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 [3].

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 length ............................................................. 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
## 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

## 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

## 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

## Winning Strategies

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

## Well-Ordered Strategy

8.1 Strategies on a Path ........................................... 46
8.2 Eventually One Strategy ........................................ 49

## Winning Regions

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

## Uniform Strategies

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

## Attractor Strategies

11.1 Existence ......................................................... 58

## 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 μ-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 $|\omega(V)| < \infty$.

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 “σ-traps”. Using σ-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 \( \text{nat} \Rightarrow \text{a option} \) with the property that the domain is an initial segment of \( \text{nat} \) (and where \( \text{a} \) is the node type). However, it turned out that coinductive lists give simpler proofs.

It is possible to represent a graph as a function \( \text{nat} \Rightarrow \text{nat} \Rightarrow \text{bool} \), 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 \( \text{UNIV} \) but rather a subset of \( \text{UNIV} \), 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 \textbf{interpret} \( \text{vmc-path} \ G \ P \ v_0 \ p \ \sigma \) to say that \( P \) is a valid maximal path in the graph \( G \) starting in \( v_0 \) and conforming to the strategy \( \sigma \) for player \( p \).

2 Auxiliary Lemmas for Coinductive Lists

Some lemmas to allow better reasoning with coinductive lists.

\textbf{theory MoreCoinductiveList}
\textbf{imports}
\hspace{1em}Main
\hspace{1em}Coinductive.Coinductive-List
\textbf{begin}

2.1 \textit{lset}

\textbf{lemma} \( \text{lset-lnth} : \ x \in \text{lset xs} \implies \exists n. \ \text{lnth xs} \ n = x \)
\hspace{1em}by \ (\text{induct rule: list.set-induct, meson lnth-0, meson lnth-Suc-LCons})

\textbf{lemma} \( \text{lset-lnth-member} : \ [ \text{lset xs} \subseteq A ; \ enat \ n < \ellength \ xs ] \implies \text{lnth xs} \ n \in A \)
\hspace{1em}using \ contra-subsetD[\text{of list.x A}] \ \text{in-lset-conv-lnth[of - xs]} \ \text{by blast}

\textbf{lemma} \( \text{lset-lnth-member-inf} : \ [ \neg \text{lfinite xs} ; \ \text{lset xs} \subseteq A ] \implies \text{lnth xs} \ n \in A \)
\hspace{1em}by \ (\text{metis contra-subsetD} \ \text{inf-lset-lnth lset-inf-lset mingle})

\textbf{lemma} \( \text{lset-intersect-lnth} : \ (\text{lset xs} \cap A \neq \emptyset) \implies \exists n. \ \text{enat} \ n < \ellength \ xs \land \text{lnth xs} \ n \in A \)
\hspace{1em}by \ (\text{metis disjoint-iff-not-equal in-lset-conv-lnth})

\textbf{lemma} \( \text{lset-ltake-Suc} \)
\hspace{1em}\textbf{assumes} \ \neg \text{lnull xs lnth xs} \ 0 = x \ \text{lset ltake (enat n) (ltl xs)} \subseteq A
\hspace{1em}\textbf{shows} \ \text{lset (ltake (enat (Suc n)) xs)} \subseteq \text{insert x A}
\hspace{1em}\textbf{proof}
\hspace{2em}\textbf{have} \ \text{lset (ltake (eSuc (enat n)) (LCons x (ltl xs)))} \subseteq \text{insert x A}
\hspace{2em}\text{using \ assms(3) \ by \ auto}
\hspace{2em}\textbf{moreover \ from \ assms(1,2) \ have} \ \text{LCons x (ltl xs)} = xs
\hspace{2em}\text{by \ (metis lnth-0 ltl-simps(2) not-lnull-conv)}

5
ultimately show \( \text{thesis by \ (simp add: eSuc-enat) } \)
\( \text{qed} \)

**lemma** \( \text{finite-lset: finite } xs \Rightarrow \neg \text{null } xs \Rightarrow \text{lset } xs \in \text{lset } xs \)

**proof** (induct rule: \( \text{finite-induct} \))

  case \( \text{LCons } xs \)

  show \( \text{case proof \ (cases) } \)

  assume \( \ast: \neg \text{null } (\text{ltl } xs) \)

  hence \( \text{lset } (\text{ltl } xs) \in \text{lset } (\text{ltl } xs) \) using \( \text{LCons.hyps(3)} \) by blast

  hence \( \text{lset } (\text{ltl } xs) \in \text{lset } xs \) by (simp add: \( \text{in-lset-ldD} \))

  thus \( \text{thesis by \ (metis } \ast \text{LCons.prems lhd-LCons-ldl last-LCons2) } \)

  \( \text{qed (metis LCons.prems lhd-LCons-ldl last-LCons llst.set-set(1)) } \)

  \( \text{qed simp} \)

**lemma** \( \text{lset-subset: \neg(\text{lset } xs \subseteq A) } \Rightarrow \exists n. \text{enat } n < \text{llength } xs \land \text{lnth } xs n \notin A \)

  by (metis \( \text{in-lset-conv-lnth subsetI} \))

2.2 \text{llength}

**lemma** \( \text{enat-Suc-llt: } \)

  assumes \( \text{enat} (\text{Suc } n) < \text{llength } xs \)

  shows \( \text{enat } n < \text{llength } (\text{ltl } xs) \)

**proof** –

  from \( \text{assms have } eSuc (\text{enat } n) < \text{llength } xs \) by (simp add: \( \text{eSuc-enat} \))

  hence \( \text{enat } n < epr e (\text{llength } xs) \) using \( \text{eSuc-le-ileI1} \) by fastforce

  thus \( \text{thesis by \ (simp add: epr e-llength) } \)

  \( \text{qed} \)

**lemma** \( \text{enat-Suc-llt-Suc: } \)

  \( \text{enat } n < \text{llength } (\text{ltl } xs) \Rightarrow \text{enat} (\text{Suc } n) < \text{llength } xs \)

  by (metis eSuc-enat ldr op-llt leD leI lnul l-ldr op)

**lemma** \( \text{infinite-small-llength} \ [\text{intr}]: \neg \text{finite } xs \Rightarrow \text{enat } n < \text{llength } xs \)

  using \( \text{enat-less \ finite-conv-llength-enat neq-iff by blast} \)

**lemma** \( \text{lnul l-0-llength: } \neg \text{lnul l } xs \Rightarrow \text{enat } 0 < \text{llength } xs \)

  using \( \text{zero-enat-def by auto} \)

**lemma** \( \text{Suc-llength: } \text{enat} (\text{Suc } n) < \text{llength } xs \Rightarrow \text{enat } n < \text{llength } xs \)

  using \( \text{dual-order.strict-trans \ enat-ord-simps(2) by blast} \)

2.3 \text{ltake}

**lemma** \( \text{ltake-lnth: } \text{ltake } n xs = \text{ltake } n ys \Rightarrow \text{enat } m < n \Rightarrow \text{lnth } xs m = \text{lnth } ys m \)

  by (metis \( \text{lnth-ltake} \))

**lemma** \( \text{lset-ltake-prefix} \ [\text{simp}]: n \leq m \Rightarrow \text{lset } (\text{ltake } n xs) \subseteq \text{lset } (\text{ltake } m xs) \)

  by \( \text{(simp add: lprefix-lsetD)} \)

**lemma** \( \text{lset-ltake: } (\forall m. m < n \Rightarrow \text{lnth } xs m \in A) \Rightarrow \text{lset } (\text{ltake } (\text{enat } n) xs) \subseteq A \)

**proof** (induct \( n \) arbitrary: \( xs \))

  case \( 0 \)

  have \( \text{ltake } (\text{enat } 0) xs = \text{LNil} \) by \( \text{(simp add: zero-enat-def)} \)
thus \textit{case} by simp

next

case (Suc $n$)

show \textit{case proof} (cases)

assume $xs \neq \text{LNil}$

then obtain $xs' \quad \text{where} \quad xs = \text{LCons} \; x \; xs'$ by (meson \textit{neg-LNil-conv})

\{ fix $m$ assume $m < n$

hence Suc $m < \text{Suc} \; n$ by simp

hence lnth $xs$ (Suc $m$) $\in A$ using Suc.prems by presburger

hence lnth $xs'$ $m$ $\in A$ using $xs$ by simp \}

hence lset (ltake (enat $n$) $xs$) $\subseteq A$ using Suc.hyps by blast

moreover have ltake (enat (Suc $n$)) $xs$ = LCons $x$ (ltake (enat $n$) $xs'$) by (metis \textit{no-types} eSuc-enat)

moreover have $x \in A$ using ltake-eSuc-LCons $\of x \; xs'$ by blast

ultimately show \textit{thesis} by simp

qed simp

qed

lemma llength-ltake': \textit{enat} $n < \text{llength} \; xs \implies \text{llength} \; (\text{ltake} \; (\text{enat} \; n) \; xs) = \text{enat} \; n$

by (metis llength-ltake min.strict-order-i)

lemma llast-ltake:

assumes \textit{enat} (Suc $n$) $< \text{llength} \; xs$

shows llast (ltake (enat (Suc $n$)) $xs$) = lnth $xs$ $n$ \textit{(is llast ?A = -)}

unfolding llast-def by (induct $n$ simp-all)

lemma lset-ltake-ltl:

\text{llset} (ltake (enat $n$) \; (ltl \; xs)) $\subseteq$ lset (ltake (enat (Suc $n$)) \; $xs$)

proof (cases)

assume \text{lnull} $xs$

then obtain $v0$ \text{where} $xs = \text{LCons} \; v0 \; (\text{ltl} \; xs)$ 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 \textit{thesis} using lset-LCons $\of v0 \; \text{ltake} \; (\text{enat} \; n) \; (\text{ltl} \; xs)$ by blast

qed (simp add: lnull-def)

2.4 ldropn

lemma ltl-ldrop: $\forall xs. \; P \; xs \implies P \; (\text{ltl} \; xs); \; P \; xs \implies P \; (\text{ldropn} \; n \; xs)$

unfolding ldropn-def by (induct $n$ simp-all)

2.5 lfinite

lemma lfinite-drop-set: lfinite $xs \implies \exists n. \; v \notin \text{lset} \; (\text{ldrop} \; n \; xs)$

by (metis ldrop-inf lmember-excl lset-lmember)

lemma index-infinite-set:

$\exists n. \; \text{lnth} \; x \; m = y; \; \forall i. \; \text{lnth} \; x \; i = y \implies (\exists m > i. \; \text{lnth} \; x \; m = y)$ $\implies y \in \text{lset} \; (\text{ldrop} \; n \; x)$

proof (induct $n$ arbitrary: $x \; m$)
case 0 thus \( \text{case using \( \text{set-nth-member-inf} \) by auto} \)
next
  case \((\text{Suc } n)\)
  obtain \(a \, xs\) where \(x = \text{LCons } a \, xs\) by (meson Suc.prems(1) lnul-limp-\text{finite} not-lnul-conv)
  obtain \(j\) where \(j > m \, \text{lnth } x\) \(j = y\) using Suc.prems(2,3) by blast
  have \(\text{lnth}\, xs\) \((j - 1) = y\) by (metis lnth-LCons' j(1,2) not-less0 \(x\))
  moreover {
    fix \(i\) assume \(\text{lnth}\, xs\) \(i = y\)
    hence \(\text{lnth}\, x\) \((\text{Suc } i) = y\) by (simp add: \(x\))
    hence \(\exists j > i. \, \text{lnth}\, xs\) \(j = y\) by (metis Suc.prems(3) Suc-lessE lnth-Suc-LCons \(x\))
  }
  ultimately show \(\text{case using Su.chyps Suc.prems(1) } x\) by auto
qed

2.6 \text{lmap}

\text{lemma} lnth-lmap-idropn:
\begin{align*}
enat\ n < \text{l\length}\ xs \implies \text{lnth} (\text{lmap } f (\text{idropn } n\ xs))\ 0\ &= \text{lnth} (\text{lmap } f\ xs)\ 0\ 
\text{by} \ (\text{simp add: lhd-idropn lnth-0-conv-lhd})
\end{align*}

\text{lemma} lnth-lmap-idropn-Suc:
\begin{align*}
enat\ (\text{Suc } n) < \text{l\length}\ xs \implies \text{lnth} (\text{lmap } f (\text{idropn } n\ xs))\ (\text{Suc } 0)\ &= \text{lnth} (\text{lmap } f\ xs)\ (\text{Suc } n)\ 
\text{by} \ (\text{metis \(\text{no-types, lifting}\) Suc-length idropn-\text{ll} leD list.map-disc-iff lnth-lmap-idropn lnth-\text{ll} lnul-idropn \text{ll-idropn \text{ll}-lmap})
\end{align*}

2.7 \text{Notation}

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

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

```
record 'a Graph =
  verts :: 'a set (V)
  arcs :: 'a Edge set (E)
```

```
abbreviation is-arc :: ('a, 'b) Graph-scheme ⇒ 'a ⇒ 'b ⇒ bool (infix1 → b0) where
  _→_ ⇒ (v,w) ∈ E G
```

```
locale Digraph =
  fixes G (structure)
  assumes valid-edge-set: E ⊆ V × V
begin
```

A node without successors is a deadend.

```
abbreviation deadend :: 'a ⇒ bool where deadend v ≡ ¬(∃ w ∈ V. v → 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.

```
coinductive valid-path :: 'a Path ⇒ bool where
  valid-path-base: valid-path LNil
  | valid-path-cons: v ∈ V ⇒ valid-path (LCons v LNil)
  | valid-path-cons: [ v ∈ V; w ∈ V; v→w; valid-path Ps; ¬lnull Ps; lhd Ps = w ] ⇒ valid-path (LCons v Ps)
```

```
inductive-simps valid-path-cons-simp: valid-path (LCons x xs)
```

```
lemma valid-path-ltl': valid-path (LCons v Ps) ⇒ valid-path Ps
  using valid-path-simps by blast
```

```
lemma valid-path-ltl: valid-path P ⇒ valid-path (ltl P)
  by (metis list.exhaust-set ltl-simps(1) valid-path-ltl')
```

```
lemma valid-path-drop: valid-path P ⇒ valid-path (ldropn n P)
  by (simp add: valid-path-ltl ltl-ldrop)
```

```
lemma valid-path-in-V: assumes valid-path P shows lset P ⊆ V
  proof
    fix x assume x ∈ lset P thus x ∈ V
    using assms by (induct rule: list.set-induct) (auto intro: valid-path.cases)
  qed
```

```
lemma valid-path-finite-in-V: [ valid-path P; enat n < llength P ] ⇒ P $ n ∈ V
  using valid-path-in-V lset-lth-member by blast
```

```
lemma valid-path-edges': valid-path (LCons v (LCons w Ps)) ⇒ v→w
  using valid-path.cases by fastforce
```

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

have $\text{enat} n < \text{lenght} P$ using $\text{assms}(2) \quad \text{enat-ord-simps}(2) \quad \text{less-trans} \quad \text{by blast}$

hence $P' \not\equiv \emptyset = P \not\equiv n$ by (simp add: $P'$-def)

moreover have $P' \not\equiv \text{Suc} 0 = P \not\equiv \text{Suc} n$

by (metis $\text{One-nat-def}$ $P'$-def $\text{Suc-eq-plus1}$ $\text{add.commute}$ $\text{assms}(2) \quad \text{ldrop}_n=\text{Suc-conv}_n=\text{ldrop}_n$)

ultimately have $\exists Ps. \ P' = \text{LCons} (P \not\equiv n) (\text{LCons} (P \not\equiv \text{Suc} n) Ps)$

by (metis $\text{P'-def}$ $\text{enat} n < \text{lenght} P$ $\quad \text{assms}(2) \quad \text{ldrop}_n=\text{Suc-conv}_n=\text{ldrop}_n$)

moreover have $\text{valid-path} P'$ by (simp add: $P'$-def $\text{assms}(1) \quad \text{valid-path-drop}$)

ultimately show $?\text{thesis using} \quad \text{valid-path-edges'} \quad \text{by blast}$

qed

lemma $\text{valid-path-coinduct}$ [consumes 1, case-names base step, coinduct pred: $\text{valid-path}$]:

assumes major: $Q \ P$

and base: $\forall v \ P. \ Q (\text{LCons} v \text{LNil}) \Longrightarrow v \in V$

and step: $\forall v \ P.\ Q (\text{LCons} v (\text{LCons} i w P)) \Longrightarrow v \rightarrow w \wedge (Q (\text{LCons} w P) \lor \text{valid-path} (\text{LCons} w P))$

shows $\text{valid-path} P$

using major proof (coinduction arbitrary: $P$


case valid-path

\{ assume $P \not= \text{LNil} \not\equiv (\exists v. \ P = \text{LCons} v \text{LNil} \wedge v \in V)$

then obtain $v w P'$ where $P = \text{LCons} v (\text{LCons} i w P')$

using $\text{neg-LNil-conv base valid-path by metis}$

hence ?case using step valid-path by auto
\}

thus ?case by blast

qed

lemma $\text{valid-path-no-deadends}$:

$\equiv \text{valid-path} P; \text{enat} (\text{Suc} i) < \text{lenght} P \Longrightarrow \text{deadend} (P \not\equiv i)$

using $\text{valid-path-edges by blast}$

lemma $\text{valid-path-ends-on-deadend}$:

$\equiv \text{valid-path} P; \text{enat} i < \text{lenght} P; \text{deadend} (P \not\equiv i) \Longrightarrow \text{enat} (\text{Suc} i) = \text{lenght} P$

using $\text{valid-path-no-deadends by (metis \text{enat-iless} \text{enat-ord-simps}(2) \quad \text{neq-iff not-less-eq})}$

lemma $\text{valid-path-prefix}$:

$\equiv \text{valid-path} P; \text{lprefix} P' P \Longrightarrow \text{valid-path} P'$

proof (coinduction arbitrary: $P'$)

\begin{itemize}
  \item case (step $v w P'' P' P'$)
  \end{itemize}

then obtain $Ps$ where $Ps: \text{LCons} v (\text{LCons} w Ps) \equiv P$ by (metis $\text{LCons-lprefix-conv}$)

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 ?case using Ps step(2) valid-path-edges' by blast

qed (metis $\text{LCons-lprefix-conv valid-path-edges simp}$)

lemma $\text{valid-path-lappend}$:

assumes valid-path $P$ valid-path $P' [\not\equiv \text{null} P; \not\equiv \text{null} P'] \Longrightarrow \text{last} P \rightarrow \text{lhd} P'$

shows valid-path (lappend $P' P'$)

proof (cases, cases)

\begin{itemize}
  \item assume $\not\equiv \text{null} P \not\equiv \text{null} P'$
  \end{itemize}

thus ?thesis using $\text{assms proof}$ (coinduction arbitrary: $P'$)

\begin{itemize}
  \item case (step $v w P'' P' P'$)
  \end{itemize}

show ?case proof (cases)
\begin{verbatim}
assume lnull (ltl P)
thus \texttt{?case using step\((1,2,3,5,6)\)}
  by (metis lhd-LCons lhd-LCons-ltl lhd-lapp end llast-singleton
          llist.collapse(1) ltl-lappend ltl-simps(2))

next
assume \texttt{~lnull (ltl P)}
moreover have ltl (lappend P P') = lappend (ltl P) P' using step \((2)\) by simp
ultimately show \texttt{?case using step}
  by (metis (no-types, lifting)
          lhd-LCons lhd-LCons-ltl lhd-lapp end llast-LCons ltl-simps(2)
          valid-path-edges' valid-path-llt)

qed

qed (metis llist.disc(1) lnull-lappend ltl-lappend ltl-simps(2))

3.4 Maximal Paths

We say that a path is maximal if it is empty or if it ends in a deadend.

**coinductive** \texttt{maximal-path} where

\begin{verbatim}
| \texttt{maximal-path-base: maximal-path LNil}
| \texttt{maximal-path-base': deadend v \implies maximal-path (LCons v LNil)}
| \texttt{maximal-path-cons: \texttt{~lnull} Ps \implies maximal-path Ps \implies maximal-path (LCons v Ps)}
\end{verbatim}

**lemma** \texttt{maximal-no-deadend: maximal-path (LCons v Ps)} \implies \texttt{~deadend} v \implies \texttt{~lnull} Ps

by (metis lhd-LCons llist.distinct(1) ltl-simps(2) maximal-path.simps)

**lemma** \texttt{maximal-ltl: maximal-path P \implies maximal-path (llt P)}

by (metis ltl-simps(1) ltl-simps(2) maximal-path.simps)

**lemma** \texttt{maximal-drop: maximal-path P \implies maximal-path (ldropn n P)}

by (simp add: maximal-ltl ltl-ldrop)

**lemma** \texttt{maximal-path-lappend:}

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

A valid path is still valid in a supergame.

**lemma** \texttt{valid-path-supergame:}

\begin{verbatim}
assumes \texttt{valid-path} P and \texttt{G'}: Digraph \texttt{G'} V \subseteq V \texttt{G}, E \subseteq \texttt{E} \texttt{G'}
shows \texttt{Digraph.valid-path \texttt{G'}} P
using \texttt{valid-path P} proof (coinduction arbitrary: P
  rule: Digraph.valid-path-coinduct[OF \texttt{G'}(1), case-names base step])
  case \texttt{base thus \texttt{?case using \texttt{G'}} valid-path-cons-simp by auto
  qed (meson \texttt{G'}(3) subset-eq valid-path-edges' valid-path-llt'}
\end{verbatim}
\end{verbatim}
hence $\exists P \geq P', P = \text{lappend } Ps I P' \land \neg \text{null } P' \land \text{maximal-path } P'$

using $\text{maximal-path}(2)$ $\text{maximal-path}(3)$ by auto

thus $\neg \text{thesis using } P \text{lappend-null by fastforce}$

qed blast

qed

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

lemma $\text{maximal-ends-on-deadend'}$:

assumes $\text{maximal-path } P \neg \text{finite } P \neg \text{null } P$

shows $\text{deadend } (\text{llast } P)$

proof

from ($\neg \text{finite } P$) ($\neg \text{null } P$) obtain $n$ where $n$: $\text{length } P = \text{enat } (\text{Suc } n)$

by (metis $\text{enat-ord}-\text{simp}(2)$ gr0-implies-Suc $\text{finite-length-enat null-0-length}$)

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

hence $\text{maximal-path } P' \text{ using } \text{assms } (1)$ maximal-drop by blast

thus $\neg \text{thesis proof } (\text{cases rule: maximal-path.cases})$

\begin{itemize}
  \item case ($\text{maximal-path-base'} v$)
  \begin{itemize}
    \item hence $\text{deadend } (\text{llast } P' \text{ unfolding } P'\text{-def by simp}$
    \item thus $\neg \text{thesis unfolding } P'\text{-def using } \text{llast-ldropn}[of $P$] \text{ n}$
    \begin{itemize}
      \item by (metis $\text{P}\text{-def ldropn-eq-LConsD local.maximal-path-base'}(1)$)
    \end{itemize}
  \end{itemize}
\end{itemize}

next

\begin{itemize}
  \item case ($\text{maximal-path-cons } P'\text{-def } v$)
  \begin{itemize}
    \item hence $\text{ldropn } (\text{Suc } n) \text{ P } = P'' \text{ unfolding } P'D'\text{-def by (metis ldrop-eSuc-lNil ldropn ld-simp}(2)$
    \item thus $\neg \text{thesis using } n \text{ maximal-path-cons } (2) \text{ by auto}$
  \end{itemize}
\end{itemize}

qed (simp add: $P'\text{-def } n \text{ ldropn-eq-LNil}$)

qed

lemma $\text{maximal-ends-on-deadend'}$: [$\text{lfinite } P; \text{deadend } (\text{llast } P)$] $\Rightarrow \text{maximal-path } P$

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

\begin{itemize}
  \item case ($\text{maximal-path } P$)
  \begin{itemize}
    \item show $\neg \text{case proof } (\text{cases})$
    \begin{itemize}
      \item assume $P' \neq \text{LNil}$
      \begin{itemize}
        \item then obtain $v \text{ P'}$ where $P' = \text{LCons } v P'$ by (meson neq-LNil-conv)
        \item show $\neg \text{thesis proof } (\text{cases})$
        \begin{itemize}
          \item assume $P' = \text{LNil}$ thus $\neg \text{thesis using } P' \text{ maximal-path } (2) \text{ by auto}$
          \end{itemize}
        \end{itemize}
    \end{itemize}
  \end{itemize}
\end{itemize}

qed simp

lemma $\text{infinite-path-is-maximal'}$: [$\text{valid-path } P;  \neg \text{infinite } P$] $\Rightarrow \text{maximal-path } P$

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 $\text{Player} = \text{Even} \mid \text{Odd}$

abbreviation $\text{other-player } p \equiv (\text{if } p = \text{Even then Odd else Even})$
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 $V \rightarrow \mathbb{N}$ with finite image.

```
notation other-player (\(-\)) [1000] 1000

lemma other-other-player | simp: p **** = p using Player.exhaust by auto
```

\[\begin{align*}
\text{notation other-player (\(-\)) [1000] 1000} \\
\text{lemma other-other-player | simp: p **** = p using Player.exhaust by auto}
\end{align*}\]

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 $V \rightarrow \mathbb{N}$ with finite image.

```

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 $V \rightarrow \mathbb{N}$ with finite image.

record 'a ParityGame = 'a Graph +
  player0 :: 'a set (V0)
  priority :: 'a ⇒ nat (ω)

locale ParityGame = Digraph G for G :: ("a, "b) ParityGame-scheme (structure) +
  assumes valid-player0-set: V0 ⊆ V
  and priorities-finite: finite (ω \cdot V)

begin

VV p is the set of nodes belonging to player p.

abbreviation VV :: Player ⇒ 'a set where VV p ≡ (if p = Even then V0 else V - V0)

lemma VV-to-V | intro: v ∈ VV p ⇒ v ∈ V using valid-player0-set by (cases p)
lemma VV-simpl1: v ∈ VV p ⇒ v ∉ VV p*** by auto
lemma VV-simpl2: v ∈ VV p*** ⇒ v ∉ VV p by auto
lemma VV-equivalence | iff: v ∈ V ⇒ v ∉ VV p ↔ v ∈ VV p*** by auto
lemma VV-cases [consumes 1]: [ v ∈ V ; v ∈ VV p ⇒ P ; v ∈ VV p*** ⇒ P ] ⇒ P by auto

3.6 Sets of Deadends

definition deadends p ≡ \{ v ∈ VV p. deadlock v \}

lemma deadends-in-V: deadends p ⊆ V unfolding deadends-def by blast

3.7 Subgames

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

definition subgame where
  subgame V' ≡ G[
    verts := V ∩ V',
    arcs := E ∩ (V' × V'),
    player0 := V0 ∩ V'
  ]

lemma subgame-V | simp: V subgame V' ⊆ V
  and subgame-E | simp: E subgame V' ⊆ E
  and subgame-ω | simp: ω subgame V' = ω

unfolding subgame-def by simp-all

lemma
  assumes V' ⊆ V
  shows subgame-V' | simp: V subgame V' = V'
    and subgame-E' | simp: E subgame V' = E ∩ (V subgame V' × V subgame V')

unfolding subgame-def using assms by auto

lemma subgame-VV | simp: ParityGame.VV (subgame V') p = V' ∩ VV p proof--
  have ParityGame.VV (subgame V') Even = V' ∩ VV Even unfolding subgame-def by auto
  moreover have ParityGame.VV (subgame V') Odd = V' ∩ VV Odd proof--

13
have \( V' \cap V - (V \cap V') = V' \cap V \cap (V - V) \) by blast
thus \( ?\text{thesis unfolding} \) \( \text{subgame-def by auto} \)
qed

ultimately show \( ?\text{thesis by simp} \)
qed
corollary \( \text{subgame-VV-subset }[\text{simp}]: \text{ParityGame}.VV \ (\text{subgame } V') \subseteq VV \) \( p \subseteq VV \) \( p \) by simp

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

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

lemma \( \text{subgame-Digraph: Digraph}(\text{subgame } V') \)
by (unfold-locales) (auto simp add: subgame-def)

lemma \( \text{subgame-ParityGame:} \)
shows \( \text{ParityGame}(\text{subgame } V') \)
proof (unfold-locales)
- show \( E_{\text{subgame } V'} \subseteq V_{\text{subgame } V'} \times V_{\text{subgame } V'} \)
  using \( \text{subgame-Digraph[unfolded Digraph-def]} \).
- show \( V0_{\text{subgame } V'} \subseteq V_{\text{subgame } V'} \) unfolding \( \text{subgame-def using valid-player0-set by auto} \)
- show \( \text{finite} \ (\omega_{\text{subgame } V'} \ V_{\text{subgame } V'}) \) by simp
qed

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

lemma \( \text{subgame-maximal-path:} \)
assumes \( V': V' \subseteq V \) \( \text{and } P: \text{maximal-path} \) \( P \subseteq V' \)

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.cohind[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 ¬ lnite 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-length)
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-length)

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 \( a \in \text{lset } (\text{Init } (\text{Init } (\text{lmap } \omega \cdot (\text{LCons } v \cdot P)))) \) using a unfolding path-inf-priorities-def by blast
  hence \( a \in \omega \cdot \text{lset } P \) by simp
  thus ?thesis using P valid-path-V priorities-finite Min-le by blast
qed

lemma path-inf-priorities-LCons:
  path-inf-priorities \( P = \text{path-inf-priorities } (\text{LCons } v \cdot P) \) (is \(?A = ?B\) )

proof
  show \(?A \subseteq ?B\) proof
    fix \( a \) assume \( a \in ?A \)
    hence \( \forall n. a \in \text{lset } (\text{Init } (\text{Init } (\text{lmap } \omega \cdot (\text{LCons } v \cdot P)))) \) unfolding path-inf-priorities-def
    using in-lset-llD[of \( a \)] by (simp add: llt-lldrop)
    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 \( \forall n. a \in \text{lset } (\text{Init } (\text{Init } (\text{lmap } \omega \cdot (\text{LCons } v \cdot P)))) \) 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 \( P = \text{path-inf-priorities } (\text{ltl } P) \)
  by (metis llist.exhaust llt-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 EVEN (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 \( p \equiv \) (if \( p = \text{Even} \) then even else odd)

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

Every path has a unique winner.

lemma paths-are-winning-for-one-player:
  assumes valid-path \( P \)
  shows winning-path \( p \) \( P \equiv \neg \text{winning-path } p ** P \)

proof (cases)
  assume \( \neg \text{Null } P \)
show ?thesis proof (cases)
   assume \( \text{finite } P \)
   thus ?thesis
     using assms finite-set valid-path-in-V
   unfolding winning-path-def
   by auto
next
   assume \( \neg \text{finite } P \)
   then obtain \( a \) where \( a \in \text{path-inf-priorities } P \land \forall b. b < a \Rightarrow b \notin \text{path-inf-priorities } P \)
   unfolding winning-path-def by auto
   hence \( \forall q. \text{winning-priority } q \leftrightarrow \text{winning-path } q P \)
   unfolding winning-path-def using \( \neg \text{null } P \) \( \neg \text{finite } P \) by (metis le-antisym not-le)
   moreover have \( \forall q. \text{winning-priority } p q \leftrightarrow \neg \text{winning-priority } p** q \) by simp
   ultimately show ?thesis by blast
qed

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

corollary winning-path-drop:
   assumes \( \text{winning-path } p \ P \text{enat } n < \text{lenght } P \)
   shows \( \text{winning-path } p \ (\ldropn n P) \)
   using assms proof (induct n)
     case \( \text{Suc } n \)
     hence \( \text{winning-path } p \ (\ldropn n P) \) using dual-order.strict-trans evat-ord-simps(2) by blast
     moreover have \( \ltl (\ldropn n P) = \ldropn (\text{Suc } n) P \) by (simp add: ldrop-eSuc-llt ltl-ldropn)
     moreover hence \( \neg \text{null } (\ldropn n P) \) using Suc.prems(2) by (metis leD lnul-lldropn lnul-lltl)
     ultimately show ?case using winning-path-ltl[of \( \text{ldropn n P} \) \( \text{Suc.prems}(2) \) by auto
qed simp

corollary winning-path-drop-add:
   assumes \( \text{valid-path } P \text{winning-path } p \ (\ldropn n P) \text{enat } n < \text{lenght } P \)
   shows \( \text{winning-path } p \ P \)
   using assms paths-are-winning-for-one-player valid-path-drop winning-path-drop by blast

lemma winning-path-LCons:
   assumes \( P: \text{winning-path } p \ P \neg \text{null } P \)
   shows \( \text{winning-path } p \ (LCons v P) \)
proof (cases)
   assume \( \text{finite } P \)
moreover have \( \text{llast } P = \text{llast } (L\text{Cons } v P) \)
using \( P(2) \) by (metis \( \text{last-LCons2 not-null-conv} \))
ultimately show \( \text{thesis using } P \) unfolding \( \text{winning-path-def} \) by simp
next
assume \( \neg \text{infinite } P \)
thus \( \text{thesis using } P \) path-inf-priorities-LCons unfolding \( \text{winning-path-def} \) by simp
qed

lemma winning-path-supergame:
assumes \( \text{winning-path } P \)
and \( G': \text{ParityGame } G' \quad VV p** \subseteq \text{ParityGame.}VV G' \quad p** \omega = \omega G' \)
shows \( \text{ParityGame.} \text{winning-path } G' \) \( \text{p } P \)
proof
interpret \( G': \text{ParityGame } G' \) using \( G'(1) \).
have \( \{ \text{finite } P; \neg \text{inull } P \} \Rightarrow \text{llast } P \in G';VV p** \text{ and } \text{inull } P \Rightarrow p = \text{Even} \)
using assms(1) unfolding \( \text{winning-path-def} \) using \( G'(2) \) by auto
thus \( \text{thesis unfolding } G': \text{winning-path-def} \)
using lnul-lfinite assms(1)
unfolding \( \text{winning-path-def} \) path-inf-priorities-def \( G'.p \_	ext{ath-inf-priorities-def} G'(3) \)
by blast
qed

end — locale ParityGame

3.10 Valid Maximal Paths

Define a locale for valid maximal paths, because we need them often.

locale vm-path = ParityGame +
fixes \( v0 \)
assumes \( P\text{-not-null} \quad |\text{simp}| : \neg \text{inull } P \)
and \( P\text{-valid} \quad |\text{simp}| : \text{valid-path } P \)
and \( P\text{-maximal} \quad |\text{simp}| : \text{maximal-path } P \)
and \( P\text{-v0} \quad |\text{simp}| : \text{lhd } P = v0 \)
begin
lemma \( P\text{-LCons} : P = L\text{Cons } v0 \ldots (ltl P) \) using lhd-LCons-ltl[OF \( P\text{-not-null} \)] by simp

lemma \( P\text{-len} \quad |\text{simp}| : \text{enat } 0 < \text{llength } P \) by (simp add: lnnull-0-length)
lemma \( P\text{-0} \quad |\text{simp}| : P \$ 0 = v0 \) by (simp add: lnth-0-lhd)
lemma \( P\text{-lth-Suc} : P \$ \text{Suc } n = \text{ltl } P \$ n \) by (simp add: lnth-lhd)
lemma \( P\text{-no-deadends} : \text{enat } (\text{Suc } n) < \text{llength } P \Rightarrow \neg \text{deadend } (P \$ n) \)
using valid-path-no-deadends by simp
lemma \( P\text{-no-deadend-v0} : \neg \text{inull } (\text{ltl } P) \Rightarrow \neg \text{deadend } v0 \)
by (metis \( P\text{-LCons} \) \( P\text{-valid edges-are-in-V (2) not-null-conv valid-path-edges} \))
lemma \( P\text{-no-deadend-v0-length} : \text{enat } (\text{Suc } n) < \text{llength } P \Rightarrow \neg \text{deadend } v0 \)
by (metis \( P\text{-0} \) \( P\text{-len} \) valid-enat-cons-simpss(2) not-less-eq valid-path-ends-on-deadend zero-less-Suc)
lemma \( P\text{-ends-on-deadend} \quad |\text{simp}| : \text{enat } n < \text{llength } P ; \text{deadend } (P \$ n) \) \( \Rightarrow \text{enat } (\text{Suc } n) = \text{llength } P \)
using P-valid valid-path-ends-on-deadend by blast

lemma \( P\text{-lnull-ltl-deadend-v0} : \text{lnull } (\text{ltl } P) \Rightarrow \text{deadend } v0 \)
using \( P\text{-LCons} \) maximal-no-deadend by force
lemma \( P\text{-lnull-ltl-LCons} : \text{lnull } (\text{ltl } P) \Rightarrow P = L\text{Cons } v0 \) LNil
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 asms 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-cong-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
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 \( p \) \( σ \) ≡ \( ∀ v ∈ VV_p. \neg \text{deadend } v \Rightarrow v \mapsto σ v \)

lemma (in ParityGame) strategyI [intro]:
  \( (\forall v. [ v ∈ VV_p; \neg \text{deadend } v ] \Rightarrow v \mapsto σ v ) \Rightarrow strategy \ p \ σ \)

unfolding strategy-def by blast

4.2 Strategy-Conforming Paths

If \( \text{path-conforms-with-strategy } p \ P \ σ \) 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 \( p \) \( \text{LNil } σ \)
  | path-conforms-LCons-LNil: path-conforms-with-strategy \( p \) \( \text{LCons } v \ct \text{LNil } σ \)
  | path-conforms-VVp: \[ v ∈ VV_p; w = σ v \Rightarrow v \mapsto σ v \] ⇒ path-conforms-with-strategy \( p \) \( \text{LCons } w \ct Ps σ \)
  \( \Rightarrow \) path-conforms-with-strategy \( p \) \( \text{LCons } v \ct (\text{LCons } w \ct Ps ) σ \)
  | path-conforms-VVpstar: \[ v / ∈ VV_p; \] path-conforms-with-strategy \( p \) \( Ps σ \)
  \( \Rightarrow \) path-conforms-with-strategy \( p \) \( \text{LCons } v \ct Ps σ \)

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 \( \text{P-conforms } [\text{simp}]: \text{path-conforms-with-strategy } p \ P \ σ \)

Similarly, 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 σ-arbitrary ≡ λv. SOME w. v → w

lemma valid-arbitrary-strategy [simp]: strategy p σ-arbitrary proof
  fix v assume ¬deadend v
  thus v → σ-arbitrary v unfolding σ-arbitrary-def using some-ex[of λw. v → w] by blast
qed

4.4 Valid Strategies

lemma valid-strategy-updates: [ strategy p σ; v0→w0 ] ⇒ strategy p (σ(v0 := w0))
  unfolding strategy-def by auto

lemma valid-strategy-updates-set:
  assumes strategy p σ \( \wedge \) v. \( v \in A; v \in VV p; ¬deadend v \) ⇒ v→σ' v
  shows strategy p (override-on σ σ' A)
  unfolding strategy-def by (metis assms override-on-def strategy-def)

lemma valid-strategy-updates-set-strong:
  assumes strategy p σ strategy p σ'
  shows strategy p (override-on σ σ' A)
  using assms(1) assms(2)[unfolded strategy-def] valid-strategy-updates-set by simp

lemma subgame-strategy-stays-in-subgame:
  assumes σ: ParityGame.strat (subgame V') p σ
  and v ∈ ParityGame.VV (subgame V') p ¬Digraph.deadend (subgame V') v
  shows σ v ∈ V'
  proof
    interpret G': ParityGame subgame V' using subgame-ParityGame .
    have σ v ∈ V (subgame V') using assms unfolding G'.strategy-def G'.edges-are-in-V(2) by blast
    thus σ v ∈ V' by (metis Diff-Iff IntE subgame-VV Player.distinct(2))
  qed

lemma valid-strategy-supergame:
  assumes σ: strategy p σ
  and σ': ParityGame.strat (subgame V') p σ'
  and G'-no-deadends: \( \wedge v. v \in V' \Rightarrow ¬Digraph.deadend (subgame V') v \)
  shows strategy p (override-on σ σ' V') (is strategy p ?σ)
  proof
    interpret G': ParityGame subgame V' using subgame-ParityGame .
    fix v assume v: v ∈ VV p ¬deadend v
    show \( \exists \sigma v \) proof (cases)
      assume v ∈ V'
      hence v ∈ G'.VV p using subgame-VV (v ∈ VV p) by blast
      moreover have ¬G'.deadend v using G'-no-deadends (v ∈ V') by blast
      ultimately have v → subgame V' σ' v using σ' unfolding G'.strategy-def by blast
      moreover have σ' v = ?σ v using v ∈ V' by simp
      ultimately show ?thesis by (metis subgame-E subsetCE)
    next
      assume v /∈ V'
  qed
4.5 Conforming Strategies

lemma \text{path-conforms-with-strategy-ltl} \text{ [intro]:}
\begin{align*}
\text{path-conforms-with-strategy} \quad &\implies \text{path-conforms-with-strategy} \quad \text{(ltl } P \text{ ) } \\
\text{by} \quad &\text{(drule path-conforms-with-strategy \_cases)} \quad \text{(simp-all add: path-conforms-with-strategy \_intros(1))}
\end{align*}

lemma \text{path-conforms-with-strategy-drop:}
\begin{align*}
\text{path-conforms-with-strategy} \quad &\implies \text{path-conforms-with-strategy} \quad \text{(ldrop n } P \text{ ) } \\
\text{by} \quad &\text{(simp add: path-conforms-with-strategy-ltl ldrop[\of } \lambda P. \text{ path-conforms-with-strategy} \quad P \quad P \text{])}
\end{align*}

lemma \text{path-conforms-with-strategy-prefix:}
\begin{align*}
\text{path-conforms-with-strategy} \quad &\implies \text{lprefix } P \quad P' \implies \text{path-conforms-with-strategy} \quad P \quad P' \quad P \quad P' \\
\text{proof} \quad &\text{(coinduction arbitrary: } P \quad P' \text{)} \\
\text{case} \quad &\text{(path-conforms-with-strategy } P \quad P' \text{)} \\
\text{thus} \quad &\text{?thesis \using path-conforms-with-strategy(2) \by auto}
\end{align*}

next \text{case path-conforms-LCons-LNil}
\text{thus} \quad &\text{?thesis \by (metis lprefix-LCons-conv lprefix-antisym lprefix-code(1) path-conforms-with-strategy(2))}
\next

\text{case} \quad &\text{(path-conforms-VVp v w )} \\
\text{thus} \quad &\text{?thesis \proof (cases)} \\
\text{assume} \quad &P' \neq \text{LNil } \land \ P' \neq \text{LCons } v \text{ LNil} \\
\text{hence} \quad &\exists Q. \ P' = \text{LCons } v \text{ LCons } w \text{ Q) } \\
\text{by} \quad &\text{(metis local.path-conforms-VVp(1) lprefix-LCons-conv path-conforms-with-strategy(2))} \\
\text{thus} \quad &\text{?thesis \using local.path-conforms-VVp(1,3,4) path-conforms-with-strategy(2) \by force}
\text{qed \ auto}
\next

\text{case} \quad &\text{(path-conforms-VVpstar v )} \\
\text{thus} \quad &\text{?thesis \proof (cases)} \\
\text{assume} \quad &P' \neq \text{LNil } \\
\text{hence} \quad &\exists Q. \ P' = \text{LCons } v \text{ Q} \\
\text{using} \quad &\text{local.path-conforms-VVpstar(1) lprefix-LCons-conv path-conforms-with-strategy(2) \by fastforce} \\
\text{thus} \quad &\text{?thesis \using local.path-conforms-VVpstar path-conforms-with-strategy(2) \by auto}
\text{qed \ simp}
\text{qed}
\text{qed}

lemma \text{path-conforms-with-strategy-irrelevant:}
\begin{align*}
\text{assumes} \quad &\text{path-conforms-with-strategy} \quad P \quad P \quad v \quad \notin \text{ lset } P
\end{align*}
shows path-conforms-with-strategy p P (σ(v := w))
using assms apply (coinduction arbitrary: P) by (drule path-conforms-with-strategy.cases) auto

lemma path-conforms-with-strategy-irrelevant-deadend:
assumes path-conforms-with-strategy p P σ deadend v ∨ v ∉ VV p valid-path P
shows path-conforms-with-strategy p P (σ(v := w))
using assms proof (coinduction arbitrary: P)
let ?σ = σ(v := w)
case (path-conforms-with-strategy P)
thus ?case proof (cases rule: path-conforms-with-strategy.cases)
case (path-conforms-VVp v' w Ps)
have w = ?σ v' proof -
  from (valid-path P) have ~deadend v'
  using local.path-conforms-VVp(1) valid-path-cons-simp by blast
with assms(2) have v' ≠ v using local.path-conforms-VVp(2) by blast
thus w = ?σ v' by (simp add: local.path-conforms-VVp(3))
qed
moreover
have ∃ P. LCons w Ps = P ∧ path-conforms-with-strategy p P σ ∧ (deadend v ∨ v ∉ VV p)
∧ valid-path P
proof -
  have valid-path (LCons w Ps)
  using local.path-conforms-VVp(1) path-conforms-with-strategy(3) valid-path-ll' by blast
  thus ?thesis using local.path-conforms-VVp(4) path-conforms-with-strategy(2) by blast
  qed ultimately show ?thesis using local.path-conforms-VVp(1,2) by blast
next
case (path-conforms-VVpstar v' Ps)
have ∃ P. path-conforms-with-strategy p Ps σ ∧ (deadend v ∨ v ∉ VV p) ∧ valid-path Ps
  using local.path-conforms-VVpstar(1,3) path-conforms-with-strategy(2,3) valid-path-ll' by blast
  thus ?thesis by (simp add: local.path-conforms-VVpstar(1,2))
  qed simp-all
  qed

lemma path-conforms-with-strategy-irrelevant-updates:
assumes path-conforms-with-strategy p P σ \v. v ∈ lset P ⇒ σ v = σ' v
shows path-conforms-with-strategy p P σ'
using assms proof (coinduction arbitrary: P)
case (path-conforms-with-strategy P)
thus ?case proof (cases rule: path-conforms-with-strategy.cases)
case (path-conforms-VVp v' w Ps)
  have w = ?σ v' using local.path-conforms-VVp(1,3) path-conforms-with-strategy(2,3) by auto
  thus ?thesis using local.path-conforms-VVp(1,4) path-conforms-with-strategy(2) by auto
  qed simp-all
  qed

lemma path-conforms-with-strategy-irrelevant'
assumes path-conforms-with-strategy p P (σ(v := w)) v ∉ lset P
shows path-conforms-with-strategy p P σ
by (metis assms fun-upd-triv fun-upd-upd path-conforms-with-strategy-irrelevant)
\begin{verbatim}
lemma path-conforms-with-strategy-irrelevant-deadend':
  assumes path-conforms-with-strategy p P (σ(v := w)) deadend v ∨ v \notin 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 \iff v \in VV p \Longrightarrow σ v = w
  by (drule path-conforms-with-strategy.cases) simp-all

lemma path-conforms-with-strategy-lappend:
  assumes
    P: \textit{finite} P \neg\textit{lnull} P path-conforms-with-strategy p P σ
    and \ P': \neg\textit{lnull} P' \textit{path-conforms-with-strategy} p P' σ
    and \textit{lappend}: \textit{lhd} P \in VV p \Longrightarrow \textit{lhd} P' = \textit{lhd} P
  shows path-conforms-with-strategy p (lappend P P') σ
  using assms proof (induct P rule: \textit{finite-induct})
  case (LCons P)
  show ?case proof (cases)
    assume \textit{lnull} (lhd P)
    then obtain v0 where v0: P = LCons v0 LNil
      by (metis LCons.prems(1) \textit{lhd}(LCons-ltl llist.collapse(1))
    have path-conforms-with-strategy p (LCons (lhd P) P') σ proof (cases)
      assume \textit{lhd} P \in VV p
      moreover with v0 have \textit{lhd} P' = σ (\textit{lhd} P)
        using LCons.prems(5) by auto
      ultimately show ?thesis
        using path-conforms-\textit{VVp}[of lhd P P \textit{lhd} P' σ]
          by (metis (no-types) LCons.prems(4) \neg\textit{lnull} P' \textit{lhd}(LCons-ltl))
    next
    assume \textit{lnull} (lhd P)
    hence *: path-conforms-with-strategy p (lappend (lhd P) P') σ
      by (metis LCons.hyps(3) LCons.prems(1) LCons.prems(2) LCons.prems(5) LCons.prems(5)
        \textit{assms}(4) \textit{assms}(5) \textit{lhd}(LCons-ltl) \textit{lappend-LCons-ltl path-conforms-with-strategy-ltl})
    have path-conforms-with-strategy p (LCons (lhd P) (lappend (lhd P) P')) σ proof (cases)
      assume \textit{lhd} P \in VV p
      moreover hence \textit{lhd} (lhd P) = σ (\textit{lhd} P)
        by (metis LCons.prems(1) LCons.prems(2) \neg\textit{lnull} (lhd P)
          \textit{lhd}(LCons-ltl path-conforms-with-strategy-start)
      ultimately show ?thesis
        using path-conforms-\textit{VVp}[of lhd P P \textit{lhd} (lhd P) σ] * (\neg\textit{lnull} (lhd P))
          by (metis \textit{lappend-code}(2) \textit{lhd}(LCons-ltl))
    next
    assume \textit{lhd} P \notin VV p
    thus ?thesis by (simp add: * path-conforms-\textit{VVpstar})
  qed
  with (\neg\textit{lnull} P) show path-conforms-with-strategy p (lappend P P') σ
    by (metis \textit{lappend-code}(2) \textit{lhd}(LCons-ltl))

24
\end{verbatim}
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 \texttt{vmc-path} and \texttt{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.

\begin{verbatim}
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
  else if v0 ∈ VV p

\end{verbatim}
then greedy-conforming-path \( p \sigma \sigma' (\sigma v0) \)
else greedy-conforming-path \( p \sigma \sigma' (\sigma' v0) \)

**private lemma** greedy-path-LNil: greedy-conforming-path \( p \sigma \sigma' v0 \neq LNil \)
using greedy-conforming-path.disc-iff list.discI(1) by blast

**private lemma** greedy-path-lhd: greedy-conforming-path \( p \sigma \sigma' v0 = LCons v P \implies v = v0 \)
using greedy-conforming-path.code by auto

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

**private corollary** greedy-path-deadend-v:
    greedy-conforming-path \( p \sigma \sigma' v0 = LCons v P \implies P = \operatorname{LNil} \leftrightarrow \text{deadend } v \)
using greedy-path-deadend-v0 greedy-path-lhd by metis

**corollary** greedy-path-deadend-v': greedy-conforming-path \( p \sigma \sigma' v0 = LCons v LNil \implies \text{deadend } v \)
using greedy-path-deadend-v by blast

**private lemma** greedy-path-ltl:
    assumes greedy-conforming-path \( p \sigma \sigma' v0 = LCons v P \)
    shows \( P = LNil \lor P = \text{greedy-conforming-path } p \sigma \sigma' (\sigma v0) \lor P = \text{greedy-conforming-path } p \sigma \sigma' (\sigma' v0) \)
    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.sel(2) ltl.simps(2))

**private lemma** greedy-path-ltl-ex:
    assumes greedy-conforming-path \( p \sigma \sigma' v0 = LCons v P \)
    shows \( P = LNil \lor (\exists v. P = \text{greedy-conforming-path } p \sigma \sigma' v) \)
using assms greedy-path-ltl by blast

**private lemma** greedy-path-ltl-VVp:
    assumes greedy-conforming-path \( p \sigma \sigma' v0 = LCons v0 P v0 \in VV p \rightarrow \text{deadend } v0 \)
    shows \( \sigma v0 = \text{lhd } P \)
using assms greedy-conforming-path.code by auto

**private lemma** greedy-path-ltl-VVpstar:
    assumes greedy-conforming-path \( p \sigma \sigma' v0 = LCons v0 P v0 \in VV p^{**} \rightarrow \text{deadend } v0 \)
    shows \( \sigma' v0 = \text{lhd } P \)
using assms greedy-conforming-path.code by auto

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

26
and greedy-path-conforms': path-conforms-with-strategy p∗∗ (greedy-conforming-path p σ σ' v0)

proof

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

show ¬null P P $ 0 = v0 by (simp-all add: nth-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' = LCons v0 P' by (metis greedy-path-LNil greedy-path-lhd neq-LNil-conv)
  hence ¬deadend v0 using asm greedy-path-deadend-v0 v0 ∈ V by blast
  from P' have 1: ¬null P' using asm list-collapse(1) v0 ∈ V greedy-path-deadend-v0 by blast
  moreover from P' (¬deadend v0) assms(2,3) v0 ∈ V have v0 → lhd P' unfolding strategy-def using greedy-path-ltl-VVp greedy-path-ltl-VVpstar by (cases v0 ∈ VV p) auto
  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 lnul l-def)
}

The conjunction of all the above.

ultimately

have ∃ P'. ?P = LCons v0 P' ∧ ¬null 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 with assms have strategy p'' σ'' by blast
  hence path-conforms-with-strategy p'' P σ'' by (cases v0 ∈ V unfolding P-def)
  proof (coinduction arbitrary; v0)
    case (path-conforms-with-strategy v0)
    show ?case proof (cases v0 ∈ VV p'')
      case True

  27
\{ \text{assume } \neg(\exists \, v. \text{greedy-conforming-path } p \, \sigma \, \sigma' \, v0 = L\text{Cons } v \, L\text{Nil}) \\
\text{with } (v0 \in V) \text{ obtain } P' \text{ where} \\
P' : \text{greedy-conforming-path } p \, \sigma \, \sigma' \, v0 = L\text{Cons } v0 \, P' \, \text{null } P' \, v0 \rightarrow \text{llhd } P' \\
\text{llhd } P' \in V \, \exists \, v. \, P' = \text{greedy-conforming-path } p \, \sigma \, \sigma' \, v \land v \in V \\
\text{using } \text{coinduction-helper by blast} \\
\text{with } (v0 \in V V \, p'' \, \text{have } \sigma'' \, v0 = \text{llhd } P'' \\
\text{using } \text{greedy-path-ltl-VVp greedy-path-ltl-VVpstar by blast} \\
\text{with } (v0 \in V V \, p'' \, P'(1, 2, 5) \text{ have } ?\text{path-conforms-VVp} \\
\text{using } \text{greedy-conforming-path.code path-conforms-with-strategy(1) by fastforce} \\
\} \text{ thus } \text{thesis by auto}
\}

\text{next} \\
\text{case False} \\
\text{thus } \text{thesis using coinduction-helper[of v0] path-conforms-with-strategy by auto} \\
\text{qed} \\
\text{qed} \}

\text{thus path-conforms-with-strategy } p \, P \, \sigma \text{ path-conforms-with-strategy } p** \, P \, \sigma' \text{ by blast} \\
\text{qed}

\text{corollary strategy-conforming-path-exists:} \\
\text{assumes } v0 \in V \text{ strategy } p \, \sigma \text{ strategy } p** \, \sigma' \\
\text{obtains } P \text{ where } \text{vmc2-path } G \, p \, v0 \, \sigma \, \sigma' \\
\text{proof} \\
\text{show } \text{vmc2-path } G \, (\text{greedy-conforming-path } p \, \sigma \, \sigma' \, v0) \, v0 \, p \, \sigma \, \sigma' \\
\text{using assms by unfold-locales (simp-all add: greedy-conforming-path-properties)} \\
\text{qed}

\text{corollary strategy-conforming-path-exists-single:} \\
\text{assumes } v0 \in V \text{ strategy } p \, \sigma \\
\text{obtains } P \text{ where } \text{vmc-path } G \, p \, v0 \, \sigma \\
\text{proof} \\
\text{show } \text{vmc-path } G \, (\text{greedy-conforming-path } p \, \sigma \text{-arbitrary } v0) \, v0 \, p \, \sigma \\
\text{using assms by unfold-locales (simp-all add: greedy-conforming-path-properties)} \\
\text{qed}

\text{end}

\text{end}

\textbf{4.7 Valid Maximal Conforming Paths}

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

\textbf{context vmc-path begin} \\
\textbf{lemma } \texttt{Ptl-conforms [simp]: path-conforms-with-strategy } p \, (\texttt{ltl } P) \, \sigma \\
\text{using } \texttt{P-conforms path-conforms-with-strategy-ltl by blast} \\
\textbf{lemma } \texttt{Pdrop-conforms [simp]: path-conforms-with-strategy } p \, (\texttt{ldropn } n \, P) \, \sigma \\
\text{using } \texttt{P-conforms path-conforms-with-strategy-drop by blast} \\
\textbf{lemma } \texttt{prefix-conforms [simp]: path-conforms-with-strategy } p \, (\texttt{ltake } n \, P) \, \sigma \\
\text{using } \texttt{P-conforms path-conforms-with-strategy-prefix by blast} \\
\textbf{lemma } \texttt{extension-conforms [simp]:}
\( (v' \in VV p \Rightarrow \sigma v' = v0) \Rightarrow \text{path-conforms-with-strategy } p \ (LCons v' P) \sigma \)

by (metis P-LCons P-conforms path-conforms-VVp path-conforms-VVpstar)

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

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

**lemma conforms-to-another-strategy:**
- **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**

**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 = vmc-path +**

- **assumes** \( v0-no-deadend \ [simp]: \neg \text{deadend } v0 \)

**begin**

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

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

**lemma P-LCons: \text{ltl } P = LCons w0 (\text{ltl } (\text{ltl } P)) \text{ 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-paths-edges')**

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

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

**lemma Ptl-edge [simp]: v0 \rightarrow lhd (\text{ltl } P) by (metis P-LCons' P-valid valid-path-edges' w0-def)**

**lemma v0-conforms: v0 \in VV p \Rightarrow \sigma v0 = w0**
- **using** path-conforms-with-strategy-start by (metis P-LCons' P-conforms)

**lemma w0-V [simp]: w0 \in V by (metis P-LCons Ptl-valid valid-path-cons-simp)**

**lemma w0-lset-P [simp]: w0 \in 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)**

**end**
context vmc-path begin

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

lemma vmc-path-conforms:
  assumes enat (Suc n) < length P P $ n \in VV p
  shows \sigma (P $ n) = P $ Suc n
proof
  define P' where P' = ldropn n P
  then interpret P': vmc-path G P' P $ n p \sigma using vmc-path-ldropn assms(1) Suc-l-length by blast
  have \neg deadend (P $ n) using assms(1) P-no-deadends by blast
  then interpret P': vmc-path-no-deadend G P' P $ n p \sigma by unfold-locales
  have \sigma (P $ n) = P'.w0 using P'.\sigma-conforms assms(2) by blast
  thus ?thesis using P'.P-Suc-0 assms(1) by simp
qed

4.9 \text{lset} Induction Schemas for Paths

Let us define an induction schema useful for proving 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. \ [vmc-path-no-deadend G P v0 p \sigma; v0 \in S; Q P \] 
  \implies \ Q (\text{lil} P) \land (vmc-path-no-deadend.w0 P) \in S
  shows \ 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 vmc-path G P v0 p \sigma by blast
  show ?case by (simp add: find.prems(3))
next
  case (step P v)
  then interpret vmc-path G P v0 p \sigma by blast
  show ?case proof (cases)
    assume lnul l (\text{lil} P)
    hence P = LCons v LNil by (metis llist disco(2) lset-cases step.hyps(2))
    thus ?thesis using step.prems(3) P-LCons by blast
next
  assume \neg lnull (\text{lil} P)
  then interpret 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
  by blast
corollary vmc-path-lset-induction-simple [case-names base step]:
assumes base: \( v_0 \in S \)
and step: \( \forall P. [\text{vmc-path-no-deadend} G P v_0 p \sigma; v_0 \in ?S; \ ?Q P] \implies ?Q (\llt \ P) \wedge \text{vmc-path-no-deadend}.w0 P \in ?S] \implies \text{lset} \subseteq ?S \) without the \( Q \) predicate.

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

lemma vmc-path-lset-induction-closed-subset [case-names VVp VVpstar v0 disjoint]:
assumes VVp: \( \forall v. [\neg \text{deadend} v; v \in VV \ P ] \implies \sigma v \in S \cup T \)
and VVpstar: \( \forall w. [\neg \text{deadend} v; v \in VV \ P^*; v \rightarrow w] \implies w \in S \cup T \)
and v0: \( v_0 \in S \)
and disjoint: \( \text{lset} \cap T = \{\} \)
shows \( \text{lset} \subseteq S \)
using disjoint proof (induct rule: vmc-path-lset-induction)

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 \( W \).
**Definition** \( \text{visits-via} \ P \ A \ W \equiv \exists n. \text{enat} \ n < \text{llength} \ P \land P \ $ n \in W \land \text{lset} \ (\text{ltake} \ (\text{enat} \ n) \ P) \subseteq A \)

**Lemma** \( \text{visits-via-monotone} : [ \text{visits-via} \ P \ A \ W ; A \subseteq A'] \implies \text{visits-via} \ P \ A' \ W \)

**Unfolding** \( \text{visits-via-def} \) by blast

**Lemma** \( \text{visits-via-visits} : \text{visits-via} \ P \ A \ W \implies \text{lset} \ P \cap W \neq {} \)

**Unfolding** \( \text{visits-via-def} \) by (meson disjoint-iff-not-equal in-lset-conv-lth)

**Lemma** \( \text{in vmc-path} \) \( \text{visits-via-trivial} : v0 \in W \implies \text{visits-via} \ P \ A \ W \)

**Unfolding** \( \text{visits-via-def} \) apply (rule exI[of - 0]) using zero-enat-def by auto

**Lemma** \( \text{visits-via-LCons} : \)

**Assumes** \( \text{visits-via} \ P \ A \ W \)

**Shows** \( \text{visits-via} \ (\text{LCons} \ v0 \ P) \ (\text{insert} \ v0 \ A) \ W \)

**Proof**

- **Obtain** \( n \) where \( n : \text{enat} \ n < \text{llength} \ P \)$ $ n \in W \land \text{lset} \ (\text{ltake} \ (\text{enat} \ n) \ P) \subseteq A \)

  **Using** \( \text{assms} \)

  **Unfolding** \( \text{visits-via-def} \) by blast

  **Define** \( P' = \text{LCons} \ v0 \ P \)

  **Have** \( \text{enat} \ (\text{Suc} \ n) < \text{llength} \ P' \)

  **Unfolding** \( P'^{-\text{def}} \)

  **By** (metis n(1) ldropn-Suc-LCons ldropn-Suc-Conv-LConsD)

  **Moreover** have \( P'^{\$ \ \text{Suc} \ n} \in W \)

  **Unfolding** \( P'^{-\text{def}} \) by \( \text{simpl add: n(2)} \)

  **Moreover** have \( \text{lset} \ (\text{ltake} \ (\text{enat} \ (\text{Suc} \ n)) \ P') \subseteq \text{insert} \ v0 \ A \)

  **Using** \( \text{lset-ltake-Suc[of - P' v0 n A]} \)

  **Unfolding** \( P'^{-\text{def}} \) by \( \text{simpl add: n(3)} \)

  **Ultimately show** ?thesis

**Qed**

**Lemma** \( \text{in vmc-path-no-deadend} \) \( \text{visits-via-ltl} : \)

**Assumes** \( \text{visits-via} \ P \ A \ W \land \text{v0} : v0 \notin W \)

**Shows** \( \text{visits-via} \ (\text{ltl} \ P) \ A \ W \)

**Proof**

- **Obtain** \( n \) where \( n : \text{enat} \ n < \text{llength} \ P \)$ $ P \$ n \in W \land \text{lset} \ (\text{ltake} \ (\text{enat} \ n) \ (\text{ltl} \ P)) \subseteq A \)

  **Using** \( \text{assms(1)[unfolded visits-via-def]} \) by blast

  **Have** \( n \neq 0 \) using \text{v0 n(2)} DiffE by force

  **Then** **Obtain** \( n' \) where \( n' : \text{Suc} \ n' = n \)

  **Using** nat.exhaust by metis

  **Have** \( \exists n'. \text{enat} \ n < \text{llength} \ (\text{ltl} \ P) \land (\text{ltl} \ P) \$ n \in W \land \text{lset} \ (\text{ltake} \ (\text{enat} n') \ (\text{ltl} \ P)) \subseteq A \)

  **Apply** (rule exI[of - n'])

  **Using** \( n' : \text{enat-Suc-ltl[of n' P]} \)

  **P-ltl-Suc-lset-ltl[of n' P] by auto**

  **Thus** ?thesis

  **Using** \( \text{visits-via-def} \) by blast

**Qed**

**Lemma** \( \text{in vm-path} \) \( \text{visits-via-deadend} : \)

**Assumes** \( \text{visits-via} \ P \ (\text{deadends} \ p) \)

**Shows** \( \text{winning-path} \ p** P \)

**Using** \( \text{assms} \)

**visits-via-deadend by blast**

5.2 Attracting Strategy from a Single Node

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

**Definition** \( \text{strategy-attracts-via} :: \text{Player} \Rightarrow 'a \text{Strategy} \Rightarrow 'a \Rightarrow 'a \text{set} \Rightarrow 'a \text{set} \Rightarrow \text{bool} \)

where
strategy-acts-via $p \sigma v0 A W \equiv \forall P. \text{vmc-path } G P v0 p \sigma \rightarrow \text{visits-via } P A W$

**lemma (in vmc-path) strategy-acts-viaE:**
- **assumes** strategy-acts-via $p \sigma v0 A W$
- **shows** visits-via $P A W$
- **using** strategy-acts-via-def assms vmc-path-axioms by blast

**lemma (in vmc-path) strategy-acts-via-SucE:**
- **assumes** strategy-acts-via $p \sigma v0 A W$
- **shows** $\exists n. \text{enat } (\text{Suc } n) < \text{lend } P \land P \not\in W \land \text{lset } (\text{ltake } (\text{enat } (\text{Suc } n)) P) \subseteq A$

**proof**
- obtain $n$ where $n$: enat $n < \text{lend } P \land P \not\in W \land \text{lset } (\text{ltake } (\text{enat } n) P) \subseteq A$
- using strategy-acts-viaE [unfolded visits-via-def] assms(1) by blast
- have $n \not= 0$ using assms(2) n(2) by (metis P-0)
- thus ?thesis using n not0-implies-Suc by blast

**qed**

**lemma (in vmc-path) strategy-acts-via-lset:**
- **assumes** strategy-acts-via $p \sigma v0 A W$
- **shows** $\text{lset } P \cap W = \{\}$

**proof**
- using assms[TTHEN strategy-acts-viaE, unfolded visits-via-def]

**by** (meson disjoint-if-not-equal lset-lnth-member subset-refl)

**lemma strategy-acts-via-v0:**
- **assumes** $\sigma; \text{strategy } p \sigma \text{strategy-acts-via } p \sigma v0 A W$
- and $v0; v0 \in V$
- **shows** $v0 \in A \cup W$

**proof**
- obtain $P$ where vmc-path $G P v0 p \sigma$
- using strategy-conforming-path-exists-single assms by blast
- then interpret vmc-path $G P v0 p \sigma$
- obtain $n$ where $n$: enat $n < \text{lend } P \land P \not\in W \land \text{lset } (\text{ltake } (\text{enat } n) P) \subseteq A$
- using $\sigma(2)[\text{unfolded strategy-acts-via-def visits-via-def}]$ 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 $v0 \in \text{lset } (\text{ltake } (\text{enat } n) P)$
      - by (metis 'n \not= 0'; P-not-null P-v0 enat-0-iff(1) lset.lnth(1) ltake.disc(2))
    - thus ?thesis using n(3) by blast
- qed

**corollary strategy-acts-not-outside:**
- $[ v0 \in V - A - W ; \text{strategy } p \sigma ] \Rightarrow \neg \text{strategy-acts-via } p \sigma v0 A W$
- using strategy-acts-via-v0 by blast

**lemma strategy-acts-viaI [intro]:**
- **assumes** $\forall P. \text{vmc-path } G P v0 p \sigma \Rightarrow \text{visits-via } P A W$
- **shows** strategy-acts-via $p \sigma v0 A W$
- unfolding strategy-acts-via-def using assms by blast

33
lemma strategy-attracts-via-no-deadends:
assumes \( v \in V \) \( v \in A - W \) strategy-attracts-via \( p \sigma v A W \)
shows \( \neg \text{deadend } v \)
proof
assume deadend \( v \)
define \( P \) where \( \lceil \text{simp}; P = LCons v LNil \rceil \)
interpret vmc-path \( G P v p \sigma \) proof
  show valid-path \( P \) using \( (v \in A - W) \) \( (v \in V) \) valid-path-base \( \) by auto
  show maximal-path \( P \) using \( \text{deadend } v \) by (simp add: maximal-path-intros \( (2) \))
  show path-conforms-with-strategy \( p \) \( P \sigma \) by (simp add: path-conforms-LCons-LNil)
qed simp-all
have visits-via \( P A W \) using \( \text{assms}(3) \) strategy-attracts-viaE by blast
moreover have length \( P = eSuc 0 \) by simp
ultimately have \( P \$ 0 \in W \) by (simp add: cnat-0-iff \( (1) \) visits-via-def)
with \( v \in A - W \) show False by auto
qed

lemma attractor-strategy-on-extends:
\[ \lceil \text{strategy-attracts-via } p \sigma v0 A W; A \subseteq A' \rceil \Rightarrow \text{strategy-attracts-via } p \sigma v0 A' W \]
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 \( (v0 \in W) \) 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 \( w0: v0 \rightarrow w0 \) \( w0 \in V V p \Rightarrow \sigma w0 = w0 \)
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 \( \lceil \text{simp}; P' = LCons v0 P \rceil \)
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 \( (v0 \rightarrow w0) \) by unfold-locales blast
have visits-via \( P' A W \) using \( \sigma(2)\) \( P'.\text{strategy-attracts-viaE} \) by blast
thus visits-via \( P A W \) using \( P'.\text{visits-via-trivial} v0 \) by simp
qed

lemma strategy-attracts-VVp:
assumes \( \sigma \): strategy \( p \sigma \) strategy-attracts-via \( p \sigma v0 A W \)
  and \( v: v0 \in A - W v0 \in V V p \neg \text{deadend } v0 \)
shows \( \sigma v0 \in A \cup W \)
proof
have \( v0 \rightarrow \sigma v0 \) using \( \sigma(1)\) \( \lfloor \text{unfolding strategy-def} \rfloor v(2,3) \) by blast
hence strategy-attracts-via \( p \sigma \) \( (\sigma v0) A W \)
using strategy-attracts-via-successor \( \sigma v(1) \) by blast
thus \( ? \text{thesis} \) using strategy-attracts-via-v0 \( (v0 \rightarrow \sigma v0) \) \( \sigma(1) \) by blast
qed

lemma strategy-attracts-VV\(\ast\)-pstar:
assumes strategy \( p \sigma v0 A W \)
and \( v0 \in A - W w0 \not\in VV p w0 \in V - A - W \)
sows \( \neg v0 \rightarrow w0 \)
by (metis assms strategy-attracts-not-outside strategy-attracts-via-successor)

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 \( \Rightarrow \) '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 \( p \sigma A W \)
shows visits-via \( p A W \)
using assms(1)[unfolded strategy-attracts-def] assms(2) strategy-attracts-viaE by blast

lemma strategy-attracts-I[intro]:
assumes \( \emptyset \)\( P v. \) [\( v \in A ; \) vmc-path \( G P v p \sigma \)] \( \Longrightarrow \) visits-via \( p A W \)
sows strategy-attracts \( p \sigma A W \)
unfolding strategy-attracts-def using assms by blast

lemma (in vmc-path) strategy-attracts-lset:
assumes strategy-attracts \( p \sigma A W \)
shows lset \( P \cap W \neq \{\} \)
using assms(1)[unfolded strategy-attracts-def] assms(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 \( P ; P = LCons v (LCons w P) \) \( v \in A - W w \not\in A \cup W \)
sows \( \neg \) visits-via \( P A W \) (is \( \neg ?A \))

proof
assume \( \neg A \)
then obtain \( n \) where \( n : \text{enat} n < \text{lheight} P P \$ n \in W \) lset \( \text{ltake} (\text{enat} n) P \) \( \subseteq A \)
unfolding visits-via-def by blast
have \( n \neq 0 \) using \( \forall v \in A - W ; n(2) \) \( P(1) \) \( \text{Diff2} \) by force
moreover have \( n \neq \text{Suc} 0 \) using \( \forall w \notin A \cup W ; n(2) \) \( P(1) \) by auto
ultimately have \( \text{Suc} (\text{Suc} 0) \leq n \) by presburger
hence lset \( \text{ltake} (\text{enat} (\text{Suc} (\text{Suc} 0))) P \) \( \subseteq A \) using n(3)
b by (meson contra-subsetD enat-ord-simps(1) lset-ltake-prefix lset-ldth-member lset-subset)
mmoreover have \( \text{enat} (\text{Suc} 0) < \text{lheight} (\text{ltake} (e\text{Suc} (\text{Suc} 0))) P \) proof
have \( * : \text{enat} (\text{Suc} (\text{Suc} 0)) < \text{lheight} P \)
using \( \text{Suc} (\text{Suc} 0) \leq n \) n(1) by (meson enat-ord-simps(2) le-less-linear less-le-trans nq-iiff)
have \( \text{lheight} (\text{ltake} (\text{enat} (\text{Suc} (\text{Suc} 0)))) P = \min (\text{enat} (\text{Suc} (\text{Suc} 0))) (\text{lheight} P) \) by simp
hence \( \text{lheight} (\text{ltake} (\text{enat} (\text{Suc} (\text{Suc} 0)))) P = \text{enat} (\text{Suc} (\text{Suc} 0)) \)

35
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:**

**Assumptions**
- \( \sigma \): strategy-attracts \( p \), \( A \)
- \( W \) strategy \( p \), \( \sigma' \)

**Shows**
- strategy-attracts \( p \) (override-on \( \sigma' \) \( (A - W) \)) \( A \) \( W \)

**Proof**
- fix \( P, v \)
- let \( \sigma = \text{override-on} \ \sigma' \ (A - W) \)
- assume \( \text{vmc-path} \ G \ P \ v \ ? \sigma \)
- then interpret \( \text{vmc-path} \ G \ P \ v \ ? \sigma \)
- assume \( v \in A \)
- hence \( P \not\subseteq A \) using \( (v \in A) \) by simp
- moreover assume \( \neg \text{visits-via} \ P \ A \ W \)
- ultimately have \( P \not\subseteq A - W \) unfolding \( \text{visits-via-def} \) by (meson DiffI P-len not-less0 lset-ltake)

**Have**
- \( \neg \text{lset} \ P \not\subseteq A - W \)
- hence \( \sigma \not\subseteq A - W \) by simp
- hence path-conforms-with-strategy \( P \) \( \sigma \)

**Using**
- path-conforms-with-strategy-irrelevant-updates \( [OF \ P\ \text{conforms}] \) by blast
hence \(\text{vmc-path } G P (P \# 0) p \sigma\)
  using conforms-to-another-strategy P-0 by blast
  thus False
  using contra \((P \# 0) \in A\) assms(1)
  by (meson vmc-path-strategy-attractsE)
qed

hence \(\exists n. \text{enat } n < \text{llength } P \land P \# n \notin A - W\) by (meson lset-subset)
then obtain \(n\) where \(n: \text{enat } n < \text{llength } P \land P \# n \notin A - W\)
\(\land i. i < n \implies \neg(\text{enat } i < \text{llength } P \land P \# i \notin A - W)\)
  using ex-least-not-le[\(\forall n. \text{enat } n < \text{llength } P \land P \# n \notin A - W\)] by blast
hence \(n\)-min: \(\forall i. i < n \implies P \# i \in A - W\)
  using dual-order-strict-trans enat-ord-simps(2) by blast
have \(n \neq 0\) using \((P \# 0) \in A - W\) n(1) by meson
then obtain \(n'\) where \(n': \text{Suc } n' = n\) using not0-implies-Suc by blast
hence \(P \# n' \in A - W\) using \(n\)-min by blast
moreover have \(P \# n' \to P \# \text{Suc } n'\) using P-valid n(1) n'-valid-path-edges by blast
moreover have \(P \# \text{Suc } n' \notin A \cup W\) proof-
  have \(P \# n \notin W\) using contr n(1) n-min unfolding visits-via-def
    by (meson Diff-subset lset-ltake subsetCE)
  thus \(?\text{thesis using } n(1) n'\) by blast
qed

ultimately have \(P \# n' \in VV\ p \land \sigma (P \# n') \neq P \# \text{Suc } n'\)
  using strategy-attracts-does-not-leave[of p \sigma A W P \# n' P \# \text{Suc } n']
    assms(1,2) by blast
  thus False
    using n(1) n' vmc-path-conforms \((P \# n') \in A - W\) by (metis override-on-apply-in)
qed

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

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

lemma (in vmc-path) attracted-path:
  assumes \(W \subseteq V\)
  and \(\sigma: \text{strategy-attracts } p \sigma A W\)
  and \(P\)-hits-A: \(\text{lset } P \cap A \neq \{\}\)
  shows \(\text{lset } P \cap W \neq \{\}\)
proof-
  obtain \(n\) where \(n: \text{enat } n < \text{llength } P \# P \# n \in A\) using P-hits-A by (meson lset-intersect-lnth)
  define \(P'\) where \(P' = \text{idropn } n\)
  interpret \(\text{vmc-path } G P P' \# P \# n \sigma \text{ unfolding } P'\)-def using vmc-path-idropn n(1) by blast
  have visits-via \(P' A W\) using \(\sigma n(2)\) strategy-attractsE by blast
  thus \(?\text{thesis unfolding } P'\)-def using visits-via-visits in-lset-idropnD[of - n P] by blast
qed

lemma (in vmc-path-no-deadend) attracted-path-step:
  assumes \(\sigma: \text{strategy } p \sigma A W\)
  and \(v\): \(\neg\text{deadend } v\) \(v\ \in A - W\ v\ \in VV\ p\)
  shows \(\sigma v\) \(v\ \in A \cup W\)
  by (metis Diff1I strategy-attracts-VVp assms strategy-attracts-def)

lemma (in vmc-path-no-deadend) attracted-path-step:
assumes $\sigma$: strategy-attracts $p \sigma A W$
and $v0$: $v0 \in A - W$
shows $w0 \in A \cup W$
by (metis (no-types) Diff1 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-attraced :: Player $\Rightarrow$ 'a set $\Rightarrow$ 'a set where
directly-attraced $p$ $S$ $\equiv$ \{ $v$ $\in$ $V - S$. $\neg$deadend $v$ $\land$
  \{ $v$ $\in$ $VV p$ $\longrightarrow$ ($\exists$ $w$. $v\rightarrow w$ $\land$ $w$ $\in$ $S$))$
$\land$ \{ $v$ $\in$ $VV p**$ $\longrightarrow$ ($\forall$ $w$. $v\rightarrow w$ $\longrightarrow$ $w$ $\in$ $S$))\}

abbreviation attractor-step $p$ $W$ $S$ $\equiv$ $W$ $\cup$ $S$ $\cup$ directly-attraced $p$ $S$

The $p$-attractor set of $W$, defined as a least fixed point.

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

6.1 directly-attraced

Show a few basic properties of directly-attraced.

lemma directly-attraced-disjoint [simp]: directly-attraced $p$ $W \cap$ $W$ = \{\}
and directly-attraced-empty [simp]: directly-attraced $p$ \{\} = \{\}
and directly-attraced-V-empty [simp]: directly-attraced $p$ $V$ = \{\}
and directly-attraced-bounded-by-V [simp]: directly-attraced $p$ $W \subseteq$ $V$
and directly-attraced-contains-no-deadends [elim]: $v$ $\in$ directly-attraced $p$ $W$ $\Longrightarrow$ $\neg$deadend $v$
unfolding directly-attraced-def by blast+

6.2 attractor-step

lemma attractor-step-empty; attractor-step $p$ \{\} \{\} = \{\}
and attractor-step-bounded-by-V; [ $W \subseteq$ $V$; $S \subseteq$ $V$ ] $\Longrightarrow$ 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 monol) auto

6.3 Basic Properties of an Attractor

lemma `attractor-unfolding`: `attractor p W = attractor-step p W` (attractor p W) unfolding `attractor-def` using `attractor-step-mono lfp-unfold` by blast
and `attractor-set-base`: `W ⊆ attractor p W` using `attractor-unfolding by auto`
lemma `attractor-set-non-empty`: `W ≠ {}` `⇒` `attractor p W ≠ {}` unfolding `attractor-def` using  `attractor-step-mono` by auto
lemma `attractor-in-V`: `W ⊆ V` `⇒` `attractor p W ⊆ V` using `attractor-lowerbound attractor-step-bounded-by-V 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 ∨ 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


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′` shows `¬ Digraph.deadend (subgame V′) v` proof (cases) assume `deadend v` have `v: v ∈ V` `¬ ?A` `¬ ?B` using `v unfolding V′-def subgame-def by simp`
\{ \textbf{fix} \ p \textbf{\'assume} \ v \in VV \ p' \} \\
\textbf{hence} \ v \in \text{attractor} \ p' \ (\text{deadends} \ p''') \\
\textbf{using} \ (\text{deadend} \ v) \ \text{attractor-set-base}[\text{of} \ \text{deadends} \ p''') \ p' \} \\
\textbf{unfolding} \ \text{deadends-def} \ by \ \text{blast} \\
\textbf{hence} \ False \ \textbf{using} \ v \ by \ (\text{cases} \ p' ; \ \text{cases} \ p) \ \text{auto} \\
\} \\
\textbf{thus} \ ?\text{thesis} \ \textbf{using} \ v \ by \ \text{blast} \\
\textbf{next} \\
\textbf{assume} \ \neg \text{deadend} \ v \\
\textbf{have} \ v : v \in V = \neg A = \neg B \ \textbf{using} \ v \ \textbf{unfolding} \ V'\text{-def} \ \text{subgame-def} \ by \ \text{simp} \\
\textbf{define} \ G' \ where\ G' = \text{subgame} V' \\
\textbf{interpret} \ G' : \text{ParityGame} \ G' \ \textbf{unfolding} \ G'\text{-def} \ \text{using} \ \text{subgame-ParityGame} \ . \\
\textbf{show} \ ?\text{thesis} \ \textbf{proof} \\
\textbf{assume} \ \text{Digraph} \ \text{deadend} \ (\text{subgame} V') \ v \\
\textbf{hence} \ G'.\text{deadend} \ v \ \textbf{unfolding} \ G'.\text{-def} \ . \\
\textbf{have} \ \text{all-in-attractor} : \wedge w. \ v \rightarrow w \Longrightarrow w \in \neg A \lor w \in \neg B \ \textbf{proof} \ (\text{rule \ contr}) \\
\textbf{fix} \ w \\
\textbf{assume} \ v \rightarrow w \ (\neg (w \in \neg A \lor w \in \neg B)) \\
\textbf{hence} \ v \in V' \ \textbf{unfolding} \ V'\text{-def} \ by \ \text{blast} \\
\textbf{hence} \ w \in V_G \ \textbf{unfolding} \ G'\text{-def} \ \text{subgame-def} \ \text{using} \ (v \rightarrow w) \ by \ \text{auto} \\
\textbf{hence} \ v \rightarrow G'.g \ \textbf{using} \ (v \rightarrow w) \ \text{assms} (2) \ \textbf{unfolding} \ G'\text{-def} \ \text{subgame-def} \ \text{by} \ \text{auto} \\
\textbf{thus} \ False \ \textbf{using} \ (G'.\text{deadend} \ v) \ \textbf{using} \ (w \in V_G) \ \textbf{by} \ \text{blast} \\
\text{qed} \\
\{ \textbf{fix} \ p' \textbf{\'assume} \ v \in VV \ p' \} \\
\{ \textbf{assume} \ \exists w. \ v \rightarrow w \ \wedge w \in \text{attractor} \ p' \ (\text{deadends} \ p''') \} \\
\textbf{hence} \ v \in \text{attractor} \ p' \ (\text{deadends} \ p''') \ \textbf{using} \ w \in VV \ p' \ \textbf{attractor-set-VVp} \ \textbf{by} \ \text{blast} \\
\textbf{hence} \ False \ \textbf{using} \ v \ \textbf{by} \ (\text{cases} \ p' ; \ \text{cases} \ p) \ \text{auto} \\
\} \\
\textbf{hence} \ \wedge w. \ v \rightarrow w \Longrightarrow w \in \text{attractor} \ p'' \ (\text{deadends} \ p''') \\
\textbf{using} \ \text{all-in-attractor} \ \text{by} \ (\text{cases} \ p' ; \ \text{cases} \ p) \ \text{auto} \\
\textbf{hence} \ v \in \text{attractor} \ p'' \ (\text{deadends} \ p''') \\
\textbf{using} \ \neg \text{deadend} \ v \ (v \in VV \ p') \ \textbf{attractor-set-VVpstar} \ \textbf{by} \ \text{auto} \\
\textbf{hence} \ False \ \textbf{using} \ v \ \textbf{by} \ (\text{cases} \ p' ; \ \text{cases} \ p) \ \text{auto} \\
\} \\
\textbf{thus} \ False \ \textbf{using} \ v \ \textbf{by} \ \text{blast} \\
\text{qed} \\
\text{qed}

6.6 Attractor Set Induction

\textbf{lemma} \ mono\text{-restriction-is-mono} : \ mono \ f \Longrightarrow \ mono \ (\lambda S. f \ (S \cap V)) \\
\textbf{unfolding} \ mono\text{-def} \ \text{by} \ (\text{meson inf\text{-}mono monoD subset\text{-}refl})

Here we prove a powerful induction schema for \text{attractor}. Being able to prove this is the only reason why we do not use \text{inductive\_set} to define the attractor set.

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

\textbf{lemma} \ \text{attractor-set-induction} \ [\text{consumes} \ 1, \ \text{case\_names} \ \text{step} \ \text{union}] : \\
\textbf{assumes} \ W \subseteq V \\
\textbf{and} \ \text{step} : \wedge S. S \subseteq V \Longrightarrow P \ S \Longrightarrow P \ (\text{attractor\_step} \ p \ W \ S) \\
\textbf{and} \ \text{union} : \wedge M. \forall S \in M. S \subseteq V \wedge P \ S \Longrightarrow P \ (\bigcup M) \\
\textbf{shows} \ P \ (\text{attractor} \ p \ W)
proof -
  let ?P = \lambda S . \ P (S \cap V)
  let ?f = \lambda S . \ \text{attractor-step} p W (S \cap 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 ?A proof (rule \text{lfp-or-dinal-induct-set})
    show \forall S . ?P S \implies ?P (W \cup (S \cap V) \cup \text{directly-attracted} p (S \cap V))
      by (metis assms(1) \text{attractor-step-bounded-by-V} \text{inf.absorb1 inf.le2 local.step})
    show \forall M . \forall S \in M . ?P S \implies ?P (\bigcup M)
      proof -
        fix M
        let ?M = \{S \cap V | S \in M\}
        assume \forall S \in M . ?P S
        hence \forall S \in ?M . S \subseteq V \wedge P S by auto
        hence *: P (\bigcup ?M) by (simp add: union)
        have \bigcup ?M = (\bigcup M) \cap V by blast
        thus ?P (\bigcup M) using * by auto
      qed
    qed (insert f-mono)
  have *: W \cup (V \cap V) \cup \text{directly-attracted} p (V \cap V) \subseteq V
    using W \subseteq V \text{ attractor-step-bounded-by-V} by auto
  have ?A \subseteq V ?B \subseteq V using * by (simp-all add: lfp-lowerbound)
  have ?A = ?f ?A using f-mono lfp-unfold by blast
  hence ?A = W \cup (?A \cap V) \cup \text{directly-attracted} p (?A \cap V) using (?A \subseteq V) by simp
  hence *: \text{attractor-step} p W ?A \subseteq ?A using (?A \subseteq V) \text{inf.absorb1 by fastforce}
  have ?B = \text{attractor-step} p W ?B using \text{attractor-step-mono lfp-unfold by blast}
  hence ?f ?B \subseteq ?B using ?B \subseteq V by (metis \text{no-types, lifting} \text{equalityD2 le-iff-inf})
  have ?A = ?B proof
    show ?B \subseteq ?A using * by (simp add: lfp-lowerbound)
  qed
  hence ?P ?B using P-A by (simp add: \text{attractor-def})
  thus \text{thesis} using ?B \subseteq V by (simp add: \text{attractor-def le-iff-inf})
  qed
end — context ParityGame
end

7 Winning Strategies

theory WinningStrategy
imports  
  Main  
  Strategy
begin

end

41
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 Strategy \( \Rightarrow \) 'a \( \Rightarrow \) bool where
\[
\text{winning-strategy } p \sigma v_0 \equiv \forall P. \text{vmc-path } G P v_0 p \sigma \rightarrow \text{winning-path } p P
\]

lemma winning-strategyI [intro]:
\[
\text{assumes } \bigwedge P. \text{vmc-path } G P v_0 p \sigma \rightarrow \text{winning-path } p P
\]
\[
\text{shows winning-strategy } p \sigma v_0
\]
\[
\text{unfolding winning-strategy-def using assms by blast}
\]

lemma (in vmc-path) paths-hits-winning-strategy-is-winning:
\[
\text{assumes } \sigma; \text{winning-strategy } p \sigma v
\]
\[
\text{and } v; v \in \text{lset } P
\]
\[
\text{shows winning-path } p P
\]
proof –
\[
\text{obtain } n \text{ where } n: \text{enat } n < \text{length } P P \& n = v \text{ using } v \text{ by (meson in-lset-convt-bddh)}
\]
\[
\text{interpret } P'. \text{vmc-path } G \text{ ldropn } n P v p \sigma \text{ using } n \text{ vmc-path-ldropn by blast}
\]
\[
\text{have winning-path } p (\text{ldropn } n P) \text{ using } \sigma \text{ by (simp add: winning-strategy-def } P'.\text{vmc-path-axioms)}
\]
\[
\text{thus } \text{thesis using winning-path-drop-add P-valid } n(1) \text{ by blast}
\]
qed

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

lemma winning-strategy-only-for-one-player:
\[
\text{assumes } \sigma; \text{strategy } p \sigma \text{ winning-strategy } p \sigma v
\]
\[
\text{and } v; v \in V
\]
\[
\text{shows False}
\]
proof –
\[
\text{obtain } P \text{ where } \text{vmc2-path } G P v p \sigma \sigma' \text{ using assms strategy-conforming-path-exists by blast}
\]
\[
\text{then interpret } \text{vmc2-path } G P v p \sigma \sigma'.
\]
\[
\text{have winning-path } p P
\]
\[
\text{using paths-hits-winning-strategy-is-winning } \sigma(2) v0-lset-P \text{ by blast}
\]
\[
\text{moreover have winning-path } p \sigma' P
\]
\[
\text{using comp-paths-hits-winning-strategy-is-winning } \sigma'(2) v0-lset-P \text{ by blast}
\]
\[
\text{ultimately show False using } P\text{-valid paths-are-winning-for-one-player by blast}
\]
qed

7.1 Deadends

lemma no-winning-strategy-on-deadends:
\[
\text{assumes } v \in V P v \text{ deadend } v \text{ strategy } p \sigma
\]
\[
\text{shows } \neg \text{winning-strategy } p \sigma v
\]
proof –
\[
\text{obtain } P \text{ where } \text{vmc-path } G P v p \sigma \text{ using strategy-conforming-path-exists-single assms by blast}
\]
\[
\text{then interpret } \text{vmc-path } G P v p \sigma .
\]
\[
\text{have } P = LCons v LNil \text{ using } P\text{-deadend-v0-LCons(deadend } v) \text{ by blast}
\]
\[
\text{hence } \neg \text{winning-path } p P \text{ unfolding winning-path-def using } (v \in V V p) \text{ by auto}
\]
\[
\text{thus } \text{thesis using winning-strategy-def vmc-path-axioms by blast}
\]
lemma winning-strategy-on-deadends:
assumes \( v \in V V P \) deadend \( v \) 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 \( P = LCons v LNil \) using \( P\)-deadend-\( v0 \)-\( LCons \) deadend \( v \) by blast
  thus winning-path \( p** \) \( P \) unfolding winning-path-def
    using \( \{ v \in V V P \} \) \( P \)-valid paths-are-winning-for-one-player by auto
qed

7.2 Extension Theorems

lemma strategy-extends-\( V V P \):
assumes \( v0 : v0 \in V V p -deadend v0 \)
and \( \sigma : strategy p \sigma \) 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 \( vmc-path G P v0 p \sigma \).
  have \( v0 \rightarrow v0 \) using \( v0 \sigma(1) \) strategy-def by blast
  hence winning-path \( p \) \( LCons v0 P \)
    using \( \sigma(2) \) extension-valid-maximal-conforming winning-strategy-def by blast
  thus winning-path \( p \) \( P \) using winning-path-ltl[of \( p \ LCons v0 P \)] by simp
qed

lemma strategy-extends-\( V V P\)star:
assumes \( v0 : v0 \in V V p** v0 \rightarrow w0 \)
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 \( p \) \( (LCons v0 P) \)
    using extension-valid-maximal-conforming VV-impl1 \( \sigma v0 \) winning-strategy-def
    by auto
  thus winning-path \( p \) \( P \) using winning-path-ltl[of \( p \ LCons v0 P \)] by auto
qed

lemma strategy-extends-backwards-\( V V P\)star:
assumes \( v0 : v0 \in V V p** \)
and \( \sigma : strategy p \sigma \backslash w, v0 \rightarrow w \implies \) 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 \( \) thesis using \( P\)-deadend-\( v0 \)-\( LCons \) winning-path-def \( v0 \) by auto
next

qed
assume ¬deadend v0
then interpret vmc-path-no-deadend G P v0 p σ by unfold-locales
interpret ltlP: vmc-path G ltl P v0 p σ using vmc-path-ltl.
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-LCons ltlP.P-LCons ltlP.P-not-null)
qed
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-locales
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 ltlP.P-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 \( r\text{-wo: } \text{well-order-on } \{ \sigma. \exists \upsilon \in S. \sigma \in \text{good } \upsilon \} \ r \)
— Every node has a good strategy.
and \( \text{good-ex: } \left( \forall \upsilon. \upsilon \in S \Rightarrow \exists \sigma. \sigma \in \text{good } \upsilon \right) \)
— good strategies are well-formed strategies.
and \( \text{good-strategies: } \left( \forall \upsilon. \sigma. \sigma \in \text{good } \upsilon \Rightarrow \text{strategy } p \sigma \right) \)
— A good strategy on \( v \) is also good on possible successors of \( v \).
and \( \text{strategies-continue: } \left( \forall \upsilon \omega. \upsilon \in S ; \upsilon \rightarrow \omega ; \upsilon \in \text{VV } p \Rightarrow \sigma \upsilon = \omega ; \sigma \in \text{good } \upsilon \right) \Rightarrow \sigma \in \text{good } \omega \)

\begin{align*}
\text{begin} \\
\text{The set of all strategies which are good somewhere.} \\
\text{abbreviation } \text{Strategies} \equiv \{ \sigma. \exists \upsilon \in S. \sigma \in \text{good } \upsilon \} \\
\text{definition minimal-good-strategy where} \\
\text{minimal-good-strategy } v \sigma \equiv \sigma \in \text{good } v \land \left( \forall \sigma'. (\sigma', \sigma) \in r - Id \Rightarrow \sigma' \notin \text{good } v \right) \\
\text{no-notation binomial (infix choose 65)} \\
\end{align*}

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

\text{definition choose where} \quad \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.

\text{definition well-ordered-strategy where} \quad \text{well-ordered-strategy } \equiv \text{override-on } \sigma\text{-arbitrary } (\lambda \upsilon. \text{choose } v \upsilon) S

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

\text{lemma } r\text{-refl [simp]: } \text{refl-on } \text{Strategies } r \\
\text{using } r\text{-wo unfolding well-order-on-def linear-order-on-def partial-order-on-def preorder-on-def by blast} \\
\text{lemma } r\text{-total [simp]: } \text{total-on } \text{Strategies } r \\
\text{using } r\text{-wo unfolding well-order-on-def linear-order-on-def by blast} \\
\text{lemma } r\text{-trans [simp]: } \text{trans } r \\
\text{using } r\text{-wo unfolding well-order-on-def linear-order-on-def preorder-on-def by blast} \\
\text{lemma } r\text{-wf [simp]: } \text{wf } (r - Id) \\
\text{using } \text{well-order-on-def } r\text{-wo by blast} \\
\text{choose always chooses a minimal good strategy on } S.

\text{lemma choose-works:} \\
\text{assumes } v \in S \\
\text{shows } \text{minimal-good-strategy } v (\text{choose } v) \\
\text{proof} \quad \\
\text{have } \text{wf } (r - Id) \text{ using } \text{well-order-on-def } r\text{-wo by blast} \\
\text{obtain } \sigma \text{ where } \sigma 1: \text{minimal-good-strategy } v \sigma \\
\text{unfolding minimal-good-strategy-def by } (\text{meson good-ex } \{ \upsilon \in S \} \text{ wf eq-minimal}) \\
\text{hence } \sigma; \sigma \in \text{good } v \land (\sigma', \sigma) \in r - Id \Rightarrow \sigma' \notin \text{good } v \\
\text{unfolding minimal-good-strategy-def by auto} \\
\{ \text{ fix } \sigma' \text{ resume minimal-good-strategy } v \sigma' \}

45
hence $\sigma' \in \text{good } \lambda \sigma. (\sigma, \sigma') \in r - Id \implies \sigma \notin \text{good } v$

unfolding minimal-good-strategy-def by auto
have $(\sigma, \sigma') \notin r - Id$ using $\sigma(1) \sigma'(2)$ by blast
moreover have $(\sigma', \sigma) \notin r - Id$ using $\sigma(2) \sigma'(1)$ by auto
moreover have $\sigma \in \text{Strategies}$ using $\sigma(1)$ ($v \in S$) by auto
moreover have $\sigma' \in \text{Strategies}$ using $\sigma'(1)$ ($v \in S$) by auto
ultimately have $\sigma' = \sigma$

using r-wo Linear-order-in-diff-Id well-order-on-Field well-order-on-def by fastforce

with $\sigma I$ have $\exists ! \sigma$. minimal-good-strategy $v \sigma$ by blast

thus $?thesis$ using $?thesis$ unfolding minimal-good-strategy-def by blast

qed

corollary
assumes $v \in S$
show $\text{choose-good}$: choose $v \in \text{good } v$
and $\text{choose-minimal}$: $\lambda \sigma'. (\sigma', \text{choose } v) \in r - Id \implies \sigma' \notin \text{good } v$
and $\text{choose-strategy}$: strategy $p$ (choose $v$)
using $\text{choose-works}$[$\{ OF \text{ assms, unfolded minimal-good-strategy-def } \}$ good-strategies by blast+

corollary $\text{choose-in-Strategies}$: $v \in S$ $\Rightarrow$ choose $v \in \text{Strategies}$ using $\text{choose-good}$ by blast

lemma well-ordered-strategy-valid: strategy $p$ well-ordered-strategy
proof

{} fix $v$ assume $v \in S$ $v \in VV$ $p \_\text{deadend } v$
moreover have strategy $p$ (choose $v$)
using $\text{choose-works}$[$\{ OF \text{ assms, unfolded minimal-good-strategy-def } \}$ good-strategies by blast
ultimately have $v \rightarrow (\lambda v. \text{choose } v v) v$ using strategy-def by blast

} thus $?thesis$ 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 path-strategies $\equiv$ lmap choose

lemma path-strategies-in-Strategies:
assumes lset $P \subseteq S$
shows lset (path-strategies $P$) $\subseteq \text{Strategies}$
using path-strategies-def assms choose-in-Strategies by auto

lemma path-strategies-good:
assumes lset $P \subseteq S$ enat $n < \text{length } P$
shows path-strategies $P$ $\$ n $\in \text{good } (P \$ n)$
by (simp add: path-strategies-def assms choose-good lset-th member)

lemma path-strategies-strategy:
assumes lset $P \subseteq S$ enat $n < \text{length } P$

shows strategy \( p \) (path-strategies \( P \) \( \subseteq n \))
using path-strategies-good assms good-strategies by blast

lemma path-strategies-monotone-Suc:
assumes \( P\colon lset P \subseteq S \) valid-path \( P \) path-conforms-with-strategy \( p \) \( P \) well-ordered-strategy
enat \( (Suc n) < llength P \)
shows (path-strategies \( P \) \( \subseteq Suc n \), path-strategies \( P \) \( \subseteq n \)) \( n \in r \)
proof -
define \( P' \) where \( P' = \text{idropn} n P \)
hence enat \( (Suc 0) < llength P' \) using \( P(4) \)
  by (metis enat-ltl-Suc ldropn-Suc-converse ldropn llength-Suc-converse ldropn 1)
then obtain \( v w P s \) where \( v w = LCons v (LCons w P s) \)
  by (metis ldropn-0 ldropn-Suc-converse ldropn llength-Suc-converse ldropn 1)
moreover have \( \text{isset} (P' \subseteq S) \) unfolding \( P'-def \) using \( P(1) \) \( \text{idropn-subset}[of n P] \) by blast
ultimately have \( v w \in S \) \( w \in S \) by auto
moreover have \( v w \rightarrow w \) using valid-path-edges\( [v w P s, \text{folded } w v] \) valid-path-drop\( [OF P(2)] \)
P'-def by blast
moreover have choose \( v \in \text{good} \) \( v \) using choose-good \( \langle v \in S \rangle \) by blast
moreover have \( v \in V V P \implies \text{choose } v w = w \) proof -
assume \( v \in V V P \)
moreover have path-conforms-with-strategy \( P' \) \( \text{well-ordered-strategy} \)
unfolding \( P'-def \) using path-conforms-with-strategy-drop \( P(3) \) by blast
ultimately have \( \text{well-ordered-strategy} v = w \) using \( \text{well-ordered-strategy-start} \)
by blast
thus \( \text{choose } v v = w \) unfolding well-ordered-strategy-def using \( w \in S \) by auto
qed

ultimately have \( \text{choose } v \in \text{good} \) \( w \) using strategies-continue by blast
hence \( * \colon (\text{choose } v w \in \text{choose } w) \not\in r \) - \( \text{Id} \) using choose-minimal \( w \in S \) by blast

have \( \text{(choose } w, \text{choose } v) \in r \) proof (cases)
assume \( \text{choose } v = \text{choose } w \)
thus \( \text{?thesis} \) using r-refl refl-onD choose-in-Strategies\( [OF w \in S] \) by fastforce
next
assume \( \text{choose } v \neq \text{choose } w \)
thus \( \text{?thesis} \) using r-total choose-in-Strategies\( [OF w \in S] \) choose-in-Strategies\( [OF w \in S] \)
by (metis (lifting) linear-order-in-diff Id r-total well-order-on-Field well-order-on-def)
qed

hence \( \text{(path-strategies } P' \subseteq Suc 0, \text{path-strategies } P' \subseteq 0) \in r \)
unfolding path-strategies-def using \( w v \) by simp
thus \( \text{?thesis} \) unfolding path-strategies-def \( P'-def \)
    using \( \text{nth-lookup-idropn} [OF Suc-llength]\( [OF P(4)] \), of choose \)
    \( \text{nth-lookup-idropn-Suc} [OF P(4)], of choose \)
    by simp
qed

lemma path-strategies-monotone:
assumes \( P\colon lset P \subseteq S \) valid-path \( P \) path-conforms-with-strategy \( p \) \( P \) well-ordered-strategy
\( n < m \) enat \( m \) < llength \( P \)
shows (path-strategies \( P \) \( \subseteq m \), path-strategies \( P \) \( \subseteq n \)) \( n \in r \)
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[of P(1,2,3)]
    by (metis (no-types) Suc.hyps(2) Suc.prems(4,5) Suc-diff-Suc Suc-inject Suc-leI 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-onl-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 "infinite 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 ∧ (∀ σ. 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-lnth[of σ'] unfolding σ-set-def by blast
  |
  fix m assume n ≤ m
  have path-strategies P $ n = path-strategies P $ m proof (rule contr)
    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) n 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-$\sigma$-map-n:

- **Assumes**: $\text{set } P \subseteq S \text{ valid-path } P \text{ path-conforms-with-strategy } P \text{ well-ordered-strategy}
- **Shows**: $\exists n. \text{ path-conforms-with-strategy } P \ (\text{ldropn } n \ P) \ (\text{path-strategies } P \upharpoonright n)$

**Proof** (cases)

- **Assume** $\neg \text{finite } P$

  then obtain $n$ where $\text{llength } P = \text{enat } n$ using $\text{finite-llength-enat}$ by blast
  hence $\text{ldropn } n \ P = \text{LNil}$ by simp
  thus $\text{thesis}$ by (metis $\text{path-conforms-LNil}$)

- **Next**

  assume $\text{finite } P$

  then obtain $n$ where $n: \forall m. n \leq m \Rightarrow \text{path-strategies } P \upharpoonright n = \text{path-strategies } P \upharpoonright m$

  using $\text{path-strategies-eventually-constant}$ by blast

  let $?\sigma = \text{well-ordered-strategy}$

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

  \{ fix $v$ assume $v \in \text{set } P'$

  hence $v \in S$ using $\text{set } P \subseteq S$ $P'$-def in-\text{set-ldropnD}$ by fastforce

  from $v \in \text{set } P'$ obtain $m$ where $m: \text{enat } m < \text{llength } P' \ P' \upharpoonright m = v$ by (meson $\text{in-lset-conv-lnth}$)

  hence $P \upharpoonright m + n = v$ unfolding $P'$-def by (simp add: $\neg \text{finite } P$, $\text{finite-small-length}$)

  moreover have $?\sigma \ v = \text{choose } v \ v$ unfolding well-ordered-strategy-def using $\text{v \in S}$ by auto

  ultimately have $?\sigma \ v = (\text{path-strategies } P \upharpoonright m \ n) \ v$

  unfolding $\text{path-strategies-def}$ using $\text{finite-small-length}$[OF $\neg \text{finite } P$] by simp

  hence $?\sigma \ v = (\text{path-strategies } P \upharpoonright n) \ v$ using $\text{n[of } m \ n]$ by simp

  \}

  moreover have $\text{path-conforms-with-strategy } P \ P' \text{ well-ordered-strategy}$

  unfolding $P'$-def by (simp add: $\text{assms}(3) \ \text{path-conforms-with-strategy-drop}$)

  ultimately show $\text{thesis}$ using $\text{path-conforms-with-strategy-irrelevant-updates}$ $P'$-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** \( \text{winning-region } p \equiv \{ v \in V. \exists \sigma. \text{strategy } p \sigma \land \text{winning-strategy } p \sigma v \} \)

**Lemma** \( \text{winning-region} \) [intro]:
- **Assumes** \( v \in V \text{ strategy } p \sigma \text{ winning-strategy } p \sigma v \)
- **Shows** \( v \in \text{winning-region } p \)
- **Using** assms unfolding \( \text{winning-region-def} \) by blast

**Lemma** \( \text{winning-region-in-V} \) [simp]: \( \text{winning-region } p \subseteq V \) unfolding \( \text{winning-region-def} \) by blast

**Lemma** \( \text{winning-region-deadends} \):
- **Assumes** \( v \in V \text{ pp deadend } v \)
- **Shows** \( v \in \text{winning-region } p \)
- **Proof**
  
  **Show** \( v \in V \) using \( (v \in VV p) \) by blast

**Proof** unfolding \( \text{winning-region-def} \) by blast

**Note** simp

9.1 Paths in Winning Regions

**Lemma** (in \( \text{vmc-path} \)) \( \text{paths-stay-in-winning-region} \):
- **Assumes** \( \sigma ': \text{strategy } p \sigma' \text{ winning-strategy } p \sigma' v0 \)
  
  and \( \sigma: \bigwedge v. v \in \text{winning-region } p \Rightarrow \sigma ' v = \sigma v \)
- **Shows** \( \text{let } P \subseteq \text{winning-region } p \)
- **Proof**
  
  **Fix** \( x \) **Assume** \( x \in \text{let } P \)
  
  **Thus** \( x \in \text{winning-region } p \) using assms \( \text{vmc-path-axioms} \)

**Proof** (induct arbitrary; \( v0 \) rule: \( \text{list-set-induct} \))

**Case** \( (\text{find } P v0) \)

**Interpret** \( \text{vmc-path } G P v0 p \sigma \) using find.prems\((4)\).

**Show** \( ?\text{case using } P-v0 \sigma ' (1) \text{ find.prems}(2) v0-V \) unfolding \( \text{winning-region-def} \) by blast

**Next**

**Case** \( (\text{step } P x v0) \)

**Interpret** \( \text{vmc-path } G P v0 p \sigma \) using step.prems\((4)\).

**Show** \( ?\text{case proof} \) (cases)

**Assume** \( \text{null} (\text{ltl } P) \)

**Thus** ?thesis using \( P-\text{null-ltl-LCons step.hyps}(2) \) by auto

**Next**

**Assume** \( \neg \text{null} (\text{ltl } P) \)

**Then** interpret \( \text{vmc-path-no-deadend } G P v0 p \sigma \) using \( P-\text{no-deadend-v0} \) by unfold-locales

**Have** \( \text{winning-strategy } p \sigma' v0 \) **Proof** (cases)

**Assume** \( v0 \in VV p \)

**Hence** \( \text{winning-strategy } p \sigma' (v0) \)

**Using** strategy-extends-VVp local.step\((4)\) step.prems\((2)\) v0-no-deadend by blast

**Moreover** have \( \sigma v0 = w0 \) using \( v0-\text{conforms} (v0 \in VV p) \) by blast

**Moreover** have \( \sigma' v0 = \sigma v0 \)

**Using** \( \sigma \) assms\((1)\) step.prems\((2)\) v0-V unfolding \( \text{winning-region-def} \) by blast

**Ultimately** show ?thesis by simp

**Next**

**Assume** \( v0 \notin VV p \)

**Thus** ?thesis using \( v0-V \text{ strategy-extends-VVpstar step}(4) \) step.prems\((2)\) by simp

**Qed**

50
thus \textit{thesis} using \textit{step.hyps(3) step(4) \sigma \text{\textit{vmc-path-lll}} by blast}

\textit{qed}

\textit{qed}

\textit{qed}

\textbf{lemma (in \textit{vmc-path}) path-hits-winning-region-is-winning:}
\begin{itemize}
  \item \textit{assumes} \sigma': strategy \ p \ \sigma' \land \forall v. v \in \text{winning-region} \ p \implies \text{winning-strategy} \ p \ \sigma' \ v
  \item \textit{and} \ \sigma: \land \forall v. v \in \text{winning-region} \ p \implies \sigma' \ v = \sigma \ v
  \item \textit{and} \ \ p: \ \text{let} \ p \ \cap \ \text{winning-region} \ p \neq \ \{\}
\end{itemize}
\text{\textit{shows} winning-path \ p \ \ p}
\textit{proof}
\begin{itemize}
  \item \textit{obtain} \ n \ \textit{where} \ n: \ \text{enat} \ n < \text{llength} \ p \ \ p \ \ \ n \in \ \text{winning-region} \ p
  \item \textit{using} \ P \ \textit{by} \ (\text{meson} \ \text{iset-intersect-lnth})
  \item \textit{define} \ p' \ \textit{where} \ p' = \text{ldropn} \ n \ p
  \item \textit{then interpret} \ P': \ \textit{vmc-path} \ G \ P' \ P \ \ n \ \ p \ \sigma
  \item \textit{unfolding} \ P'-\textit{def using} \ \textit{vmc-path-ldropn} \ n(1) \ \textit{by blast}
  \item \textit{have} winning-strategy \ p \ \sigma'(v \ \text{lset} \ n(2) \ n(2)) \ \textit{by blast}
  \item \textit{hence} \ \text{let} \ p' \subseteq \text{winning-region} \ p
  \item \textit{using} \ P',\textit{paths-stay-in-winning-region}[OF \ \sigma'(1) - \sigma]
  \item \textit{by blast}
  \item \textit{hence} \ \land \forall v. v \in \text{iset} \ p' \implies \sigma \ v = \sigma' \ v \ \textit{using} \ \sigma \ \textit{by auto}
  \item \textit{hence} \ \text{path-conforms-with-strategy} \ p \ \ p' \ \sigma'
  \item \textit{using} \ \textit{path-conforms-with-strategy-irrelevant-updates} \ P'.P\text{-conforms}
  \item \textit{by blast}
  \item \textit{then interpret} \ P': \ \textit{vmc-path} \ G \ P' \ P \ \ n \ \ p \ \sigma' \ \textit{using} \ P'.\textit{conforms-to-another-strategy} \ \textit{by blast}
  \item \textit{have} winning-path \ p \ \ p' \ \textit{using} \ \sigma'(2) \ n(2) \ P',\textit{vmc-path-axioms} \ \textit{winning-strategy-def} \ \textit{by blast}
  \item \textit{thus} winning-path \ p \ \ p \ \textit{unfolding} \ P'-\textit{def using} \ \textit{winning-path-drop-add} \ n(1) \ \textit{P-valid} \ \textit{by blast}
\end{itemize}
\textit{qed}

\textbf{9.2 Irrelevant Updates}

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

\textbf{lemma winning-strategy-updates:}
\begin{itemize}
  \item \textit{assumes} \ \sigma: \ \text{strategy} \ p \ \sigma \ \text{winning-strategy} \ p \ \sigma \ v0
  \item \textit{and} \ v: \ v \notin \text{winning-region} \ p \ v\rightarrow v
\end{itemize}
\textit{shows} winning-strategy \ p \ \sigma(v := w) \ v0
\textit{proof}
\begin{itemize}
  \item \textit{fix} \ p \ \textit{assume} \ \textit{vmc-path} \ G \ p0 \ p \ (\sigma(v := w))
  \item \textit{then interpret} \ \textit{vmc-path} \ G \ p0 \ p \ \sigma(v := w)
  \item \textit{have} \ \land \forall v'. v' \in \text{winning-region} \ p \implies \sigma' \ v' = (\sigma(v := w)) \ v' \ \textit{using} \ v \ \textit{by auto}
  \item \textit{hence} \ v \notin \text{iset} \ p \ \textit{using} \ \textit{paths-stay-in-winning-region} \ \sigma \ \textit{unfolding} \ \textit{winning-region-def} \ \textit{by blast}
  \item \textit{hence} \ \text{path-conforms-with-strategy} \ p \ \sigma
  \item \textit{using} \ \textit{P-conforms path-conforms-with-strategy-irrelevant'} \ \textit{by blast}
  \item \textit{thus} \ \textit{winning-path} \ p \ \textit{using} \ \textit{conforms-to-another-strategy} \ \sigma(2) \ \textit{winning-strategy-def} \ \textit{by blast}
\end{itemize}
\textit{qed}

\textbf{9.3 Extending Winning Regions}

\textbf{lemma winning-region-extends-VVp:}
\begin{itemize}
  \item \textit{assumes} \ v: v \in \ VV \ p \ v\rightarrow v \ \textit{and} \ w: w \in \text{winning-region} \ p
\end{itemize}
\textit{shows} \ v \in \text{winning-region} \ p
proof (rule contr)
  obtain σ where σ: strategy p σ winning-strategy p σ w
      using w unfolding winning-region-def by blast
  let ?σ = σ(v := w)
  assume contra: v /∈ winning-region p
  moreover have strategy p ?σ using valid-strategy-updates σ(I) (v→w) by blast
  moreover hence winning-strategy p ?σ v
      using winning-strategy-updates σ contm vs strategy-extends-backwards-VVp
      by auto
  ultimately show False using (v→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

end

10 Uniform Strategies

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

theory UniformStrategy
imports
  Main
  ActingStrategy WinningStrategy WellOrderedStrategy WinningRegion
begin

context ParityGame begin

10.1 A Uniform Attractor Strategy

lemma merge-attractor-strategies:
  assumes S ⊆ V
  and strategies-ex: ∀v. v ∈ S ⟹ ∃σ. strategy p σ ∧ strategy-attracts-via p σ v S W
  shows ∃σ. strategy p σ ∧ strategy-attracts p σ S W
proof
  define good where good v = {σ. strategy p σ ∧ strategy-attracts-via p σ v S W} for v
  let ?G = {σ. ∃v ∈ S − W. σ ∈ 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 ⊆ V using (S ⊆ V) by blast
next
  show ∀v. v ∈ S − W ⟹ ∃σ. σ ∈ good v unfolding good-def using strategies-ex by blast
next
  show ∀v. σ ∈ good v ⟹ strategy p σ unfolding good-def by blast
next
  fix v w σ assume v: v ∈ S − W v→w v ∈ VV p ⟹ σ v = w σ ∈ good v
  hence σ: strategy p σ strategy-attracts-via p σ v S W unfolding good-def by simp-all

52
hence \( \text{strategy-} \text{attracts-via} \ p \ \sigma \ \text{w S W using} \ \text{strategy-} \text{attracts-via} \ \text{successor} \ v \ \text{by} \ \text{blast} \)

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

\text{qed (insert r)}

have \( \text{S-W-no-deadends: } \bigwedge \ v. \ v \in S - W \implies \neg \text{deadend} \ v \)

using \( \text{strategy-} \text{attracts-via} \ \text{no-deadends} \ \text{of} \ S \ W \ \text{strategies-ex} \)

by (metis (no-typ es) \text{Diff-iff S-V rev-subsetD})

{ 
  \text{fix} \ v0 \ \text{assume} \ v0 \in S 
  \text{fix} \ P \ \text{assume} \ P; \ \text{vmc-path} \ G \ P \ v0 \ p \ \text{well-ordered-strategy} 
  \text{then interpret} \ \text{vmc-path} \ G \ P \ v0 \ p \ \text{well-ordered-strategy} . 
  \text{have} \ \text{visits-via} \ P \ S \ W \ \text{proof} \ \text{(rule} \ \text{contr})

  \text{assume} \ \text{contra:} \ \neg \ \text{visits-via} \ P \ S \ W 

  \text{have} \ \text{insert} \ v0 \ S = S \ \text{using} \ \text{step.hyps}(2) \ \text{by} \ \text{blast} 

  \text{hence} *: \ \neg \ \text{visits-via-LCons} \ \text{of} \ \text{ltl} \ P \ S \ W \ v0 \ \text{by} \ \text{auto} 

  \text{have} \ \text{w0} \ \in \ S \cup \ W \ \text{proof} \ \text{(cases)}

  \text{assume} \ v0 \in VV p 

  \text{hence} \ \text{well-ordered-strategy} \ v0 = w0 \ \text{using} \ \text{v0-conforms} \ \text{by} \ \text{blast} 

  \text{hence} \ \text{choose} \ v0 = w0 \ \text{using} \ \text{step.hyps}(2) \ \text{well-ordered-strategy-def} \ \text{by} \ \text{auto} 

  \text{moreover have} \ \text{strategy-} \text{attracts-via} \ p \ \text{(choose} \ v0) \ v0 \ S \ W 

  \text{using} \ \text{choose-good good-def} \ \text{step.hyps}(2) \ \text{by} \ \text{blast} 

  \text{ultimately show} \ \text{thesis} 

  \text{by} \ \text{(metis strategy-} \text{attracts-via} \ \text{successor} \ \text{strategy-} \text{attracts-via-v0} \ \text{strategy-} \text{attracts-via-successor} \ \text{strategy-} \text{attracts-via-v0} \ \text{v0-edge-w0 w0-V}) 

  \text{qed (metis DiD1 asms(2) \text{step.hyps}(2) \text{strategy-} \text{attracts-via-successor} \ \text{strategy-} \text{attracts-via-v0} \ \text{v0-edge-w0 w0-V})} 

  \text{with} * * * \ \text{show} \ \text{?case by} \ \text{blast} 

  \text{qed} 

  \text{have} \ \neg \ \text{finite} \ P \ \text{proof} 

  \text{assume} \ \text{finite} \ P 

  \text{hence} \ \text{deadend} \ (\text{last} \ P) \ \text{using} \ \text{P-maximal} \ \text{P-not-null} \ \text{maximal-ends-on-deadend} \ \text{by} \ \text{blast} 

  \text{moreover have} \ \text{last} \ P \in S - W \ \text{using} \ \text{lset} \ P \subseteq S - W \ \text{P-not-null} \ \text{ifinite} \ P \ \text{finite-lset} 

  \text{by} \ \text{blast} 

  \text{ultimately show} \ \text{False} \ \text{using} \ \text{S-W-no-deadends} \ \text{by} \ \text{blast} 

  \text{qed} 

  \text{obtain} \ n \ \text{where} \ n: \ \text{path-conforms-with-strategy} \ p \ (\text{ldropn n P}) \ \text{(path-strategies} \ P \ \text{§} \ n) 

  \text{using} \ \text{path-eventually-conforms-to-} \sigma \ \text{map-n[OF lset} \ P \subseteq S - W \ \text{P-valid P-conforms]} 

  \text{by} \ \text{blast} 

  \text{define} \ \sigma' \ \text{where} \ \text{simp}: \ \sigma' = \text{path-strategies} \ P \ \text{§} \ n 

  \text{define} \ \text{P'} \ \text{where} \ \text{simp}: \ \text{P'} = \text{ldropn n P} 

53
interpret \text{vme-path} G P' lhd P' p σ'
proof
show \neg \text{null } P' \text{ unfolding } P' \text{-def}
  using \neg \text{finite } P' \text{ finite-lldropn null-imp-finite } by \text{ blast}
qed (simp-all add: n)
have \text{ strategy } p \ σ' \text{ unfolding } σ' \text{-def}
  using path-strategies-strategy \{ \text{ lset } P \subseteq S - W \} \neg\text{finite } P' \text{ infinite-small-length }
  by \text{ blast}
moreover have \text{ strategy-attracts-via } p \ σ' \text{ unfolding } P' \text{-def } S W \text{ proof } -
  have \text{ lset } P' \subseteq S - W \text{ using } \{ \text{ lset-nth-member-inf } \text{ by } \text{ blast } \}
  hence \ σ' \in good (P \ $ n) 
  using path-strategies-good σ' \text{-def } \{ \text{ lset } P \subseteq S - W \} \text{ by blast}
  hence \text{ strategy-attracts-via } p \ σ' (P \ $ n) \text{ unfolding good-def } by \text{ blast}
thus \text{ ?thesis unfolding } P' \text{-def using } P-0 \text{ by (simp add: } \neg\text{finite } P' \text{ infinite-small-length })
qed
moreover from \{ \text{ lset } P \subseteq S - W \} \text{ have } \text{ lset } P' \subseteq S - W
unfolding P' \text{-def using } \text{ lset-lldropn-subset[of n P]} \text{ by blast}
ultimately show False using strategy-attracts-via-lset by \text{ blast}
qed
\}
thus \text{ ?thesis using well-ordered-strategy-valid } by \text{ blast}
qed

10.2 A Uniform Winning Strategy

Let \text{ S be the winning region of player } p. \text{ Then there exists a uniform winning strategy on } S.

lemma merge-winning-strategies:
  shows \exists σ. \text{ strategy } p \ σ \land (\forall v \in \text{ winning-region } p. \text{ winning-strategy } p \ σ \ v)
proof -
define \text{ good } where \text{ good } v = \{ \sigma. \text{ strategy } p \ σ \land \text{ winning-strategy } p \ σ \ v \} \text{ for } v
let \ G = \{ \sigma. \exists v \in \text{ winning-region } p. \ σ \in \text{ good } v \}
obtain r where r: \text{ well-order-on } \ G r using \text{ well-order-on } by \text{ blast}
have \text{ no-VVp-deadends } : \bigwedge v. [ v \in \text{ winning-region } p; v \in VV p ] \implies \neg \text{ deadend } v
  using \text{ no-winning-strategy-on-deadends unfolding winning-region-def } by \text{ blast}
interpret \text{ WellOrderedStrategies } G \text{ winning-region } p p \text{ good } r \text{ proof }
show \bigwedge v. v \in \text{ winning-region } p \implies \exists \sigma. \ σ \in \text{ good } v
  unfolding \text{ good-def winning-region-def } by \text{ blast}
next
show \bigwedge v. σ. \ σ \in \text{ good } v \implies \text{ strategy } p \ σ \text{ unfolding good-def } by \text{ blast}
next
fix v w σ assume v: v \in \text{ winning-region } p \implies w v \in VV p \implies σ v = w \ σ \in \text{ good } v
hence σ: \text{ strategy } p \ σ \text{ winning-strategy } p \ σ \text{ unfolding good-def } by \text{ simp-all}
hence \text{ winning-strategy } p \ σ \ w \text{ proof } (\text{ cases })
  assume *: v \in VV p
  hence **: σ v = w using v(3) \text{ by blast}
  have \neg \text{ deadend } v using \text{ no-VVp-deadends } (v \in VV p) v(1) \text{ by blast}
  with * ** show \text{ ?thesis using strategy-extends-VVp } σ \text{ by blast}
next
assume \ \ v \notin VV p
thus \textit{thesis} using strategy-extends-VVp\textsuperscript{star} \(\sigma \langle v \rightarrow w \rangle\) by blast

\textit{qed}

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

\textit{qed} (insert winning-region-in-Vv)

\{
  \begin{align*}
  \text{fix } v0 & \text{ assume } v0 \in \text{winning-region } p \\
  \text{fix } P & \text{ assume } P: \text{vmc-path } G P v0 p \text{ well-ordered-strategy} \\
  \text{then interpret } & \text{vmc-path } G P v0 p \text{ well-ordered-strategy} .
  \end{align*}
\}

\textit{have} \(\text{lset } P \subseteq \text{winning-region } p\) \textit{proof} (\textit{induct rule: \text{vmc-path-lset-induction-simple})}

\textit{case} (\text{step } P v0)

\textit{interpret} \(\text{vmc-path-no-leaf-end } G P v0 p \text{ well-ordered-strategy using step.hyps(1)}\) .

\{ \text{assume } v0 \in VVp \}

\textit{have} \(\text{well-ordered-strategy } v0 = w0\) using \(v0\)-conforms by blast

\textit{hence} choose \(v0 v0 = w0\) by \(\text{simp add: step.hyps(2) well-ordered-strategy-def}\)

\textit{hence} choose \(v0 \in \text{good } w0\) using strategies-continue choose-good step.hyps(2) by simp

\textit{thus} \textit{case} unfolding good-def winning-region-def using \(w0\)-V by blast

\textit{qed} (insert \(v0 \in \text{winning-region } p\))

\textit{have} \(\text{winning-path } P P\) \textit{proof} (\textit{rule contr})

\textit{assume} contra: \(\neg \text{winning-path } P P\)

\textit{have} \(\neg \text{finite } P\) \textit{proof}

\textit{assume} finite \(P\)

\textit{hence} deadend (last \(P\)) using maximal-ends-on-deadend by simp

\textit{moreover} have \(\text{last } P \in \text{winning-region } P\)

\textit{using} \(\text{lset } P \subseteq \text{winning-region } P\) \(P\)-not-null \(\text{finite } P\) \(\text{finite-lset}\) by blast

\textit{moreover} have \(\text{last } P \in VVp\)

\textit{using} \(\text{contn paths-are-winning-for-one-player } \neg \text{finite } P\)

\textit{unfolding} winning-path-def by simp

\textit{ultimately show} \(\text{False}\) using \(\neg \text{VvP-deadend}\) by blast

\textit{qed}

\textit{obtain} \(n\) where \(n: \text{path-conforms-with-strategy } P \langle \text{idropn } n \ P \rangle\) \(\text{path-strategies } P \not\in \ n\)

\textit{using} \(\text{path-eventually-conforms-to-} \sigma \text{-map-} n\langle \text{OF lset } P \subseteq \text{winning-region } p\rangle \ P\)-valid \(P\)-conforms\)

\textit{by blast}

\textit{define} \(\sigma'\) where \(\text{simp: } \sigma' = \text{path-strategies } P \not\in \ n\)

\textit{define} \(P'\) where \(\text{simp: } P' = \text{idropn } n \ P\)

\textit{interpret} \(P': \text{vmc-path } G P' \thd P' P' \sigma'\) \textit{proof}

\textit{show} \(\neg \text{\text{finite } P'}\) using \(\neg \text{finite } P\) unfolding \(P'\)-def

\textit{using} \(\text{finite-idropn } \text{null-imp-finite}\) by blast

\textit{qed} \(\text{simp-all add: } n\)

\textit{have} strategy \(P \sigma'\) unfolding \(\sigma'\)-def

\textit{using} path-strategies-strategy \(\text{lset } P \subseteq \text{winning-region } p\) \(\neg \text{finite } P\) by blast

\textit{moreover} have \(\text{winning-strategy } P \sigma' \langle \text{idropn } P' \rangle\) \textit{proof}

\textit{have} \(P' \not\in \ n\) \(\text{in } \text{winning-region } p\)

\textit{using} \(\text{lset } P \subseteq \text{winning-region } p\) \(\neg \text{finite } P\) \(\text{lset-nth-member-inf}\) by blast

\textit{hence} \(\sigma' \in \text{good } P' \not\in \ n\)

\textit{using} path-strategies-good choose-good \(\sigma'\)-def \(\neg \text{finite } P\) \(\text{lset } P \subseteq \text{winning-region } p\)

\textit{by blast}

\textit{55}
hence winning-strategy $p \sigma'$ ($P \not\subseteq n$) unfolding good-def by blast
thus ?thesis
  unfolding $P'$-def using P-0 ($\neg$finite $P$) by (simp add: infinite-small-length lhs-ldropn)
qed
ultimately have winning-path $p$ $P'$ unfolding winning-strategy-def
  using $P'$,vmc-path-axioms by blast
moreover have $\neg$finite $P'$ using ($\neg$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-VV$p$ for $VV p**$ nodes, which was still missing.

lemma winning-region-extends-VV$p$star:
  assumes $v$ : $v \in VV p**$ and $w$: $\forall v \cdot v \Rightarrow w \Rightarrow w \in$ winning-region $p$
  shows $v \in$ winning-region $p$
proof -
  obtain $\sigma$ where $\sigma$: strategy $p \sigma$ $\forall v \cdot v \in$ winning-region $p \Rightarrow$ winning-strategy $p \sigma v$
    using merge-winning-strategies by blast
  have winning-strategy $p \sigma v$ using strategy-extends-backwards-VV$p$star[OF $v \sigma(1)$] $\sigma(2)$ $w$ by blast
  thus ?thesis unfolding winning-region-def using $v \sigma(1)$ by blast
qed

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

lemma removing-winning-region-induces-no-deadends:
  assumes $v \in V$ - winning-region $p$ $\neg$deadend $v$
  shows $\exists w \in V$ - winning-region $p$ $v \Rightarrow w$
  using assms winning-region-extends-VV$p$ winning-region-extends-VV$p$star 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-VV$p$:
assumes \( \sigma \): strategy \( p \sigma \) strategy-attracts \( p \sigma S W \)
and \( \forall \sigma_0: \sigma_0 \in Vp \sigma_0 \in \text{directly-attracted} p \sigma S \sigma_0 \notin S \)
shows \( \exists \sigma_0: \sigma_0 \in \text{strategy-attracts-via} p \sigma_0 (\text{insert} \ \sigma_0 S) W \)

proof –
from \( v0(1,2) \) obtain \( w \) where \( v0 \to w \in S \) using \( \text{directly-attracted-def} \) by blast
from \( \sigma_0 \in S \) have \( \text{strategy-attracts-via} p \sigma_0 S \) \( \text{W unfolding} \) \( \text{strategy-attracts-def} \) by blast
let \( \sigma = \sigma_0(v0 := w) \) — Extend \( \sigma \) to the new node.

have \( \text{strategy} p \ ? \sigma \) \( \text{using} \ \sigma(1) \ (v0 \to w) \) \( \text{valid-strategy-updates} \) by blast
moreover have \( \text{strategy-attracts-via} p \ ? \sigma_0 (\text{insert} \ \sigma_0 S) W \) proof

fix \( P \)
assume \( \text{vme-path} G P v0 p \ ? \sigma \)
then interpret \( \text{vme-path} G P v0 p \ ? \sigma \).

have \( \neg \text{deadend} v0 \) using \( (v0 \to w) \) by blast
then interpret \( \text{vme-path-no-deadend} G P v0 p \ ? \sigma \) by unfold-locales

define \( P'' \) where \( \text{[simp]} \): \( P'' = \text{ltl} P \)

have \( \text{ltl} P'' = w \) using \( v0(1) \) \( v0\text{-conforms} \) \( w0\text{-def} \) by auto

hence \( \text{vme-path} G P'' w p \ ? \sigma \) using \( \text{vme-path-ltl} \) by (simp add: \( w0\text{-def} \))

have \( *: v0 \notin S - W \) using \( (v0 \notin S) \) by blast

have \( \text{override-on} (\sigma(v0 := w)) \ (S - W) = ? \sigma \)
by (rule ext) \( \text{metis} \ (\text{fun-upd-def} \ \text{override-on-def}) \)

hence \( \text{strategy-attracts} p \ ? \sigma S W \)
using \( \text{strategy-attracts-irrelevant-override} \ [OF \ \sigma(2,1) \ (\text{strategy} p \ ? \sigma)] \) by simp

hence \( \text{strategy-attracts-via} p \ ? \sigma S W \) \( \text{unfolding} \) \( \text{strategy-attracts-def} \)
using \( (w \in S) \) by blast

hence \( \text{visits-via} P'' S W \) \( \text{unfolding} \) \( \text{strategy-attracts-via-def} \)
using \( \text{vme-path} G P'' w p \ ? \sigma \) by blast
thus \( \text{visits-via} P (\text{insert} \ v0 S) W \)

using \( \text{visits-via-LCons} [\text{of} \ \text{ltl} P S W v0] \) \( \text{P-LCons} \) by simp

qed

ultimately show \( ? \text{thesis} \) by blast

qed

lemma \( \text{strategy-attracts-extends-VVpstar} \):
assumes \( \sigma: \text{strategy-attracts} p \sigma S W \)
and \( \forall \sigma_0: \sigma_0 \notin Vp \sigma_0 \in \text{directly-attracted} p S \)
shows \( \text{strategy-attracts-via} p \sigma_0 (\text{insert} \ v0 S) W \)

proof

fix \( P \)
assume \( \text{vme-path} G P v0 p \sigma \)
then interpret \( \text{vme-path} G P v0 p \sigma \).

have \( \neg \text{deadend} v0 \) using \( v0(2) \) \( \text{directly-attracted-contains-no-deadends} \) by blast
then interpret \( \text{vme-path-no-deadend} G P v0 p \sigma \) by unfold-locales

have \( \text{visits-via} (\text{ltl} P) S W \)
using \( \text{vme-path-strategy-attractsE} [OF \ \text{vme-path-ltl} \ \sigma_0] \ (\text{directly-attracted-def} \) by simp
thus \( \text{visits-via} P (\text{insert} \ v0 S) W \) \( \text{using} \ \text{visits-via-LCons} [\text{of} \ \text{ltl} P S W v0] \) \( \text{P-LCons} \) by simp

qed

lemma \( \text{attractor-has-strategy-single} \):
assumes \( W \subseteq V \)
and \( \forall \sigma_0: \sigma_0 \in \text{attractor} p W \) \( \text{(is -} \in ?A) \)
shows $\exists \sigma. \text{strategy } p \sigma \land \text{strategy-attr acts-via } p \sigma v0 \{ A W$

using asms proof (induct arbitrary; v0 rule: attractor-set-induction)

case (step $S$)
have $v0 \in W \implies \exists \sigma. \text{strategy } p \sigma \land \text{strategy-attr acts-via } p \sigma v0 \{ \}$ $W$
using strategy-attracts-via-trivial valid-arbitrary-strategy by blast

moreover {
assume *: $v0 \in \text{directly-attracted } p S v0 \notin S$
from asms (1) step.hyps (1) step.hyps (2)
have $\exists \sigma. \text{strategy } p \sigma \land \text{strategy-attr acts } p \sigma S W$
using merge-attractor-strategies by auto
with *
have $\exists \sigma. \text{strategy } p \sigma \land \text{strategy-attr acts-via } p \sigma v0 \{ \}$ $W$
using strategy-attracts-extends-VVp strategy-attracts-extends-VVpsstar by blast
}

ultimately show ?case
using step.prems step.hyps (2)
attractor-strategy-on-extends[of $p - v0$ insert $v0 S W W \cup S \cup \text{directly-attracted } p S$]
attractor-strategy-on-extends[of $p - v0 S S W W \cup S \cup \text{directly-attracted } p S$]
attractor-strategy-on-extends[of $p - v0 \{ \} S W W \cup S \cup \text{directly-attracted } p S$]
by blast

next
case (union $M$)
hence $\exists S. S \in M \land v0 \in S$ by blast
thus ?case by (meson Union-upper attractor-strategy-on-extends union.hyps)

qed

11.1 Existence

Prove that every attractor set has an attractor strategy.

theorem attractor-has-strategy:
assumes $W \subseteq V$
shows $\exists \sigma. \text{strategy } p \sigma \land \text{strategy-attr acts } p \sigma (\text{attractor } p W) W$

proof –
let $?A = \text{attractor } p W$

have $?A \subseteq V$ by (simp add: $W \subseteq V$ attractor-in-V)

moreover
have $\forall u. u \in ?A \implies \exists \sigma. \text{strategy } p \sigma \land \text{strategy-attr acts-via } p \sigma u ?A W$
using $W \subseteq V$ attractor-has-strategy-single by blast

ultimately show ?thesis using merge-attractor-strategies $W \subseteq V$ by blast

qed

end — context ParityGame

end

12 Positional Determinacy of Parity Games

theory PositionalDeterminacy

imports
  Main
AttractionStrategy
begin

correct ParityGame begin

12.1 Induction Step

The proof of positional determinacy is by induction over the size of the finite set \( \omega \setminus V \), 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: \( \forall v. v \in V \implies \lnot \text{deadend } v \)
- **and** III: \( \forall (G :: (a, b) \text{ ParityGame-scheme) v.} \)
  - \( \lnot (\text{card (} \omega_G \setminus V_G) < \text{card (} \omega \setminus V; v \in V_G; \text{ ParityGame } G; \) \)
  - \( \forall v. v \in V_G \implies \lnot \text{digraph.deadend } G v \)
  - \( \implies \exists p. v \in \text{ParityGame.winning-region } G p \)
- **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 min-prio = \( \min (\omega \setminus V) \)
**have** \( \exists p. \text{winning-priority } p \min-prio \text{ by auto} \)
**then obtain** \( p \text{ where } p: \text{winning-priority } p \min-prio \text{ by blast} \)

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 \( K = U \cup (\omega \setminus \{ \min-prio \}) \)
**define** \( V' \) where \( V' = U - \text{attractor } p K \)

**define** \( G' \) where \( \text{simp}: G' = \text{subgame } V' \)
**interpret** \( G': \text{ParityGame } G' \text{ using } \text{subgame-ParityGame by simp} \)

**have** \( \forall v. v \in V \implies v \in U \leftrightarrow v \notin \text{winning-region } p^{**} \)
**unfolding** U-def W1-def by blast

**have** \( V' \subseteq V \text{ unfolding } U-def V'-def \text{ by blast} \)
**hence** \( \text{simp}: V_{G'} = V' \text{ unfolding } G'-def \text{ by simp} \)

**have** \( V_{G'} \subseteq V \text{ E } G' \subseteq E \omega_{G'} = \omega \text{ unfolding } G'-def \text{ by (simp-all add: subgame-} \omega) \)
**have** \( G'.VVp = V' \cap VVp \text{ unfolding } G'-def \text{ using } \text{subgame-VV by simp} \)
It turns out the winning region of player $p^{**}$ is empty, so we have a strategy for player $p$.

$$\text{have } v \in G'.\text{-winning-region } p \text{ proof}$$

It follows that: $\forall v \in V, w \in V' \implies \neg G'.\text{-deadend} v \implies G'.\text{-deadend} v$ proof.

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

$$\text{obtain } \sigma W1 \text{ where } \sigma W1:$$

strategy $p^{**} \sigma W1 \forall v, w \in W1 \implies \text{winning-strategy } p^{**} \sigma W1 v$

unfolding $W1$-def using merge-winning-strategies by blast

$$\{ \text{fix } v \text{ assume } v \in V_G' \}$$

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

$$\text{have } G'.\text{-winning-strategy: } \exists p. v \in G'.\text{-winning-region } p \text{ proof}$$

$$\text{have card } (\omega_G \cdot V_G') < \text{card } (\omega \cdot V)$$

$$\{ \text{assume min-prio } \in \omega_G \cdot V_G' \}$$

then obtain $v$ where $v, w \in V_G', \omega_G; v = \text{min-prio}$ by blast

$$\text{hence } v \in \omega \cdot V \cdot \{ \text{min-prio } \text{using } (\omega_G \cdot V_G') = \omega \text{ by simp} \}$$

$$\text{hence } \text{False using } V.-\text{def } K\text{-def attractor-set-base } (V_G' = V' \cdot v(1))$$

by (metis DiffD1 DiffD2 IntI contra-subsetD)

$$\}$$

$$\text{hence min-prio } \not\in \omega_G \cdot V_G'$$

moreover have min-prio $\in \omega \cdot V$

unfolding min-prio-def using priorities-finite Min-in assms(1) by blast

$$\text{moreover have } \omega_G \cdot V_G' \subseteq \omega \cdot V$$

unfolding $G'.\text{-def}$ by simp

ultimately show $?thesis$ by (metis priorities-finite psubset1 psubset-card-mono)

$$\text{qed}$$

thus $?thesis$ using $\exists p. v \in V_G' \implies G'.\text{-no-deadends } G'.\text{ParityGame-axioms}$ by blast

$$\text{qed}$$

It turns out the winning region of player $p^{**}$ is empty, so we have a strategy for player $p$.
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{let } P \subseteq V \rangle \text{ subgame-maximal-path } \langle V' \subseteq V \rangle \text{ 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 } \langle V' \subseteq V \rangle \text{ by simp}

moreover have \(\forall v, v \in \text{let } P \Rightarrow \sigma' v = \sigma v\) using \text{let } P \subseteq V \rangle \sigma'-def by auto

ultimately show \(?\text{thesis}\)

using \(G'.\text{path-conforms-with-strategy-irrelevant-updates}\) by blast

qed

ultimately have \(G'.\text{winning-path } P^{**} P\)

using \(\sigma(2) G'.\text{winning-strategy-def } G'.\text{valid-maximal-conforming-path-0 } P\)-\text{0-not-null}\) by blast

moreover have \(G'.VV p^{**} \subseteq VV p^{**}\) using \text{subgame-VV-subset } G'-def by blast

ultimately show \(\text{False}\)

using \(G'.\text{winning-path-supergame}[\text{of } P^{**}] \omega_{G'} = \omega\)

\((\sim\text{winning-path } P^{**} P) \text{ ParityGame-axioms}\)

by blast

qed

moreover have \(v \in V\) using \(\langle V G' \subseteq V \rangle \langle w \in V_{G'} \rangle\) by blast

ultimately have \(v \in W1\) unfolding \(W1\)-def \text{ winning-region-def}\) by blast

thus \(\text{False}\) using \(\langle v \in V_{G'} \rangle\) \text{ using } U-def \text{ V'}-def \langle V_{G'} = V' \rangle \langle v \in V_{G'} \rangle\) by blast

qed

} note \(\text{recursion = this}\)

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.

obtain \(\sigma1\)

where \(\sigma1\) where strategy \(p \sigma1\)

strategy-attracts \(p \sigma1 \langle \text{attractor } p K) K\)

using \text{attractor-has-strategy}[\text{of } K p]\text{ K-def } U-def \text{ by auto}\n
Next, on \(G'\) we follow the winning strategy whose existence we proved earlier.

have \(G'.\text{winning-region } p = V_{G'}\) using \(\text{recursion}\) unfolding \(G'.\text{winning-region-def}\) by blast

then obtain \(\sigma2\)

where \(\sigma2\) where \(\forall v, v \in V_{G'} \Rightarrow G'.\text{strategy } p \sigma2\)

\(\forall v, v \in V_{G'} \Rightarrow G'.\text{winning-strategy } p \sigma2 v\)

using \(G'.\text{merge-winning-strategies}\) by blast

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

define \(\text{suc}\) where \(\text{suc } v = \langle \text{SOME } w. v \rightarrow w \land (v \in W1 \lor w \notin W1) \rangle \) for \(v\)

Compose the three pieces.

define \(\sigma\) where \(\sigma = \text{override-on } \langle \text{override-on } suc \sigma2 V' \rangle \sigma1 \langle \text{attractor } p K - K)\)

have \text{attractor } p K \cap W1 = \{\} \text{ proof } (\text{rule } \text{contr})
assume \(attractor \ p \ K \cap W1 \neq \{\}\)
then obtain \(v\) where \(v \in attractor \ p \ K \cap W1\) by blast
hence \(v \in V\) using \(W1-def\) \(\text{winning-region-def}\) by blast
obtain \(P\) where \(\text{vmc2-path} \ G \ P \ p \ \sigma I \ \sigma W1\)
using \(\text{strategy-conforming-path-exists} \ \sigma W1(I) \ \sigma I(I) \ v \in V\) by blast
then interpret \(\text{vmc2-path} \ G \ P \ v \ \sigma I \ \sigma W1\).
have \(\text{strategy-attracts-via} \ p \ \sigma I \ v \ (\text{attractor} \ p \ K) \ K\) using \(v(1) \ \sigma I(2) \ \text{strategy-attracts-def}\) by blast
hence \(\neg \ \set P \cap K \neq \\{\\}\) using \(\text{strategy-attracts-viaE}\) \(\text{visits-via-visits}\) by blast
hence \(\neg \ \set P \subseteq W1 \ \text{unfolding} \ K\)-def \(U\)-def by blast
thus \(\text{False}\) unfolding \(W1\)-def using \(\text{comp.path-s-stay-in-winning-region} \ \sigma W1 \ v(2)\) by auto
qed

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

have \(\sigma-\sigma1: \\{w, v \in attractor \ p \ K \ - \ K \implies \sigma v = \sigma I v \ \text{unfolding} \ \sigma\)-def by simp
have \(\sigma-\sigma2: \\{w, v \in V' \implies \sigma v = \sigma 2 v \ \text{unfolding} \ V'-\text{def}\) by auto
have \(\sigma-K: \\{w, v \in K \cup W1 \implies \sigma v = \text{succ} v \ \text{proof}\)
- fix \(v\) assume \(v \in K \cup W1\)
- hence \(v \notin V' \ \text{unfolding} \ V'-\text{def} \ \text{using} \ \text{attractor-set-base}\) by auto
with \(v\) show \(\sigma v = \text{succ} v \ \text{unfolding} \ \sigma\)-def \(U\)-def using \(\text{attractor} p \ K \cap W1 = \{\}\)
by (melis (mono-tags, lifting) Diff-iff IntI UnE override-on-def override-on-emptyset)
qed

Show that \(\text{succ}\) succeeds in avoiding entering \(W1\).

\{
  \text{fix} \ v\ \text{assume} \ v \in VV \ p \\
  \text{hence} \ \neg \ \text{deadend} \ v \ \text{using} \ \text{no-deadends}\) by blast
have \(\exists w, v \rightarrow w \wedge (v \in W1 \ \wedge w \notin W1)\) proof \(\text{cases}\)
  assume \(v \in W1\)
  thus \(\text{thesis}\) using \(\text{no-deadends} \ (\neg \text{deadend} \ v)\) by blast
next
  assume \(v \notin W1\)
show \(\text{thesis}\) proof \(\text{rule case}\)
    assume \(\neg (\exists w, v \rightarrow w \wedge (v \in W1 \ \wedge w \notin W1))\)
    hence \(\exists w, v \rightarrow w \implies \text{winning-strategy} \ p** \ \sigma W1 w\) using \(\sigma W1(2)\) by blast
    hence \(\text{winning-strategy} \ p** \ \sigma W1 v\)
    using \(\text{strategy-extends-backwards} \ VVpstar \ \sigma W1(1)\) \(v \in VV \ p\) by simp
    hence \(v \in W1 \ \text{unfolding} \ W1\)-def \(\text{winning-region-def}\) using \(\sigma W1(1) \ (\neg \text{deadend} \ v)\) by blast
    thus \(\text{False}\) using \(v \notin W1\) by blast
qed
qed
hence \(v \rightarrow \text{succ} v \ v \in W1 \ \vee \ \text{succ} v \notin W1 \ \text{unfolding} \ \text{succ-def}\)
  using \(\text{someI-ex}[\lambda w, v \rightarrow w \wedge (v \in W1 \ \vee w \notin W1)]\) by blast+
\} note \(\text{sucworks} = \text{this}\)

have \(\text{strategy} \ p \ \sigma\)
proof
  fix \(v\) assume \(v \in VV \ p \ \neg \text{deadend} \ v\)
  hence \(v \in attractor \ p \ K \ - \ K \implies v \rightarrow \sigma v\) using \(\sigma-\sigma1 \ \sigma I(1) v\) \(\text{unfolding}\) \(\text{strategy-def}\) by auto
moreover have \(v \in V' \implies v \rightarrow \sigma v\) proof
  assume \(v \in V'\)
moreover have \(v \in VG\) using \(v \in V' \ (VG = V')\) by blast
moreover have \( v \in G',VV p \) using \( (G',VV p = V' \cap VV p) \ (v \in V') \ (v \in VV p) \) by blast
moreover have \( \neg\text{Digraph.deadend } G' \ v \) using \( G'\text{-no-deadends } (v \in V G') \) by blast
ultimately have \( v \rightarrow G',\sigma2 \ v \) using \( \sigma2(1) G'\text{-strategy-def} [\text{of } p \sigma2] \) by blast
with \( (v \in V') \) show \( v \rightarrow \sigma \ v \) using \( (E _{(G')} \subseteq E) \sigma\sigma2 \) by \( \text{metis subsetCE} \)
qed

moreover have \( v \in K \cup W1 \) \( v \rightarrow \sigma \ v \) using \( \text{succ-works}(1) v \sigma-K \) by auto
moreover have \( v \in V \) using \( (v \in VV p) \) by blast
ultimately show \( v \rightarrow \sigma \ v \) using \( \text{V-decomp} \) by blast
qed

have \( \sigma\text{-attracts: } \text{strategy-attracts } p \sigma (\text{attractor } p K) \ K \) proof–
have \( \text{strategy-attracts } p \ (\text{override-on } \sigma \sigma1 (\text{attractor } p K - K)) (\text{attractor } p K) \ K \)
using \( \text{strategy-attracts-irrelevant-override } \sigma1 \ (\text{strategy } p \sigma) \) by blast
moreover have \( \sigma = \text{override-on } \sigma \sigma1 (\text{attractor } p K - K) \)
by \( \text{(rule ext) (simp add: override-on-def } \sigma\sigma1) \)
ultimately show \( \text{?thesis by simp} \)
qed

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

have \( \forall v \in V - W1. \text{ winning-strategy } p \sigma v \) proof \( \text{(intro ball winning-strategy1)} \)
fix \( v \ P \) assume \( P: v \in V - W1 \) \text{vmc-path } G P v p \sigma
interpret \( \text{vmc-path } G P v p \sigma \) using \( P(2) \).

have \( \text{let } P \subseteq V - W1 \)
proof \( \text{(induct rule: vmc-path-let-induction-closed-subset)} \)
fix \( v \text{ assume } v \in V - W1. \neg\text{deadend } v v \in VV p \)
show \( \sigma v \in V - W1 \cup \{} \) proof \( \text{(rule contr)} \)
assume \( \neg\text{?thesis} \)

hence \( \sigma v \in W1 \)
using \( \text{strategy } p \sigma \ (\neg\text{deadend } v (v \in VV p)) \)
unfolding \( \text{strategy-def} \) by blast

hence \( v \notin K \) using \( \text{succ-works}(2)[\text{OF } (v \in VV p)] (v \in V - W1) \sigma-K \) by auto
moreover have \( v \notin \text{attractor } p K - K \)
assume \( v \in \text{attractor } p K - K \)

hence \( \sigma v \in \text{attractor } p K \)

using \( \text{attracted-strategy-step} (\text{strategy } p \sigma) \sigma\text{-attracts} (\neg\text{deadend } v (v \in VV p)) \)
\( \text{attractor-set-base} \)
by blast
thus \( \text{False using } (\sigma v \in W1) \) \( (\text{attractor } p K \cap W1 = \{} \) by blast
qed

moreover have \( v \notin V' \) proof
assume \( v \in V' \)
have \( \sigma2 v \in V G' \) proof \( \text{(rule } G'\text{-valid-strategy-in-V[of } p \sigma2 \ v)) \)

have \( v \in V G' \) using \( \text{\( (V G' = V' \cap VV p) \) by simp} \)
thus \( \neg G'\text{-deadend } v \) using \( G'\text{-no-deadends by blast} \)
show \( G'\text{-strategy } p \sigma2 \) using \( \sigma2(1) (v \in V G') \) by blast

show \( v \in G'VV p \) using \( (v \in VV p) (G'VV p = V' \cap VV p) (v \in V') \) by simp
qed

hence \( \sigma v \in V G' \) using \( (v \in V') \sigma\sigma2 \) by simp
thus \( \text{False using } (V G' = V' (\sigma v \in W1) V'\text{-def U-def by blast} \)

qed
ultimately show \( \text{False using } (v \in V - W1) \) \( \text{V-decomp} \) by blast.
qed

next

fix v w assume v ∈ V − W1 ¬deadend v v ∈ VV p∗∗ v→w

show w ∈ V − W1 ∪ { }

proof (rule contr)

assume ¬thesis

hence w ∈ W1 using (v→w) by blast

let ?σ = σW1(v := w)

have winning-strategy p∗∗ σ w w using (w ∈ W1) σW1 (2) by blast

moreover have ¬(∃σ. strategy p∗∗ σ ∧ winning-strategy p∗∗ σ v)

using (v ∈ V − W1) unfolding W1-def winning-region-def by blast

ultimately have winning-strategy p∗∗ ?σ w

using winning-strategy-updates[wp∗∗ σ W1 w v w] σW1 (1) (v→w)

unfolding winning-region-def by blast

moreover have strategy p∗∗ ?σ using (v→w) σW1 (1) valid-strategy-updates by blast

ultimately have winning-strategy p∗∗ ?σ v

by auto

hence v ∈ W1 unfolding W1-def winning-region-def

using strategy-extends-backwards- VVp [wp∗∗ ?σ w]

(v ∈ VV p∗∗) (v→w)

by auto

hence v ∈ W1 unfolding W1-def winning-region-def

using (strategy p∗∗ ?σ) (v ∈ V − W1) by blast

thus False using (v ∈ V − W1) by blast

qed

qed (insert P (1), simp-all)

This concludes the proof of lset P ⊆ V − WI.

hence lset P ⊆ attractor p K ∪ V' using V-decomp by blast

have ¬finite P

using no-deadends lfinite lset maximal-ends-on-deadend[of P] P-maximal P-not-null lset P-V

by blast

Every σ-conforming path starting in V − WI is winning. We distinguish two cases:

1. P eventually stays in V'. Then P is winning because σ2 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 ∃ n. lset (ldropn n P) ⊆ V'

The first case: P eventually stays in V'.

then obtain n where n: lset (ldropn n P) ⊆ V' by blast

define P' where P' = ldropn n P

hence lset P' ⊆ V' using n by blast

interpret vme-path V' vme-path G' P' lhd P' p σ2 proof

show ¬null P' unfolding P'-def

using ¬finite P' lfinite lset maxnull lset null-imp-finite by blast

show G'.valid-path P' proof–

have valid-path P' unfolding P'-def by simp

thus ?thesis using subgame-valid-path lset P' ⊆ V' G'-def by blast

65
The second case: P

hence path-inf-priorities-def unfolding min-prio

assume thus thesis using subgame-maximal-path \( \{ \text{set } P' \subseteq V' \mid V' \subseteq V \} \) G'-def by blast

qed

show G'.maximal-path P' proof -

have maximal-path P' unfolding P'-def by simp

thus ?thesis using subgame-maximal-path \( \{ \text{set } P' \subseteq V' \mid V' \subseteq V \} \) G'-def by blast

qed

show G'.path-conforms-with-strategy p P' \( \sigma \neq 2 \) proof -

have path-conforms-with-strategy p P' \( \sigma \) unfolding P'-def by simp

hence path-conforms-with-strategy p P' \( \sigma \neq 2 \)

using path-conforms-with-strategy-irrelevant-updates \( \{ \text{set } P' \subseteq V' \mid \sigma \neq 2 \} \)

by blast

thus ?thesis using subgame-path-conforms-with-strategy \( \{ \text{set } P' \subseteq V' \mid V' \subseteq V \} \) G'-def by blast

qed

qed simp

have G'.winning-strategy p \( \sigma \neq 2 \) (ldd P')

using \( \{ \text{set } P' \subseteq V' \} \) vmc-path'.P-not-null \( \sigma \neq 2 \)

by blast

hence G'.winning-path p P' using G'.winning-strategy-def vmc-path'.vmc-path-axioms by blast

moreover have G'.VV \( \forall V' \subseteq V' \subseteq \forall \) unfolding G'-def using subgame-VV by simp

ultimately have winning-path p P' using G'.winning-path-supergame[of p P' G] \( \omega G' = \omega \) ParityGame-axioms by blast

thus ?thesis unfolding P'-def using infinite-small-length[of P p n] P-valid

by blast

next

assume asm: \( \neg (\exists n. \text{ lset } (\text{ldropn } n P) \subseteq V') \)

The second case: P visits K infinitely often. Then \( \text{min-prio} \) occurs infinitely often on P.

have min-prio \( \in \) path-inf-priorities P unfolding path-inf-priorities-def proof (intro Collect1 all1)

fix n

obtain k1 where k1: ldropn n P \( \notin \) k1 \( \notin \) V' using asm by (metis lset-lnth subset1)

define k2 where k2 = k1 + n

interpret vmc-path G ldropn k2 P \( \notin \) k2 \( \notin \) P \( \notin \) k2 \( \notin \) \( \sigma \)

using vmc-path-ldropn infinite-small-length \( \neg \)finite P by blast

have P \( \notin \) k2 \( \notin \) V' unfolding k2-def

using k1 lhd-exists \( \neg \)finite k1 \( \notin \) V' unfolding k2-def

hence P \( \notin \) k2 \( \notin \) \( \text{attractor } p K \) using \( \neg \)finite P \( \notin \) \( \text{set } P \subseteq V - W1 \)

by (metis Diff1 U-def V'-def lset-nth-member-inf)

then obtain k3 where k3: ldropn k2 P \( \notin \) k3 \( \notin \) K

using \( \sigma \)-attracts strategy-attractsE unfolding G'.visits-via-def by blast

define k4 where k4 = k3 + k2

hence P \( \notin \) k4 \( \notin \) K

using k3 lhd-exists \( \neg \)finite k3 \( \notin \) V' unfolding k2-def

moreover have k4 \( \geq \) n unfolding k4-def k2-def

using le-add2 le-bvms by blast

moreover have ldropn n P \( \notin \) k4 \( \notin \) n \( \notin P \) \( \notin \) (k4 \( \notin \) n) \( \notin \) n

using lhd-exists \( \neg \)finite k4 \( \notin \) V' unfolding k2-def

ultimately have ldropn n P \( \notin \) k4 \( \notin \) n \( \notin \) K by simp

66
hence \( \text{iset} \ (\text{lop} \ n \ P) \cap K \neq \{\} \)
using \( \lnot \text{finite} \ P \) by blast
thus \( \text{min-prio} \in \text{iset} \ (\text{lop} \ n \ (\text{imap} \ \omega \ P)) \) unfolding \( K \)-def by auto
qed
thus \( ?\text{thesis} \) unfolding winning-path-def
using path-inf-priorities-at-least-min-prio[of \( P \)-valid, folded \( \text{min-prio} \)-def]
(\( \text{winning-priority} \ \text{p} \ \text{min-prio} \) \( \lnot \text{finite} \ P \) )
by blast
qed

hence \( \forall v \in V. \exists p. \sigma. \text{strategy} \ p \ \sigma \land \text{winning-strategy} \ p \ \sigma \ v \)
unfolding \( W1 \)-def winning-region-def using \( \text{strategy} \ p \ \sigma \) by blast
hence \( \exists p. \sigma. \text{strategy} \ p \ \sigma \land \text{winning-strategy} \ p \ \sigma \ v \)
using \( \sigma \) by blast
thus \( ?\text{thesis} \) unfolding winning-region-def using \( \sigma \in V \) by blast
qed

12.2 Positional Determinacy without Deadends

\textbf{theorem} positional-strategy-exists-without-deadends:
\begin{itemize}
  \item \textbf{assumes} \( v \in V \land v \in V \implies \lnot \text{deadend} \ v \)
  \item \textbf{shows} \( \exists p. \ v \in \text{winning-region} \ p \)
  \item \textbf{using} \( \text{assms ParityGame-axioms} \)
\end{itemize}
by (induct card \( (\omega \ V) \) arbitrary; \( G \ v \) rule: nat-less-induct)
\( \text{(rule ParityGame.\textit{positional-strategy-induction-step}, simp-all)} \)

12.3 Positional Determinacy with Deadends

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

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

Remove the attractor sets of the sets of deadends.

define $V'$ where $V' = V - A \text{Even} - A \text{Odd}$

hence $V' \subseteq V$ by blast

show $\exists\theta$ thesis proof (cases)

assume $v \theta \in V'$

define $G'$ where $G' = \text{subgame } V'$

interpret $G'$: ParityGame $G'$ unfolding $G'$-def using subgame-ParityGame.

have $V'_G = V'$ unfolding $G'$-def using $V' \subseteq V$ by simp

hence $v \theta \in V'_G$ unfolding $v \theta \in V'$ by simp

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

fix $v$ assume $v \in V_G$

moreover have $V' = V - A \text{Even} - A \text{Odd}$ using $V'$-def by simp

ultimately show $\neg G'.\text{deadend } v$

using subgame-without-deadends $(v \in V_G)\$ unfolding $A$-def $G'$-def by blast

qed

ultimately obtain $p \sigma$ where $\sigma: G'.\text{strategy } p \sigma G'.\text{winning-strategy } p \sigma v \theta$

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

unfolding $G'.\text{winning-region-def}$

by blast

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

fix $v$ assume $v \in V'$

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

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

qed

obtain $\sigma$-attr

where $\sigma$-attr: strategy $p \sigma$-attr strategy-attracts $p \sigma$-attr $(A \ p)$ (deadends $p**$)

using attractor-has-strategy[OF deadends-in-$V$] unfolding $A$-def by blast

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

have $\sigma'$-is-on-$V'$: $\forall v. \ v \in V' \implies \sigma' = v = v$

unfolding $V'$-def $\sigma'$-def $A$-def by (cases $p$) simp-all

have strategy $p \sigma'$ proof -

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

unfolding $\sigma'$-def override-on-def by (rule ext) simp

moreover have strategy $p$ (override-on $\sigma$-attr $\sigma V'$)

using valid-strategy-supergame $\sigma$-attr $(I)\sigma(I) V'$-no-deadends $(V'_G = V')$

unfolding $G'$-def by blast

ultimately show $\exists\theta$ thesis by (simp add: valid-strategy-only-in-$V$ $V'$-def override-on-def)

qed

moreover have winning-strategy $p \sigma' v \theta$ proof (rule winning-strategy1, rule contr)

fix $P$ assume vmc-path $G P v \theta p \sigma'$

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

interpret $\text{vmc-path-no-deadend}$ $G P v \theta p \sigma'$

using $V'$-no-deadends $(v \theta \in V')$ by unfold-locales

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 $v \in V' \neg \text{deadend } v v \in VV p$
hence \( v \in G',VV \vdash p \) unfolding \( G'\)-def by (simp add: \( v \in V' \))
mOREover have \( \neg G'\)\,-deadend \( v \) using \( V'\)-no-deadends \( (v \in V') \wedge (V' G' = V') \) by blast
moreover have \( G',\text{strategy} p \sigma' \)
using \( G',\text{valid-strategy-only-in} V \sigma'\)-def \( \sigma'\)-is-\( \sigma\)-on-\( V' \sigma(1) \wedge (V' G' = V') \) by auto
ultimately show \( \sigma' v \in V' \cup A p \) using subgame-strategy-stays-in-subgame
unfolding \( G'\)-def by blast

next
fix \( v w \) assume \( v \in V' \neg\)\,-deadend \( v \in VV p** v \rightarrow w \)
have \( w \notin A p** \) proof
assume \( w \in A p** \)
hence \( v \in A p** \) unfolding \( A\)-def
using \( \langle v \in VV p** \rangle \langle w \rightarrow w \rangle \) attractor-set-\( VVp \) by blast
thus False using \( \langle v \in V' \rangle \) unfolding \( V'\)-def by (cases \( p \)) auto
qed
thus \( w \in V' \cup A p \) unfolding \( V'\)-def using \( \langle w \rightarrow w \rangle \) by (cases \( p \)) auto
next
show \( lset P \cap A p = \{ \} \) proof (rule contr)
assume \( lset P \cap A p \neq \{ \} \)
have strategy-attracts \( p \) (override-on \( \sigma' \) \( \sigma\)-attr \( (A p - \text{deadends} p**))\)

\( \langle A p \rangle \)

\( \langle \text{deadends} p** \rangle \)
using strategy-attracts-irrelevant-override[OF \( \sigma\)-attr \( (2) \langle \sigma\)-attr \( \langle (1) \) strategy \( p \sigma' \)\) by blast
moreover have override-on \( \sigma' \) \( \sigma\)-attr \( (A p - \text{deadends} p**) = \sigma' \)
by (rule ext, unfold \( \sigma'\)-def, cases \( p \) (simp-all add; override-on-def)
ultimately have strategy-attracts \( p \sigma' \) \( A p \) (deadends \( p** \)) by simp
hence \( lset P \cap \text{deadends} p** \neq \{ \} \)
using \( \langle lset P \cap A p \neq \{ \rangle \) : \text{attracted-path}[OF \text{deadends-in-} V] by simp
thus False using contra visits-deadend[of \( p** \)] by simp
qed
qed (insert \( \langle v0 \in V' \rangle \))

then interpret \( \text{vmc-path} G' P v0 p \sigma' \)
unfolding \( G'\)-def using subgame-path-vmc-path[OF \( \langle V' \subseteq V' \rangle \) by blast
have \( G',\text{path-conforms-with-strategy} p P \sigma \) proof
have \( \forall v. v \in lset P \rightarrow \sigma' v = \sigma v \)
using \( \sigma'\)-is-\( \sigma\)-on-\( V' \langle V' G' = V' \rangle \) set-P-V by blast
thus \( G',\text{path-conforms-with-strategy} p P \sigma \)
using \( P\)-conforms \( G',\text{path-conforms-with-strategy-irrelevant-updates} \) by blast
qed
then interpret \( \text{vmc-path} G' P v0 p \sigma \) using conforms-to-another-strategy by blast
have \( G',\text{winning-path} p P \)
using \( \sigma(2) [\text{unfolded} G',\text{winning-strategy-def}] \) \( \text{vmc-path-axioms} \) by blast
from \( \neg \text{winning-path} p P \)
\( G',\text{winning-path-supergame}[\text{OF this ParityGame-axioms, unfolded} G'\)-def] \)
subgame-\( VV\)-subset[of \( p** V' \)]
subgame-\( \omega\)[of \( V' \)]
show False by blast
qed
ultimately show \( ? \text{thesis unfolding} \text{ winning-region-def} \text{ using} \langle v0 \in V' \rangle \) by blast
next
assume \( v0 \notin V' \)
then obtain \( p \) where \( v_0 \in \text{attractor } p \) (\( \text{deadends } p^{**} \))

unfolding \( V' \)-def \( A \)-def using \( (v_0 \in V) \) by blast

thus \(?\text{thesis unfolding } \) winning-region-def

using lemma-path-to-deadend \( (v_0 \in V) \) by blast

qed

qed

12.4 The Main Theorem: Positional Determinacy

Prove the main theorem: The winning regions of player \text{EVEN} and \text{ODD} are a partition of the set of nodes \( V \).

\textbf{theorem partition-into-winning-regions:}

shows \( V = \text{winning-region Even} \cup \text{winning-region Odd} \)

and \( \text{winning-region Even} \cap \text{winning-region Odd} = \{\} \)

\textbf{proof}

show \( V \subseteq \text{winning-region Even} \cup \text{winning-region Odd} \)

by \( (\text{rule subsetI}) \) (metis \( \text{full-types} \) \( \text{Un-i other-other-player positional-strategy-exists} \))

next

show \( \text{winning-region Even} \cup \text{winning-region Odd} \subseteq V \)

by \( (\text{rule subsetI}) \) (meson \( \text{Un-iff subsetCE winning-region-in-V} \))

next

show \( \text{winning-region Even} \cap \text{winning-region Odd} = \{\} \)

using \( \text{winning-strategy-only-for-one-player[of Even]} \)

unfolding \( \text{winning-region-def} \) by auto

qed

end — context ParityGame

end

13 Defining the Attractor with \texttt{inductive_set}

\textbf{theory AttractorInductive}

\textbf{imports}

\( \text{Main} \)

\( \text{Attractor} \)

\textbf{begin}

context ParityGame begin

In section 6 we defined \text{attractor} manually via \texttt{lfp}. We can also define it with \texttt{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 \texttt{attractor-inductive}

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

\texttt{inductive-set attractor-inductive :: Player => 'a set => 'a set for p :: Player and W :: 'a set where}
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** $\text{attractor-}p\ W = \text{attractor}\ p\ V$

**Proof**

- Fix $v$ such that $v \in \text{attractor-}p\ W$

  - Thus $v \in \text{attractor-}p\ W$ [induct rule: attractor-inductive-induct]
    - **Case** (Base $v$) such that $v \in \text{attractor-}p\ W$
      - **Next**
        - **Case** ($\forall v \in V\ p$) such that $v \in \text{attractor-}p\ V$ [induct rule: attractor-set-VVp]
          - **Next**
            - **Case** ($\forall v \in V\ p\star$) such that $v \in \text{attractor-}p\ V\star$
              - **QED**

  - **QED**

**Proof**

- Define $P$ where $P\ S \iff S \subseteq \text{attractor-}p\ W$ for $S$

  - From ($W \subseteq V$) have $P\ (\text{attractor-}p\ W)$ [induct rule: attractor-set-induction]
    - **Case** (Step $S$)
      - Hence $S \subseteq \text{attractor-}p\ W$ using $P\text{-def}$ by simp
      - Have $W \cup S \cup \text{directly-}p\ W \subseteq \text{attractor-}p\ W$ [cases rule: VV-cases]
        - Fix $v$ such that $v \in W \cup S \cup \text{directly-}p\ W$
          - Moreover
            - Assume $v \in W$ hence $v \in \text{attractor-}p\ W$ by blast
          - Moreover
            - Assume $v \in S$ hence $v \in \text{attractor-}p\ W$
              - By (meson ($S \subseteq \text{attractor-}p\ p\ W$) rev-subsetD)
          - Moreover
            - Assume $v \in \text{directly-}p\ S$
              - Hence $v \in V$ using ($S \subseteq V$) attractor-step-boundedy-by-V by blast
              - Hence $v \in \text{attractor-}p\ W$ [cases rule: VV-cases]
                - Assume $v \in VV\ p$
                  - Hence $\exists w, v \rightarrow w \land w \in S$ using $v\text{-attracted directly-}\text{attracted-def}$ by blast
                  - Hence $\exists w, v \rightarrow w \land w \in \text{attractor-}p\ W$
                    - Using ($S \subseteq \text{attractor-}p\ W$) by blast
                      - Thus thesis by (simp add: ($v \in VV\ p$) attractor-inductive.VVp)
                - **Next**
                  - Assume $v \in VV\ p\star$
                    - Hence $\exists w, v \rightarrow w \rightarrow w \in S$ using $v\text{-attracted directly-}\text{attracted-def}$ by blast
                    - Have $\neg\text{deadend } v$ using $v\text{-attracted directly-}\text{attracted-def}$ by blast
                    - Show thesis proof [rule contr]
                      - Assume $v \notin \text{attractor-}p\ W$
                        - Hence $\exists w, v \rightarrow w \land w \notin \text{attractor-}p\ W$
                          - By (metis attractor-inductive.VVpstar ($v \in VV\ p\star$) (\neg\text{deadend } v))
                        - Hence $\exists w, v \rightarrow w \land w \notin S$ using ($S \subseteq \text{attractor-}p\ W$) by (meson subsetCE)
thus False using * by blast
qed
qed
)
ultimately show v ∈ attractor-inductive p W by (meson UnE)
qed
thus P (W ∪ S ∪ directly-attracted p S) using P-def by simp
qed (simp add: P-def Sup-least)
thus ?thesis using P-def by simp
qed
qed
end
end

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-digraph.verts (to-pre-digraph G)
  by (metis * edges-are-in-V(1) pre-digraph.ext-inject pre-digraph.surjective prod.collapse to-pre-digraph-def )

72
show head (to-pre-digraph G) e ∈ pre-digraph.verts (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 nonmulti-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 nonmulti-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-on1 no-multi-arcs)
moreover have $\forall a \in \text{pre-digraph.ares } G$.

$\text{iso-verts iso } (\text{tail } G a) = \text{iso-tail iso } (\text{iso-ares iso } a)$

$\land \text{iso-verts iso } (\text{head } G a) = \text{iso-head iso } (\text{iso-ares iso } a)$

unfolding $\text{iso-def by } (\text{simp add: arc-to-ends-def})$

ultimately have digraph-isomorphism iso
unfolding digraph-isomorphism-def using arc-to-ends-def wf-digraph-axioms by blast

moreover have $\text{to-pre-digraph } (\text{from-pre-digraph } G) = \text{app-iso iso } G$
unfolding to-pre-digraph-def from-pre-digraph-def iso-def app-iso-def by (simp-all add: arcs-ends-def)

ultimately show $\text{thesis unfolding digraph-iso-def by blast}$

qed

end
References


