma** (in nomulti-digraph) graph-conversion-bij2: digraph-iso G (to-pre-digraph (from-pre-digraph G))

**proof**

define iso

**where** iso = |
  iso-verts = id :: 'a ⇒ 'a,
  iso-arcs = arc-to-ends G,
  iso-head = snd,
  iso-tail = fst
|

have inj-on (iso-verts iso) (pre-digraph.verts G) unfolding iso-def by auto

moreover have inj-on (iso-arcs iso) (pre-digraph.arcs G)

**unfolding** iso-def arc-to-ends-def by (simp add: arc-to-ends-def inj-onI no-multi-arcs)

moreover have ∀ a ∈ pre-digraph.arcs G.
  iso-verts iso (tail G a) = iso-tail iso (iso-arcs iso a)
  ∧ iso-verts iso (head G a) = iso-head iso (iso-arcs iso a)

**unfolding** iso-def by (simp add: arc-to-ends-def)

ultimately have digraph-isomorphism iso

**unfolding** digraph-isomorphism-def using arc-to-ends-def wf-digraph-axioms by blast

moreover have to-pre-digraph (from-pre-digraph G) = app-iso iso G

73
unfolding to-pre-digraph-def from-pre-digraph-def iso-def app-iso-def by (simp-all add: arcs-ends-def)

ultimately show \( \wedge \text{thesis unfolding digraph-iso-def by blast} \)
qed

end
References


