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 llength ........................................ 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
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Positional Strategies</td>
<td>20</td>
</tr>
<tr>
<td>4.1</td>
<td>Definitions</td>
<td>20</td>
</tr>
<tr>
<td>4.2</td>
<td>Strategy-Conforming Paths</td>
<td>20</td>
</tr>
<tr>
<td>4.3</td>
<td>An Arbitrary Strategy</td>
<td>20</td>
</tr>
<tr>
<td>4.4</td>
<td>Valid Strategies</td>
<td>21</td>
</tr>
<tr>
<td>4.5</td>
<td>Conforming Strategies</td>
<td>22</td>
</tr>
<tr>
<td>4.6</td>
<td>Greedy Conforming Path</td>
<td>25</td>
</tr>
<tr>
<td>4.7</td>
<td>Valid Maximal Conforming Paths</td>
<td>28</td>
</tr>
<tr>
<td>4.8</td>
<td>Valid Maximal Conforming Paths with One Edge</td>
<td>29</td>
</tr>
<tr>
<td>4.9</td>
<td>iset Induction Schemas for Paths</td>
<td>30</td>
</tr>
<tr>
<td>5</td>
<td>Attracting Strategies</td>
<td>31</td>
</tr>
<tr>
<td>5.1</td>
<td>Paths Visiting a Set</td>
<td>31</td>
</tr>
<tr>
<td>5.2</td>
<td>Attracting Strategy from a Single Node</td>
<td>32</td>
</tr>
<tr>
<td>5.3</td>
<td>Attracting strategy from a set of nodes</td>
<td>35</td>
</tr>
<tr>
<td>6</td>
<td>Attractor Sets</td>
<td>38</td>
</tr>
<tr>
<td>6.1</td>
<td>directly-attracted</td>
<td>38</td>
</tr>
<tr>
<td>6.2</td>
<td>attractor-step</td>
<td>38</td>
</tr>
<tr>
<td>6.3</td>
<td>Basic Properties of an Attractor</td>
<td>39</td>
</tr>
<tr>
<td>6.4</td>
<td>Attractor Set Extensions</td>
<td>39</td>
</tr>
<tr>
<td>6.5</td>
<td>Removing an Attractor</td>
<td>39</td>
</tr>
<tr>
<td>6.6</td>
<td>Attractor Set Induction</td>
<td>40</td>
</tr>
<tr>
<td>7</td>
<td>Winning Strategies</td>
<td>41</td>
</tr>
<tr>
<td>7.1</td>
<td>Deadends</td>
<td>42</td>
</tr>
<tr>
<td>7.2</td>
<td>Extension Theorems</td>
<td>43</td>
</tr>
<tr>
<td>8</td>
<td>Well-Ordered Strategy</td>
<td>44</td>
</tr>
<tr>
<td>8.1</td>
<td>Strategies on a Path</td>
<td>46</td>
</tr>
<tr>
<td>8.2</td>
<td>Eventually One Strategy</td>
<td>49</td>
</tr>
<tr>
<td>9</td>
<td>Winning Regions</td>
<td>49</td>
</tr>
<tr>
<td>9.1</td>
<td>Paths in Winning Regions</td>
<td>50</td>
</tr>
<tr>
<td>9.2</td>
<td>Irrelevant Updates</td>
<td>51</td>
</tr>
<tr>
<td>9.3</td>
<td>Extending Winning Regions</td>
<td>51</td>
</tr>
<tr>
<td>10</td>
<td>Uniform Strategies</td>
<td>52</td>
</tr>
<tr>
<td>10.1</td>
<td>A Uniform Attractor Strategy</td>
<td>52</td>
</tr>
<tr>
<td>10.2</td>
<td>A Uniform Winning Strategy</td>
<td>54</td>
</tr>
<tr>
<td>10.3</td>
<td>Extending Winning Regions</td>
<td>56</td>
</tr>
<tr>
<td>11</td>
<td>Attractor Strategies</td>
<td>56</td>
</tr>
<tr>
<td>11.1</td>
<td>Existence</td>
<td>58</td>
</tr>
<tr>
<td>12</td>
<td>Positional Determinacy of Parity Games</td>
<td>58</td>
</tr>
<tr>
<td>12.1</td>
<td>Induction Step</td>
<td>59</td>
</tr>
</tbody>
</table>
1 Introduction

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

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

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

1.1 Formal Introduction

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

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

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

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

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

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

1.2 Overview

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

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

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

The main theorem is in section 12.4.
1.3 Technical Aspects

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

We also explored representing paths as functions nat ⇒ 'a option with the property that the domain is an initial segment of nat (and where 'a is the node type). However, it turned out that coinductive lists give simpler proofs.

It is possible to represent a graph as a function 'a ⇒ 'a ⇒ 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 UNIV :: 'a set but rather a subset of UNIV :: 'a set, it becomes easier to reason about subgraphs.

Another point is that we make extensive use of locales, in particular to represent maximal paths conforming to a specific strategy. Thus proofs often start with interpret vmc-path G P v0 p σ to say that P is a valid maximal path in the graph G starting in v0 and conforming to the strategy σ for player p.

2 Auxiliary Lemmas for Coinductive Lists

Some lemmas to allow better reasoning with coinductive lists.

theory MoreCoinductiveList
imports Main Coinductive.List begin

2.1 lset

lemma lset-lnth: x ∈ lset xs ⇒ ∃ n. Inth xs n = x
by (induct rule: llist.set-induct, meson lnth-0, meson lnth-Suc-LCons)

lemma lset-lnth-member: [ lset xs ⊆ A; enat n < llength xs ] ⇒ lnth xs n ∈ A
using contra-subsetD[ of lset xs A ] in-lset-cov-lnth[of - xs] by blast

lemma lset-lnth-member-inf: [ ¬finite xs; lset xs ⊆ A ] ⇒ lnth xs n ∈ A
by (metis contra-subsetD inf-lset-lnth lset-inf-llist memel)

lemma lset-intersect-lnth: lset xs ∩ A ≠ {} ⇒ ∃ n. enat n < llength xs ∧ lnth xs n ∈ A
by (metis disjoint-if-not-equal in-lset-cov-lnth)

lemma lset-ltake-Suc:
assumes ¬null xs lnth xs 0 = x lset (ltake (enat n) (ltl xs)) ⊆ A
shows lset (ltake (enat (Suc n)) xs) ⊆ insert x A
proof –
have lset (ltake (enat n)) (LCons x (ltl xs)) ⊆ insert x A
using assms(3) by auto
moreover from assms(1,2) have LCons x (ltl xs) = xs
by (metis lnth-0 ltl-simps(2) not-null-lset)

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

qed

lemma lfinite-lset: \(\text{lfinite } x s \implies \neg\text{null } x s \implies \text{llast } x s \in \text{lset } x s\)

proof (induct rule: lfinite-induct)

case \((\text{LCons } x s)\)

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

assume \(\ast\): \(\neg\text{null } (\text{Lt } x s)\)

hence \(\text{llast } (\text{Lt } x s) \in \text{lset } (\text{Lt } x s)\) using \(\text{LCons.hyps(3)}\) by blast

hence \(\text{llast } (\text{Lt } x s) \in \text{lset } x s\) by \((\text{simp add: in-lset-LHD})\)

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

qed \((\text{metis LCons.prems lhd-LCons-ltl llast-LCons list.set-set-set(1)})\)

qed simp

lemma lset-subset: \(\neg(\text{lset } x s \subseteq A) \implies \exists n. \text{enat } n < \text{llength } x s \land \text{lnth } x s n \notin A\)

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

2.2 llength

lemma enat-Suc-ltl:

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

shows \(\text{enat } n < \text{llength } (\text{Lt } x s)\)

proof -

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

hence \(\text{enat } n < \text{epr e d } (\text{llength } x s)\) using \(\text{eSuc-le-ilI1}\) by fastforce

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

qed

lemma enat-ltl-Suc:

\(\text{enat } n < \text{llength } (\text{Lt } x s)\) = \(\text{enat } (\text{Suc } n) < \text{llength } x s\)

by \((\text{metis eSuc-enat ldr op-ltl leD leI lnul l-ldr op})\)

lemma innite-small-llength [intro]: \(\neg\text{lnite } x s\) = \(\text{enat } n < \text{llength } x s\)

using \(\text{enat-iless lnite-c onv-llength-enat ne q-i}\) by blast

lemma lnul l-0-llength:

\(\neg\text{lnul l } x s\) = \(\text{enat } 0 < \text{llength } x s\)

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

lemma Suc-llength:

\(\text{enat } (\text{Suc } n) < \text{llength } x s\) = \(\text{enat } n < \text{llength } x s\)

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

2.3 ltake

lemma ltake-lnth: \(\text{ltake } n x s = \text{ltake } n y s\) = \(\text{enat } m < n \implies \text{lnth } x s m = \text{lnth } y s m\)

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

lemma lset-ltake-prefix [simp]: \(n \leq m\) = \(\text{lset } (\text{ltake } n x s) \subseteq \text{lset } (\text{ltake } m x s)\)

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

lemma lset-ltake: \(\text{ltake } m. m < n \implies \text{lnth } x s m \in A\) = \(\text{lset } (\text{ltake } (\text{enat } n) x s) \subseteq A\)

proof (induct n arbitrary: \(x s\))

case \(0\)

have \(\text{ltake } (\text{enat } 0) x s \in \text{LNil}\) by \((\text{simp add: zem-enat-def})\)
thus \( \text{?case by simp} \)

next

case (Suc \( n \))
show \( \text{?case proof (cases)} \)
assume \( \text{xs} \neq \text{LNil} \)
then obtain \( \text{xs}' \) where \( \text{xs} = \text{LCons} \ \text{xs}' \) by (meson neq-LNil-conv)
{ fix \( m \) assume \( m < n \)
  hence Suc \( m < \) Suc \( n \) by simp
  hence \( \text{lnth xs} \) (Suc \( m \)) \( \in \text{A} \) using Suc.prems by presburger
  hence \( \text{lnth xs}' \) \( m \) \( \in \text{A} \) using \( \text{xs} \) by simp
}

hence lset \( (\text{ltake (enat } n)) \text{xs}') \subseteq \text{A} \) using Suc.prems \( \text{xs} \) by force
ultimately show \( \text{?thesis by simp} \)
qed simp

qed

lemma llength-ltake': \( \text{enat} \ n < \text{llength} \ \text{xs} \implies \text{llength} (\text{ltake (enat } n)) \text{xs}) = \text{enat} \ n \)
by (metis llength-ltake min.strict-order-iff)

lemma llast-ltake:
assumes \( \text{enat} \ (\text{Suc } n) < \text{llength} \ \text{xs} \)
shows \( \text{llast} (\text{ltake (enat } (\text{Suc } n))) \text{xs}) = \text{lnth} \ \text{xs} \ n \)
\( \text{is llast } \ ?A = \cdot \)
unfolding llast-def using llength-ltake' [OF assms] by (auto simp add: lnth-ltake)

lemma lset-ltake-ltl:
\( \text{lset} (\text{ltake (enat } n)) (\text{ltl} \ \text{xs})) \subseteq \text{lset} (\text{ltake (enat } (\text{Suc } n))) \text{xs}) \)
proof (cases)
assume \( \neg\text{lnull} \ \text{xs} \)
then obtain \( v0 \) where \( \text{xs} = \text{LCons} \ v0 (\text{ltl} \ \text{xs})) \)
by (metis lhd-LCons-ltl)

hence \( \text{ltake (eSuc (enat } n))) \text{xs}) = \text{LCons} \ v0 (\text{ltake (enat } n)) (\text{ltl} \ \text{xs})) \)
by (metis ltake-eSuc-LCons)

hence lset (\text{ltake (enat } (\text{Suc } n))) \text{xs}) = lset (\text{LCons} \ v0 (\text{ltake (enat } n)) (\text{ltl} \ \text{xs}))) \)
by (simp add: eSuc-enat)
thus \( \text{?thesis using lset-LCons[of v0 ltake (enat } n)) (\text{ltl} \ \text{xs})] \)
by blast
qed \( \text{simp add: lnull-def} \)

2.4 ldropn

lemma ltl-ldrop:
\[ (\forall \text{xs}. \text{P} \ \text{xs} \implies \text{P} (\text{ltl} \ \text{xs}); \text{P} \ \text{xs}) \implies \text{P} (\text{ldropn } n \ \text{xs}) \]
unfolding ldropn-def by \( \text{induct } n \) simp-all

2.5 lfinite

lemma lfinite-drop-set:
\( \text{lfinite} \ \text{xs} \implies \exists \text{n}. \ v \notin \text{lset} (\text{ldropn } n \ \text{xs}) \)
by (metis ldrop-inf-lmember-excl1 lset-lmember)

lemma index-infinite-set:
\[ (\neg\text{lfinite } x; \text{lnth} \ x \ m = y; \forall i. \text{lnth} \ x \ i = y \implies (\exists \text{m} > i. \text{lnth} \ x \ m = y)) \implies y \in \text{lset} (\text{ldropn } n \ x) \]
proof \( \text{induct } n \) arbitrary: \( x \ \text{m} \)

\[7\]
case 0 thus \textit{case using \texttt{iset-nth-member-inf} by \texttt{auto}}
next
case (Suc n)
obtain \texttt{a xs where} x = \texttt{LCons a xs} by \texttt{(meson Suc.prems(1) lnull-imp-finite not-lnull-conv)}
obtain j \texttt{where} j > m \texttt{lnth x j = y using Suc.prems(2,3) by blast}
have \texttt{lnth xs (j - 1) = y by (metis lnth-LCons' j(1,2) not-less0 x)}
moreover {
  \texttt{fix i assume \texttt{lnth xs i = y}}
  \texttt{hence \texttt{lnth x (Suc i) = y by (simp add: x)}}
  \texttt{hence \exists j > i. \texttt{lnth xs j = y by (metis Suc.prems(3) Suc-lessE lnth-Suc-LCons x)}}
}
ultimately show \textit{?case using Suc.hyps Suc.prems(1) x by \texttt{auto}}
qed

2.6 \texttt{lmap}

\textbf{lemma} \texttt{lnth-lmap-ldropn:}
\begin{align*}
\texttt{enat n < llength xs} & \Rightarrow \texttt{lnth (lmap f (ldropn n xs)) 0 = lnth (lmap f xs) n} \\
\texttt{by (simp add: lhd-ldropn lnth-0-conv-lhd)}
\end{align*}

\textbf{lemma} \texttt{lnth-lmap-ldropn-Suc:}
\begin{align*}
\texttt{enat (Suc n) < llength xs} & \Rightarrow \texttt{lnth (lmap f (ldropn n xs)) (Suc 0) = lnth (lmap f xs) (Suc n)} \\
\texttt{by (metis \texttt{(no-types, lifting) Suc-length ldropn-ll leD list.map-disc iff lnth-lmap-ldropn lnth-ll lnnull ldropn ll-ldropn \ll-lmap})}
\end{align*}

2.7 \textbf{Notation}

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

\textbf{notation} \texttt{lnth (infix $\texttt{$}$ 61)}

end

3 Parity Games

\textbf{theory} \textit{ParityGame}
\textbf{imports} Main MoreCoinductiveList
\textbf{begin}

3.1 Basic definitions

\texttt{a} is the node type. Edges are pairs of nodes.

\textbf{type-synonym} \texttt{a Edge = a \times a}

A path is a possibly infinite list of nodes.

\textbf{type-synonym} \texttt{a Path = a list}
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_1)
  arcs :: 'a Edge set (E_1)
abbreviation is-arc :: ('a, 'b) Graph-scheme ⇒ 'a ⇒ 'a ⇒ bool (infix !→ 60) where
  v !→ G w ≡ (v,w) ∈ E G
locale Digraph =
  fixes G (structure)
  assumes valid-edge-set: E ⊆ V × V
begin
lemma edges-are-in-V [intro]: v!→w v ∈ V v!→w w ∈ V using valid-edge-set by blast+
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; ¬hlull 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 Ps ⇒ valid-path (ltl Ps)
  by (metis l list.exhaust-sel ltl-simps (1) valid-path-ltl')
lemma valid-path-drop: valid-path Ps ⇒ valid-path (ldropn n Ps)
  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: llist.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 $P' = \text{ldropn} n P$

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

hence $P' \$ 0 = P \$ n by (simp add: $P'$-def)

moreover have $P' \$ \text{Suc} \ 0 = P \$ \text{Suc} n$

by (metis One-nat-def $P'$-def $P'$-eq-plus1 \text{add-commute} \text{assms}(2) \text{ldropn-Suc-conv-ldropn})

ultimately have $\exists Ps. \ P' = \text{LCons} (P \$ n) (\text{LCons} (P \$ \text{Suc} n) Ps)$

by (metis $P'$-def \text{enat} \text{n} < \text{lenght} P; \text{assms}(2) \text{ldropn-Suc-conv-ldropn})

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

ultimately show \( \ \text{thesis using} \ \text{valid-path-edges}' \) by blast

qed


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

assumes major: $Q \ P$

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

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

shows valid-path $P$

using major proof (coinduction arbitrary: $P$)

case valid-path

\{
\begin{itemize}
\item assume $P \neq \text{LNil} \not\exists v. \ P = \text{LCons} v \ \text{LNil} \land v \in V$
\item then obtain $v \ w \ P'$ where $P = \text{LCons} v (\text{LCons} w P')$
\item using \text{neg-LNil-conv base valid-path} by metis
\end{itemize}

hence case using step valid-path by auto
\}

thus case by blast

qed


lemma valid-path-no-deadends:

\[ \text{valid-path} P; \ \text{enat} \ (\text{Suc} \ i) < \text{lenght} P \] \rightarrow \lnot \text{deadend} (P \$ i)

using valid-path-edges by blast


lemma valid-path-ends-on-deadend:

\[ \text{valid-path} P; \ \text{enat} i < \text{lenght} P; \ \text{deadend} (P \$ i) \] \rightarrow \text{enat} (\text{Suc} i) = \text{lenght} P

using valid-path-no-dead-ends by (metis \text{enat-less} \text{enat-ord-simps}(2) \text{neg-iff} \text{not-less-eq})


lemma valid-path-prefix: \[ \text{valid-path} P; \ \text{lprefix} P' P \] \rightarrow \text{valid-path} $P'$

proof (coinduction arbitrary: $P'$ $P$)

case (step $v \ w \ P'' P' P$)

then obtain $Ps$ where $Ps: \ \text{LCons} v (\text{LCons} w Ps) = P$ 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 $P'' P''$) (LCons w Ps) using Ps step(1,3) by auto

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

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


lemma valid-path-lappend:

assumes valid-path $P$ valid-path $P' \ [ \not\text{lnull} P; \ \not\text{lnull} P' ] \rightarrow \text{lappd} P'

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

proof (cases, cases)

assume \( \not\text{lnull} P \ \not\text{lnull} P' \)

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

\{ case $v \ w \ P'' P' P$

\show case proof (cases)

\}
A valid path is still valid in a supergame.

**3.4 Maximal Paths**

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

**coinductive maximal-path where**

- maximal-path-base : maximal-path LNil
- maximal-path-base' : deadend v \implies maximal-path (LCons v LNil)
- maximal-path-cons : \neg lnull P' \implies maximal-path Ps \implies maximal-path (LCons v Ps)

**lemma maximal-no-deadend :** maximal-path (LCons v Ps) \implies \neg deadend v \implies \neg lnull Ps

**lemma maximal-nil :** maximal-path P \implies maximal-path (lnil P)

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

**lemma maximal-lappend :**

- assumes \neg lnull P', maximal-path P'
- shows maximal-path (lappend P P')

**proof (cases)**

- assumes \neg lnull P
- thus ?thesis using assms proof (coinduction arbitrary: P' P rule: maximal-path.coinduct)
- case (maximal-path P' P')
  - let ?P = lappend P P'
  - show ?case proof (cases ?P = LNil v (\exists v. ?P = LCons v LNil v deadend v))
    - case False
      - then obtain Ps v where P: ?P = LCons v Ps by (meson neg-LNil-conv)
      - hence Ps = lappend (lnil P) P' by (simp add: lappend-lnil maximal-path(1))
hence \(\exists P s_1 P'. Ps = lappend P s_1 P' \land \neg \text{null } P' \land \text{maximal-path } P'\)
using \text{maximal-path}(2) \text{maximal-path}(3) by auto
thus \(?thesis \text{ using } P\ lappend-\text{null} \text{ by fastforce}
qed blast
qed

\text{lemma maximal-ends-on-deadend:}
\text{ assumes maximal-path } P \ \text{finite } P \ \neg \text{null } P
\text{ shows deadend } (llast P)
\text{ proof – }
\text{ from } (\text{finite } P) (-\text{null } P) \text{ obtain } n \text{ where } n: \text{length } P = \text{enat } (\text{Suc } n)
\text{ by } (\text{metis enat-or-simps(2) gr0-implies-Suc finite-length-enat null-0-length})
\text{ define } P' \text{ where } P' = \text{ldropn } n P
\text{ hence maximal-path } P' \text{ using } \text{assms(1)} \text{ maximal-drop by blast}
thus \(?thesis \text{ proof (cases rule: maximal-path.cases)}
\text{ case } (\text{maximal-path-base' } v)
\text{ hence deadend } (llast P') \text{ unfolding } P'^{\text{def by simp}}
\text{ thus } ?thesis \text{ unfolding } P'^{\text{def by llast-ldropn[of } n P] } n
\text{ by } (\text{metis } P'^{\text{def ldropn-eq-LCons local.maximal-path-base'(1)})
next
\text{ case } (\text{maximal-path-cons } P' \ v)
\text{ hence ldropn } (\text{Suc } n) \ P = P'' \text{ unfolding } P'^{\text{def by (metis ldrop-eSuc Lil-ldropn Lil-simps(2))}}
\text{ thus } ?thesis \text{ using } n \text{ maximal-path-cons}(2) \text{ by auto}
\text{ qed (simp add: } P'^{\text{def n ldropn-eq-LNil)}
\text{ qed}

\text{lemma maximal-ends-on-deadend': [ \text{finite } P; \text{deadend } (llast P) ] \implies maximal-path } P
\text{ proof (coinduction arbitrary: } P \text{ rule: maximal-path.coinduct)}
\text{ case } (\text{maximal-path } P)
\text{ show } ?\text{case proof (cases)}
\text{ assume } P \neq LNil
\text{ then obtain } v P' \text{ where } P' = LCons v P' \text{ by } (\text{meson neq-LNil-conv})
\text{ show } ?\text{thesis proof (cases)}
\text{ assume } P' = LNil \text{ thus } ?\text{thesis using } P' \text{ maximal-path}(2) \text{ by auto}
\text{ qed (metis } P' \text{ finite-LCons llast-LCons llist.collapse(1) maximal-path(I,2))
\text{ qed simp
\text{ qed}

\text{lemma infinite-path-is-maximal: [ } \text{valid-path } P; \neg \text{finite } P ] \implies maximal-path } P
\text{ by (coinduction arbitrary: } P \text{ rule: maximal-path.coinduct)}
\text{ (cases rule: valid-path.cases, auto)}
\text{ qed

end — locale Digraph

3.5 Parity Games

Parity games are games played by two players, called \text{Even} and \text{Odd}.

\text{datatype } \text{Player } = \text{Even } | \text{Odd}

\text{abbreviation other-player } p \equiv (\text{if } p = \text{Even} \text{ 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 \(V \to \mathbb{N}\) with finite image.

Record `ParityGame = 'a Graph +
player0 :: 'a set (V0)
priority :: 'a ⇒ nat (ω)
` ParityGame = Digraph \(G\) for \(G :: ('a,'b) ParityGame-scheme (structure) +
assumes valid-player0-set: V0 ⊆ V
and priorities-finite: finite (ω ' V)
` ParityGame = Digraph \(G\)

3.6 Sets of Deadends

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

3.7 Subgames

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

**Definition**
\[\text{subgame where}
\text{subgame } V' \equiv \]
\[\text{verts} := V \cap V',
\text{ars} := E \cap (V' × V'),
\text{player0} := V0 \cap V'
\]

**Lemma**
\[\text{subgame-}V [simp]: V_{\text{subgame } V'} \subseteq V\]
and \[\text{subgame-}E [simp]: E_{\text{subgame } V'} \subseteq E\]
and \[\text{subgame-}ω: \omega_{\text{subgame } V'} = ω\]
unfolding \text{subgame-def by simp-all}\n
**Lemma**
\[\text{assumes } V' \subseteq V\]
\[\text{shows }\text{subgame-}V'[simp]: V_{\text{subgame } V'} = V'
\text{and }\text{subgame-}E'[simp]: E_{\text{subgame } V'} = E \cap (V_{\text{subgame } V'} × V_{\text{subgame } V'})\]
unfolding \text{subgame-def using asms by auto}\n
**Lemma**
\[\text{subgame-}VV [simp]: \text{ParityGame}.VV (\text{subgame } V') p = V' \cap VV p\]
proof—
have \[\text{ParityGame}.VV (\text{subgame } V') \text{Even } = V' \cap VV \text{Even}\]
unfolding \text{subgame-def by auto}\nmoreover have \[\text{ParityGame}.VV (\text{subgame } V') \text{Odd } = V' \cap VV \text{Odd}\] proof—
have $V' \cap V = (V \cap V') = V' \cap V \cap (V - V')$ by blast
thus ?thesis unfolding subgame-def by auto
qed
ultimately show ?thesis by simp
qed

corollary subgame-VV-subset [simp]: ParityGame.VV (subgame $V'$) $p \subseteq VV p$ by simp

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

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

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

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

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

lemma subgame-maximal-path:
  assumes $V': V' \subseteq V$ and P: maximal-path P lset P $\subseteq V'$
shows Digraph.maximal-path (subgame \( V' \)) \( P \)

proof –

have \( \text{lset } P \subseteq \text{subgame } V' \), unfolding subgame-def using \( P(2) \) \( V' \), by auto

with \( P(1) \) \( V' \), show \( \text{?thesis} \)

by (induction arbitrary: \( P \) rule: Digraph.maximal-path.coinduct[OF subgame-Digraph])

cases rule: maximal-path.cases, auto

qed

3.8 Priorities Occurring Infinitely Often

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

definition path-inf-priorities :: \( \text{\`a Path } \Rightarrow \text{nat set} \)

path-inf-priorities \( P \) \( \equiv \{ k . \forall n . k \in \text{lset} (\text{ldropn n } (\text{lmap } \omega P)) \} \)

Because \( \omega \) 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 \( P \), \( \neg \text{finite } P \)

shows \( \exists k . k \in \text{path-inf-priorities } P \)

proof –

Define a map from indices to priorities on the path.

define \( f \) where \( f \, i = \omega \, (P \, \$ \, i) \) for \( i \)

have \( \text{range } f \subseteq \omega \, ' \, V \) 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 \( n0 \): \( \neg \text{(finite } \{ n . f \, n = f \, n0 \}) \)

using pigeonhole-infinite[of UNIV \( f \)] by auto

define \( k \) where \( k = f \, n0 \)

The priority \( k \) occurs infinitely often.

have \( \text{lmap } \omega \, P \, \$ \, n0 = k \) unfolding f-def k-def

using assms(2) by (simp add: infinite-small-length)

moreover {

fix \( n \) assume \( \text{lmap } \omega \, P \, \$ \, n = k \)

have \( \exists n' > n . f \, n' = k \) unfolding k-def using \( n0 \) infinite-nat-iff-unbounded by auto

hence \( \exists n' > n . \text{lmap } \omega \, P \, \$ \, n' = k \) unfolding f-def

using assms(2) by (simp add: infinite-small-length)

}

ultimately have \( \forall n . k \in \text{lset} (\text{ldropn n } (\text{lmap } \omega P)) \)

using index-infinite-set[of \( \text{lmap } \omega P \, n0 \, k \) \( P(2) \)] finite-lmap

by blast

thus \( \text{?thesis} \) unfolding path-inf-priorities-def by blast

qed

lemma path-inf-priorities-at-least-min-prio:

assumes \( P \): valid-path \( P \) and \( a : a \in \text{path-inf-priorities } P \)
shows $\min (\omega \cdot V) \leq a$

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

qed

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

proof
- show $?A \subseteq ?B$ proof
  - fix a assume $a \in ?A$
  - hence $\forall n. a \in \text{set } (\text{ldropn } n \times (\text{lmap } \omega \text{ (LCons } v \text{ P))))$
  - unfolding path-inf-priorities-def
  - using in-set-llD[of a] by (simp add: ltl-ldropn)
  - 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 $\forall n. a \in \text{set } (\text{ldropn } (\text{Suc } n) \times (\text{lmap } \omega \text{ (LCons } v \text{ P))))$
  - unfolding path-inf-priorities-def by blast
  - thus $a \in ?A$ unfolding path-inf-priorities-def by simp

qed

qed

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

3.9 Winning Condition

Let $G = (V, E, V_0, \omega)$ be a parity game. An infinite path $v_0, v_1, \ldots$ in $G$ is winning for player 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 $p \equiv (\text{if } p = \text{Even then even else odd})$

definition winning-path :: Player $\Rightarrow \'a \text{ Path } \Rightarrow \text{bool}$ where
winning-path $p$ $P$ $\equiv$
  $\langle \lnot \text{finite } P \land (\exists a \in \text{path-inf-priorities } P.
  (\forall b \in \text{path-inf-priorities } P. a \leq b) \land \text{winning-priority } p \ a) \rangle$
  $\lor (\lnot \text{null } P \land \text{finite } P \land \text{llast } P \in \text{VV } p^{**})$
  $\lor (\text{null } P \land p = \text{Even})$

Every path has a unique winner.

lemma paths-are-winning-for-one-player:
assumes valid-path $P$
shows winning-path $p$ $P$ $\iff$ winning-path $p^{**}$ $P$
proof (cases)
- assume $\lnot \text{null } P$

16
show \( ?\)thesis proof (cases)

\begin{itemize}
  \item assume \( \text{finite} \ P \)
  \item thus \( ?\)thesis
    \begin{itemize}
      \item using \( \text{assms} \ \text{finite-set} \ \text{valid-path-in-V} \)
      \item unfolding \( \text{winning-path-def} \)
        \begin{itemize}
          \item by auto
        \end{itemize}
    \end{itemize}
\end{itemize}

next

\begin{itemize}
  \item assume \( \neg \text{finite} \ P \)
  \item then obtain \( a \) where \( a \in \text{path-inf-priorities} \ P \land b < a \implies b \notin \text{path-inf-priorities} \ P \)
    \begin{itemize}
      \item using \( \text{assms} \ \text{ex-least-nat-le}\,[\lambda a. \ a \in \text{path-inf-priorities} \ P] \ \text{path-inf-priorities-is-nonempty} \)
        \begin{itemize}
          \item by \( \text{blast} \)
        \end{itemize}
    \end{itemize}
  \end{itemize}

hence \( \forall q. \ \text{winning-priority} \ q \ a \iff \text{winning-path} \ q \ P \)

unfolding \( \text{winning-path-def} \) using \( \neg \text{null} \ P \) \( \neg \text{finite} \ P \), by \( \text{metis le-antisym not-le} \)

moreover have \( \forall q. \ \text{winning-priority} \ q \ P \iff \neg \text{winning-priority} \ q \ast \ P \) by \( \text{simp} \)

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

qed

\begin{itemize}
  \item \text{corollary} \ \text{winning-path-drop}: \ \text{assumes} \ \text{winning-path} \ p \ \text{P} \ \neg \text{null} \ P \ \neg \text{null} \ \text{ltl} \ P \)
    \begin{itemize}
      \item shows \ \text{winning-path} \ p \ \text{ltl} \ P \)
        \begin{itemize}
          \item proof (cases)
            \begin{itemize}
              \item assume \( \text{finite} \ P \)
              \item moreover have \( \text{last} \ P = \text{last} \ \text{ltl} \ P \)
                \begin{itemize}
                  \item using \( P(2,3) \) by \( \text{metis \ last-LCons2 \ ltl-simps}(2) \ \text{not-null-conv} \)
                \end{itemize}
            \end{itemize}
        \end{itemize}
    \end{itemize}
\end{itemize}

next

\begin{itemize}
  \item assume \( \neg \text{finite} \ P \)
  \item thus \( ?\)thesis using \( \text{winning-path-def} \ \text{path-inf-priorities-llt} \ P(1,2) \) by auto
\end{itemize}

qed

\begin{itemize}
  \item \text{corollary} \ \text{winning-path-drop-add}: \ \text{assumes} \ \text{valid-path} \ P \ \text{winning-path} \ p \ \text{ldropn} \ n \ P \) \( \text{enat} \ n < \text{l length} \ P \)
    \begin{itemize}
      \item shows \ \text{winning-path} \ p \ \text{ldropn} \ n \ P \)
        \begin{itemize}
          \item using \( \text{assms} \ \text{proof} \ (\text{induct} \ n) \)
            \begin{itemize}
              \item case \( \text{Suc} \ n \)
              \item hence \( \text{winning-path} \ p \ \text{ldropn} \ n \ P \) using \( \text{dual-order.strict-trans \ evat-ord-simps}(2) \) by \( \text{blast} \)
              \item moreover have \( \text{llt} \) \( \text{ldropn} \ n \ P \) = \( \text{ldropn} \) \( \text{Suc} \) \( n \) \( \text{P} \)
                \begin{itemize}
                  \item by \( \text{simp \ add: \ ldrop-eSuc-llt \ llt-ldropn} \)
                \end{itemize}
              \item moreover hence \( \neg \text{null} \) \( \text{ldropn} \ n \ P \) using \( \text{Suc.prem}(2) \) by \( \text{metis leD \ null-ldropn \ null-ltl} \)
            \end{itemize}
        \end{itemize}
    \end{itemize}
\end{itemize}

ultimately show \( \text{?case using \ \text{winning-path-llt}}(p \ \text{ldropn} \ n \ P \) \( \text{Suc.prem}(2) \)

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

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

lemma \( \text{winning\text{-}path\text{-}super\text{game}}: \)
assumes \( \text{winning\text{-}path} P P \)
and \( G': \text{ParityGame} G' \text{VV} p^{* * } \subseteq \text{ParityGame} . \text{VV} G' p^{* * } \omega = \omega G' \)
shows \( \text{ParityGame}. \text{winning\text{-}path} G' P P \)
proof
interpret \( G': \text{ParityGame} G' \) using \( G'(1) . \)
have \( \left[ \text{finite} P ; \neg \text{null} P \right] \implies \text{l\text{-}last} P \in G'. \text{VV} p^{* * } \) and \( \text{null} P \implies p = \text{Even} \)
using \( \text{assms}(1) \) unfolding \( \text{winning\text{-}path-def} \) using \( G'(2) \) by auto
thus \( ?\text{thesis} \) unfolding \( G'. \text{winning\text{-}path-def} \)
using \( \text{null\text{-}imp}\cdot \text{finite} \) \( \text{assms}(1) \)
unfolding \( \text{winning\text{-}path-def} \) \( \text{path\text{-}inf\text{-}priorities\text{-}def} G' \)
\( \text{path\text{-}inf\text{-}priorities\text{-}def} G'(3) \)
by blast
qed

end — locale ParityGame

3.10 Valid Maximal Paths

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

locale \( \text{vm\text{-}path} = \text{ParityGame} + \)
fixes \( P v0 \)
assumes \( P\text{-not\text{-}null} \) (simp): \( \neg \text{null} P \)
and \( P\text{-valid} \) (simp): \( \text{valid\text{-}path} P \)
and \( P\text{-maximal} \) (simp): \( \text{maximal\text{-}path} P \)
and \( P\text{-v0} \) (simp): \( \text{lhd} P = v0 \)
begin
lemma \( P\text{-LCons}: P = L\text{Cons} v0 \) (\( \text{null} P \)) using \( \text{lhd}\cdot L\text{Cons}\cdot \text{l\text{-}ltl}(O F P\text{-not\text{-}null}) \) by simp

lemma \( P\text{-len} \) (simp): \( \text{enat} 0 < \text{llength} P \) by (simp add: \( \text{null\text{-}0}\cdot \text{llength} \))
lemma \( P\text{-0} \) (simp): \( P \$ 0 = v0 \) by (simp add: \( \text{int\text{-}0}\cdot \text{conv}\cdot \text{lhd} \))
lemma \( P\text{-int\text{-}Suc}: P \$ \text{Suc} n = \text{llt} P \$ n \) by (simp add: \( \text{int\text{-}llt} \))
lemma \( P\text{-no\text{-}deadends}: \text{enat} (\text{Suc} n) < \text{llength} P \implies \neg \text{deadend} (P \$ n) \)
using \( \text{valid\text{-}path}\cdot \text{no\text{-}deadends} \) by simp
lemma \( P\text{-no\text{-}deadend\text{-}v0}: \neg \text{null} (\text{llt} P) \implies \neg \text{deadend} v0 \)
by (metis \( P\text{-LCons} \) \( P\text{-valid} \) \( \text{edges\text{-}are\text{-}in\text{-}V}(2) \) \( \text{not\text{-}null\text{-}conv} \) \( \text{valid\text{-}path\text{-}edges} \))
lemma \( P\text{-no\text{-}deadend\text{-}v0\cdot \text{length}: \text{enat} (\text{Suc} n) < \text{llength} P \implies \neg \text{deadend} v0 \)
by (metis \( P\text{-0} \) \( P\text{-len} \) \( P\text{-valid} \) \( \text{enat\text{-}ori\text{-}simp\text{-}2} \) \( \text{not\text{-}less\text{-}eq} \) \( \text{valid\text{-}path\text{-}ends\text{-}on\text{-}deadend} \) \( \text{zero\text{-}less\text{-}Suc} \))
lemma \( P\text{-ends\text{-}on\text{-}deadend}: [ \text{enat} n < \text{llength} P ; \text{deadend} (P \$ n) ] \implies \text{enat} (\text{Suc} n) = \text{llength} P \)
using \( \text{P\text{-valid}} \) \( \text{valid\text{-}path\text{-}ends\text{-}on\text{-}deadend} \) by blast

lemma \( P\text{-null\text{-}ltl\text{-}deadend\text{-}v0}: \text{null} (\text{llt} P) \implies \text{deadend} v0 \)
using \( P\text{-LCons} \) \( \text{maximal\text{-}no\text{-}deadend} \) by force
lemma \( P\text{-null\text{-}ltl\text{-}L\text{Cons}}: \text{null} (\text{llt} P) \implies P = L\text{Cons} v0 \text{LNil} \)
using P-LCons lnull-def by metis
lemma P-deadend-v0-LCons: deadend v0 \implies P = LCons v0 LNil
   using P-lnull-ltl-LCons P-no-deadend-v0 by blast

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

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

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

lemma extension-valid [simp]: v \rightarrow v0 \implies valid-path (LCons v P)
   using valid-path-cons by auto
lemma extension-maximal [simp]: maximal-path (LCons v P)
   by (simp add: maximal-path-cons)
lemma lappend-maximal [simp]: maximal-path (lappend P' P)
   by (simp add: maximal-path-lappend)

lemma v0-V [simp]: v0 \in V by (metis P-LCons P-valid valid-path-cons-simp)
lemma v0-lset-P [simp]: v0 \in lset P using P-not-null P-v0 llist.set-set(1) by blast
lemma v0-VV [simp]: v0 \in VV P \neq \{} by blast
lemma lset-ltl-P-V [simp]: lset (ltl P) \subseteq V using (simp add: valid-path-in-V)
lemma lset-ltl-P-V [simp]: lset (ltl P) \subseteq V using (simp add: valid-path-in-V)

lemma finite-llast-deadend [simp]: finite P \implies deadend (llast P)
   using P-maximal P-not-null maximal-ends-on-deadend by blast
lemma finite-llast-V [simp]: finite P \implies llast P \in V
   using P-not-null finite-lset lset-P-V by blast

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

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

end

end
4 Positional Strategies

theory Strategy
imports
  Main
  ParityGame
begin

4.1 Definitions

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

type-synonym 'a Strategy = 'a ⇒ 'a

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

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

lemma (in ParityGame) strategyI [intro]:
  \( (∀ v. \ [ v ∈ VV\ p\]; ¬deadend \( v\) \] \(⇒ \) \( v\) → \( σ\ v\)) \(⇒ \) strategy \( p\ σ\)

unfolding strategy-def by blast

4.2 Strategy-Conforming Paths

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

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

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

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

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 \( \sigma \)-arbitrary \( \equiv \lambda v. \text{SOME } w. v \rightarrow w \)

lemma valid-arbitrary-strategy [simp]: strategy \( p \) \( \sigma \)-arbitrary proof
    fix \( v \) assume \( \neg \text{deadend } v \)
    thus \( v \rightarrow \sigma \)-arbitrary \( v \) unfolding \( \sigma \)-arbitrary-def using some-ex[of \( \lambda w. v \rightarrow w \)] by blast qed

4.4 Valid Strategies

lemma valid-strategy-updates: [ strategy \( p \) \( \sigma \); \( v \rightarrow w \) ] \( \implies \) strategy \( p \) \( (\sigma(v0 := w0)) \)
    unfolding strategy-def by auto

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

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

lemma subgame-strategy-stays-in-subgame:
    assumes \( \sigma \): \( \text{ParityGame.subgame } V' \) \( \text{strategy } p \) \( \sigma \)
    and \( v \in \text{ParityGame.VV } (\text{subgame } V') \) \( \neg \text{Digraph.deadend } (\text{subgame } V') \) \( v \)
    shows \( \sigma \) \( v \in V' \)
proof
    interpret \( G': \text{ParityGame.subgame } V' \) using \( \text{subgame-ParityGame} \).
    have \( \sigma \) \( v \in \text{subgame } V' \) using assms unfolding \( G'.strategy-def \) \( G'.edges-are-in-V(2) \) by blast
    thus \( \sigma \) \( v \in V' \) by (metis \( \text{Diff} \) iff \( \text{IntE.subgame-VV Player.distinguished}(2) \))
qed

lemma valid-strategy-supergame:
    assumes \( \sigma \): strategy \( p \) \( \sigma \)
    and \( \sigma' \): \( \text{ParityGame.strategy } (\text{subgame } V') \) \( p \) \( \sigma' \)
    and \( G'\)-no-deadends: \( \land v. v \in V' \implies \neg \text{Digraph.deadend } (\text{subgame } V') \) \( v \)
    shows strategy \( p \) (override-on \( \sigma \) \( \sigma' \) \( V' \)) (is strategy \( p \) \( ?\sigma \))
proof
    interpret \( G': \text{ParityGame.subgame } V' \) using \( \text{subgame-ParityGame} \).
    fix \( v \) assume \( v \in VV \) \( \neg \text{deadend } v \)
    show \( \exists v. ?\sigma v \) proof (cases)
        assume \( v \in V' \)
        hence \( v \in G'.VV \) using subgame-VV \( (v \in VV \implies) \) by blast
        moreover have \( \neg G'.deadend v \) using \( G'\)-no-deadends \( (v \in V' \implies) \) by blast
        ultimately have \( v \rightarrow \text{subgame } V' \) \( \sigma' v \) using \( \sigma' \) unfolding \( G'.strategy-def \) by blast
        moreover have \( \sigma' v = ?\sigma v \) using \( v \in V' \) by simp
        ultimately show \( \exists \text{thesis } \) by (metis subgame-E subsetCE)
    next
        assume \( v \in V' \)
next

21
thus \( ?\text{thesis using } v \sigma \text{ unfolding strategy-def by simp} \)

\( \text{qed} \)

\( \text{qed} \)

\text{lemma valid-strategy-in-V: } [ \text{strategy } p \sigma; v \in V V p; \neg \text{deadend } v ] \implies \sigma v \in V 

\text{unfolding strategy-def using valid-edge-set by auto} 

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

\text{unfolding strategy-def using edges-are-in-V(1) by auto} 

4.5 Conforming Strategies 

\text{lemma path-conforms-with-strategy-ltl [intro]:} 

\begin{align*}
\text{path-conforms-with-strategy } P \sigma & \implies \text{path-conforms-with-strategy } (ltl P) \sigma \\
\text{by } (drule path-conforms-with-strategy_cases) & (\text{simp-all add: path-conforms-with-strategy_intros(1)})
\end{align*} 

\text{lemma path-conforms-with-strategy-drop:} 

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

\text{lemma path-conforms-with-strategy-prefix:} 

\begin{align*}
\text{path-conforms-with-strategy } P \sigma & \implies \text{prefix } P' P \implies \text{path-conforms-with-strategy } P \sigma \\
\text{proof } (\text{coinduction arbitrary: } P P') & \\
\text{case } (\text{path-conforms-with-strategy } P P') & \\
\text{thus } ?\text{case proof } (\text{cases rule: path-conforms-with-strategy_cases}) & \\
\text{case path-conforms-LNil} & \\
\text{thus } ?\text{thesis using path-conforms-with-strategy(2) by auto} & \\
\text{next} & \\
\text{case path-conforms-LCons-LNil} & \\
\text{thus } ?\text{thesis by } (\text{metis prefix-LCons-conv prefix-antisym prefix-code(1) path-conforms-with-strategy(2)}) & \\
\text{next} & \\
\text{case (path-conforms-VVp v w)} & \\
\text{thus } ?\text{thesis proof } (\text{cases}) & \\
\text{assume } & P' \neq LNil \land P' \neq LCons v LNil & \\
\text{hence } & \exists Q. P' = LCons v (LCons u Q) & \\
\text{by } (\text{metis local.path-conforms-VVp(1) prefix-LCons-conv path-conforms-with-strategy(2)}) & \\
\text{thus } ?\text{thesis using local.path-conforms-VVp(1,3,4) path-conforms-with-strategy(2) by force} & \\
\text{qed auto} & \\
\text{next} & \\
\text{case (path-conforms-VVpstar v)} & \\
\text{thus } ?\text{thesis proof } (\text{cases}) & \\
\text{assume } & P' \neq LNil & \\
\text{hence } & \exists Q. P' = LCons v Q & \\
\text{using local.path-conforms-VVpstar(1) prefix-LCons-conv path-conforms-with-strategy(2) by fastforce} & \\
\text{thus } ?\text{thesis using local.path-conforms-VVpstar path-conforms-with-strategy(2) by auto} & \\
\text{qed simp} & \\
\text{qed} & \\
\text{lemma path-conforms-with-strategy-irrelevant:} & \\
\text{assumes } & \text{path-conforms-with-strategy } p P \sigma v \notin \text{set } P \end{align*}
shows \( \text{path-conforms-with-strategy} \ P \ (\sigma(v := w)) \)
using \( \text{assms} \) apply (coinduction arbitrary: \( P \)) by (drule path-conforms-with-strategy.cases) auto

**Lemma** \( \text{path-conforms-with-strategy-irrelevant-deadend} \):

**Assumes** \( \text{path-conforms-with-strategy} \ P \ \sigma \ \text{deadend} \ v \lor v \notin \mathsf{VV} \ P \)

**Shows** \( \text{path-conforms-with-strategy} \ P \ (\sigma(v := w)) \)
using \( \text{assms} \) proof (coinduction arbitrary: \( P \))

let \( ?\sigma = \sigma(v := w) \)
case (path-conforms-with-strategy \( P \))

thus ?case proof (cases rule: path-conforms-with-strategy.cases)
case (path-conforms-VVp \( v'w Ps \))

have \( w = ?\sigma v' \) proof

from (valid-path \( P \)) have \( \neg\text{deadend} v' \)

using local.path-conforms-VVp(1) valid-path-cons simp by blast

with \( \text{assms}(2) \) have \( v' \neq v \) using local.path-conforms-VVp(2) by blast
thus \( w = ?\sigma v' \) by (simp add: local.path-conforms-VVp(3))

qed

moreover

have \( \exists P. L\text{Cons} w Ps = P \land \text{path-conforms-with-strategy} \ P \ \sigma \land (\text{deadend} v \lor v \notin VV P) \land \text{valid-path} P \)

proof

have valid-path (LCons w Ps)

using local.path-conforms-VVp(1) path-conforms-with-strategy(3) valid-path-ltl' 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 \( \exists P. \text{path-conforms-with-strategy} \ P Ps \ \sigma \land (\text{deadend} v \lor v \notin VV P) \land \text{valid-path} Ps \)

using local.path-conforms-VVpstar(1,3) path-conforms-with-strategy(2,3) valid-path-ltl' by blast

thus \(?thesis \) by (simp add: local.path-conforms-VVpstar(1,2))

qed simp-all

qed

**Lemma** \( \text{path-conforms-with-strategy-irrelevant-updates} \):

**Assumes** \( \text{path-conforms-with-strategy} \ P \ \sigma \land v \in \text{lset} P \rightarrow \sigma v = \sigma' v \)

**Shows** \( \text{path-conforms-with-strategy} \ P \ \sigma' \)
using \( \text{assms} \) 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 \( w = \sigma' v' \) using local.path-conforms-VVp(1,3) path-conforms-with-strategy(2) by auto
thus \(?thesis \) using local.path-conforms-VVp(1,4) path-conforms-with-strategy(2) by auto

qed simp-all

qed

**Lemma** \( \text{path-conforms-with-strategy-irrelevant'} \):

**Assumes** \( \text{path-conforms-with-strategy} \ P \ (\sigma(v := w)) \ v \notin \text{lset} P \)

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

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

lemma path-conforms-with-strategy-lappend:
assumes P: lfinite P ¬lnull P path-conforms-with-strategy p P σ
and P': ¬lnull P' path-conforms-with-strategy p P' σ
and conforms: llast P ∈ VV p =⇒ σ (llast P) = lhd P'
shows path-conforms-with-strategy p (lappend P P') σ
using assms proof (induct P rule: lfinite-induct)
case (LCons P)
  show ?case proof (cases)
  assume lnull (llt P)
  then obtain v0 where v0: P = LCons v0 LNil
  by (metis LCons.prems(1) lhd-LCons-ltl list.collapse(1))
  have path-conforms-with-strategy p (LCons (lhd P) P') σ proof (cases)
    assume lhd P ∈ VV p
    moreover with v0 have lhd P' = σ (lhd P)
      using LCons.prems(5) by auto
    ultimately show ?thesis
      using path-conforms-VVp[of lhd P p lhd P' σ]
      by (metis (no-types) LCons.prems(4) ¬lnull P' lhd-lCons-ltl)
  next
  assume ¬lnull (llt P)
  hence *: path-conforms-with-strategy p (lappend (llt P) P') σ
  by (metis LCons.hyps(3) LCons.prems(1) LCons.prems(2) LCons.prems(5) LCons.prems(5)
       assms(4) assms(5) lhd-LCons-ltl llast-LCons2 path-conforms-with-strategy-ltl)
  have path-conforms-with-strategy p (LCons (lhd P) (lappend (llt P) P')) σ proof (cases)
    assume lhd P ∈ VV p
    moreover hence lhd (llt P) = σ (lhd P)
    by (metis LCons.prems(1) LCons.prems(2) ¬lnull (llt P);
        lhd-LCons-ltl path-conforms-with-strategy-start)
    ultimately show ?thesis
      using path-conforms-VVp[of lhd P p lhd (llt P) σ] *
      (¬lnull (llt P))
      by (metis lappend-code(2) lhd-LCons-ltl)
  next
  assume lhd P ∉ VV p
  thus ?thesis by (simp add: * path-conforms-VVpstar)
  qed
  thus ?thesis by (simp add: v0)
next
  assume ¬lnull (llt P)
  hence *: path-conforms-with-strategy p (lappend (llt P) P') σ
  by (metis LCons.hyps(3) LCons.prems(1) LCons.prems(2) LCons.prems(5) LCons.prems(5)
       assms(4) assms(5) lhd-LCons-ltl llast-LCons2 path-conforms-with-strategy-ltl)
  have path-conforms-with-strategy p (LCons (lhd P) (lappend (llt P) P')) σ proof (cases)
    assume lhd P ∈ VV p
    moreover hence lhd (llt P) = σ (lhd P)
    by (metis LCons.prems(1) LCons.prems(2) ¬lnull (llt P);
        lhd-LCons-ltl path-conforms-with-strategy-start)
    ultimately show ?thesis
      using path-conforms-VVp[of lhd P p lhd (llt P) σ] *
      (¬lnull (llt P))
      by (metis lappend-code(2) lhd-LCons-ltl)
  next
  assume lhd P ∉ VV p
  thus ?thesis by (simp add: * path-conforms-VVpstar)
  qed
with (¬lnull P) show path-conforms-with-strategy p (lappend P P') σ
by (metis lappend-code(2) lhd-LCons-ltl)
lemma path-conforms-with-strategy-VVpstar:
  assumes lset P ⊆ VV p ++
  shows path-conforms-with-strategy p P σ
  using assms proof (coinduction arbitrary: P)
  case (path-conforms-with-strategy P)
  moreover have ∨ v Ps. P = LCons v P "⇒ ?case using path-conforms-with-strategy by auto
  ultimately show ?case by (cases P = LNil, simp) (metis lnull-def not-llnulI-not-nll-cong)
qed

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

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

4.6 Greedy Conforming Path

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

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

context begin

private primcorec greedy-conforming-path :: Player ⇒ 'a Strategy ⇒ 'a Strategy ⇒ 'a ⇒ 'a Path
where
  greedy-conforming-path p σ σ' v0 =
  LCons v0 (if dendend v0
  then LNil
  else if v0 ∈ VV p
  qed simp-all

25
then greedy-conforming-path p σ σ′ (σ v0)
else greedy-conforming-path p σ σ′ (σ ′ v0))

private lemma greedy-path-LNil: greedy-conforming-path p σ σ′ v0 ≠ LNil
using greedy-conforming-path.disc-iff list.discI(1) by blast

private lemma greedy-path-lhd: greedy-conforming-path p σ σ′ v0 = LCons v P =⇒ v = v0
using greedy-conforming-path.code by auto

private lemma greedy-path-deadend-v0: greedy-conforming-path p σ σ′ v0 = LCons v P =⇒ P = LNil ↔ 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 σ σ′ v0 = LCons v P =⇒ P = LNil ↔ deadend v
using greedy-path-deadend-v0 greedy-path-lhd by metis
corollary greedy-path-deadend-v: greedy-conforming-path p σ σ′ v0 = LCons v LNil =⇒ deadend v
using greedy-path-deadend-v by blast

private lemma greedy-path-lhd:
assumes greedy-conforming-path p σ σ′ v0 = LCons v P
shows P = LNil ∨ P = greedy-conforming-path p σ σ′ (σ v0) ∨ P = greedy-conforming-path p σ σ′′ (σ′ v0)
apply (insert assms, frule greedy-conforming-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-lhd-ex:
assumes greedy-conforming-path p σ σ′ v0 = LCons v P
shows P = LNil ∨ (∃ v. P = greedy-conforming-path p σ σ′ v)
using assms greedy-path-lhd by blast

private lemma greedy-path-lhd-VVp:
assumes greedy-conforming-path p σ σ′ v0 = LCons v0 P v0 ∈ VV p ~deadend v0
shows σ v0 = lhd P
using assms greedy-conforming-path.code by auto

private lemma greedy-path-lhd-VVpstar:
assumes greedy-conforming-path p σ σ′ v0 = LCons v0 P v0 ∈ VV p* ~deadend v0
shows σ′ v0 = lhd P
using assms greedy-conforming-path.code by auto

private lemma greedy-conforming-path-properties:
assumes v0 ∈ V strategy p σ strategy p* σ′
sshows
    greedy-path-not-null: ~null (greedy-conforming-path p σ σ′ v0)
and greedy-path-v0: greedy-conforming-path p σ σ′ v0 $ 0 = v0
and greedy-path-valid: valid-path (greedy-conforming-path p σ σ′ v0)
and greedy-path-maximal: maximal-path (greedy-conforming-path p σ σ′ v0)
and greedy-path-conforms: path-conforms-with-strategy p (greedy-conforming-path p σ σ′ v0) σ

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

proof

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

show ¬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′: ?P = LCons v0 P′ by (metis greedy-path-LNil greedy-path-lhd neg-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'' using (v0 ∈ V) unfolding P-def
  proof (coinduction arbitrary: v0)
    case (path-conforms-with-strategy v0)
    show ?case proof (cases v ∈ VV p'')
      case True

27
\{ assume \( \neg (\exists v. \text{greedy-conforming-path } p \sigma \sigma' \; v0 = LCons v LNil) \) with \((v0 \in V)\) obtain \( P' \) where
\begin{align*}
P' & = \text{greedy-conforming-path } p \sigma \sigma' \; v0 = LCons v0 P' \; \text{hull } P' \; v0 \rightarrow \lhd P' \\text{lhd } P' \in V \exists v. \; P' & = \text{greedy-conforming-path } p \sigma \sigma' \; v \wedge v \in V \\
\end{align*}
\text{using coinduction-helper by blast with } (v0 \in VV \; P'' \; p'' \; \text{have } \sigma'' \; v0 = \lhd P' \\
\text{using greedy-path-ltl-VV}p \; \text{greedy-path-ltl-VV}p \star \text{by blast with } (v0 \in VV \; P'' \; P'(1,2,5) \; \text{have } \text{?path-conforms-VV}p \}
\text{using greedy-conforming-path.code path-conforms-with-strategy(1) by fastforce }
\}
\text{thus } \text{?thesis by auto}
next
\text{case False}
\text{thus } \text{?thesis using coinduction-helper[of v0] path-conforms-with-strategy by auto qed}
\text{qed}
\}
\text{thus path-conforms-with-strategy } p \; P \; \sigma \; \text{path-conforms-with-strategy } p \; p'' \; P \; \sigma' \; \text{by blast+ qed}
\}
corollary \text{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'
\text{using assms by unfold-locales (simp-all add: greedy-conforming-path-properties)}
\text{qed}
\}
corollary \text{strategy-conforming-path-exists-single:}
\text{assumes } v0 \in V \; \text{strategy } p \;
\text{obtains } P \; \text{where } \text{vmc-path } G \; P \; v0 \; p \;
\text{proof}
\text{show } \text{vmc-path } G \; (\text{greedy-conforming-path } p \; \sigma \; \text{arbitrary } v0) \; v0 \; p \;
\text{using assms by unfold-locales (simp-all add: greedy-conforming-path-properties)}
\text{qed}
\end{align*}
\}
end
end

4.7 Valid Maximal Conforming Paths

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

context \text{vmc-path begin}
lemma \text{Plt-conforms [simp]: path-conforms-with-strategy } p \; (\text{ltl } P) \; \sigma
\text{using } P\text{-conforms path-conforms-with-strategy-ltl by blast}
lemma \text{Pldrop-conforms [simp]: path-conforms-with-strategy } p \; (\text{ldrop } n \; P) \; \sigma
\text{using } P\text{-conforms path-conforms-with-strategy-drop by blast}
lemma \text{prefix-conforms [simp]: path-conforms-with-strategy } p \; (\text{ltake } n \; P) \; \sigma
\text{using } P\text{-conforms path-conforms-with-strategy-prefix by blast}
lemma \text{extension-conforms [simp]:}
end
end

28
\( v' \in VV \implies \sigma v' = v0 \) \( \implies \) 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 \( v' \rightarrow v0 \ v' \in VV \implies \sigma v' = v0 \)
shows \( vmc-path \ G \ (LCons v' \ P) \ v' \ p \ \sigma \)
using assms by unfold-locales simp-all

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

**lemma** conforms-to-another-strategy:
path-conforms-with-strategy \( p \ \sigma' \implies\) vmc-path \( G \ p0 \ p \ \sigma' \)
using P-not-null P-valid P-maximal P-v0 by unfold-locales blast+
end

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

### 4.8 Valid Maximal Conforming Paths with One Edge

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

**locale** vmc-path-no-deadend = vmc-path +
assumes v0-no-deadend [simp]: \( \neg \text{deadend} \ v0 \)
begin
definition \( w0 \equiv \text{lhd} \ (ltl \ P) \)

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

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

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

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

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

**lemma** P-Suc-0: \( P \ \$ \ Suc \ 0 = w0 \) by (simp add: P-Suc Ptl-0 w0-def)

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

**lemma** vtl-conforms: \( v0 \in VV \implies \sigma v0 = w0 \)
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]: \( w0 \in \text{lset} \ P \) by (metis P-LCons' lset-intros(1) lset-intros(2))

**lemma** vmc-path-ltl [simp]: \( vmc-path \ G \ (ltl \ P) \ w0 \ p \ \sigma \) by (unfold-locales) (simp-all add: w0-def)
end

29
context vmc-path begin

lemma vmc-path-lnull-llt-no-deadend:
¬lnull (ltl P) ⟹ vmc-path-no-deadend G P v0 p σ
using P-0 P-no-deadends by (unfold-locales) (metis enat-llt-Suc lnul-l-length)

lemma vmc-path-conforms:
assumes enat (Suc n) < llength P P $ n ∈ VV p
shows σ (P $ n) = P $ Suc n
proof
  define P' where P' = ldopn n P
  then interpret P': vmc-path G P' P $ n p σ using vmc-path-ldopn assms(1) Suc-length by blast
  have ¬deadend (P $ n) using assms(1) P-no-deadends by blast
  then interpret P': vmc-path-no-deadend G P' P $ n p σ by unfold-locales
  have σ (P $ n) = P',w0 using P'.v0-conforms assms(2) by blast
  thus ?thesis using P'.P-Suc-0 assms by simp
qed

4.9 lset Induction Schemas for Paths

Let us define an induction schema useful for proving lset P ⊆ S.

lemma vmc-path-lset-induction [consumes 1, case-names base step]:
  assumes Q P
  and base: v0 ∈ S
  and step-assumption: \[ P \] v0. [ vmc-path-no-deadend G P v0 p σ; v0 ∈ S; Q P ] ⟹ Q (llt P) ∧ (vmc-path-no-deadend.w0 P) ∈ S
  shows lset P ⊆ S
proof
  fix v assume v ∈ lset P
  thus v ∈ 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 σ by blast
    show ?case by (simp add: find.prems(3))
  next
    case (step P v)
    then interpret vmc-path G P v0 p σ by blast
    show ?case proof (cases)
      assume lnul (llt P)
      hence P = LCons v LNil by (metis llist.disc(2) lset-cases step.hyps(2))
      thus ?thesis using step.prems(3) P-LCons by blast
    next
    assume ¬lnul (llt P)
    then interpret vmc-path-no-deadend G P v0 p σ
    using vmc-path-lnull-llt-no-deadend by blast
    show v ∈ S
      using step.hyps(3) step-assumption[OF vmc-path-no-deadend-axioms (v0 ∈ S) (Q P)]
      vmc-path-llt
    by blast
\[ ?Q P ; v0 \in ?S ; \wedge P v0 . [ \text{vmc-path-no-deadend} G P v0 p \sigma ; v0 \in ?S ; ?Q P ] \implies ?Q (\ltl P) \land \text{vmc-path-no-deadend} w0 P \in ?S ] \implies \text{lset} P \subseteq ?S \text{without the Q predicate.} \]

corollary vmc-path-lset-induction-simple [case-names base step]:
  assumes base: v0 \in S
  and step: \wedge P v0 . [ \text{vmc-path-no-deadend} G P v0 p \sigma ; v0 \in S ]
  \implies \text{vmc-path-no-deadend} w0 P \in S
  shows \text{lset} P \subseteq S
  using assms vmc-path-lset-induction[of \lambda P. True] by blast

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

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

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.
\[
\text{definition} \hspace{1em} \text{visits-via } P \ A \ W \equiv \exists n. \ \text{enat } n < \text{llength } P \land \text{P } \# n \in W \land \text{lset } (\text{ltake } (\text{enat } n) \ P) \subseteq A
\]

\[
\text{lemma} \ \text{visits-via-monotone: [ visits-via } P \ A \ W; A \subseteq A' ] \implies \text{visits-via } P \ A' \ W
\]

\[
\text{unfolding visits-via-def by blast}
\]

\[
\text{lemma} \ \text{visits-via-visits: visits-via } P \ A \ W \implies \text{lset } P \cap W \neq \{\}
\]

\[
\text{unfolding visits-via-def by meson disjoint-iff-not-equal in-lset-conv-lnth}
\]

\[
\text{lemma} \ (\text{in vmc-path}) \ \text{visits-via-trivial: } v0 \in W \implies \text{visits-via } P \ A \ W
\]

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

\[
\text{lemma} \ (\text{in vmc-path-no-deadend}) \ \text{visits-via-ltl}
\]

\[
\text{assumes visits-via } P \ A \ W
\]

\[
\text{and} \ v0: \ v0 \notin W
\]

\[
\text{shows visits-via } (\text{ltl } P) \ A \ W
\]

\[
\text{proof –}
\]

\[
\text{obtain } n \ \text{where } n: \ \text{enat } n < \text{llength } P \ P \# n \in W \land \text{lset } (\text{ltake } (\text{enat } n) \ P) \subseteq A
\]

\[
\text{using assms unfolding visits-via-def by blast}
\]

\[
\text{define } P' \text{ where } P' = \text{LCons } v0 \ P
\]

\[
\text{have enat } (\text{Suc } n) < \text{llength } P' \text{ unfolding } P'-def
\]

\[
\text{by (metis n(1)) ldropn-Suc-LCons ldropn-Suc-conv-ldropn-ldropn-conv-LConsD)
\]

\[
\text{moreover have } P' \# \text{Suc } n \in W \text{ unfolding } P'-def \text{ by (simp add: n(2))}
\]

\[
\text{moreover have lset } (\text{ltake } (\text{enat } (\text{Suc } n)) \ P') \subseteq \text{insert } v0 \ A
\]

\[
\text{using lset-ltake-Suc[of P' v0 n A] unfolding P'-def by (simp add: n(3))}
\]

\[
\text{ultimately show } \text{?thesis unfolding visits-via-def } P'-def \text{ by blast}
\]

\[
\text{qed}
\]

\[
\text{lemma} \ (\text{in vmc-path-no-deadend}) \ \text{visits-via-ltl}
\]

\[
\text{assumes visits-via } P \ A \ W
\]

\[
\text{and} \ v0: \ v0 \notin W
\]

\[
\text{shows visits-via } (\text{ltl } P) \ A \ W
\]

\[
\text{proof –}
\]

\[
\text{obtain } n \ \text{where } n: \ \text{enat } n < \text{llength } (\text{ltl } P) \ P \# n \in W \land \text{lset } (\text{ltake } (\text{enat } n) \ (\text{ltl } P)) \subseteq A
\]

\[
\text{using assms unfolding visits-via-def by blast}
\]

\[
\text{have } n \neq 0 \ \text{using v0 n(2) DiffE by force}
\]

\[
\text{then obtain } n' \ \text{where } n' = n \ \text{using nat.exhaust by metis}
\]

\[
\text{have } \exists n'. \ \text{enat } n' < \text{llength } (\text{ltl } P) \land (\text{ltl } P) \# n' \in W \land \text{lset } (\text{ltake } (\text{enat } n') \ (\text{ltl } P)) \subseteq A
\]

\[
\text{apply (rule exI[of - n'])}
\]

\[
\text{using n' enat-Suc-ltl[of n' P] P-ltl-Suc-lset-ltake-ltl[of n' P] by auto}
\]

\[
\text{thus } \text{?thesis using visits-via-def by blast}
\]

\[
\text{qed}
\]

\[
\text{lemma} \ (\text{in vm-path}) \ \text{visits-via-deadend}
\]

\[
\text{assumes visits-via } P \ A \ (\text{deadends } p)
\]

\[
\text{shows winning-path p** P}
\]

\[
\text{using assms visits-via-visits visits-deadend by blast}
\]

5.2 Attracting Strategy from a Single Node

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

\[
\text{definition} \ \text{strategy-attracts-via :: Player } \Rightarrow \text{ 'a Strategy } \Rightarrow \text{ 'a } \Rightarrow \text{ 'a set } \Rightarrow \text{ 'a set } \Rightarrow \text{ bool where}
\]

32
strategy-attracts-via $p \sigma v_0 A W \equiv \forall P. \text{vmc-path } G P v_0 p \sigma \rightarrow \text{visits-via } P A W$

**Lemma (in vmc-path) strategy-attracts-viaE:**
- **assumes** strategy-attracts-via $p \sigma v_0 A W$
- **shows** visits-via $P A W$
- **using** strategy-attracts-via-def **assms** vmc-path-axioms by blast

**Lemma (in vmc-path) strategy-attracts-via-SucE:**
- **assumes** strategy-attracts-via $p \sigma v_0 A W v_0 \not\in W$
- **shows** $\exists n. \text{enat } (\text{Suc } n) < \text{length } P \land P \ni n \in W \land \text{lsset } (\text{ltake } (\text{enat } (\text{Suc } n)) P) \subseteq A$
- **proof**
  - **obtain** $n$ where $n: \text{enat } n < \text{length } P P \ni n \in W \land \text{lsset } (\text{ltake } n P) \subseteq A$
    - **using** strategy-attracts-viaE [unfolded visits-via-def] **assms** (1) by blast
  - **have** $n \not= 0$ using **assms** (2) n(2) by (metis P-0)
  - **thus** \text{?thesis} using $n \not= 0$-implies-Suc by blast
- **qed**

**Lemma (in vmc-path) strategy-attracts-via-lset:**
- **assumes** strategy-attracts-via $p \sigma v_0 A W$
- **shows** $\text{lset } P \cap W \not= \emptyset$
- **using** **assms** THEN strategy-attracts-viaE, unfolded visits-via-def by (meson disjoint-i-not-equal lset-lnth-member subset-refl)

**Lemma strategy-attracts-via-v0:**
- **assumes** $\sigma: \text{strategy } p \sigma \text{ strategy-attracts-via } p \sigma v_0 A W$
  - and $v_0: v_0 \in V$
- **shows** $v_0 \in A \cup W$
- **proof**
  - **obtain** $P$ where $\text{vmc-path } G P v_0 p \sigma$ using strategy-conforming-path-exists-single **assms** by blast
    - **then** interpret $\text{vmc-path } G P v_0 p \sigma$.
    - **obtain** $n$ where $n: \text{enat } n < \text{length } P P \ni n \in W \land \text{lsset } (\text{ltake } n P) \subseteq A$
      - **using** $\sigma(2)[\text{unfolded strategy-attracts-via-def visits-via-def}] \text{vmc-path-axioms}$ by blast
    - **show** \text{?thesis} proof (cases $n = 0$)
      - **case** True **thus** \text{?thesis} using n(2) by simp
      - **next**
        - **case** False
          - **hence** lhd $(\text{ltake } n P) = \text{lhd } P$ by (simp add: enat-0-iff (1))
          - **hence** $v_0 \in \text{lsset } (\text{ltake } n P)$
            - by (metis \(n \not= 0\), P-not-null P-v0 enat-0-iff (1) lset.lset-set(1) ltake.disc(2))
          - **thus** \text{?thesis} using $n(3)$ by blast
- **qed**
- **corollary** strategy-attracts-not-outside:
  - $[ v_0 \in V \setminus A \setminus W; \text{strategy } p \sigma ] \Rightarrow \neg \text{strategy-attracts-via } p \sigma v_0 A W$
  - **using** strategy-attracts-via-v0 by blast

**Lemma strategy-attracts-viaI [intro]:**
- **assumes** $\forall P. \text{vmc-path } G P v_0 p \sigma \rightarrow \text{visits-via } P A W$
- **shows** strategy-attracts-via $p \sigma v_0 A W$
- **unfolding** strategy-attracts-via-def **using** **assms** by blast

33
lemma strategy-attracts-via-no-deadends:
assumes \( v \in V \) \( v \in A - W \) strategy-attracts-via \( p \sigma v A W \)
show \( \lnot \text{deadend } v \)
proof
assume \( \text{deadend } v \)
define \( P \) where \( \text{simp} : P = LCons v LNil \)
interpret \( \text{vmc-path } G P v p \sigma \) 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 \( \text{simp add: maximal-path-intros} (2) \)
  show path-conforms-with-strategy \( p P \) \( \sigma \) \( \) by \( \text{simp add: path-conforms-LCons-LNil} \)
qed \( \text{simp-all} \)

have visits-via \( P A W \) using \( \text{assms} (3) \) strategy-attracts-viaE \( \) by blast
moreover have \( \text{length } P = eSuc 0 \) by simp
ultimately have \( P \$ 0 \in W \) by \( \text{simp add: enat-0-Iff} (1) \) visits-via-def
with \( v \in A - W \) show \( \text{False} \) by auto
qed

lemma attractor-strategy-on-extends:
\[ \text{strategy-attracts-via } p \sigma v0 A W ; A \subseteq A' \Rightarrow \text{strategy-attracts-via } p \sigma v0 A' W \]
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 \( v0 \in W \) \( \text{vmc-path } G P v0 p \sigma \)
then interpret \( \text{vmc-path } G P v0 p \sigma \) by blast
show visits-via \( P A W \) using \( \text{visits-via-trivial} \) using \( \langle v0 \in W \rangle \) by blast
qed

lemma strategy-attracts-via-successor:
assumes \( \sigma : \text{strategy } p \sigma \text{ strategy-attracts-via } p \sigma v0 A W \)
  and \( v0 : v0 \in A - W \)
  and \( w0 : v0 \rightarrow w0 \) \( v0 \in VV p \Rightarrow \sigma v0 = w0 \)
shows \( \text{strategy-attracts-via } p \sigma w0 A W \)
proof
fix \( P \) assume \( \text{vmc-path } G P w0 p \sigma \)
then interpret \( \text{vmc-path } G P w0 p \sigma \) .
define \( P' \) where \( \text{simp} : P' = LCons v0 P \)
then interpret \( \text{vmc-path } G P' v0 p \sigma \) .
using extension-valid-maximal-conforming \( w0 \) by blast
interpret \( P' : \text{vmc-path-no-deadend } G P' v0 p \sigma \) 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 \( P' : \text{visits-via-ll} v0 \) by simp
qed

lemma strategy-attracts-VVp:
assumes \( \sigma : \text{strategy } p \sigma \text{ strategy-attracts-via } p \sigma v0 A W \)
  and \( v : v0 \in A - W v0 \in VV p \text{ -deadend } v0 \)
shows \( \sigma v0 \in A \cup W \)
proof
have \( v0 \rightarrow v \) using \( \sigma (1) [\text{unfolding strategy-def} ] v (2,3) \) by blast
hence strategy-attracts-via \( p \sigma (\sigma v0) A W \)

\[34\]
5.3 Attracting strategy from a set of nodes

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

**definition** strategy-attracts :: Player \( \Rightarrow \) 'a Strategy \( \Rightarrow \) 'a set \( \Rightarrow \) bool

where

\[
\text{strategy-attracts} \ p \ \sigma \ W v0 \equiv \forall v0 \in A. \ \text{strategy-attracts-via} \ p \ \sigma \ W v0
\]

**lemma (in vmc-path)** strategy-attractsE:

**assumes** strategy-attracts \( p \ \sigma \ A \ W v0 \in A \)

**shows** visits-via \( P \ A W \)

**using** \( \text{assms}(1)[\text{unfolded strategy-attracts-def}] \) \( \text{assms}(2) \) strategy-attracts-viaE by blast

**lemma (in vmc-path)** strategy-attracts-lset:

**assumes** strategy-attracts \( p \ \sigma \ A \ W v0 \in A \)

**shows** \( \text{lset} P \cap W \neq \{\} \)

**using** \( \text{assms}(1)[\text{unfolded strategy-attracts-def}] \) \( \text{assms}(2) \) 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** \( P\colon P = LCons v (LCons w P') \ v \in A - W w \notin A \cup W \)

**shows** \( \neg \text{visits-via} \ P \ A W \) (is \( \neg \?A \))

**proof** 

**assume** \( \?A \)

**then obtain** \( n \) where \( n\colon \text{enat} n < \text{lheight} P P \ S n \in W \text{lset} \ (\text{ltake} \ (\text{enat} n) P) \subseteq A \)

**unfolding visits-via-def by blast**

**have** \( n \neq 0 \) using \( \ (v \in A - W) \ n(2) \ P(1) \text{ DiffD2 by force} \)

**moreover have** \( n \neq \text{Suc} 0 \ ) using \( \ (w \notin A \cup W \ n(2) \ P(1) \text{ by auto} \)

**ultimately have** \( \text{Suc} (\text{Suc} 0) \leq n \) by presburger

**hence** \( \text{lset} \ (\text{ltake} \ (\text{enat} (\text{Suc} (\text{Suc} 0))) P) \subseteq A \) using \( n(3) \)

**by** \( \text{meson contra-subsetD enat-ord-simps(1) lset-ltake-prefix lset-lth-member lset-subset} \)

**moreover have** \( \text{enat} (\text{Suc} 0) < \text{lheight} (\text{ltake} (eSuc (eSuc 0)) P) \)

**proof** 

**have s\colon \text{enat} (\text{Suc} (\text{Suc} 0)) < \text{lheight} P**

**using** \( \ (\text{Suc} (\text{Suc} 0) \leq n) \) by \( \text{meson enat-ord-simps(2) le-less-linear less-le-trans nq-if} \)

**have** \( \text{lheight} (\text{ltake} (\text{enat} (\text{Suc} (\text{Suc} 0)))) P) = \text{min} (\text{enat} (\text{Suc} (\text{Suc} 0))) (\text{lheight} P) \) by simp

**hence** \( \text{lheight} (\text{ltake} (\text{enat} (\text{Suc} (\text{Suc} 0)))) P) = \text{enat} (\text{Suc} (\text{Suc} 0)) \)
Given an attracting strategy \( \sigma \)

**lemma strategy-attracts-does-not-leave:**
- **assumes** \( \sigma \): strategy-attracts \( p \ \sigma \ A W \) strategy \( p \ \sigma \)
  - and \( v: v \rightarrow w v \in A - W w \notin A \cup W \)
- **shows** \( v \in VV p \land \sigma v \neq w \)
- **proof** (rule ccontr)
  - **assume** ccontra: \((v \in VV p \land \sigma v \neq w)\)
  - **define** \( \sigma' \) where \( \sigma' = \sigma\)-arbitrary\((v := w)\)
  - **hence** strategy \( p \ast \sigma' \) using \((v \rightarrow w)\)
  - **then obtain** \( P \) where \( P; vmc2-path G P v p \sigma \sigma' \)
    - using \((v \rightarrow w)\) strategy-conforming-path-exists \((2)\) using blast
  - **then interpret** vmc2-path \( G P v p \sigma \sigma' \).
  - interpret vmc-path-no-deadend \( G P v p \sigma \) using \((v \rightarrow w)\) by unfold-locales blast
  - interpret \( \operatorname{comp} \) : vmc-path-no-deadend \( G P v p \sigma \) \( \sigma' \) using \((v \rightarrow w)\) by unfold-locales blast
  - have \( w = \operatorname{v0} \) using and \( \sigma' \)-def \( \operatorname{v0}\)-conforms comp.\( \operatorname{v0}\)-conforms by (cases \( v \in VV p \)) auto
  - **hence** \( \neg \text{visits-via} P A W \)
    - using strategy-attracts-invalid-path \([\text{of} P v w \\text{lll} (\text{lll} P)]\) \((2,3)\) P-LCons' by simp
  - **thus** False by (meson DiffE \((1)\) strategy-attractsE \((2)\))
- **qed**

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

**lemma strategy-attracts-irrelevant-override:**
- **assumes** strategy-attracts \( p \ \sigma \ A W \) strategy \( p \ \sigma \)
  - **shows** strategy-attracts \( p \) \( \langle \text{override-on} \ \sigma' \ \sigma \ (A - W) \rangle \ A W \)
- **proof** (rule strategy-attractsI, rule ccontr)
  - **fix** \( P v \)
  - **let** \( \sigma = \text{override-on} \ \sigma' \ \sigma \ (A - W) \)
  - **assume** vmc-path \( G P v p ? \sigma \)
  - **then interpret** vmc-path \( G P v p ? \sigma \).
  - **assume** \( v \in A \)
  - **hence** \( P \olec 0 \ A \) using \((v \in A)\) by simp
    - moreover assume ccontra: \( \neg \text{visits-via} P A W \)
  - ultimately have \( P \olec 0 \ A - W \) unfolding visits-via-def by (meson DiffI P-len not-less0 lset-ltake)
  - have \( \lnot \text{lset} P \subseteq A - W \) proof
    - assume \( \lset P \subseteq A - W \)
    - **hence** \( \land \forall. v \in \text{lset} P \implies \text{override-on} \ \sigma' \ \sigma \ (A - W) \) \( v = \sigma v \) by auto
    - **hence** path-conforms-with-strategy \( p \ \sigma \)
      - using path-conforms-with-strategy-Irrelevant-updates \([OF P\text{-conforms}]\) by blast

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 \textit{vmc-path} \ G \ P \ (P \ \& \ 0) \ p \ \sigma \\
using \textit{conforms-to-another-strategy} \ P-0 \ by \ blast \\
thus \textit{False} \\
using \textit{contra} \ (P \ \& \ 0 \ \in \ A; \ \text{assms}(1)) \\
by \ (\textit{meson \ \textit{vmc-path} \ . \ \textit{strategy-attractsE}}) \\
\text{qed} \\
hence \exists n. \ \text{enat} \ n \ < \ llength \ P \ \& \ P \ \notin \ A - W \ by \ (\textit{meson \ lset-subset}) \\
then \text{obtain} \ n \ where \ n: \ \text{enat} \ n \ < \ llength \ P \ \& \ P \ \notin \ A - W \\
\forall i. \ i < n \ \Rightarrow \ \neg(\text{enat} \ i \ < \ llength \ P \ \& \ P \ \notin \ A - W) \\
using \textit{ex-least-not-le} \ [\forall n. \ \text{enat} \ n \ < \ llength \ P \ \& \ P \ \notin \ A - W] \ by \ blast \\
hence \ n\text{-min}: \ \forall i. \ i < n \ \Rightarrow \ P \ \notin \ A - W \\
using \textit{dual-order-strict-trans} \ \textit{enat-ord-simps}(2) \ by \ blast \\
\text{have} \ n \neq 0 \ using \ (P \ \& \ 0 \ \in \ A - W) \ \text{assms}(1) \ by \ \textit{meson} \\
then \text{obtain} \ n' \ where \ n': \ \text{Suc} \ n' = n \ using \ \textit{not0-implies-Suc} \ by \ blast \\
hence \ P \ \& \ n' \in \ A - W \ using \ \textit{n-min} \ by \ blast \\
moreover \ \text{have} \ P \ \& \ n' \rightarrow P \ \& \ Suc \ n' \ using \ \textit{P-valid} \ n(1) \ \textit{n-valid-path-edges} \ by \ blast \\
moreover \ \text{have} \ P \ \& \ Suc \ n' \notin A \ \cup W \ \text{proof} - \\
\text{have} \ P \ \& \ n' \notin W \ using \ \textit{contra} \ n(1) \ \textit{n-min} \ \textit{unfolding} \ \textit{visits-via-def} \\
by \ (\textit{meson \ Diff-subset \ lset-ltake \ subsetCE}) \\
thus \ \textit{thesis} \ using \ n(1) \ n' \ by \ blast \\
\text{qed} \\
ultimately \ \text{have} \ P \ \& \ n' \in VV \ p \ \& \ \sigma \ (P \ \& \ n') \neq P \ \& \ Suc \ n' \\
using \ \textit{strategy-attracts-does-not-leave} \ [of \ p \ \sigma \ A \ P \ P \ \& \ n' \ P \ \& \ Suc \ n'] \\
\text{assms}(1,2) \ by \ blast \\
thus \ \textit{False} \\
using \ n(1) \ n' \ \textit{vmc-path-conforms} \ (P \ \& \ n' \ \in \ A - W) \ by \ (\textit{metis override-on-apply-in}) \\
\text{qed} \\

\textbf{lemma} \ \textit{strategy-attracts-trivial} \ [\textit{simp}]: \ \textit{strategy-attracts} \ p \ \sigma \ W \ W \\
by \ (\textit{simp \ add: \ strategy-attracts-def \ strategy-attracts-via-trivial}) \\
If a \ \sigma\text{-conforming} \ \textit{path} \ \text{hits} \ \text{an \ \textit{attractor}} \ A, \ \text{it \ \textit{will \ \textit{visit}}} \ \textit{W}.

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

\textbf{lemma} \ \textit{strategy-attracted-step}: \\
\text{assumes} \ \sigma: \ \textit{strategy} \ p \ \sigma \ \textit{strategy-attracts} \ p \ \sigma \ A \ W \\
and \ \textit{V0}: \ \neg\textit{deadend} \ \textit{V0} \ \textit{V0} \ \in \ A - W \ \textit{V0} \ \in \ VV \ p \\
\text{shows} \ \sigma \ \textit{V0} \ \in \ A \ \cup \ W \\
by \ (\textit{metis \ DiffI \ strategy-attracts-VPp \ assms \ strategy-attracts-def}) \\

\textbf{lemma} \ (\textit{in \ \textit{vmc-path-no-deadend}) \ \textit{attracted-path-step}:
assumes $\sigma$: strategy-attracts $p\;\sigma\;A\;W$
and $v0: v0 \in A - W$
shows $v0 \in A \cup W$
by (metis (no-types) DiffD1 P-LCons' $\sigma$ strategy-attractsE strategy-attracts-invalid-path $v0$)

end — context ParityGame
end

6 Attractor Sets

theory Attractor
imports
  Main
  AttractingStrategy
begin
Here we define the $p$-attractor of a set of nodes.

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

definition directly-attacked :: Player $\Rightarrow$ 'a set $\Rightarrow$ 'a set where
  directly-attacked $p\;S \equiv \{ v \in V - S. \neg$deadend $v \land$
  $(v \in V V p \quad \exists w. v \rightarrow w \land w \in S))$
  $\land (v \in V V p^{**} \quad \forall w. v \rightarrow w \rightarrow w \in S)\}$

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

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-attacked

Show a few basic properties of directly-attacked.

lemma directly-attacked-disjoint $\quad$ [simp]: directly-attacked $p\;W \cap W = \{}$
and directly-attacked-empty $\quad$ [simp]: directly-attacked $p\;\{} = \{}$
and directly-attacked-V-empty $\quad$ [simp]: directly-attacked $p\;V = \{}$
and directly-attacked-bounded-by-V $\quad$ [simp]: directly-attacked $p\;W \subseteq V$
and directly-attacked-contains-no-deadends $\quad$ [elim]: $v \in$ directly-attacked $p\;W \Longrightarrow \neg$deadend $v$

unfolding directly-attacked-def by blast+

6.2 attractor-step

lemma attractor-step-empty: attractor-step $p\;\{} \;\{} = \{}$
and attractor-step-bounded-by-V $\quad$ [ \( W \subseteq V \); $S \subseteq V \] $\quad$ implies 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 monoI) 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 lp-res-unfold by blast

Lemma attractor-lowerbound: attractor-step p W S ⊆ S ⇒ attractor p W ⊆ S

unfolding attractor-def using attractor-step-mono by (simp add: lfp-lowerbound)

Lemma attractor-set-non-empty: W ≠ {} ⇒ attractor p W ≠ {}

and attractor-set-base: W ⊆ attractor p W

using attractor-unfolding 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. 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

Lemma removing-attractor-induces-no-deadends:

assumes v ∈ S − attractor p W v → w w ∈ S \w. [ v ∈ VV p; v → w ] ⇒ w ∈ S

shows ∃ w ∈ S − attractor p W. v → w

proof –

have v ∈ V using ⟨v → w⟩ by blast

thus ?thesis proof (cases rule: VV-cases)

assume v ∈ VV p

thus ?thesis using attractor-set-VVp assms by blast

next

assume v ∈ VV p***

thus ?thesis using attractor-set-VVpstar assms by (metis Diff_iiff edges-are-in-V(2))

qed

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

Lemma subgame-without-deadends:

assumes V' − def: V' = V − attractor p (deadends p** − attractor p** (deadends p****))

(is V' = V − ?A − ?B)

and v: v ∈ V subgame V'

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
\{ \text{fix } p \prime \text{ assume } v \in V V \, p' \}

\text{hence } v \in \text{attractor p'} (\text{deadends p'**})

\text{using } (\text{deadend } v) \text{ attractor-set-base[of deadends p'** p']}

\text{unfolding deadends-def by blast}

\text{hence } \text{False using } v \text{ by } (\text{cases } p' ; \text{ cases } p) \text{ auto}
\}

\text{thus } \text{?thesis using } v \text{ by blast}

next

\text{assume } \neg \text{deadend } v

\text{have } v \in V = \neg A = \neg B \text{ using } v \text{ unfolding } V'\text{-def subgame-def by simp}

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

\text{interpret } G'; \text{ ParityGame } G' \text{ unfolding } G'\text{-def using subgame-ParityGame} .

\text{show } \text{?thesis proof}

\text{assume } \text{Digraph. deadend } (\text{subgame } V') \, v

\text{hence } G'.\text{deadend } v \text{ unfolding } G'\text{-def} .

\text{have all-in-attractor: } \forall w. \, v \mapsto w \implies w \in \neg A \lor w \in \neg B \text{ proof } (\text{rule contr})

\text{fix } w

\text{assume } v \mapsto w \neg (w \in \neg A \lor w \in \neg B)

\text{hence } w \in V' \text{ unfolding } V'\text{-def by blast}

\text{hence } w \in V_G \text{ unfolding } G'\text{-def subgame-def using } (v \mapsto w) \text{ by auto}

\text{hence } v \mapsto G' \, w \text{ using } (v \mapsto w) \text{ assms(2) unfolding } G'\text{-def subgame-def by auto}

\text{thus } \text{False using } (G'.\text{deadend } v) \text{ using } (w \in V_G) \text{ by blast}

\text{qed}

\{ \text{fix } p \prime \text{ assume } v \in V V \, p' \}

\{ \text{assume } \exists w. \, v \mapsto w \land w \in \text{attractor p'} (\text{deadends p'**})

\text{hence } v \in \text{attractor p'} (\text{deadends p'**}) \text{ using } w \in V V \, p' \text{ attractor-set-VVp by blast}

\text{hence } \text{False using } v \text{ by } (\text{cases } p' ; \text{ cases } p) \text{ auto}
\}

\text{hence } \forall w. \, v \mapsto w \implies w \in \text{attractor p'** (deadends p'****)}

\text{using all-in-attractor by } (\text{cases } p' ; \text{ cases } p) \text{ auto}

\text{hence } v \in \text{attractor p'** (deadends p'****)}

\text{using } (\neg \text{deadend } v) \, (v \in V V \, p') \text{ attractor-set-VVpstar by auto}

\text{hence } \text{False using } v \text{ by } (\text{cases } p' ; \text{ cases } p) \text{ auto}
\}

\text{thus } \text{False using } v \text{ by blast}

\text{qed}

\text{qed}

\textbf{6.6 Attractor Set Induction}

\textbf{Lemma mono-restriction-is-mono: mono } f \implies \text{mono } (\lambda S. \, f (S \cap V))

\text{unfolding } \text{mono-def by } (\text{meson inf-mono monoD subset-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 \url{https://lists.cam.ac.uk/pipermail/cl-isabelle-users/2015-October/msg00123.html}

\textbf{Lemma attractor-set-induction [consumes 1, case-names step union]:}

\text{assumes } W \subseteq V

\text{and step: } \forall S. \, S \subseteq V \implies P \, S \implies P \, (\text{attractor-set } p \, W \, S)

\text{and union: } \forall M. \, \forall S \in M. \, S \subseteq V \land P \, S \implies P \, (\bigcup M)

\text{shows } P \, (\text{attractor } p \, W)

\text{40}
proof –
let $\mathcal{A} = \lambda S. P (S \cap V)$
let $\mathcal{B} = \lambda S. \text{attractor-step } p W (S \cap V)$
let $\mathcal{C} = \lambda P$ $\mathcal{A}$
let $\mathcal{D} = \lambda P (\text{attractor-step } p W)$

have $\mathcal{A}$-mono: mono $\mathcal{A}$
  using $\text{mono-restriction-is-mono}$ of $\text{attractor-step } p W$ $\text{attractor-step-mono}$ by simp

have $\mathcal{P}$-A: $\mathcal{A}$ $\mathcal{P}$ proof (rule $\text{lfp-or ordinal-induct-set}$)
  show $\bigwedge S. ?P S \Longrightarrow ?P (W \cup (S \cap V) \cup \text{directly-attracted } p (S \cap V))$
    by (metis assms(1) $\text{attractor-step-bounded-by-V}$ inf.absorb1 inf.le2 local.step)
  show $\bigwedge M. \forall S \in M. ?P S \Longrightarrow ?P (\bigcup M)$ 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 $\mathcal{A} \subseteq ?B \subseteq V$ using $*$ by (simp add: lfp-lowerbound)

have $\mathcal{A} = ?B$ using $\mathcal{A}$-mono $\text{lfp-unfold}$ by blast
hence $\mathcal{A} = W \cup (?A \cap V) \cup \text{directly-attracted } p (?A \cap V)$ using $\mathcal{A} \subseteq V$ by simp
hence $*: \text{attractor-step } p W ?A \subseteq ?A$ using $\mathcal{A} \subseteq V$ inf.absorb1 by fastforce

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

have $\mathcal{A} = ?B$ proof
  show $\mathcal{A} \subseteq ?B$ using $\mathcal{B} \subseteq ?B$ by (simp add: lfp-lowerbound)
  show $?B \subseteq ?A$ using $*$ by (simp add: lfp-lowerbound)
  qed

hence $?P ?B$ using $\mathcal{P}$-A by (simp add: $\text{attractor-def}$)
thus $?\text{thesis}$ using $\mathcal{B} \subseteq V$ by (simp add: $\text{attractor-def}$ le_iff_inf)
qed

end — context ParityGame

end

7 Winning Strategies

theory WinningStrategy
imports
  Main
  Strategy
begin

end
context ParityGame begin

Here we define winning strategies.

A strategy is winning for player $p$ from $v0$ if every maximal $\sigma$-path starting in $v0$ is winning.

definition winning-strategy :: Player $\Rightarrow$ 'a Strategy $\Rightarrow$ 'a $\Rightarrow$ bool where
  winning-strategy $p$ $\sigma$ $v0$ $\equiv$ $\forall P. \text{vmc-path } G P v0 p \sigma \longrightarrow$ winning-path $p$ $P$

lemma winning-strategyI [intro]:
  assumes $\bigwedge P. \text{vmc-path } G P v0 p \sigma \Longrightarrow$ winning-path $p$ $P$
  shows winning-strategy $p$ $\sigma$ $v0$

unfolding winning-strategy-def using assms by blast

lemma (in vmc-path) paths-hits-winning-strategy-is-winning:
  assumes $\sigma$: winning-strategy $p$ $\sigma$ $v$
    and $v$: $v \in \text{lset } P$
  shows winning-path $p$ $P$

proof –
  obtain $n$ where $n$: $\text{enat } n < \text{length } P P \$ n = v using $v$ by (meson in-lset-conv-lth)
  interpret $P'$: $\text{vmc-path } G \text{Idropn } n P v p \sigma$ using $n$ $\text{vmc-path-Idropn}$ by blast
  have winning-path $p$ ($\text{Idropn } n P$) using $\sigma$ by (simp add: winning-strategy-def $P'.\text{vmc-path-axioms}$)
  thus ?thesis using winning-path-drop-add $P$-valid $n(1)$ by blast
qed

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

lemma winning-strategy-only-for-one-player:
  assumes $\sigma$: strategy $p$ $\sigma$ winning-strategy $p$ $\sigma$ $v$
    and $\sigma'$: strategy $p$ $\sigma'$ winning-strategy $p$ $\sigma'$ $v$
    and $v$: $v \in V$
  shows False

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

7.1 Deadends

lemma no-winning-strategy-on-deadends:
  assumes $v$: $v \in \text{VV } p \text{ deadend } v$ strategy $p$ $\sigma$
  shows $\neg$winning-strategy $p$ $\sigma$ $v$

proof –
  obtain $P$ where $\text{vmc-path } G P v p \sigma$ using strategy-conforming-path-exists-single assms by blast
  then interpret $\text{vmc-path } G P v p \sigma$.
  have $P = LCons v LNil$ using $P$-deadend-$v0$-$LCons$ (deadend $v$) by blast
  hence $\neg$winning-path $p$ $P$ unfolding winning-path-def using ($v \in \text{VV } p$) by auto
  thus ?thesis using winning-strategy-def $\text{vmc-path-axioms}$ by blast
lemma winning-strategy-on-deadends:
  assumes \( v \in VV_p \) deadend \( v \) strategy \( p \) \( \sigma \)
  shows winning-strategy \( p \) \( \sigma \) \( v \)

proof
  fix \( P \) assumes \( \text{vmc-path } G \ P \ v \) \( p \) \( \sigma \)
  then \( \text{interpret } \text{vmc-path } G \ P \ v \) \( p \) \( \sigma \).
  have \( P = LCons v LNil \) using P-deadend-v0-LCons deadend \( v \) by blast
  thus winning-path \( p \) \( \sigma \) \( P \) unfolding winning-path-def
  using \( \{ v \in VV_p \} \) P-valid paths-are-winning-for-one-player by auto
qed

7.2 Extension Theorems

lemma strategy-extends-VVp:
  assumes \( v0 : v0 \in VV_p \) ~deadend \( v0 \)
  and \( \sigma : \text{strategy } p \) \( \sigma \) winning-strategy \( p \) \( \sigma \) \( v0 \)
  shows winning-strategy \( p \) \( \sigma \) \( (v0 v0) \)

proof
  fix \( P \) assume \( \text{vmc-path } G \ P \ (\sigma v0) \) \( p \)
  then \( \text{interpret } \text{vmc-path } G \ P \ v0 \) \( p \) \( \sigma \).
  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-VVpstar:
  assumes \( v0 : v0 \in VV_p \) \( \star v0 \rightarrow w0 \)
  and \( \sigma : \text{strategy } p \) \( \sigma \) \( \star \)
  shows winning-strategy \( p \) \( \sigma \) \( v0 \)

proof
  fix \( P \) assume \( \text{vmc-path } G \ P \ (\sigma v0) \) \( p \)
  then \( \text{interpret } \text{vmc-path } G \ P \ v0 \) \( p \) \( \sigma \).
  have \( \text{winning-path } p \) \( LCons v0 p \)
  using \( \text{extension-valid-maximal-conforming VV-impl1 } \sigma \) \( v0 \) 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-VVpstar:
  assumes \( v0 : v0 \in VV_p \) \( \star v0 \rightarrow w \)
  and \( \sigma : \text{strategy } p \) \( \sigma \) \( \star \)
  shows winning-strategy \( p \) \( \sigma \) \( v0 \)

proof
  fix \( P \) assume \( \text{vmc-path } G \ P v0 p \) \( \sigma \)
  then \( \text{interpret } \text{vmc-path } G \ P v0 p \) \( \sigma \).
  have \( \text{winning-path } p \) \( P \) \( \sigma \) \( v0 \)
  show winning-path \( p \) \( P \) proof (cases)
  assume \( \text{deadend } v0 \)
  thus \( \text{thesis } \) 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 w0 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 Ptl-not-null)
qed

end — context ParityGame
end

8 Well-Ordered Strategy

theory WellOrderedStrategy
imports
  Main
  Strategy
begin

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

The following locale formalizes this idea.

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

locale WellOrderedStrategies = ParityGame +
fixes S :: 'a set
and p :: Player
— The set of good strategies on a node v
and good :: 'a ⇒ 'a Strategy set
and r :: ('a Strategy × 'a Strategy) set
assumes S-V: S ⊆ V
— r is a wellorder on the set of all strategies which are good somewhere.
Every node has a good strategy.

Good strategies are well-formed strategies.

A good strategy on $v$ is also good on possible successors of $v$.

Good strategies are well-formed strategies.

A good strategy on $v$ is also good on possible successors of $v$.

Among the good strategies on $v$, choose the minimum.

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.

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

Choose always chooses a minimal good strategy on $S$.

Lemma choose-works:

Proof:

Choose always chooses a minimal good strategy on $S$.

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

unfolding \text{minimal-good-strategy-def} by \text{auto}

have $(\sigma, \sigma') \notin r - \text{Id} \text{ using } \sigma(1) \sigma'(2) \text{ by } \text{blast}$

moreover have $(\sigma', \sigma) \notin r - \text{Id} \text{ using } \sigma(2) \sigma'(1) \text{ by } \text{auto}$

moreover have $\sigma \in \text{Strategies using } \sigma(1) \text{ by } \text{auto}$

moreover have $\sigma' \in \text{Strategies using } \sigma'(1) \text{ by } \text{auto}$

ultimately have $\sigma' = \sigma$

using \text{r-wo Linear-order-in-diff-Id well-order-on-Field well-order-on-def} by \text{fastforce}

} with $\sigma$ have $\exists ! \sigma. \text{minimal-good-strategy } v \sigma \text{ by } \text{blast}$

thus $?\text{thesis using theI' [of minimal-good-strategy } v, \text{folded choose-def] by blast}$

qed

corollary

assumes $v \in S$

shows \text{choose-good: choose } v \in \text{good } v$

and \text{choose-minimal: } \forall \sigma. \ (\sigma', \text{choose } v) \in r - \text{Id} \implies \sigma' \notin \text{good } v$

and \text{choose-strategy: strategy } p \ (\text{choose } v)$

using \text{choose-works}[\text{OF assms, unfolded minimal-good-strategy-def}] \text{good-strategies by blast+}$

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

lemma \text{well-ordered-strategy-valid: strategy } p \text{ well-ordered-strategy}$

proof -

{ fix $v$ assume $v \in S \ v \in VV \ p \text{-deadend } v$

moreover have \text{strategy } p \ (\text{choose } v)$

using \text{choose-works}[\text{OF } v \in S, \text{unfolded minimal-good-strategy-def}, \text{THEN conjunct1}] \text{good-strategies by blast}$

ultimately have $v \rightarrow (\lambda v. \text{choose } v) \ v \text{ using strategy-def by blast}$

}

thus $?\text{thesis unfolding well-ordered-strategy-def using valid-strategy-updates-set by force}$

qed

8.1 Strategies on a Path

Maps a path to its strategies.

definition \text{path-strategies} \equiv \text{lmap choose}$

lemma \text{path-strategies-in-Strategies:}$

assumes $\text{lset } P \subseteq S$

shows $\text{lset } (\text{path-strategies } P) \subseteq \text{Strategies}$

using \text{path-strategies-def assms choose-in-Strategies by auto}$

lemma \text{path-strategies-good:}$

assumes $\text{lset } P \subseteq S \ \text{enat } n < \text{length } P$

shows $\text{path-strategies } P \ s n \in \text{good } (P \ s n)$

by $(\text{simp add: path-strategies-def assms choose-good lset-nth-member})$

lemma \text{path-strategies-strategy:}$

assumes $\text{lset } P \subseteq S \ \text{enat } n < \text{length } P$
shows strategy p (path-strategies P $ n)
using path-strategies-good assms good-strategies by blast

lemma path-strategies-monotone-Suc:
assumes P: lset P ⊆ S valid-path P path-conforms-with-strategy p P well-ordered-strategy
e nat (Suc n) < llength P
shows (path-strategies P $ Suc n, path-strategies P $ n) ∈ r
proof -
define P' where P' = ldropn n P
  by (metis enat-ltl Suc ldrop-n Suc-ltl ldropn-Suc-conv-ldropn llist-disc(2) lnull-0-llength ltl-ldropn)
then obtain v w P's where v' = LCons v (LCons w P)
  by (metis ldropn-0 ldropn-Suc-conv-ldropn ldropn-llnull-0-llength)
moreover have lset P' ⊆ S unfolding P'-def using P(1) lset-ldropn-subset[of n P] by blast
ultimately have v ∈ S w ∈ S by auto
moreover have v→w using valid-path-edges"of v w P's, folded vw] valid-path-drop[OF P(2)] P'-def by blast
moreover have choose v ∈ good v using choose-good ⟨v ∈ S⟩ by blast
moreover have v ∈ VV P ⟹ choose v = w proof -
  assume v ∈ VV P
  moreover have path-conforms-with-strategy p P' well-ordered-strategy
  unfolding P'-def using path-conforms-with-strategy-drop P(3) by blast
  ultimately have well-ordered-strategy v = w using vw path-conforms-with-strategy-start by blast
  thus choose v = w unfolding well-ordered-strategy-def using ⟨v ∈ S⟩ by auto
qed

ultimately have choose v ∈ good w using strategies-continue by blast

hence ∗: (choose v, choose w) ∉ r - Id using choose-minimal ⟨w ∈ S⟩ by blast

have (choose w, choose v) ∈ r proof (cases)
  assume choose v = choose w
  thus ?thesis using r-refl refl-onD choose-in-Strategies[OF v ∈ S] by fastforce
next
  assume choose v ≠ choose w
  by (metis (lifting) Linear-order-in-diff-Id r-wo well-order-on-Field well-order-on-def)
qed

hence (path-strategies P' $ Suc 0, path-strategies P' $ 0) ∈ r
unfolding path-strategies-def using vw by simp
thus ?thesis unfolding path-strategies-def P'-def

by simp

lemma path-strategies-monotone:
assumes P: lset P ⊆ S valid-path P path-conforms-with-strategy p P well-ordered-strategy
n < m enat m < llength P
shows (path-strategies P $ m, path-strategies P $ n) ∈ r
using assms proof (induct m - n arbitrary: n m)
case (Suc d)
show \(?\)case proof (cases)
  assume \(d = 0\)
  thus \(?\)thesis using path-strategies-monotone-Suc[OF P(1,2,3)]
    by (metis (no-types) Suc.hyps(2) Suc.prems(4,5) Suc-diff-Suc Suc-inject Suc-le1 diff-is-0-eq
diff0-imp-equal)
next
  assume \(d \neq 0\)
  have \(m \neq 0\) using Suc.hyps(2) by linarith
  then obtain \(m'\) where \(m' = m\) using notD-implies-Suc by blast
  hence \(d = m' - n\) using Suc.hyps(2) by presburger
  moreover hence \(n < m'\) using \(d \neq 0\) by presburger
  ultimately have (path-strategies P $ m', path-strategies P $ n) \(\in\) r
    using Suc.hyps(1)[of \(m'\), OF P(1,2,3)] Suc.prems(5) dual-order.strict-trans-enat-onsimps(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 assms-def)
  qed
  qed simp

lemma path-strategies-eventually-constant:
  assumes \(-\)finite P let P \(\subseteq\) S valid-path P path-conforms-with-strategy P P well-ordered-strategy
  shows \(\exists\ n. \forall m \geq n\). path-strategies P $ n = path-strategies P $ m
proof -
  define \(\sigma\)-set where \(\sigma\)-set = lset (path-strategies P)
  have \(\exists\sigma, \sigma \in \sigma\)-set unfolding \(\sigma\)-set-def path-strategies-def
    using assms(1) finite-lmap lset-nth-member-inf by blast
  then obtain \(\sigma'\) where \(\sigma'\) \(\in\) \(\sigma\)-set \(\prod\) \(\tau\). \((\tau, \sigma') \in r\) \(\Rightarrow\) \(\tau \notin \sigma\)-set
    using wfE-min[of \(r - Id\) - \(\sigma\)-set] by auto
  obtain \(n\) where \(n\): path-strategies P $ n = \sigma'
    using \(\sigma'(1)\) lset-lnth[of \(\sigma'\)] unfolding \(\sigma\)-set-def by blast
  \{
    fix m assume \(n \leq m\)
    have \(\)path-strategies P $ n = path-strategies P $ m proof (rule contr)
      assume *: \(\)path-strategies P $ \(n \neq\) path-strategies P $ m
      with \((n \leq m)\) have \(n < m\) using le-imp-less-or-eq by blast
      with path-strategies-monotone have (path-strategies P $ m, path-strategies P $ n) \(\in\) r
        using assms by (simp add: infinite-small-lenghth)
      with * have (path-strategies P $ m, path-strategies P $ n) \(\in\) r - Id by simp
      with \(\sigma'(2)\) have path-strategies P $ m \(\notin\) \(\sigma\)-set by blast
      thus \(\)False unfolding \(\sigma\)-set-def path-strategies-def
        using assms(1) finite-lmap lset-nth-member-inf by blast
      qed
    }\}
  thus \(?\)thesis by blast
  qed
8.2 Eventually One Strategy

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

**lemma path-eventually-conforms-to-σ-map-n:**
  * assumes $lset P \subseteq S$ valid-path $P$ path-conforms-with-strategy $p$ $P$ well-ordered-strategy
  * shows $\exists n. \text{path-conforms-with-strategy } P \ (\text{ldropn } n \ P) \ (\text{path-strategies } P \ \# \ n)$

**proof (cases)**
  * assume $\text{finite } P$
    * then obtain $n$ where $\text{llength } P = \text{enat } n$ using $\text{finite-length-enat by blast}$
    * hence $\text{ldropn } n \ P = \text{LNil by simp}$
    * thus $\exists \sigma$ by $(\text{metis path-conforms-LNil})$
  * next
    * assume $\neg \text{finite } P$
      * then obtain $n$ where $\text{n : } \forall m. \ n \leq m \implies \text{path-strategies } P \ \# \ n = \text{path-strategies } P \ \# \ m$
        * using $\text{path-strategies-eventually-constant assms by blast}$
      * let $\sigma = \text{well-ordered-strategy}$
      * define $P'$ where $P' = \text{ldropn } n \ P$
        * fix $v$ assume $v \in lset P'$
          * hence $v \in S$ using $\text{lset } P \subseteq S$ $P'$-def in-lset-ldropn by fastforce
            * from $v \in lset P'$ obtain $m$ where $m: \text{enat } m < \text{llength } P' \ P' \ \# \ m = v$ by $(\text{meson in-lset-conv-llth})$
              * hence $P \ \# \ n \ \& \ m \ = \ v$ unfolding $P'$-def by $(\text{simp add: } \neg \text{finite } P, \text{finite-small-length})$
                * moreover have $\sigma v = \text{choose } v v$ unfolding well-ordered-strategy-def using $(v \in S)$ by auto
                  * ultimately have $\sigma v = (\text{path-strategies } P \ \# \ m \ \& \ n) v$
                    * unfolding $\text{path-strategies-def using } \text{finite-small-length} [OF \ (\neg \text{finite } P)]$ by simp
                      * hence $\sigma v = (\text{path-strategies } P \ \# \ n) v$ using $n[\text{of } m \ \& \ n]$ by simp
                }
          * moreover have $\text{path-conforms-with-strategy } p \ P' \ \text{well-ordered-strategy}$
            * unfolding $P'$-def by $(\text{simp add: assms(3) path-conforms-with-strategy-drop})$
              * ultimately show $\exists \sigma$ by $(\text{metis path-conforms-irrelevant-updates } P'$-def by blast)
      qed
end — WellOrderedStrategies

end

9 Winning Regions

**theory WinningRegion**

**imports**
  * Main
  * WinningStrategy

**begin**

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

**context ParityGame begin**
**Definition** winning-region \( p \equiv \{ v \in V. \exists \sigma. \text{strategy } p \sigma \land \text{winning-strategy } p \sigma v \} \)

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

**Lemma** winning-region-in-V \( \text{simp} \)
- **Assumes** \( v \in V p \text{ deadend } v \)
- **Shows** \( v \in \text{winning-region } p \)
- **Using** unfolding winning-region-def by blast

**Lemma** winning-region-deadends:
- **Assumes** \( v \in V p \text{ deadend } v \)
- **Shows** \( v \in \text{winning-region } p \)
- **Proof**

**Lemma** (in vmc-path) paths-stay-in-winning-region:
- **Assumes** \( \sigma': \text{strategy } p \sigma' \text{ winning-strategy } p \sigma' v_0 \)
- 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**

**9.1 Paths in Winning Regions**

**Lemma** (in vmc-path) paths-stay-in-winning-region:
- **Assumes** \( \sigma': \text{strategy } p \sigma' \text{ winning-strategy } p \sigma' v_0 \)
- 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**

**9.1 Paths in Winning Regions**

**Lemma** (in vmc-path) paths-stay-in-winning-region:
- **Assumes** \( \sigma': \text{strategy } p \sigma' \text{ winning-strategy } p \sigma' v_0 \)
- 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**

**Proof**
- **Fix** \( x \)**
- **Assume** \( x \in \text{let } P \)
- **Thus** \( x \in \text{winning-region } p \)**
- **Using** assms vmc-path-axioms
- **Proof (induct arbitrary: } v_0 \text{ rule: list-set-induct)***
  - **Case** \( (\text{find } P v_0) \)
    - **Interpret** vmc-path \( G P v_0 p \sigma \text{ using find.prem } (4) . \)
    - **Show** \( ? \text{ case using } P-v_0 \sigma' (1) \text{ find.prem } (2) v_0 p \text{ unfolding } \text{winning-region-def } \text{ by blast} \)
  - **Next**
    - **Case** \( (\text{step } P x v_0) \)
      - **Interpret** vmc-path \( G P v_0 p \sigma \text{ using step.prem } (4) . \)
      - **Show** \( ? \text{ case proof (cases)***
        - **Assume** \( \text{null } (l t l P) \)
          - **Thus** \( \text{thesis using } P-\text{null-ltl-LCons step.hyp } (2) \text{ by auto} \)
        - **Next**
          - **Assume** \( \text{null } (l t l P) \)
            - **Then interpret** vmc-path-no-deadend \( G P v_0 p \sigma \text{ using } P-\text{no-deadend-v0 } \text{ by unfold-locale} \)
              - **Have** \( \text{winning-strategy } p \sigma' v_0 \text{ proof (cases)***
                - **Assume** \( v_0 \in V V p \)
                  - **Hence** \( \text{winning-strategy } p \sigma' (v_0 v_0) \)
                    - **Using** strategy-extends-VVp local.step (4) step.prem (2) v0-no-deadend by blast
                      - **Moreover have** \( \sigma v_0 = w_0 \text{ using v0-conforms } (v_0 v_0 \in V V p) \text{ by blast} \)
                      - **Moreover have** \( \sigma' v_0 = \sigma v_0 \)
                        - **Using** \( \sigma \text{ assms } (1) \text{ step.prem } (2) v_0 \text{ unfolding } \text{winning-region-def } \text{ by blast} \)
                          - **Ultimately show** \( \text{thesis by simp} \)
                        - **Next**
                          - **Assume** \( v_0 / \in V V p \)
                            - **Thus** \( \text{thesis using } v_0 V \text{ strategy-extends-VVpstar step } (4) \text{ step.prem } (2) \text{ by simp} \)
                            - **Qed**
thus \( ?\)thesis using \( \text{step.hyps(3) step(4)} \) \( \sigma \) \( \text{vmc-path-lll} \) by blast

qed

qed

lemma \((\text{in vmc-path})\) \( \text{path-hits-winning-region-is-winning} \):
assumes \( \sigma' : \text{strategy } p \sigma' \land v, v \in \text{winning-region } p \implies \text{winning-strategy } p \sigma' v \)
and \( \sigma : \land v, v \in \text{winning-region } p \implies \sigma' v = \sigma v \)
and \( P : \text{let } P \cap \text{winning-region } p \neq \{\} \)
shows \( \text{winning-path } p P \)

proof
obtain \( n \) where \( n : \text{enat } n < \ell\text{length } p P \) \( n \in \text{winning-region } p \)
using \( p \) by \((\text{meson lset-intersect-lnth})\)
define \( P' \) where \( P' = \text{idropn } n P \)
then interpret \( P' : \text{vmc-path } G P' P \) \( n \in P \sigma \)
unfolding \( P' \text{-def using } \text{vmc-path-idropn } n(1) \) by blast
have \( \text{winning-strategy } p \sigma'(P \) \( n) \) \( \text{using } \sigma'(2) \) \( n(2) \) by blast
hence \( \text{let } P' \subseteq \text{winning-region } p \)
using \( P' \text{-paths-stay-in-winning-region}[OF } \sigma'(1) - \sigma] \)
by blast
hence \( \land v, v \in \ell P' \implies \sigma v = \sigma' v \) \( \text{using } \sigma \) by auto
hence \( \text{path-conforms-with-strategy } p P' \sigma' \)
using \( \text{path-conforms-with-strategy-irrelevant-updates } P' P\text{-conforms} \)
by blast
then interpret \( P' : \text{vmc-path } G P' P \) \( n \in P \sigma \)
using \( P' \text{-conforms-to-another-strategy by blast} \)
have \( \text{winning-path } p P' \) \( \text{using } \sigma'(2) \) \( n(2) P' \text{,vmc-path-axioms winning-strategy-def by blast} \)
thus \( \text{winning-path } p P' \) \( \text{unfolding } P' \text{-def using } \text{winning-path-drop-add } n(1) P\text{-valid by blast} \)
qed

9.2 Irrelevant Updates

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

lemma \( \text{winning-strategy-updates} \):  
assumes \( \sigma : \text{strategy } p \sigma \) \( \text{winning-strategy } p \sigma v0 \)
and \( v : v \not\in \text{winning-region } p v \rightarrow w \)
shows \( \text{winning-strategy } p (\sigma(v := w)) v0 \)

proof
fix \( P \) assume \( \text{vmc-path } G P \) \( v0 p (\sigma(v := w)) \)
then interpret \( \text{vmc-path } G P \) \( v0 p (\sigma(v := w)) \) .
have \( \land v', v' \in \text{winning-region } p \implies \sigma v' = (\sigma(v := w)) v' \) \( \text{using } v \) by auto
hence \( v \not\in \ell P \text{ using } v \text{paths-stay-in-winning-region } \sigma \text{ unfolding } \text{winning-region-def by blast} \)
thus \( \text{path-conforms-with-strategy } p P \sigma \)
using \( \text{P-conforms path-conforms-with-strategy-irrelevant' by blast} \)
thus \( \text{winning-path } p P \) \( \text{using } \text{conforms-to-another-strategy } \sigma(2) \) \( \text{winning-strategy-def by blast} \)
qed

9.3 Extending Winning Regions

lemma \( \text{winning-region-extends-VVp} \):  
assumes \( v : v \in V V p v \rightarrow w \) and \( w : w \in \text{winning-region } p \)
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 contr: 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 contains v 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
  AttractingStrategy 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. σ ∈ good v =⇒ strategy p σ unfolding good-def by blast
next
fix v w σ assume v: v ∈ S − W w → 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
hence strategy-attracts-via $p$ $\sigma$ $w$ $SW$ using strategy-attracts-via-successor $v$ by blast
thus $\sigma \in$ good $w$ unfolding good-def using $\sigma(1)$ by blast

qed (insert r)

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

{  
fix $v0$ assume $v0 \in S$
fix $P$ assume $P$: vmc-path $G$ $P$ $v0$ $p$ well-ordered-strategy
then interpret vmc-path $G$ $P$ $v0$ $p$ well-ordered-strategy .
have visits-via $P$ $S$ $W$ proof (rule contr)
  assume contra: $\neg$visits-via $P$ $S$ $W$
  have $S$-$W$-no-deadends: $\forall v. v \in S - W \implies \neg$deadend $v$
thus $\sigma \in$ good $w$ unfolding good-def using $\sigma(1)$ by blast

{  
fix $v0$ assume $v0 \in S$
fix $P$ assume $P$: vmc-path $G$ $P$ $v0$ $p$ well-ordered-strategy
then interpret vmc-path $G$ $P$ $v0$ $p$ well-ordered-strategy .
have visits-via $P$ $S$ $W$ proof (rule contr)
  assume contra: $\neg$visits-via $P$ $S$ $W$
  have $S$-$W$-no-deadends: $\forall v. v \in S - W \implies \neg$deadend $v$
thus $\sigma \in$ good $w$ unfolding good-def using $\sigma(1)$ by blast

have $\neg$finite $P$ proof
  assume $\neg$finite $P$
  hence deadend ($llast P$) using $P$-maximal $P$-not-null maximal-ends-on-deadend by blast
  moreover have $llast P \in S - W$ using $\neg$finite $P$ $\neg$finite-$lset$
by blast
  ultimately show False using $S$-$W$-no-deadends by blast
qed

obtain $n$ where $n$: path-conforms-with-strategy $p$ ($ldropn n P$) (path-strategies $P$ $\subseteq$ $n$)
using path-eventually-conforms-to-$\sigma$-map-$n$[OF $\forall S$ $W$ $P$-valid $P$-conforms]
by blast

define $\sigma'$ where [simp]: $\sigma' =$ path-strategies $P$ $\subseteq$ $n$
define $P'$ where [simp]: $P' = ldropn n P$

53
interpret \texttt{mnc-path} \( G \) \( P' \) \( \mathbf{bld} \) \( P' \) \( p \sigma' \)

proof
\begin{itemize}
\item show \( \lnot \text{null} \) \( P' \) unfolding \( P' \)-def
  \begin{itemize}
  \item using \( \lnot \text{finite} \) \( P \) \( \text{finite-idropn null-imp-finite} \) by \texttt{blast}
  \end{itemize}
\end{itemize}
\texttt{qed} (simp-all add; \( n \))

have strategy \( p \sigma' \) unfolding \( \sigma' \)-def
\begin{itemize}
\item using path-strategies-strategy \( \{ \text{set} \ P \subseteq S - W \} \) \( \lnot \text{finite} \) \( P \) \( \text{infinite-small-length} \) by \texttt{blast}
\end{itemize}
moreover have strategy-attracts-via \( p \sigma' \) \( \{ \text{bld} \ P' \} \) \( S \) \( W \) proof —
\begin{itemize}
\item have \( P \) \( \text{\& n} \) \( \in \) \( S - W \) using \( \{ \text{set} \ P \subseteq S - W \} \) \( \lnot \text{finite} \) \( P \) \( \text{set-nth-member-inf} \) by \texttt{blast}
\item hence \( \sigma' \) \( \in \) \( \text{good} \) \( (P \) \( \text{\& n} \) \)
  \begin{itemize}
  \item using path-strategies-good \( \sigma' \)-def \( \lnot \text{finite} \) \( P \) \( \{ \text{set} \ P \subseteq S - W \} \) by \texttt{blast}
  \end{itemize}
\item hence strategy-attracts-via \( p \sigma' \) \( (P \) \( \text{\& n} \) \) \( S \) \( W \) unfolding good-def by \texttt{blast}
\end{itemize}
thus \( \text{thesis} \) unfolding \( P' \)-def using \( P \cdot 0 \) by \( \text{(simp add; } \lnot \text{finite} \) \( P \) \( \text{infinite-small-length}) \)
\texttt{qed}

moreover from \( \{ \text{set} \ P \subseteq S - W \} \) have \( \text{\& s} \subseteq S - W \)
\begin{itemize}
\item unfolding \( P' \)-def using \( \text{idropn-subset}[\text{of } n \ P] \) by \texttt{blast}
\end{itemize}
ultimately show \( \text{False} \) using strategy-attracts-via-set by \texttt{blast}
\texttt{qed}

\}
thus \( \text{thesis} \) using \( \text{well-ordered-strategy-valid} \) by \texttt{blast}
\texttt{qed}

\subsection{10.2 A Uniform Winning Strategy}

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

\textbf{lemma} merge-winning-strategies:
\begin{itemize}
\item shows \( \exists \sigma. \) strategy \( p \sigma \land (\forall v \in \text{winning-region } p. \) winning-strategy \( p \sigma v) \)
\item proof —
\item define \textit{good} where \textit{good} \( v = \{ \sigma. \) strategy \( p \sigma \land \text{winning-strategy } p \sigma v \} \) \textit{for} \( v \)
\item let \( ?G = \{ \sigma. \exists v \in \text{winning-region } p. \sigma \in \text{good } v \} \)
\item obtain \( r \) where \( r: \text{well-order-on } ?G \) \( r \) using \( \text{well-order-on} \) by \texttt{blast}
\end{itemize}

\begin{itemize}
\item have no-\( \text{VVp-deadends}: \forall v. [ v \in \text{winning-region } p; v \in \text{VV } p ] \implies \lnot \text{deadend } v \)
\item using no-winning-strategy-on-deadends unfolding winning-region-def by \texttt{blast}
\end{itemize}

\begin{itemize}
\item interpret \texttt{WellOrderedStrategies} \( G \) \( \text{winning-region } p \) \( p \) \( \text{good } r \) proof
\item show \( \forall v. v \in \text{winning-region } p \implies \exists \sigma. \sigma \in \text{good } v \)
\item unfolding good-def winning-region-def by \texttt{blast}
\end{itemize}

\begin{itemize}
\item next
\item show \( \forall v. \sigma. \sigma \in \text{good } v \implies \text{strategy } p \sigma \text{ unfolding good-def} \) by \texttt{blast}
\item next
\item fix \( v \) \( w \) \( \sigma \) assume \( v: v \in \text{winning-region } p \implies w \) \( v \in \text{VV } p \implies \sigma \) \( v = w \) \( \sigma \in \text{good } v \)
\item hence \( \sigma: \) strategy \( p \sigma \) winning-strategy \( p \sigma v \) unfolding good-def by simp-all
\item hence winning-strategy \( p \sigma w \) proof (cases)
\item assume \( \ast: v \in \text{VV } p \)
\item hence \( \ast\ast: \sigma v = w \) using \( v(3) \) by \texttt{blast}
\item have \( \lnot \text{deadend } v \) using no-\( \text{VVp-deadends} \) \( (v \in \text{VV } p) \) \( v(1) \) by \texttt{blast}
\item with \( \ast \ast \) show \( \text{thesis} \) using strategy-extends-\( \text{VVp } \sigma \) by \texttt{blast}
\item next
\item assume \( v \notin \text{VV } p \)
\end{itemize}
thus \textit{thesis} using \textit{strategy-extends-VVpstar} \(\sigma \langle v \rightarrow w \rangle\) by blast

\textbf{qed}

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

\textbf{qed} (insert \textit{winning-region-in-Vr})
hence winning-strategy $p \sigma' (P \& n)$ unfolding good-def by blast
thus $\exists \sigma$ unfolding $P'$-def using $P$-def ($\neg$finite $P$) by (simp add: infinite-small-length lhd-idropn)
  qed
ultimately have winning-path $p$ $P'$ unfolding winning-strategy-def
  using $P'$,vc-m 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 $\exists \sigma$ 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-$VV p^{**}$:
  assumes $v : v \in V V p^{**}$ and $w : \wedge v' \in \sigma \Rightarrow w \in \text{winning-region } p$
  shows $v \in \text{winning-region } p$
proof-
  obtain $\sigma$ where $\sigma : \text{strategy } p \sigma \wedge v \in \text{winning-region } p \Rightarrow \text{winning-strategy } p \sigma v$
    using merge-winning-strategies by blast
  have $\text{winning-strategy } p \sigma v$ using strategy-extends-backwards-$VV p^{**}$[OF $\sigma(1)$] $\sigma(2)$ $w$ by blast
  thus $\exists \exists \sigma$ unfolding $\text{winning-region-def}$ using $\sigma(1)$ by blast
  qed

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

lemma removing-winning-region-induces-no-deadends:
  assumes $v : v \in V - \text{winning-region } p - \text{deadend } v$
  shows $\exists w \in V - \text{winning-region } p, v \leadsto w$
  using assms winning-region-extends-$VV p$ winning-region-extends-$VV p^{**}$ 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 $v0$: $v0 \in VV p v0 \in directly-attracted p S v0 \notin S$
shows $\exists \sigma$. strategy $p \sigma \land$ strategy-attracts-via $p \sigma v0$ (insert $v0 S$) $W$

proof –
from $v0(1,2)$ obtain $w$ where $v0 \rightarrow w w \in S$ using directly-attracted-def by blast
from $w \in S$ $\sigma(2)$ have strategy-attracts-via $p \sigma w S W$ unfolding strategy-attracts-def by blast
let $? = \sigma(v0 := w)$ — Extend $\sigma$ to the new node.
have strategy $p ?\sigma$ using $\sigma(1)$ $(v0 \rightarrow w)$ valid-strategy-updates by blast
moreover have strategy-attracts-via $p ?\sigma v0$ (insert $v0 S$) $W$ proof

fix $P$
assume vmc-path $G P v0 p ?\sigma$
then interpret vmc-path $G P v0 p ?\sigma$.
have $\neg$deadend $v0$ using $(v0 \rightarrow w)$ by blast
then interpret vmc-path-no-deadend $G P v0 p ?\sigma$ by unfold-locales
define $P''$ where $\exists [\text{simp}]; P'' = \text{ltl } P$
have $\exists [\text{simp}]; P'' = w$ using $v0(1)$ $v0$-conforms $w0$-def by auto
hence vmc-path $G P'' w p ?\sigma$ using vmc-path-llt by $(\text{simp add: } w0$-def $)$

have $\exists [\text{simp}]; P'' = w$ using $v0(1)$ $v0$-conforms $w0$-def by auto
hence strategy-attracts $p ?\sigma S W$
using strategy-attracts-inrelevant-overide $[\text{OF } \sigma(2,1) \text{ strategy } p ?\sigma]$ by simp
hence strategy-attracts-via $p ?\sigma w S W$ unfolding strategy-attracts-def
using $(w \in S)$ by blast
hence visits-via $P'' S W$ unfolding strategy-attracts-via-def
using vmc-path $G P'' w p ?\sigma$ by blast
thus visits-via $P (\text{insert } v0 S) W$
using visits-via-LCons[of ltl $P S W$ $v0]$ $P$-LCons by simp
qed
ultimately show !thesis by blast
qed

lemma strategy-attracts-extends-VVpstar:
assumes $\sigma$: strategy-attracts $p \sigma S W$
and $v0$: $v0 \notin VV p v0 \in directly-attracted p S$
shows strategy-attracts-via $p \sigma v0$ (insert $v0 S$) $W$

proof
fix $P$
assume vmc-path $G P v0 p \sigma$
then interpret vmc-path $G P v0 p \sigma$.
have $\neg$deadend $v0$ using $v0(2)$ directly-attracted-contains-no-deadends by blast
then interpret vmc-path-no-deadend $G P v0 p \sigma$ by unfold-locales
have visits-via $(ltt P) S W$
using vmc-path strategy-attractsE $[\text{OF } \text{vmc-path-llt } \sigma] v0$ directly-attracted-def by simp
thus visits-via $P (\text{insert } v0 S) W$ unfolding strategy-attracts-via-def
using visits-via-LCons[of ltl $P S W$ $v0]$ $P$-LCons by simp
qed

lemma attractor-has-strategy-single:
assumes $W \subseteq V$
and $v0$-def: $v0 \in$ attractor $p W$ $(is \ - \ ?A)$
shows $\exists \sigma. \text{strategy } p \sigma \land \text{strategy-attracts-via } p \sigma v_0 ?A W$
using assms proof (induct arbitrary; $v_0$ rule: attractor-set-induction)
case (step $S$)
have $v_0 \in W \implies \exists \sigma. \text{strategy } p \sigma \land \text{strategy-attracts-via } p \sigma v_0 \{\}$ $W$
using strategy-attracts-via-trivial valid-arbitrary-strategy by blast
moreover {
assume $*$: $v_0 \in \text{directly-attracted } p S v_0 \notin S$
from assms (1) step.hyps (1) step.hyps (2)
have $\exists \sigma. \text{strategy } p \sigma \land \text{strategy-attracts } p \sigma S W$
using merge-attractor-strategies by auto
with $*$
have $\exists \sigma. \text{strategy } p \sigma \land \text{strategy-attracts-via } p \sigma v_0 \{\text{insert } v_0 S\} W$
using strategy-attracts-extends-VVp strategy-attracts-extends-VVpstar by blast
}
ultimately show ?case
using step.prems step.hyps (2)
attactor-strategy-on-extends [of $p - v_0$ insert $v_0$ $S W W \cup S \cup \text{directly-attracted } p S$]
attactor-strategy-on-extends [of $p - v_0$ $S W W \cup S \cup \text{directly-attracted } p S$]
attactor-strategy-on-extends [of $p - v_0 \{\}$ $W W S \cup S \cup \text{directly-attracted } p S$]
by blast
next
case (union $M$)
hence $\exists S. S \in M \land v_0 \in S$ 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 $\exists \sigma. \text{strategy } p \sigma \land \text{strategy-attracts } p \sigma \{\text{attractor } p W\} W$
proof –
let ?A = attractor $p W$
have $?A \subseteq V$ by (simp add: $W \subseteq V$ attractor-in-V)
moreover
have $\forall v. v \in ?A \implies \exists \sigma. \text{strategy } p \sigma \land \text{strategy-attracts-via } p \sigma v ?A W$
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

58
12.1 Induction Step

The proof of positional determinacy is by induction over the size of the finite set $\omega \ni V$, 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: $\forall v. v \in V \Rightarrow \neg \text{deadend } v$
- and IH: $\forall (G :: (\langle a, b \rangle) \text{ ParityGame-scheme}) v. [ \text{card } (\omega G \cdot V G) < \text{card } (\omega \cdot V); v \in V G; \text{ParityGame } G; \forall v. v \in V G \Rightarrow \neg \text{Digraph.deadend } G v ] \Rightarrow \exists p. v \in \text{ParityGame.winning-region } G p$

- Shows $\exists p. v \in \text{winning-region } p$

**Proof**

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

- Define $\text{min-prio}$ where $\text{min-prio} = \text{Min } (\omega \ni V)$
- Have $\exists p. \text{winning-priority } p \text{ min-prio by auto}$
- Then obtain $p$ where $p. \text{winning-priority } p \text{ min-prio by blast}$

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 $K = U \cap (\omega - \{ \text{min-prio} \})$
- Define $V'$ where $V' = U - \text{Attractor } p K$

- Define $G'$ where $[\text{simp}]: G' = \text{subgame } V'$
- Interpret $G': \text{ParityGame } G' \text{ using subgame-ParityGame by simp}$

- Have $U$-equiv: $\forall v. v \in V \Rightarrow v \in U \iff v \notin \text{winning-region } p^{**}$
- Unfolding $U$-def $W1$-def by blast

- Have $V' \subseteq V$ unfolding $U$-def $V'$-def by blast
- Hence $[\text{simp}]: V_{G'} = V'$ unfolding $G'$-def by simp

- Have $V_{G'} \subseteq V \ E_{G'} \subseteq E \omega_{G'} = \omega$ unfolding $G'$-def by (simp-all add: subgame-$\omega$)
- Have $G'.VVp = V' \cap VVp$ unfolding $G'$-def using subgame-$VV$ by simp
have $V$-decomp: $V = \text{attractor } p K \cup V' \cup WI$ proof –
have $V \subseteq \text{attractor } p K \cup V' \cup WI$
unfolding $V'$-def $U$-def by blast
moreover have attractor $p K \subseteq V$
using attractor-in-$V'$[of $K$] unfolding $K$-def $U$-def by blast
ultimately show $\exists$thesis
unfolding $W1$-def winning-region-def using $\langle V' \subseteq V \rangle$ by blast
qed

have $G'$-no-deadends: $\forall v. v \in V_{G'} \implies \neg G'.\text{deadend } v$ proof –
fix $v$ assume $v \in V_{G'}$
hence $\ast: v \in U \implies \text{attractor } p K$ using $\langle V_{G'} = V' \rangle$ $V'$-def by blast
moreover hence $\exists w \in U. v \mapsto w$
using removing-winning-region-induces-no-deadends[of $v \ast$] no-deadends $U$-equiv $U$-def by blast
moreover have $\forall w. \{ v \in V Vp; v \mapsto w \} \implies w \in U$
using $\ast$ $U$-equiv $\text{winning-region-extends}$-$VVp$ by blast
ultimately have $\exists w \in V'. v \mapsto w$
using $U$-equiv $\text{winning-region-extends}$-$VVp$ removing-attractor-induces-no-deadends $V'$-def by blast
thus $\neg G'.\text{deadend } v$ using $\langle v \in V_{G'} \rangle \langle V' \subseteq V \rangle$ by simp
qed

By definition of $WI$, we obtain a winning strategy on $WI$ for player $p^\ast$.

obtain $\sigma W1$ where $\sigma W1$:
strategy $p^\ast \sigma W1 \forall v. v \in WI \implies$ winning-strategy $p^\ast \sigma W1 v$
unfolding $WI$-def using merge-winning-strategies by blast

$\{ $
fix $v$ assume $v \in V_{G'}$

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

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

It turns out the winning region of player $p^\ast$ is empty, so we have a strategy for player $p$

have $v \in G'.\text{winning-region } p$ proof (rule contr)
This concludes the proof of $lset P \subseteq V'$.

**hence $G'$, valid-path $P$ using subgame-valid-path by simp**

**moreover have $G'$, maximal-path $P$**
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 \( \sigma 1 \)**

**Where \( \sigma 1: \text{strategy} \ p \ \sigma 1 \)**

- \( \text{strategy-attracts} \ p \ \sigma 1 \ (\text{attractor} \ p \ K) \ K \)
- \( \text{using} \ \text{attractor-has-strategy}[of \ K \ p] \ K\text{-def} \ U\text{-def} \ \text{by auto} \)

Next, on \( G' \) we follow the winning strategy whose existence we proved earlier.

**Have \( G'.\text{winning-region} \ p = V_{G'} \)** **Using** **Recursion** **Unfolding** **G'.\text{winning-region-def}** **By blast**

**Then obtain \( \sigma 2 \)**

**Where \( \sigma 2: \ \lnot \forall v, v \in V_{G'} \implies G'.\text{strategy} \ p \ \sigma 2 \)**

**Using** **G'.merge-winning-strategies** **By blast**

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

**Define** **suc c** **Where** **suc c \( v = (\text{SOME} w, v \mapsto w \land (v \in W1 \lor v \not\in W1)) \)** **For** \( v \)

Compose the three pieces.

**Define** **\( \sigma \)** **Where** **\( \sigma = \text{override-on} (\text{override-on suc c} \sigma 2 V') \ \sigma 1 \ (\text{attractor} \ p \ K - K) \)**

**Have** **attractor \ p \ K \cap W1 = {}** **Proof** **(rule contr)**
Assume \( \text{attractor } p \ K \cap W1 \neq \{\} \)

Then obtain \( v \) where \( v\in \text{attractor } p \ K \ v \in W1 \) by blast

Hence \( v \in V \) using \( \text{WL-def winning-region-def} \) by blast

Obtain \( P \) where \( \text{vmc2-path } G \ P \ v \ p \ \sigma I \ \sigma W1 \)

Using \( \text{strategy-conforming-path-exists } \sigma W1(1) \ \sigma I(1) \ v \in V \) by blast

Then interpret \( \text{vmc2-path } G \ P \ v \ p \ \sigma I \ \sigma W1 \).

Having \( \text{strategy-attracts-via } p \ \sigma I \ v \ (\text{attractor } p \ K) \ K \text{ using } v(1) \ \sigma I(2) \) \( \text{strategy-attracts-def} \) by blast

Hence \( \text{let } P \cap K \neq \{\} \) using \( \text{strategy-attracts-viaE visits-visits} \) by blast

Hence \( \neg \text{let } P \subseteq W1 \text{ unfolding } K\text{-def } U\text{-def} \) by blast

Thus \( \text{False unfolding } \text{WL-def} \) using \( \text{comp.path-stay-in-winning-region } \sigma W1 \ v(2) \) by auto

Qed

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

- \( \text{have } \sigma: \sigma = 1 \)
  - \( \exists v, v \in \text{attractor } p \ K - K \implies \sigma = 1 \ v \text{ unfolding } \sigma\text{-def} \) by simp
  - \( \text{have } \sigma: \sigma = 2 \)
  - \( \exists v, v \in V' \implies \sigma = 2 \ v \text{ unfolding } \sigma\text{-def } V'\text{-def} \) by auto

- \( \text{have } \sigma: \sigma = 3 \)
  - \( \exists v, v \in K \cup W1 \implies \sigma = v \text{ succ } v \text{ proof} - \)
    - Fix \( v \) assume \( v, v \in K \cup W1 \)
    - Hence \( v \notin V' \text{ unfolding } V'\text{-def} \text{ U-def using } \text{attractor-set-base} \) by auto

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

By \( \text{metis} (\text{mono-tags, lifting}) \text{ Diff-if IntI UnE override-on-def override-on-emptyset} \)

Qed

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

\[
\{ \text{fix } v \text{ assume } v, v \in V V p \}
\]

Hence \( \neg \text{deadend } v \text{ using } \neg \text{deadend}\) by blast

Have \( \exists w, v \rightarrow w \land (v \in W1 \lor w \notin W1) \) proof (cases)

Assume \( v \in W1 \)

Thus \( \text{thesis using } \neg \text{deadend } v \) by blast

Next

Assume \( v \notin W1 \)

Show \( \text{thesis proof (rule ccontr)} \)

Assume \( \neg (\exists w, v \rightarrow w \land (v \in W1 \lor w \notin W1)) \)

Hence \( \exists w, v \rightarrow w \implies \text{winning-strategy } p** \sigma W1 w \text{ using } \sigma W1(2) \text{ by blast} \)

Hence \( \text{winning-strategy } p** \sigma W1 v \)

Using \( \text{strategy-extends-backwards-VVpstar } \sigma W1(1) \) \( \forall v \in V V p \) by simp

Hence \( v \in W1 \text{ unfolding } \text{WL-def} \text{ winning-region-def } \) \( \sigma W1(1) \neg \text{deadend } v \) by blast

Thus \( \text{False using } \neg \text{deadend } v \text{ by blast} \)

Qed

Qed

Hence \( v \rightarrow \text{succ } v \ v \in W1 \lor \text{succ } v \notin W1 \text{ unfolding succ-def} \)

Using \( \text{someI-ex } [\lambda w. v \rightarrow w \land \{v \in W1 \lor w \notin W1\}] \) by blast+

\} \text{ note succ-works = this} \]

Have \( \text{strategy } p \ \sigma \)

Proof

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

Hence \( v \in \text{attractor } p \ K - K \implies v \rightarrow \sigma v \) using \( \sigma - \sigma I \ \sigma I(1) \ v \text{ unfolding } \text{strategy-def} \) by auto

Moreover have \( v \in V' \implies v \rightarrow \sigma v \text{ proof} - \)

Assume \( v \in V' \)

Moreover have \( v \in V G \) using \( \forall v \in V' (V G' = V') \text{ by blast} \)

63
moreover have \( v \in G' \setminus V V p \) using \( (G' \setminus V V p = V' \cap V V p) \) \( \langle v \in V' \rangle \) \( \langle v \in V V p \rangle \) by blast

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

ultimately have \( v \rightarrow G', \sigma 2 v \) using \( \sigma 2(1) \) \( G' \text{-strategy-def [of } \sigma \sigma 2] \) by blast

with \( \langle v \in V' \rangle \) show \( v \rightarrow \sigma v \) using \( (E G', \subseteq \langle E \rangle \sigma \sigma 2) \) by \( \text{metis subsetCE} \)

qed

moreover have \( v \in K \cup WI \implies v \rightarrow \sigma v \) using \( \text{suc works}(1) \) \( \sigma -K \) by auto

moreover have \( v \in V \) using \( \langle v \in V V p \rangle \) by blast

ultimately show \( v \rightarrow \sigma v \) using \( V \text{-decomp by blast} \)

qed

have \( \sigma \text{-attracts: strategy-attracts } p \sigma \) (attractor \( p K \)) \( K \) proof–

have strategy-attracts \( p \) (override-on \( \sigma \sigma 1 \) \( \text{attractor } p K - K \)) \( \text{(attractor } p K \)) \( K \)

using strategy-attracts-irrelevant-override \( \sigma 1 \) strategy \( p \sigma \) by blast

moreover have \( \sigma = \text{override-on } \sigma \sigma 1 \) \( \text{(attractor } p K - K \) )

by (rule \( \text{equiv} \) ) (simp add: override-on-def \( \sigma \sigma 1 \))

ultimately show \?thesis by simp

qed

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

have \( \forall v \in V - WI \) winning-strategy \( \sigma v \) \( \sigma v \) \( \text{proof} \) (intro ballI winning-strategyI)

fix \( v P \) assume \( P \) \( v \in V - WI \) \( \text{vmc-path } G P v p \sigma \)

interpret \( \text{vmc-path } G P v p \sigma \) using \( P(2) \).

have \( \text{lset } P \subseteq V - WI \)

proof (induct rule: \( \text{vmc-path-lset-induction-closed-subset} \))

fix \( v \) assume \( \langle v \in V - WI \rangle \neg \text{deadend } v \) \( v \in V V p \)

show \( \sigma v \in V - WI \cup \{ \} \) \( \text{proof} \) (rule \( \text{constr} \))

assume \( \neg \?thesis \)

hence \( \sigma v \in WI \)

using \( \langle \text{strategy } p \sigma \rangle \neg \text{deadend } v \) \( \langle v \in V V p \rangle \)

unfolding \( \text{strategy-def by blast} \)

hence \( v \notin K \) using \( \text{suc works}(2) \) [OF \( \langle v \in V V p \rangle \) \( \langle v \in V - WI \rangle \) \( \sigma -K \) ] by auto

moreover have \( v \notin \text{attractor } p K - K \) proof

assume \( v \in \text{attractor } p K - K \)

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

using \( \text{attracted-strategy-step \langle strategy } p \sigma \rangle \sigma \text{-attracts } \neg \text{deadend } v \) \( \langle v \in V V p \rangle \)

attractor-set-base

by blast

thus \( \text{False using } (\sigma v \in WI) \) \( \text{attractor } p K \cap WI = \{ \} \) by blast

qed

moreover have \( v \notin V' \) proof

assume \( v \in V' \)

have \( \sigma 2 v \in V G \) proof (rule \( G' \text{-valid-strategy-in-V[of } p \sigma 2 v \) )

have \( v \in V G \) using \( \langle V G = V' \rangle \) \( \langle v \in V' \rangle \) by simp

thus \( \neg G' \text{-deadend } v \) using \( G' \text{-no-deadends by blast} \)

show \( G' \text{-strategy } p \sigma 2 \) using \( \sigma 2(1) \) \( \langle v \in V G \rangle \) by blast

show \( v \in G' \setminus V V p \) using \( \langle v \in V V p \rangle \) \( \langle G' \setminus V V p = V' \cap V V p \rangle \) \( \langle v \in V' \rangle \) by simp

qed

hence \( \sigma v \in V G \) using \( \langle v \in V' \rangle \) \( \sigma \sigma 2 \) by simp

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

qed

ultimately show \( \text{False using } \langle v \in V - WI \rangle V \text{-decomp by blast} \)
qed
next

fix v w assume v ∈ V − WI ¬deadend v v ∈ V* v → w

show w ∈ V − WI ∪ { }

proof (rule contr)
  assume ¬thesis
  hence w ∈ WI using (v → w) by blast
  let ?σ = σWI(v := w)
  have winning-strategy p** σ WI w using (w ∈ WI) σ WI (2) by blast
  moreover have ¬(∃σ, strategy p** σ ∧ winning-strategy p** σ v)
    using (v ∈ V − WI) unfolding WI-def winning-region-def by blast
  ultimately have winning-strategy p** ?σ w
    using winning-strategy-updates [of p** σ WI w v w] σ WI (1) (v → w)
  unfolding winning-region-def by blast
  moreover have strategy p** ?σ using (v → w) σ WI (1) valid-strategy-updates by blast
  ultimately have winning-strategy p** ?σ v
    using strategy-extends-backwards-VVp [of v p** ?σ w]
    (v ∈ V* p** (v → w)) by auto
  hence v ∈ WI unfolding WI-def winning-region-def
    using (strategy p** ?σ) (v ∈ V − WI) by blast
  thus False using (v ∈ V − WI) 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 finite-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 ′: vme-path G′ P ′ lhd P′ p σ2 proof
show ¬null P′ unfolding P′-def
  using ¬finite P finite-ldropn 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
The second case: P

qed

show G'.maximal-path P' proof
  have maximal-path P' unfolding P'-def by simp
  thus ?thesis using subgame-maximal-path \{set P' ⊆ V' \land V' ⊆ V\} G'-def by blast
qed

show G'.path-conforms-with-strategy p P' σ 2 proof
  have path-conforms-with-strategy p P' σ unfolding P'-def by simp
  hence path-conforms-with-strategy p P' σ 2
    using path-conforms-with-strategy-irrelevant-updates \{set P' ⊆ V' \land σ 2\}
    by blast
  thus ?thesis
    using subgame-path-conforms-with-strategy \{set P' ⊆ V' \land V' ⊆ V\} G'-def
    by blast
qed

qed simp

have G'.winning-strategy p σ 2 (ldropn P')
  using \{set P' ⊆ V' \land vmc-path'.P-not-null σ 2\} \land \{ldropn P' \land V' = V' \land list.set-set\{1\}\}
  by blast
hence G'.winning-path p P' using G'.winning-strategy-def vmc-path'.vmc-path-axioms by blast
moreover have G'.VV p** ⊆ VV p** unfolding G'-def using subgame-VV by simp
ultimately have winning-path p P'
  using G'.winning-path-supergame[of p P' G] \omega G' = \omega \ ParityGame-axioms by blast
thus ?thesis
  unfolding P'-def
  using infinite-small-length[of P p n] P-valid
  winning-path-drop-add[of p P n] P-valid
  by blast
next
assume asm: \neg(\exists n. \ lset \{ldropn n P\} ⊆ V')

The second case: P visits K infinitely often. Then min-prio occurs infinitely often on P.

have min-prio ∈ path-inf-priorities P
unfold path-inf-priorities-def proof (intro Collect1 all1)
  fix n
  obtain k1 where k1: ldropn n P $ k1 \notin V' using asm by (metis list.set-lhd subsetI)
  define k2 where k2 = k1 + n
  interpret vmc-path G ldropn k2 P $ k2 p σ
  using vmc-path-ldropn infinite-small-length (~\{finite\} P) by blast
  have P $ k2 \notin V' unfolding k2-def
  using k1 lhd P infinite-small-length[of P (~\{finite\} P)] by simp
  hence P $ k2 ∈ attractor p K using (~\{finite\} P) \land \{set P \subseteq V - W1\}
    by (metis Diff1 U-def V'-def set-lhd-member-inf)
  then obtain k3 where k3: ldropn k2 P $ k3 K
    using σ-attracts strategy-attractsE unfolding G'.visits-via-def by blast
  define k4 where k4 = k3 + k2
  hence P $ k4 ∈ K
    using k3 lhd P infinite-small-length[of P (~\{finite\} P)] by simp
  moreover have k4 ≥ n unfolding k4-def k2-def
    using le-add2 le-trns by blast
  moreover have ldropn n P $ k4 - n = P $ (k4 - n) + n
    using lhd P infinite-small-length[of P (~\{finite\} P)] by blast
  ultimately have ldropn n P $ k4 - n ∈ K by simp

66
hence \( \text{lset} \ (\text{ldropn n P}) \cap K \neq \{\} \)
using \((\neg \text{finite P}) \land \text{finite-ldropn in-lset-conv-lth} [\text{of} \ \text{ldropn n P \ \& \ k4 = n}]\)
by blast
thus \(\text{min-prio} \in \text{lset} \ (\text{ldropn n (lmap} \ \omega P))\) 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 min-prio-def]
(winning-priority p min-prio \((\neg \text{finite P})\)) by blast
qed

12.2 Positional Determinacy without Deadends

Theorem:

\text{positional-strategy-exists-without-deadends}:
assumes \(v \in V \land v \in V \Rightarrow \neg \text{deadend} v\)
shows \(\exists p. \ v \in \text{winning-region p}\)
proof -
{ \text{fix p.} \ v0 \in \text{winning-region p} }
{ \text{define A where} \ A = \text{attractor p} \ (\text{deadends p**})} 
assume v0-in-attractor: v0 \in \text{attractor p} \ (\text{deadends p**})
then obtain \(\sigma\) where \(\sigma\): strategy p \ \sigma \ \text{strategy-attracts p} \ \sigma \ A \ (\text{deadends p**})
using attractor-has-strategy [of \text{deadends p** p}] A-def deadends-in-V by blast
have \(A \subseteq V\) using A-def using attractor-in-V deadends-in-V by blast
hence \(A = \text{deadends p**} \subseteq V\) by auto

have winning-strategy p \ \sigma \ v0 proof (unfold winning-strategy-def, intro allI implI)
fix \(P\) assume vmc-path G P v0 p \(\sigma\)
then interpret vmc-path G P v0 p \(\sigma\).
show winning-path p \(P\)
using visits-deadend [of \(p**\) \(\sigma(2)\)] strategy-attracts-lset v0-in-attractor
unfolding A-def by simp
qed
hence \(\exists p \ \sigma. \ \text{strategy p} \ \sigma \ \land \ \text{winning-strategy p} \ \sigma \ v0\) using \(\sigma\) by blast
}
note lemma-path-to-deadend = this

12.3 Positional Determinacy with Deadends

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

Theorem:

\text{positional-strategy-exists}:
assumes \(v0 \in V\)
shows \(\exists p. \ v0 \in \text{winning-region p}\)
proof -
{ \text{fix p.} \ v0 \in \text{winning-region p} }
{ \text{define A where} \ A = \text{attractor p} \ (\text{deadends p**})} 
assume v0-in-attractor: v0 \in \text{attractor p} \ (\text{deadends p**})
then obtain \(\sigma\) where \(\sigma\): strategy p \(\sigma\) \text{strategy-attracts p} \(\sigma\) A \ (\text{deadends p**})
using attractor-has-strategy [of \text{deadends p** p}] A-def deadends-in-V by blast
have \(A \subseteq V\) using A-def using attractor-in-V deadends-in-V by blast
hence \(A = \text{deadends p**} \subseteq V\) by auto

have winning-strategy p \(\sigma\) \(v0\) proof (unfold winning-strategy-def, intro allI implI)
fix \(P\) assume vmc-path G P v0 p \(\sigma\)
then interpret vmc-path G P v0 p \(\sigma\).
show winning-path p \(P\)
using visits-deadend [of \(p**\) \(\sigma(2)\)] strategy-attracts-lset v0-in-attractor
unfolding A-def by simp
qed
hence \(\exists p \ \sigma. \ \text{strategy p} \ \sigma \ \land \ \text{winning-strategy p} \ \sigma \ v0\) using \(\sigma\) by blast
}
define $A$ where $A \ p = \text{attractor } p \ (\text{deadends } p^{**})$ 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 $\text{thesis proof (cases)}$

assume $\emptyset \in V'$

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

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

have $V_G' = V' \text{ unfolding } G'\text{-def using } (V' \subseteq V)$ by simp

hence $\emptyset \in V_G'$ using $\emptyset \in V'$ by simp

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

fix $v$ assume $v \in V_G'$

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

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

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

qed

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

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

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

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

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

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

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

qed

obtain $\sigma\text{-attr}$

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

using $\text{attractor-has-strategy } (O F \text{ deadends-in } V) \text{ unfolding } A\text{-def by blast}$

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

have $\sigma'\text{-is-on- } V' \colon \forall v. \ v \in V' \implies \sigma' v = \sigma v$

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

have strategy $p \sigma' \text{ proof}$

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

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

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

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

unfolding $G'\text{-def by blast}$

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

qed

moreover have $\text{winning-strategy } p \sigma' \emptyset$ proof (rule $\text{winning-strategy', rule contr}$)

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

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

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

using $V'\text{-no-deadends } \emptyset \emptyset \in V'$ by unfold-locales

assume contra: $\neg \text{winning-path } p$

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

fix $v$ assume $v \in V' \neg \text{deadend } v v \in VV p$
hence \( v \in G', \forall v \in V \) unfolding \( G' \)-def by (simp add: \( v \in V \))
moreover have \( \neg G' \)-deadend \( v \) using \( \forall \text{no-deadends} (v \in V) \) \( G' = V \) by blast
moreover have \( G' \)-strategy \( v \) \( \sigma' \)
  using \( G' \)-valid-strategy-only-in-\( V \) \( \sigma' \)-def \( \sigma' \)-is-\( \sigma \)-on-\( V' \) \( (1) \) \( G' = V \) 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 \( v \in V' \) \( \neg \text{deadend} \ v \in V V \ p** \ v \rightarrow w \)
have \( w \notin A \ p** \) proof
  assume \( w \in A \ p** \)
  hence \( v \in A \ p** \) unfolding \( A \)-def
    using \( \forall v \in V V \ p** \ (v \rightarrow w) \) attractor-set-\( V V p \) by blast
thus False using \( v \in V' \) unfolding \( V' \)-def by (cases \( p \)) auto
qed
thus \( w \in V' \cup A \ p \) unfolding \( V' \)-def using \( v \rightarrow w \) by (cases \( p \)) auto
next
show lset \( P \cap A \ p = \{ \} \) proof (rule contr)
  assume \( lset \ P \cap A \ p \neq \{ \} \)
  have strategy-attracts \( p \ (\text{override-on-} \sigma' \  \sigma \)-attr \ (A \ p \ - \ \text{deadends} \ p**)) \( A \ p \)
    (deadends \ p**)
    using strategy-attracts-irrelevant-override[OF \( \sigma \)-attr \( \sigma' \) \( \sigma \)-attr \( (2) \) \( \sigma \)-attr \( (1) \) \( \sigma' \)]
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 \( p \  \sigma' \ (A \ p) \  \text{deadends} \ p** \) by simp
  hence lset \( P \cap \text{deadends} \ p** \neq \{ \} \)
    using \( lset \ P \cap A \ p \neq \{ \} \) : attracted-path[OF deadends-in-\( V \)] by simp
thus False using contra visits-deadend[OF \( p** \)] by simp
qed
qed (insert \( \forall v \in V' \))

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

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

thus \(?thesis\) unfolding winning-region-def

using lemma-path-to-deadend \( \langle v_0 \in V \rangle \) 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}:

\textbf{shows} \( V = \text{winning-region } \text{EVEN} \cup \text{winning-region } \text{ODD} \)

and \( \text{winning-region } \text{EVEN} \cap \text{winning-region } \text{ODD} = \{\} \)

\textbf{proof}

\textbf{show} \( V \subseteq \text{winning-region } \text{EVEN} \cup \text{winning-region } \text{ODD} \)

by \( (\text{rule subsetI}) \) (metis (full-typ es) Un-if f other-other-player positional-strategy-exists)

next

\textbf{show} \( \text{winning-region } \text{EVEN} \cup \text{winning-region } \text{ODD} \subseteq V \)

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

next

\textbf{show} \( \text{winning-region } \text{EVEN} \cap \text{winning-region } \text{ODD} = \{\} \)

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

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}

\hspace{1em} Main

\hspace{1em} Attractor

\textbf{begin}

context ParityGame begin

In section 6 we defined \textit{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 \textit{attractor-inductive}

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

\texttt{inductive-set} \textit{attractor-inductive} :: \( \text{Player} \Rightarrow 'a \text{ set} \Rightarrow 'a \text{ set} \)

for \( p :: \text{Player} \) and \( W :: 'a \text{ set} \) 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 $\text{attractor-}\text{inductive}\ p\ W = \text{attractor}\ p\ W$

**proof**

show $\text{attractor-}\text{inductive}\ p\ W \subseteq \text{attractor}\ p\ W$ proof

fix $v$ assume $v \in \text{attractor-}\text{inductive}\ p\ W$

thus $v \in \text{attractor}\ p\ W$ proof (induct rule: attractor-inductive.induct)

 case (Base $v$) thus ?case using attractor-set-base by auto

next

 case ($\forall v$) thus ?case using attractor-set-$\forall v$ by auto

next

 case ($\exists v$) thus ?case using attractor-set-$\exists v$ by auto

qed

show $\text{attractor}\ p\ W \subseteq \text{attractor-}\text{inductive}\ p\ W$

proof

  define $P$ where $P S \leftrightarrow S \subseteq \text{attractor-}\text{inductive}\ p\ W$ for $S$

  from ($W \subseteq V$) have $\text{attractor-}\text{set-induction}$ proof (induct rule: attractor-set-induction)

  case (step $S$)

  hence $S \subseteq \text{attractor-}\text{inductive}\ p\ W$ using $P$-def by simp

  have $W \cup S \cup \text{directly-attracted}\ p\ S \subseteq \text{attractor-}\text{inductive}\ p\ W$ proof

  fix $v$ assume $v \in W \cup S \cup \text{directly-attracted}\ p\ S$

  moreover

  { assume $v \in W$ hence $v \in \text{attractor-}\text{inductive}\ p\ W$ by blast }

  moreover

  { assume $v \in S$ hence $v \in \text{attractor-}\text{inductive}\ p\ W$

  by (meson $S \subseteq \text{attractor-}\text{inductive}\ p\ W$ rev-subsetD) }

  moreover

  { assume $v$-attracted: $v \in \text{directly-attracted}\ p\ S$

  hence $v \in V$ using $S \subseteq V$ attractor-step-bounded-by-$V$ by blast

  hence $v \in \text{attractor-}\text{inductive}\ p\ W$ proof (cases rule: $\forall v$-cases)

  assume $v \in V V p$

  hence $\exists w. v \rightarrow w \wedge w \in S$ using $v$-attracted directly-attracted-def by blast

  hence $\exists w. v \rightarrow w \wedge w \in \text{attractor-}\text{inductive}\ p\ W$

  using $(S \subseteq \text{attractor-}\text{inductive}\ p\ W)$ by blast

  thus ?thesis by (simp add: ($v \in V V p$) attractor-inductive.$V V p$)

next

 assume $v \in V V p^{**}$

 hence $\forall w. v \rightarrow w \rightarrow w \in S$ using $v$-attracted directly-attracted-def by blast

 have $\neg$deadend $v$ using $v$-attracted directly-attracted-def by blast

 show ?thesis proof (rule contr)

 assume $v \notin \text{attractor-}\text{inductive}\ p\ W$

 hence $\exists w. v \rightarrow w \wedge w \notin \text{attractor-}\text{inductive}\ p\ W$

 by (metis attractor-inductive.$V V p^{**}$ $v \in V V p^{**}$ $\neg$deadend $v$)

 hence $\exists w. v \rightarrow w \wedge w \notin S$ using $(S \subseteq \text{attractor-}\text{inductive}\ p\ W)$ by (meson subsetCE)
thus False using * by blast
qed

ultimately show v ∈ attractor-inductive p W by (meson UnE)

\*
\*
show head (to-pre-digraph G) \( e \in pre\text{-}digraph\text{.verts} (to-pre-digraph G) \)
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 \( \Rightarrow \) 'a Graph
where from-pre-digraph \( G \equiv \emptyset \)
\( \text{Graph.verts} = pre\text{-}digraph\text{.verts} G \),
\( \text{Graph.arcs} = \text{arcs-ends} G \)

context nomulti-digraph begin
interpretation is-Digraph: Digraph from-pre-digraph G proof –
{ 
  fix \( v \) \( w \) assume \((v, w) \in E\) from-pre-digraph G
  then obtain \( e \) where \( e \in pre\text{-}digraph\text{.arcs} G \) tail G \( e = v \) head G \( e = w \)
unfolding from-pre-digraph-def by auto
  hence \((v, w) \in V\) from-pre-digraph G \( \times \) 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: \( G = \text{from-pre-digraph} (\text{to-pre-digraph G}) \)
unfolding to-pre-digraph-def from-pre-digraph-def arcs-ends-def arc-to-ends-def by auto

lemma (in nomulti-digraph) graph-conversion-bij2: digraph-iso G (to-pre-digraph (from-pre-digraph G))
proof –
define iso
where iso = \( \emptyset \)
  iso-verts = id :: 'a \( \Rightarrow \) '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 $\forall a \in \text{pre-digraph.\ arcs } G$.
iso-verts iso (tail $G\ a$) = iso-tail iso (iso-arcs iso $a$)
\land iso-verts iso (head $G\ a$) = iso-head iso (iso-arcs iso $a$)

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

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

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

ultimately show \$thesis unfolding digraph-iso-def by blast
qed

end
References


