Positional Determinacy of Parity Games

Christoph Dittmann
c christoph.dittmann@tu-berlin.de
August 16, 2018

We present a formalization of parity games (a two-player game on directed graphs) and a proof of their positional determinacy in Isabelle/HOL. This proof works for both finite and infinite games. We follow the proof in [2], which is based on [3].

Contents

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

2 Auxiliary Lemmas for Coinductive Lists 5
   2.1 lset ......................................................... 5
   2.2 length ..................................................... 6
   2.3 take ......................................................... 6
   2.4 ldropn ...................................................... 7
   2.5 infinite ..................................................... 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
1 Introduction

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

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

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

1.1 Formal Introduction

Formally, a parity game is $G = (V, E, V_0, \omega)$, where $(V, E)$ is a directed graph, $V_0 \subseteq V$ is the set of Even nodes, and $\omega : V \rightarrow \mathbb{N}$ 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 “σ-traps”. Using σ-traps could be worth exploring, because it has the potential to simplify our formalization.

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

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

The main theorem is in section 12.4.
1.3 Technical Aspects

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

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

It is possible to represent a graph as a function \( \textit{nat} \Rightarrow \textit{nat} \Rightarrow \textit{bool} \), 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 \( \textit{UNIV} :: \textit{a} \) set but rather a subset of \( \textit{UNIV} :: \textit{a} \) 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 \( \textit{vmc-path G P v0 p \sigma} \) to say that \( P \) is a valid maximal path in the graph \( G \) starting in \( v_0 \) and conforming to the strategy \( \sigma \) for player \( p \).

2 Auxiliary Lemmas for Coinductive Lists

Some lemmas to allow better reasoning with coinductive lists.

theory MoreCoinductiveList

imports 
Main 
Coinductive.Coinductive-List

begin

2.1 lset

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

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

lemma lset-lnth-member-inf: \[ \neg \text{finite } xs; \text{lset } xs \subseteq A \] \( \Rightarrow \text{lth } xs n \in A \)
by (metis contra-subsetD inf-lset-lnth lset-inf-lset mingle)

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

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

\textbf{lemma} \textit{lnfinite-lset}: \textit{finite} \textit{xs} $\implies$ \textit{~null} \textit{xs} $\implies$ \textit{last} \textit{xs} $\in$ \textit{lset} \textit{xs}

\textbf{proof} (induct rule: \textit{lnfinite-induct})
  \begin{itemize}
  \item case \textit{(LCons} \textit{xs})
  \item show \textit{?case} proof (cases)
    \begin{itemize}
    \item assume \textit{*}: \textit{~null} \textit{(ltl} \textit{xs})
    \item hence \textit{last} \textit{(ltl} \textit{xs}) $\in$ \textit{lset} \textit{(ltl} \textit{xs}) \textbf{using} \textit{LCons.hyps(3)} \textbf{by} blast
    \item hence \textit{last} \textit{(ltl} \textit{xs}) $\in$ \textit{lset} \textit{xs} \textbf{by} (simp add: \textit{in-lset-ltlD})
    \end{itemize}
  \item thus \textit{thesis} by (metis \textit{*} \textit{LCons.prems lhd-LCons-ltl last-LCons2})
  \item qed (metis \textit{LCons.prems lhd-LCons-ltl last-LCons llist.set-set(1)})
  \item qed simp
\end{itemize}

\textbf{lemma} \textit{lset-subset}: \textit{~((lset} \textit{xs} $\subseteq$ \textit{A}) $\implies$ \exists n. \textit{enat} \textit{n} $<$ \textit{llength} \textit{xs} \land \textit{lnth} \textit{xs} \textit{n} $\notin$ \textit{A} \textbf{by} (metis \textit{in-lset-c onv-lnth subsetI})

2.2 \textit{llength}

\textbf{lemma} \textit{enat-Suc-ltl}: \begin{itemize}
\item assumes \textit{enat} \textit{(Suc} \textit{n}) $<$ \textit{llength} \textit{xs}
\item shows \textit{enat} \textit{n} $<$ \textit{llength} \textit{(ltl} \textit{xs})
\item proof
  \begin{itemize}
  \item from \textit{assms} have \textit{eSuc} \textit{(enat} \textit{n}) $<$ \textit{llength} \textit{xs} \textbf{by} (simp add: eSuc-enat)
  \item hence \textit{enat} \textit{n} $<$ \textbf{epr e} \textit{(llength} \textit{xs}) \textbf{using} eSuc-le-i lleI1 \textbf{by} fastforce
  \item thus \textit{thesis} by (simp add: epr e-llength)
  \end{itemize}
\item qed
\end{itemize}

\textbf{lemma} \textit{enat-ltl-Suc}: \textit{enat} \textit{n} $<$ \textit{llength} \textit{(ltl} \textit{xs}) $\implies$ \textit{enat} \textit{(Suc} \textit{n}) $<$ \textit{llength} \textit{xs} \textbf{by} (metis eSuc-enat ldr op-ltl leD leI lnul l-ldr op)

\textbf{lemma} \textit{innite-smal l-l length} [intro]: \textit{~lninite} \textit{xs} $\implies$ \textit{enat} \textit{n} $<$ \textit{llength} \textit{xs} \textbf{by} blast

\textbf{lemma} \textit{lnul l-0-l length}: \textit{~lnul l} \textit{xs} $\implies$ \textit{enat} \textit{0} $<$ \textit{llength} \textit{xs} \textbf{by} auto

\textbf{lemma} \textit{Suc-l length}: \textit{enat} \textit{(Suc} \textit{n}) $<$ \textit{llength} \textit{xs} $\implies$ \textit{enat} \textit{n} $<$ \textit{llength} \textit{xs} \textbf{by} (dual-or der.strict-tr ans enat-or d-simps (2)) \textbf{by} blast

2.3 \textit{ltake}

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

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

\textbf{lemma} \textit{lset-ltake}: ($\forall$ \textit{m}. \textit{n} $<$ \textit{lnth} \textit{xs} \textit{m} $\in$ \textit{A}) $\implies$ \textit{lset} \textit{((ltake} \textit{(enat} \textit{n}) \textit{xs}) $\subseteq$ \textit{A} \textbf{by} (induct \textit{n} arbitrary: \textit{xs})
  \begin{itemize}
  \item case \textit{0}
  \item have \textit{ltake} \textit{(enat} \textit{0}) \textit{xs} $=$ \textit{LNil} \textbf{by} (simp add: \textit{zero-enat-def})
  \end{itemize}
thus ?case by simp
next
  case (Suc n)
  show ?case proof (cases)
    assume xs ≠ LNil
    then obtain x xs' where xs: xs = LCons x xs' by (meson neq-LNil-conv)
    { fix m assume m < n
      hence Suc m < Suc n by simp
      hence lnth xs (Suc m) ∈ A using Suc.prems by presburger
      hence lnth xs' m ∈ A using xs by simp }
    hence lset (ltake (enat n) xs') ⊆ A using Suc.hyps by blast
    moreover have ltake (enat (Suc n)) xs = LCons x (ltake (enat n) xs')
      using xs ltake-eSuc-LCons[of - x xs'] by (metis (no_types) eSuc-enat)
    moreover have x ∈ A using Suc.prems xs by force
    ultimately show ?thesis by simp
  qed simp
qed

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

lemma llast-ltake:
  assumes enat (Suc n) < llength xs
  shows llast (ltake (enat (Suc n)) xs) = lnth xs n (is llast ?A = -)
  unfolding llast-def using llength-ltake'[OF assms] by (auto simp add: lnth-ltake)
lemma lset-ltake-lltl: lset (ltake (enat n) (lltl xs)) ⊆ lset (ltake (enat (Suc n)) xs)
proof (cases)
  assume ¬lnull xs
  then obtain v0 where xs: xs = LCons v0 (lltl xs) by (metis ltl-LCons-lltl)
  hence ltake (eSuc (enat n)) xs = LCons v0 (ltake (enat n) (lltl xs))
    by (metis ltake-eSuc-LCons)
  hence lset (ltake (enat (Suc n)) xs) = lset (LCons v0 (ltake (enat n) (lltl xs)))
    by (simp add: eSuc-enat)
  thus ?thesis using lset-LCons[of v0 ltake (enat n) (lltl xs)] by blast
  qed (simp add: lnull-def)

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

2.5 lfinite
lemma lfinite-ldrop-set: lfinite xs ⇒ ∃n. v ∉ lset (ldropn n xs)
  by (metis ldrop-inf lmember-cexd1 lset-lmember)
lemma index-infinite-set:
  [¬lfinite x; lnth x m = y; (∀i. lnth x i = y ⇒ (∃m > i. lnth x m = y) ) ] ⇒ y ∈ lset (ldropn n x)
proof (induct n arbitrary: x m)
case 0  thus case using bset-nth-member-inf by auto
next
case (Suc n)
obtain a xs where x: x = LCons a xs by (meson Suc.prems(1) lnull-imp-lfinite not-lnull-conv)
obtain j where j: j > m nth x j = y using Suc.prems(2,3) by blast
have nth xs (j - 1) = y by (metis nth-LCons' j (1,2) not-less0 x)
moreover {
  fix i assume nth xs i = y
  hence nth x (Suc i) = y by (simp add: x)
  hence \(\exists j > i. \text{nth} \, x \, j \, = \, y\) by (metis Suc.prems(3) Suc-lessE nth-Suc-LCons x)
}
ultimately show case using Suc.hyps Suc.prems(1) x by auto
qed

2.6 lmap

lemma nth-lmap-idropn:
enat n < llength xs \(\Longrightarrow\) nth (lmap f (ldropn n xs)) 0 = nth (lmap f xs) n
by (simp add: lhd-ldropn nth-0-conv-lhd)

lemma nth-lmap-idropn-Suc:
enat (Suc n) < llength xs \(\Longrightarrow\) nth (lmap f (ldropn n xs)) (Suc 0) = nth (lmap f xs) (Suc n)
by (metis (no_types, lifting) Suc-length ldropn-ll leD list.map-disc iff nth-lmap-idropn
nth-ll lnull-ldropn lll-ldropn lll-lmap)

2.7 Notation

We introduce the notation $ to denote nth.

notation nth (infix $ 61)
3.2 Graphs

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

```plaintext
record 'a Graph =
  verts :: 'a set (V₁)
  arcs :: 'a Edge set (E₁)
abbreviation is-arc :: ('a, 'b) Graph-scheme ⇒ 'a ⇒ 'a ⇒ bool (infix₁ ⇒ₙ 60) where
  v ⇒ₙ 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⇒ₙ w → v ∈ V v⇒ₙ 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 ⇒ₙ 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.

```plaintext
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⇒ₙ w; valid-path Ps; ¬lmul Ps; lhd Ps = w ]
  ⇒ valid-path (LCons v Ps)
inductive-simps valid-path-cons-simp: valid-path (LCons x xs)

lemma valid-path-ltl': valid-path (LCons v Ps) ⇒ valid-path Ps
  using valid-path-simps by blast
lemma valid-path-ltl: valid-path P ⇒ valid-path (ltl P)
  by (metis llist.exhaust-set ltl-simps(1) valid-path-ltl')
lemma valid-path-drop: valid-path P ⇒ valid-path (ldropn n P)
  by (simp add: valid-path-ltl ltl-ldrop)

lemma valid-path-in-V: assumes valid-path P shows lset P ⊆ V proof
  fix x assume x ∈ lset P thus x ∈ V
  using assms by (induct rule: list.set-induct) (auto intro: valid-path.cases)
qed
lemma valid-path-finite-in-V: [ valid-path P; enat n < llength P ] ⇒ P $ n ∈ V
  using valid-path-in-V lset-lnth-member by blast
lemma valid-path-edges': valid-path (LCons v (LCons w Ps)) ⇒ v⇒ₙ 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 –
```

9
define \( P' \) where \( P' = \text{ldropn} \ n \ P \)

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

hence \( P' \ \text{is} \ 0 = P \ \text{by} \ (\text{simp add: } P' \text{-def}) \)

moreover have \( P' \ 0 = P \ 0 \)

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

ultimately have \( \exists P' \ P = \text{LCons} \ (P \ 0) \ (\text{LCons} \ (P \ 0) \ P) \)

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

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

qed

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

assumes major: \( Q \ P \)

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

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

shows valid-path \( P \)

using major proof (coinduction arbitrary: \( P \)

case valid-path

\{ assume \( P \neq \text{LNil} \ \neg (\exists v. \ P = \text{LCons} \ v \ \text{LNil} \land v \in V) \)

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

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

hence \( \text{case using} \ step \text{ valid-path by auto} \)

\}

thus \( \text{case by blast} \)

qed

lemma \( \text{valid-path-no-deadends} \): \( \forall \text{valid-path} \ P; \ \text{enat} \ (\text{Suc} \ i) < \text{length} \ P \)

\( \Longrightarrow \neg \text{deadend} \ (P \ 0) \)

using valid-path-edges by blast

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

\( \forall \text{valid-path} \ P; \ \text{enat} \ i < \text{length} \ P; \ \text{deadend} \ (P \ 0) \)

\( \Longrightarrow \text{enat} \ (\text{Suc} \ i) = \text{length} \ P \)

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

lemma \( \text{valid-path-prefix} \): \( \forall \text{valid-path} \ P; \ \text{prefix} \ P' \ P \)

\( \Longrightarrow \text{valid-path} \ P' \)

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

case \( \text{step} \ v \ w \ P' \ P \ P' \)

then obtain \( P' w \) where \( P' = \text{LCons} \ v \ (\text{LCons} \ w \ P) \) by (metis \( \text{LCons-prefix-conv} \)

hence valid-path \( (\text{LCons} \ w \ P) \) using valid-path-ltl step(2) by blast

moreover have \( \text{prefix} \ (\text{LCons} \ w \ P') (\text{LCons} \ w \ P) \) using \( \text{P' step}(1,3) \) by auto

ultimately show \( \text{case using} \ P' \text{ step}(2) \text{ valid-path-edges'} \) by blast

qed (metis \( \text{LCons-prefix-conv valid-path-ends-simp} \))

lemma \( \text{valid-path-lappend} \):\n
assumes valid-path \( P \) valid-path \( P' \) \( \neg lnull \ P; \ \neg lnull \ P' \)

shows valid-path \( (\text{lappend} \ P' P) \)

proof (cases, cases)

assume \( \neg lnull \ P; \ \neg lnull \ P' \)

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

case \( \text{step} \ v \ w \ P'' P' P \)

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

qed
assume \( \text{lnull} (\text{ltl} P) \)
thus \(?\text{case using} \) \(\text{step}(1,2,3,5,6)\)
  by \((\text{metis lhd-LCons lhd-LCons-ltl lhd-lappend \text{llast-singleton} \linebreak llist\text{.collapse}(1) \text{ltl-lappend ltl-simps}(2))\)

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

qed

qed \((\text{metis l list.\text{disc}(1) lnul l-lapp end ltl-lapp end ltl-simps}(2))\)

A valid path is still valid in a supergame.

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

### 3.4 Maximal Paths

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

**coinductive** \(\text{maximal-path} \) where
  \(\text{maximal-path-base}: \text{maximal-path} \text{ LNil}\)
  \| \(\text{maximal-path-base}': \text{deadend} v \implies \text{maximal-path} (\text{LCons} v \text{LNil})\)
  \(\text{maximal-path-cons}: \neg \text{lnull} Ps \implies \text{maximal-path} Ps \implies \text{maximal-path} (\text{LCons} v Ps)\)

**lemma** \(\text{maximal-no-deadend}: \text{maximal-path} (\text{LCons} v Ps) \implies \neg \text{deadend} v \implies \neg \text{lnull} Ps\)
  by \((\text{metis lhd-LCons llist.distinct}(1) \text{ltl-simps}(2) \text{maximal-path.simps})\)

**lemma** \(\text{maximal-ltl}: \text{maximal-path} P \implies \text{maximal-path} (\text{ltl} P)\)
  by \((\text{metis ltl-simps}(1) \text{ltl-simps}(2) \text{maximal-path.simps})\)

**lemma** \(\text{maximal-drop}: \text{maximal-path} P \implies \text{maximal-path} (\text{ldropn} n P)\)
  by \((\text{simp add: maximal-ltl ltl-\text{ldrop}})\)

**lemma** \(\text{maximal-path-lappend}:\)
  assumes \(\neg \text{lnull} P' \) maximal-path \(P'\)
  shows \(\text{maximal-path} (\text{lappend} P P')\)
proof (cases)
  assume \(\neg \text{lnull} P\)
thus \(?\text{thesis using} \) \(\text{assms} \) \(\text{proof}\) (coinduction arbitrary: \(P' P\) rule: \(\text{maximal-path.coinduct}\))
  case \(\text{maximal-path} P' P\)
  let \(P = \text{lappend} P P'\)
  show \(?\text{case proof} (\text{cases} ?P = \text{LNil} \lor (\exists v. ?P = \text{LCons} v \text{LNil} \land \text{deadend} v))\)
  case False
    then obtain \(Ps v\) where \(P; ?P = \text{LCons} v Ps\) by \((\text{meson neg-LNil-conv})\)
    hence \(Ps = \text{lappend} (\text{ltl} P) P'\) by \((\text{simp add: lappend-llt maximal-path}(1))\)
hence \( \exists P s, P, P_\prime. P = l\text{append} P s, P_\prime \wedge \neg l\text{null} P_\prime \wedge \text{maximal-path} P_\prime \)

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

thus \(? \text{thesis} \) using P l\text{append}-lnull by fastforce

qed blast

qed

qed (simp add: \textassms(2) l\text{append}-lnull[of P P_\prime])

\textbf{lemma} maximal-ends-on-deadend:

\textbf{assumes} maximal-path P \text{finite} P \neg lnull P

\textbf{shows} deadend (l\text{last} P)

\textbf{proof} -

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

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

define P_\prime where P_\prime = l\text{drop} n P

hence maximal-path P_\prime using \assms(1) maximal-drop by blast

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

\textcase (maximal-path-base \ v)

hence deadend (l\text{last} P_\prime) unfolding P_\prime-def by simp

thus \(? \text{thesis} \) unfolding P_\prime-def using ll\text{last}-\text{ldropn}[of n P] n

by (metis P_\prime-def l\text{dropn}-eq-LConsD local.maximal-path-base\'(1))

next

\textcase (maximal-path-cons P_\prime v)

hence l\text{dropn} (Suc n) P = P_\prime unfolding P_\prime-def by (metis l\text{dropn}-eq-Suc ll\text{ldropn} ll\text{simps}(2))

thus \(? \text{thesis} \) using n maximal-path-cons(2) by auto

qed (simp add: P_\prime-def n l\text{dropn}-eq-LNil)

qed

\textbf{lemma} maximal-ends-on-deadend\': [ \text{finite} P; deadend (l\text{last} P) ] \implies maximal-path P

\textbf{proof} (coinduction arbitrary: P rule: maximal-path.coinduct)

\textcase (maximal-path P)

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

\textassume P \neq LNil

then obtain v P_\prime where P_\prime = LCons v P_\prime by (meson neq-LNil-conv)

show \(? \text{thesis} \) proof (cases)

\textassume P_\prime = LNil thus \(? \text{thesis} \) using P_\prime maximal-path(2) by auto

qed (metis P_\prime \text{finite}-LCons ll\text{last}-LCons l\text{list}.collapse(1) maximal-path(1,2))

qed simp

\textbf{lemma} infinite-path-is-maximal: [ valid-path P; \neg \text{finite} P ] \implies maximal-path P

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

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

end — locale Digraph

\textbf{3.5 Parity Games}

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

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

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

**Notation**

- other-player (-**)** \([1000]\) \([1000]\)
- lemma other-other-player | simp: \(p **\;= p\) using Player.exhaust by auto

**Record**

\[
\begin{align*}
'a\text{ ParityGame} = 'a\text{ Graph } + \\
\text{ player0 } :: 'a\text{ set } (V_0) \\
\text{ priority } :: 'a \Rightarrow \text{nat} (\omega)
\end{align*}
\]

**Locale**

\[
\begin{align*}
\text{ParityGame} = \text{Digraph}\; G \text{ for } G :: (', a, ', b)\text{ ParityGame}-\text{scheme (structure)} + \\
\text{ assumes valid-player0-set: } V_0 \subseteq V \\
\text{ and priorities-finite: } \text{finite} (\omega \cdot V)
\end{align*}
\]

**Begin**

\[
\begin{align*}
VV\;p \text{ is the set of nodes belonging to player } p.
\end{align*}
\]

**Abbreviation**

\[
\begin{align*}
VV :: \text{Player } \Rightarrow \text{'a set where } VV\;p \equiv (\text{if } p = \text{Even} \text{ then } V_0 \text{ else } V - V_0)
\end{align*}
\]

**Lemma**

\[
\begin{align*}
\text{V-to-V [intro]: } v \in VV\;p \implies v \in V \text{ using valid-player0-set by (cases } p) \text{ auto}
\end{align*}
\]

**Lemma**

\[
\begin{align*}
\text{V-impl1: } v \in VV\;p \implies v \notin VV\;p** \text{ by auto}
\end{align*}
\]

**Lemma**

\[
\begin{align*}
\text{V-equivalence [iff]: } v \in V \implies v = VV\;p \iff v \in VV\;p** \text{ by auto}
\end{align*}
\]

**Lemma**

\[
\begin{align*}
\text{V-cases [consumes 1]: } [ v \in V ; v \in VV\;p \implies P ; v \in VV\;p** \implies P ] \implies P \text{ by auto}
\end{align*}
\]

### 3.6 Sets of Deadends

**Definition**

\[
\begin{align*}
deadends\;p \equiv \{ v \in VV\;p, \text{ deadend } v\}
\end{align*}
\]

**Lemma**

\[
\begin{align*}
deadends\;in-V: \text{deadends } p \subseteq V \text{ unfolding deadends-def by blast}
\end{align*}
\]

### 3.7 Subgames

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

**Definition**

\[
\begin{align*}
\text{subgame where} \\
& \text{ subgame } V' \equiv G[ \] \\
& \text{ verts } :: V \cap V', \\
& \text{ arcs } :: E \cap (V' \times V'), \\
& \text{ player0 } :: V_0 \cap V'
\end{align*}
\]

**Lemma**

\[
\begin{align*}
\text{subgame-V [simp]: } V_{\text{subgame}}\;V' \subseteq V \\
\text{ and subgame-E [simp]: } E_{\text{subgame}}\;V' \subseteq E \\
\text{ and subgame-\omega: } \omega_{\text{subgame}}\;V' = \omega
\end{align*}
\]

**Unfolding**

\[
\begin{align*}
\text{subgame-def by simp-all}
\end{align*}
\]

**Lemma**

\[
\begin{align*}
\text{ assumes } V' \subseteq V \\
\text{ shows subgame-V' [simp]: } V_{\text{subgame}}\;V' = V' \\
\text{ and subgame-E' [simp]: } E_{\text{subgame}}\;V' = E \cap (V_{\text{subgame}}\;V' \times V_{\text{subgame}}\;V')
\end{align*}
\]

**Unfolding**

\[
\begin{align*}
\text{subgame-def assms by auto}
\end{align*}
\]

**Lemma**

\[
\begin{align*}
\text{subgame-VV [simp]: } \text{ParityGame}.VV (\text{subgame } V') \equiv V' \cap VV\;p \text{ proof--} \\
\text{ have ParityGame}.VV (\text{subgame } V') \text{ Even } = V' \cap VV\;p \text{ Even unfolding subgame-def by auto}
\end{align*}
\]

**Moreover**

\[
\begin{align*}
\text{have ParityGame}.VV (\text{subgame } V') \text{ Odd } = V' \cap VV\;Odd \text{ proof--}
\end{align*}
\]
have \( V' \cap V - (V0 \cap V') = V' \cap V - (V - V0) \) by blast

thus \(?\)thesis unfolding subgame-def by auto

qed

ultimately show \(?\)thesis by simp

qed

corollary subgame-VV-subset [simp]: \( \text{ParityGame} \cdot (\text{subgame} \ V') p \subseteq \text{VV} \ p \) by simp

lemma subgame-finite [simp]: finite \( \omega_{\text{subgame}} V' \cdot V_{\text{subgame}} V' \) proof –

have \( \omega' \cdot V_{\text{subgame}} V' \) 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 \( E_{\text{subgame}} V' \subseteq \text{subgame} V' \times V_{\text{subgame}} V' \)

using subgame-Digraph [unfolded Digraph-def].

show \( V0_{\text{subgame}} V' \subseteq \text{subgame} V' \) 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 \( P: \text{valid-path} \ lset P \subseteq V' \)

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 , \( \text{case-names} \ III \)]

\( \text{case} \ III \)

thus \(?\)case proof (cases rule: valid-path.cases)

\( \text{case} \ (\text{valid-path-cons} \ v \ w \ Ps) \)

moreover hence \( v \in V_{\text{subgame}} V' \) \( w \in V_{\text{subgame}} V' \) using \( III(2) \) by auto

moreover hence \( v \rightarrow_{\text{subgame}} V' \) \( w \) using local.valid-path-cons(4) subgame-def by auto

moreover have \( \text{valid-path} \ Ps \) using \( III(1) \) valid-path-ltl' 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 \( P: \text{maximal-path} \ lset P \subseteq V' \)

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

3.8 Priorities Occurring Infinitely Often

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

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

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

lemma path-inf-priorities-is-nonempty:
  assumes P: valid-path P ¬finite P
  shows ∃k. k ∈ path-inf-priorities P
proof

  Define a map from indices to priorities on the path.

  define f where f i = ω (P $ i) for i
  have range f ⊆ ω ' V unfolding f-def
    using valid-path-in-V[OF P(1)] lset-nth-member-inf[OF P(2)]
    by blast
    hence finite (range f)
      using priorities-finite finite-subset by blast
  then obtain n0 where n0: ¬(finite {n. f n = f n0})
    using pigeonhole-infinite[of UNIV f] by auto
  define k where k = f n0

  The priority k occurs infinitely often.

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

lemma path-inf-priorities-at-least-min-prio:
  assumes P: valid-path P and a: a ∈ path-inf-priorities P
shows Min (ω, V) ≤ a
proof -
  have a ∈ set (ldropn 0 (lmap ω (LCons v P))) using a unfolding path-inf-priorities-def by blast
  hence a ∈ ω' set P by simp
  thus ?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 (LCons v P) (is ?A = ?B)
proof
  show ?A ⊆ ?B proof
    fix a assume a ∈ ?A
    hence ∀n. a ∈ set (ldropn n (lmap ω (LCons v P)))
      unfolding path-inf-priorities-def
      using in-set-ltlD[of a] by (simp add: ltl-ldr opn)
    thus a ∈ ?B unfolding path-inf-priorities-def by blast
  qed
next
  show ?B ⊆ ?A proof
    fix a assume a ∈ ?B
    hence ∀n. a ∈ set (ldropn (Suc n) (lmap ω (LCons v P)))
      unfolding path-inf-priorities-def by blast
    unfolding path-inf-priorities-def by simp
  qed
qed
corollary path-inf-priorities-ltl: path-inf-priorities P = path-inf-priorities (ltl P)
  by (metis l list.exhaust ltl-simps path-inf-priorities-LCons)

3.9 Winning Condition

Let G = (V, E,V0,ω) be a parity game. An infinite path v0,v1,... 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 ≡ (if p = Even then even else odd)

definition winning-path :: Player ⇒ 'a Path ⇒ bool where
  winning-path p P ≡
  (¬finite P ∧ (∃a ∈ path-inf-priorities P. (∃b ∈ path-inf-priorities P. a ≤ b) ∧ winning-priority p a))
  ∨ (¬null P ∧ finite P ∧ llast P ∈ VV p**)
  ∨ (ltl P ∧ p = Even)

Every path has a unique winner.

lemma paths-are-winning-for-one-player:
  assumes valid-path P
  shows winning-path p P ←→ ¬winning-path p** P
proof (cases)
  assume ¬null P

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

assume \( \text{lfinite} P \)

thus \( \text{thesis} \)

using assms lfinite-lset valid-path-in-V

unfolding winning-path-def

by auto

next

assume \( \neg \text{lfinite} P \)

then obtain \( a \) where \( a \in \text{path-inf-priorities} P \wedge b < a \Rightarrow b \notin \text{path-inf-priorities} P \)

using assms ex-least-nat-le[of \( \lambda a. a \in \text{path-inf-priorities} P \)] path-inf-priorities-is-nonempty

by blast

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

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

moreover have \( \forall q. \text{winning-priority} p q \leftrightarrow \neg \text{winning-priority} p \star q \) by simp

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

qed

qed (simp add: winning-path-def)

lemma winning-path-\text{ltl}:

assumes \( P: \text{winning-path} p P \neg \text{null} P \neg \text{null} (\text{ltl} P) \)

shows \( \text{winning-path} p (\text{ltl} P) \)

proof (cases)

assume lfinite P

moreover have \( \text{llast} P = \text{llast} (\text{ltl} P) \)

using \( P(2,3) \) by (metis llast-LCons2 ltl-simps(2) not-null-conv)

ultimately show \( \text{thesis} \) using \( P \) by (simp add: winning-path-def)

next

assume \( \neg \text{lfinite} P \)

thus \( \text{thesis} \) using winning-path-def path-inf-priorities-ltl P (1,2) by auto

qed

corollary winning-path-drop:

assumes \( \text{winning-path} p P \text{ enat} n < llength P \)

shows \( \text{winning-path} p (\text{ldropn} n P) \)

using assms proof (induct n)

case (Suc n)

hence \( \text{winning-path} p (\text{ldropn} n P) \)

using dual-order.strict-trans evat-ord-simps(2) by blast

moreover have \( \text{ltl} (\text{ldropn} n P) = \text{ldropn} (\text{Suc} n) P \)

by (simp add: ldrop-eSuc-llt llt-ldropn)

moreover hence \( \neg \text{null} (\text{ldropn} n P) \)

using Suc.prems(2) by (metis leD null-ldropn lnull-lllt)

ultimately show \( \text{thesis} \) using winning-path-\text{ltl}(of p ldropn n P) Suc.prems(2) by auto

qed simp

corollary winning-path-drop-add:

assumes \( \text{valid-path} P \text{ winning-path} p (\text{ldropn} n P) \text{ enat} n < llength P \)

shows \( \text{winning-path} p P \)

using assms paths-are-winning-for-one-player valid-path-drop winning-path-drop by blast

lemma winning-path-LCons:

assumes \( P: \text{winning-path} p P \neg \text{null} P \)

shows \( \text{winning-path} p (\text{LCons} v P) \)

proof (cases)

assume lfinite P
moreover have \( \ell \text{last } P = \text{last } (\text{LCons } v P) \)
using \( P(2) \) by (metis \( \text{last-LCons2 } \) not-null-conv)
ultimately show \( \forall \text{thesis using } P \) unfolding winning-path-def by simp
next
assume \( \neg \text{finite } P \)
thus \( \forall \text{thesis using } P \) path-inf-priorities-LCons unfolding winning-path-def by simp
qed

lemma winning-path-supergame:
assumes winning-path \( P \) P
and \( G': \text{ParityGame } G' \VV p** \subseteq \text{ParityGame} \VV G' \VV p** \omega = \omega G' \)
shows \( \text{ParityGame}.\text{winning-path } G' P P \)
proof
interpret \( G': \text{ParityGame } G' \) using \( G'(1) \).
have [ finite P; \( \neg \text{null } P \) ] \( \Rightarrow \) \( \text{last } P \in G'.\VV \VV p** \) and \( \text{null } P \Rightarrow p = \text{Even} \)
using assms(1) unfolding winning-path-def using \( G'(2) \) by auto
thus \( \forall \text{thesis unfolding } G'.\text{winning-path-def} \)
using \( \text{null-imp-finite assms(1)} \)
unfolding winning-path-def path-inf-priorities-def \( G'.\text{path-inf-priorities-def } G'(3) \)
by blast
qed
end — locale ParityGame

3.10 Valid Maximal Paths

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

locale vm-path = ParityGame +
fixes \( v0 \)
assumes \( P\text{-not-null } [\text{simp}]: \neg \text{null } P \)
and \( P\text{-valid } [\text{simp}]: \text{valid-path } P \)
and \( P\text{-maximal } [\text{simp}]: \text{maximal-path } P \)
and \( P\text{-v0 } [\text{simp}]: \ellhd P = v0 \)
begin

lemma P-LCons: \( P = \text{LCons } v0 (\text{ltl } P) \) using \( \ellhd\text{-LCons-ltl}[OF \) P-not-null] by simp

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

lemma P-lnull-ltl-deadend-v0: \( \text{lnull } (\text{ltl } P) \Rightarrow \text{deadend } v0 \)
using P-LCons maximal-no-deadend by force
lemma P-lnull-ltl-LCons: \( \text{lnull } (\text{ltl } P) \Rightarrow P = \text{LCons } v0 \text{ LNil} \)
lemma P-deadend-\(\omega\)-LCons: \(\text{deadend } v_0 \implies P = L\text{Cons } v_0.L\text{Nil}\)
using \(P\text{-null-ltl-LCons } P\text{-no-deadend-}\omega\) 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 Pdr-op-valid [simp]: valid-path (ldr opn n P) using valid-path-dr-op by auto
lemma Pdr-op-maximal [simp]: maximal-path (ldr opn n P) using maximal-dr-op by auto

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

lemma extension-valid [simp]: \(v \rightarrow v_0 = \implies \) valid-path (LCons v P) using P-not-null P-v0 P-valid valid-path-cons simp by blast
lemma extension-maximal [simp]: maximal-path (LCons v P) by (simp add: maximal-path-cons)

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

lemma v0-V [simp]: \(v_0 \in V\) by (metis P-LCons P-valid valid-path-cons-simp)
lemma v0-lset-P [simp]: \(v_0 \in \text{lset } P\) using P-not-null P-v0 llist.set-set(1) by blast
lemma v0-VV: \(v_0 \in V \Rightarrow v_0 \in V\) by simp

lemma lset-ltl-P-V [simp]: lset (ltl P) \(\subseteq V\) by (simp add: valid-path-in-V)
lemma lset-lnit-P-V [simp]: lset (lnit P) \(\subseteq V\) by (simp add: valid-path-in-V)

lemma finite-lnit-deadend [simp]: \(\text{finite } P \implies \text{deadend } (\text{lnit } P)\)
using P-maximal P-not-null maximal-ends-on-deadend by blast
lemma finite-lnit-V [simp]: \(\text{finite } P \implies \text{last } P \in V\)
using P-not-null finite-lnit 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 \text{deadends } p
using assms by (meson lset-intersect-lnth)
hence \(\ast\): 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-cov-lnth)
hence llast P \in \text{deadends } p using n(2) by simp
moreover have \(\text{finite } P\) using \(\ast\) llength-eq-enat-finiteD by force
ultimately show \(\ast\)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 \( VV \).
definition (in ParityGame) strategy :: Player => 'a Strategy => bool where
strategy p \( \sigma \) ≡ \( \forall v \in VV \). \( \neg \)deadend \( v \) \implies v \( \mapsto \) \( v \sigma v \)

lemma (in ParityGame) strategyI [intro]:
(\( \forall v. [ v \in VV \& \neg \)deadend \( v ] \implies v \mapsto v \sigma v ) \implies strategy p \( \sigma \)

unfolding strategy-def by blast

4.2 Strategy-Conforming Paths

If \( path-conforms-with-strategy \) \( p \) \( P \) \( \sigma \) holds, then we call \( P \) a \( \sigma \)-path. This means that \( P \) follows \( \sigma \) 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 \( \sigma \)
| path-conforms-LCons-LNil: path-conforms-with-strategy p (LCons v LNil) \( \sigma \)
| path-conforms-VVp: [ v \in VV \& w = \sigma v ] path-conforms-with-strategy p (LCons w Ps) \( \sigma \) ]
\implies path-conforms-with-strategy p (LCons v (LCons w Ps)) \( \sigma \)
| path-conforms-VVpstar: [ v \notin VV \& path-conforms-with-strategy p Ps \( \sigma \) ]
\implies path-conforms-with-strategy p (LCons v Ps) \( \sigma \)

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 \( p \sigma \) assumes P-conforms [simp]: path-conforms-with-strategy p \( P \sigma \)

Similarly, define a locale for valid maximal paths that conform to given strategies for both
players.
locale vmc2-path = comp?; vmc-path G P \( v0 \) \( p \& p \sigma \) \( \sigma' \) + vmc-path G P \( v0 \) \( p \sigma \)
for \( G P \) \( v0 \) \( p \sigma \) \( \sigma' \)

4.3 An Arbitrary Strategy

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

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

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

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

**4.4 Valid Strategies**

**lemma** valid-strategy-updates: [ strategy \( p \sigma \); \( v \Rightarrow w \) ] \( \Rightarrow \) strategy \( p (\sigma(v0 := w0)) \)

**unfolding** strategy-def by auto

**lemma** valid-strategy-updates-set:

- assumes strategy \( p \sigma \) \( \land \) \( v \in A; v \in VV \) \( \Rightarrow \neg \text{deadend } v \) \( \Rightarrow v \Rightarrow \sigma' \)
- shows strategy \( p (\text{override-on } \sigma \sigma' A) \)

**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 \( p (\text{override-on } \sigma \sigma' A) \)

**using** assms(1) assms(2)[unfolded strategy-def] valid-strategy-updates-set by simp

**lemma** subgame-strategy-stays-in-subgame:

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

**proof**

- interpret \( G' : \text{ParityGame} \) subgame \( V' \) using subgame-ParityGame.
  - have \( \sigma v \in V' \) subgame-ParityGame using assms unfolding \( G'.\text{strategy-def } G'.\text{edges-are-in-V(2)} \) by blast
  - thus \( \sigma v \in V' \) by (metis Diff-iff IntE subgame-VV Player.distinct(2))

**qed**

**lemma** valid-strategy-supergame:

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

**proof**

- interpret \( G' : \text{ParityGame} \) subgame \( V' \) using subgame-ParityGame.
  - fix \( v \) assume \( v \in VV \) \( \neg \text{deadend } v \)
  - show \( v \Rightarrow ? \sigma v \) proof (cases)
    - assume \( v \in V' \)
      - hence \( v \in G' ; VV \) using subgame-VV \( \langle v \in VV \rangle \) by blast
      - moreover have \( \neg G'.\text{deadend } v \) using \( G' \)-no-deadends \( \langle v \in V' \rangle \) by blast
      - ultimately have \( v \Rightarrow \text{subgame } V' \sigma' v \) using \( \sigma' \text{-unfolding } G'.\text{strategy-def} \) by blast
      - moreover have \( \sigma' v = ? \sigma v \) using \( \langle v \in V' \rangle \) by simp
      - ultimately show ?thesis by (metis subgame-E subsetCE)

**next**

- assume \( v \notin V' \)

21
thus \textit{thesis} using $v \sigma$ unfolding strategy-def by simp

qed

qed

lemma valid-strategy-in-V: \begin{align*}
\text{strategy } p \sigma; \forall v \in V V p; \neg \text{deadend } v \implies \sigma v \in V
\end{align*}

unfolding strategy-def using valid-edge-set by auto

lemma valid-strategy-only-in-V: \begin{align*}
\text{strategy } p \sigma; \forall v, v' \in V \implies \sigma v = \sigma v'
\end{align*}

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

4.5 Conforming Strategies

lemma path-conforms-with-strategy-ltl [intro]:
\begin{align*}
\text{path-conforms-with-strategy } p P \sigma \implies \text{path-conforms-with-strategy } p (\text{ltl } P) \sigma
\end{align*}

by (drule path-conforms-with-strategy_cases) (simp-all add: path-conforms-with-strategy_intros(1))

lemma path-conforms-with-strategy-drop:
\begin{align*}
\text{path-conforms-with-strategy } p P \sigma \implies \text{path-conforms-with-strategy } p (\text{ldropn } n P) \sigma
\end{align*}

by (simp add: path-conforms-with-strategy_ltl ldropn[of $\lambda P$. path-conforms-with-strategy $P \sigma$])

lemma path-conforms-with-strategy-prefix:
\begin{align*}
\text{path-conforms-with-strategy } p P \sigma \implies \text{lprefix } P P' \implies \text{path-conforms-with-strategy } p P' \sigma
\end{align*}

proof (coinduction arbitrary: $P P'$)
\begin{itemize}
  \item thus \textit{case proof} (cases rule: path-conforms-with-strategy_cases)
    \item path-conforms-LNil
    \begin{itemize}
      \item thus \textit{thesis} using path-conforms-with-strategy(2) by auto
    \end{itemize}
  \end{itemize}

next
\begin{itemize}
  \item path-conforms-LCons-LNil
  \begin{itemize}
    \item thus \textit{thesis} by (metis lprefix-LCons-conv lprefix-antisym lprefix-code(1) path-conforms-with-strategy(2))
  \end{itemize}
\end{itemize}

next
\begin{itemize}
  \item case (path-conforms-VVp $v w$)
    \begin{itemize}
      \item thus \textit{thesis} proof (cases)
        \begin{itemize}
          \item assumes $P' \neq LNil \land P' \neq LCons v LNil$
          \item hence $\exists Q. P' = LCons v (LCons w Q)$
            \begin{itemize}
              \item by (metis local.path-conforms-VVp(1) lprefix-LCons-conv path-conforms-with-strategy(2))
              \item thus \textit{thesis} using local.path-conforms-VVp(1,3,4) path-conforms-with-strategy(2) by force
            \end{itemize}
        \end{itemize}
    \end{itemize}
  \end{itemize}
\end{itemize}

next
\begin{itemize}
  \item case (path-conforms-VVpstar $v$)
    \begin{itemize}
      \item thus \textit{thesis} proof (cases)
        \begin{itemize}
          \item assumes $P' \neq LNil$
          \item hence $\exists Q. P' = LCons v Q$
            \begin{itemize}
              \item using local.path-conforms-VVpstar(1) lprefix-LCons-conv path-conforms-with-strategy(2) by fastforce
              \item thus \textit{thesis} using local.path-conforms-VVpstar path-conforms-with-strategy(2) by auto
            \end{itemize}
        \end{itemize}
    \end{itemize}
\end{itemize}

qed simp

qed

lemma path-conforms-with-strategy-irrelevant:
\begin{align*}
\text{assumes path-conforms-with-strategy } p P \sigma v \notin \text{lset } P
\end{align*}
shows \( \text{path-conforms-with-strategy } p \ P \ (\sigma(v := w)) \)

using \( \text{assms } \) apply (coinduction arbitrary: \( P \)) by (drule path-conforms-with-strategy.cases) auto

lemma \( \text{path-conforms-with-strategy-irrelevant-deaden}\):
  assumes \( \text{path-conforms-with-strategy } p \ P \ \sigma \ \text{deaden} \ v \lor v \notin VV P \ \text{valid-path } P \)
  shows \( \text{path-conforms-with-strategy } p \ P \ (\sigma(v := w)) \)
  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-\( \text{VVpv} v \ w Ps \))
    have \( w = ?\sigma \ w' \) proof
      from (valid-path \( P \)) have \( \neg \text{deaden } w' \)
      using local.path-conforms-VVpv(1) valid-path-cons-simp by blast
      with assms(2) have \( v' \neq v \) using local.path-conforms-\( VVpv(2) \) by blast
      thus \( w = ?\sigma \ w' \) by (simp add: local.path-conforms-VVpv(3))
    qed
  qed
  moreover
  have \( \exists P. LCons w Ps = P \land \text{path-conforms-with-strategy } p \ P \ \sigma \land (\text{deaden } v \lor v \notin VV P) \land \text{valid-path } P \)
  proof
    have valid-path (LCons w Ps)
    using local.path-conforms-VVpv(1) path-conforms-with-strategy(3) valid-path-ltl' by blast
    thus \(?\thesis \) using local.path-conforms-VVpv(4) path-conforms-with-strategy(2) by blast
  qed
  ultimately show \(?\thesis \) using local.path-conforms-VVpv(1,2) by blast

next
  case (path-conforms-\( \text{VVpstar} v \ w Ps \))
  have \( \exists P. \text{path-conforms-with-strategy } p Ps \ \sigma \land (\text{deaden } v \lor v \notin VV P \land \text{valid-path } Ps \)
  using local.path-conforms-\( \text{VVpstar} (1,3) \) path-conforms-with-strategy(2,3) valid-path-ltl' by blast
  thus \(?\thesis \) by (simp add: local.path-conforms-\( \text{VVpstar} (1,2) \))
  qed simp-all
  qed

lemma \( \text{path-conforms-with-strategy-irrelevant-updates} \):
  assumes \( \text{path-conforms-with-strategy } p \ P \ \sigma \land \forall v. v \in \text{iset } P \implies \sigma v = \sigma' v \)
  shows \( \text{path-conforms-with-strategy } p \ P \ \sigma' \)
  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-\( \text{VVpv} v \ w Ps \))
    have \( w = \sigma' v' \) using local.path-conforms-VVpv(1,3) path-conforms-with-strategy(2) by auto
    thus \(?\thesis \) using local.path-conforms-VVpv(1,4) path-conforms-with-strategy(2) by auto
    qed simp-all
  qed

lemma \( \text{path-conforms-with-strategy-irrelevant} \):
  assumes \( \text{path-conforms-with-strategy } p \ P \ (\sigma(v := w)) \ v \notin \text{iset } P \)
  shows \( \text{path-conforms-with-strategy } p \ P \ \sigma \)
  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 (\sigma(v := w)) deadend v \lor v \notin VV p valid-path P
shows path-conforms-with-strategy p P \sigma
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)) \sigma \implies v \in VV p \implies \sigma v = w
by (drule path-conforms-with-strategy.cases) simp-all

lemma path-conforms-with-strategy-lappend :
assumes P: infinite P \lnot lnull P path-conforms-with-strategy p P \sigma
and P': infinite P' path-conforms-with-strategy p P' \sigma
and lappends: llast P \in VV p \implies (\sigma (llast P)) = lhd P'
shows path-conforms-with-strategy p (lappend P P') \sigma
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 llist.collapse(1))
have path-conforms-with-strategy p (LCons (lhd P) P') \sigma proof (cases)
assume lhd P \in VV p
moreover with v0 have lhd P' = \sigma (lhd P)
using LCons.prems(5) by auto
ultimately show ?thesis
using path-conforms-VV p[of lhd P p lhd P' \sigma]
by (metis (no-types) LCons.prems(4) \lnot lnull P' lhd-LCons-llt)
next
assume lhd P \notin VV p
thus ?thesis using path-conforms-VV pstar using LCons.prems(4) v0 by blast
qed
thus ?thesis by (simp add: v0)
next
assume \lnot lnull (llt P)
hence*: path-conforms-with-strategy p (lappend (llt P) P') \sigma
by (metis LCons.hyps(3) LCons.prems(1) LCons.prems(2) LCons.prems(5) LCons.prems(5)
assms(4) assms(5) lhd-LCons-llt llast-LCons path-conforms-with-strategy-llt)
have path-conforms-with-strategy p (LCons (lhd P) (lappend (llt P) P')) \sigma proof (cases)
assume lhd P \in VV p
moreover hence lhd (llt P) = \sigma (lhd P)
by (metis LCons.prems(1) LCons.prems(2) \lnot lnull (llt P); lhd-LCons-llt path-conforms-with-strategy-start)
ultimately show ?thesis
using path-conforms-VV p[of lhd P p lhd (llt P) \sigma] \ast \lnot lnull (llt P)
by (metis lappend-code(2) lhd-LCons-llt)
next
assume lhd P \notin VV p
thus ?thesis by (simp add: * path-conforms-VV pstar)
qed
with \lnot lnull P show path-conforms-with-strategy p (lappend P P') \sigma
by (metis lappend-code(2) lhd-LCons-llt)
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 \( \forall \sigma' P. P = LCons v P \sigma' \implies \) ?case using path-conforms-with-strategy by auto
  ultimately show ?case by (cases P = LNil, simp) (metis lnull-def not-lnull-conv)
qed

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

lemma (in vmc-path) subgame-path-vmc-path:
  assumes \( V' \subseteq V \) 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 by blast
    show G'.maximal-path P using subgame-maximal-path V' P-maximal P by blast
    show G'.path-conforms-with-strategy P P σ 
      using subgame-path-conforms-with-strategy V' P-conforms P by blast
    qed simp-all
  qed

4.6 Greedy Conforming Path

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

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

case

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

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-ltl:
**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-path-lhd)
**apply** (cases deadend v0, simp add: greedy-conforming-path.code)
by (metis (no-types, lifting) greedy-conforming-path.sel(2) ltl.simps(2))

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

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

**private lemma** greedy-path-ltl-VVpstar:
**assumes** greedy-conforming-path p σ σ′ v0 = LCons v0 P v0 ∈ V 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 σ strategy p σ strategy p** σ′
**shows**
greedy-path-not-null: ~null (greedy-conforming-path p σ σ′ v0)
and greedy-path-∅: greedy-conforming-path p σ σ′ v0 $ ∅ = 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
and greedy-path-conforms' : path-conforms-with-strategy p** (greedy-conforming-path p σ σ' v0)

proof

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

show ~null P P $ 0 = v0 by (simp-all add: nth-0-conv-lhd)

{ fix v0 assume v0 \in V
  let ?P = greedy-conforming-path p σ σ' v0
  assume asm: ~∃ v. ?P = LCons v LNil
  obtain P' where P' : ?P = LCons v0 P' by (metis greedy-path-LNil greedy-path-lhd neq-LNil-conv)
  hence ~deadend v0 using asm greedy-path-deadend-v0 $ v0 \in V$ by blast
  from P' have 1 : ~null P' using asm list-collapse(1) $ v0 \in V$ greedy-path-deadend-v0 by blast
  moreover from P' (~deadend v0) assms(2,3) $ v0 \in V$ have v0 \rightarrow ld P' unfolding strategy-def using greedy-path-ltl-VVp greedy-path-ltl-VVpstar
    by (cases v0 \in VV p) auto
  moreover hence ld P' \in V by blast
  moreover hence \exists v. P' = greedy-conforming-path p σ σ' v \land v \in 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 \exists P', ?P = LCons v0 P' \land ~null P' \land v0 \rightarrow ld P' \land ld P' \in V
      \land (\exists v. P' = greedy-conforming-path p σ σ' v \land v \in 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 \in 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 \in 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 \land σ'' = σ) \lor (p'' = p** \land σ'' = σ')
    moreover with assms have strategy p'' σ'' by blast
  hence path-conforms-with-strategy p'' P σ'' using (v0 \in V) unfolding P-def
  proof (coinduction arbitrary: v0)
    case (path-conforms-with-strategy v0)
    show ?case proof (cases v0 \in 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 v P' \backslash\text{null} & v0 &\to \text{lhd} P' \\
\text{lhd} P' &\in V \exists v. P' = \text{greedy-conforming-path} p \sigma \sigma' v \land v \in V 
\end{align*}
using coinduction-helper by blast \\
with (v0 \in V V p') p'' have \sigma'' v0 = \text{lhd} P' \\
using greedy-path-ltl-VVp greedy-path-ltl-VVpstar by blast \\
with (v0 \in V V p') P'(1, 2, 5) have ?path-conforms-VVp \\
using greedy-conforming-path.code path-conforms-with-strategy(1) by fastforce 
\}

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

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

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

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

\text{end}

\text{end}

\textbf{4.7 Valid Maximal Conforming Paths}

Now is the time to add some lemmas to the locale vmc-path.

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

\text{end}
\((v' \in VVp \implies \sigma v' = v0) \implies \text{path-conforms-with-strategy } p \ (LCons v' P) \sigma\)

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

**Lemma** extension-valid-maximal-conforming:

assumes \(v' \rightarrow v0\) \(v' \in VVp \implies \sigma v' = v0\)

shows \(\text{vmc-path } G \ (LCons v' P) v' p \sigma\)

using assms by unfold-locales simp-all

**Lemma** vmc-path-ldropn:

assumes \(\text{enat } n < \text{lenght } P\)

shows \(\text{vmc-path } G \ (\text{ldropn } n \ P) \ (P \nsuc n) \ p \sigma\)

using assms by unfold-locales (simp-all add: lhd-ldropn)

**Lemma** conforms-to-another-strategy:

\(\text{path-conforms-with-strategy } p \ P \sigma' \implies \text{vmc-path } G \ v0 \ p \sigma'\)

using \(P\text{-not-null } P\text{-valid } P\text{-maximal } P\text{-v0}\) by unfold-locales blast+

end

**Lemma** (in ParityGame) valid-maximal-conforming-path-0:

assumes \(\neg\text{lnull } P\) \(\text{valid-path } P\) \(\text{maximal-path } P\) \(\text{path-conforms-with-strategy } p \ P \sigma\)

shows \(\text{vmc-path } G \ P \ (P \nsuc 0) \ p \sigma\)

using assms by unfold-locales (simp-all add: lnth-0-convs lhd)

4.8 Valid Maximal Conforming Paths with One Edge

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

**locale** vmc-path-no-deadend = vmc-path +

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

begin

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

**Lemma** Ptl-not-null [simp]: \(\neg\text{lnull } (\text{ltl } P)\)

using P-LCons P-maximal maximal-no-deadend v0-no-deadend by metis

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

**Lemma** P-LCons': \(P = \text{LCons } v0 \ (\text{LCons } w0 \ (\text{ltl } (\text{ltl } P)))\) using P-LCons Ptl-LCons by simp

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

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

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

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

**Lemma** v0 conforms: \(v0 \in VVp \implies \sigma v0 = w0\)

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

end
context vmc-path begin

lemma vmc-path-lnull-ltl-no-deadend:
¬lnull (ltl P) ⇒ vmc-path-no-deadend G P v0 p σ
using P-0 P-no-deadends by (unfold-locales) (metis enat-ltl-Suc lnull-0-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’ = ldropn n P
  then interpret P’: vmc-path G P’ $ n p σ using vmc-path-ldropn 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’ $ n p σ by unfold-locales
  have σ (P $ n) = P’,w0 using P’.vt-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 (ltl 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 lnull (ltl P)
  hence P = LCons v LNil by (metis llist-disc(2) lset-cases step.hyps(2))
  thus ?thesis using step.prems(3) P-LCons by blast
next
  assume ¬lnull (ltl P)
  then interpret vmc-path-no-deadend G P v0 p σ
  using vmc-path-lnull-ltl-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-lll
  by blast

30
corollary \( \text{vmc-path-lset-induction-simple} \) [\( \text{case-names base step} \)]:

assumes base: \( v_0 \in S \)

and step: \( \forall v_0. \ [ \text{vmc-path-no-deadend} \ G P v_0 p \sigma; v_0 \in ?S; ?Q P ] \implies ?Q (\lll P) \wedge \text{vmc-path-no-deadend}.w_0 \ P \in ?S \) \implies lset \( P \subseteq ?S \) without the \( Q \) predicate.

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

lemma \( \text{vmc-path-lset-induction-closed-subset} \) [\( \text{case-names VVp VVpstar v0 disjoint} \)]:

assumes VVp: \( \forall v. \ [ v \in S; \neg \text{deadend} v; v \in VVp ] \implies \sigma v \in S \cup T \)

and VVpstar: \( \forall w. \ [ w \in S; \neg \text{deadend} v; v \in VVp \star; v \rightarrow w ] \implies w \in S \cup T \)

and v0: \( v_0 \in S \)

and disjoint: \( \text{lset} \ P \cap T = \{ \} \)

shows \( \text{lset} \ P \subseteq S \)

using \( \text{disjoint \ proof} \) (\( \text{induct \ rule: \ vmc-path-lset-induction} \))

case (step \ P v0)

interpret \( \text{vmc-path-no-deadend} \ G P v_0 p \sigma \) using \( \text{step.hyps(1)} \).

have \( \text{lset} \ (\lll P) \cap T = \{ \} \) using \( \text{step.hyps(3)} \)

by \( \text{meson \ disjoint-eq-subset-Compl \ lset-lll \ order-trans} \)

moreover have \( w_0 \in S \cup T \)

using \( \text{assms(1,2)\[of \ w_0]} \) \( \text{step.hyps(2)} \) \( \text{v0-no-deadend \ v0-conforms} \)

by \( \text{cases \ v0 \in VVp} \) simp-all

ultimately show \( ?\text{case \ using \ step.hyps(3) \ w0-lset-P \ by \ blast} \)

qed (\( \text{insert} \ v0 \))

5 Attracting Strategies

theory \( \text{AttractingStrategy} \)

imports

\( \text{Main} \)

\( \text{Strategy} \)

begin

Here we introduce the concept of attracting strategies.

context \( \text{ParityGame} \) begin

5.1 Paths Visiting a Set

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

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

\textbf{unfolding} \textit{visits-via-def} \text{ by blast}

\textbf{lemma} \textit{visits-via-visits}: \(\text{visits-via} \ P \ A \ W \implies \text{lset} \ P \cap W \neq \{\}\)

\textbf{unfolding} \textit{visits-via-def} \text{ by } (\text{meson disjoint-iff-not-equal in-lset-conv-lnth})

\textbf{lemma} \textit{visits-via-visits-trivial}: \(v0 \in W \implies \text{visits-via} \ P \ A \ W\)

\textbf{unfolding} \textit{visits-via-def} \text{ by } (\text{meson disjoint-i-not-equal in-lset-conv-lnth})

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

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

\textbf{lemma} \textit{visits-via-LCons}: \assumes \text{visits-via} \ P \ A \ W \shows \text{visits-via} \ (\text{LCons} \ v0 \ P) \ (\text{insert} \ v0 \ A) \ W

\textbf{proof} –

\begin{itemize}
  \item obtain \(n\) where \(n: \text{enat} \ n < \text{llength} \ P \ \text{\$} \ n \in \ W \ \text{lset} \ (\text{ltake} \ (\text{enat} \ n) \ P) \subseteq A\)
  \item using \assms \text{unfolding} \textit{visits-via-def} \text{ by blast}
  \item define \(P' = \text{LCons} \ v0 \ P\)
  \item have \(\text{enat} \ (\text{Suc} \ n) < \text{llength} \ P\) \text{ unfolding} \(P'\text{-def}\)
  \item by (\text{metis n(1) ldropn-Suc-LCons ldropn-Suc-conv-ldropn ldropn-quant-LConsD})
  \item moreover have \(P' \text{\$} \text{Suc} \ n \in W\) \text{ unfolding} \(P'\text{-def}\) \text{ by } (\text{simp add: n(2)})
  \item moreover have \(\text{lset} \ (\text{ltake} \ (\text{enat} \ (\text{Suc} \ n)) \ P') \subseteq \text{insert} \ v0 \ A\)
  \item using \text{lset-ltake-Suc[of \ P \ v0 \ n \ A]} \text{ unfolding} \(P'\text{-def}\) \text{ by } (\text{simp add: n(3)})
  \item ultimately show \text{thesis} \text{ unfolding} \textit{visits-via-def} \(P'\text{-def}\) \text{ by blast}
\end{itemize}

\textbf{qed}

\textbf{lemma} \textit{in-vmc-path-no-deadend}: \textit{visits-via-llt}: \assumes \text{visits-via} \ P \ A \ W

\text{and} \(v0: v0 \notin W\)

\shows \text{visits-via} \ (\text{llt} \ P) \ A \ W

\textbf{proof} –

\begin{itemize}
  \item obtain \(n\) where \(n: \text{enat} \ n < \text{llength} \ P \ \text{\$} \ n \in \ W \ \text{lset} \ (\text{ltake} \ (\text{enat} \ n) \ P) \subseteq A\)
  \item using \assms(1)[unfolded \textit{visits-via-def}] \text{ by blast}
  \item have \(n \neq 0\) using \(v0 \text{\$} n(2)\) \text{ DiffE by force}
  \item then obtain \(n'\) where \(n' = n\) using \text{nat.exhaust} \text{ by metis}
  \item have \(\exists n'. \ \text{enat} \ n' < \text{llength} \ (\text{llt} \ P) \ \land \ (\text{llt} \ P) \ \text{\$} \ n' \in W \ \land \ \text{lset} \ (\text{ltake} \ (\text{enat} \ n') \ (\text{llt} \ P)) \subseteq A\)
  \item applying (\text{rule exI[of \ - \ n']})
  \item using \(n' \text{\$} \text{enat-llt}[of \ n' \ P] \ (\text{llt-conv-Suc-lset-ltake-llt}[of \ n' \ P])\) \text{ by auto}
  \item thus \text{thesis} \text{ using} \textit{visits-via-def} \text{ by blast}
\end{itemize}

\textbf{qed}

\textbf{lemma} \textit{in-vmpath}: \textit{visits-via-deadend}: \assumes \text{visits-via} \ P \ A \ (\text{deadends} \ p)

\shows \text{winning-path} \ P \bullet P

\textbf{using} \assms \textit{visits-via-visits-deadend by blast}

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

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

\textbf{definition} \textit{strategy-attracts-via} :: \(\text{Player} \Rightarrow \text{a Strategy} \Rightarrow \text{a} \Rightarrow \text{a set} \Rightarrow \text{a set} \Rightarrow \text{bool}\) \text{ where}
strategy-attracts-via \( p \sigma v0 A W \equiv \forall P. \text{vmc-path } G P v0 p \sigma \rightarrow \text{visits-via } P A W \)

**Lemma (in vmc-path) strategy-attracts-viaE:**

**Assumes** strategy-attracts-via \( p \sigma v0 A W \)

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

**Using** strategy-attracts-via-def **Assms** vmc-path-axioms by blast

**Lemma (in vmc-path) strategy-attracts-via-SucE:**

**Assumes** strategy-attracts-via \( p \sigma v0 A W \)

**Shows** \( \exists n. \text{enat } (\text{Suc } n) < \text{llength } P \land P \subset W \land \text{lset } (\text{ltake } (\text{enat } (\text{Suc } n)) P) \subseteq A \)

**Proof**

- **Obtain** \( n \) where \( n. \text{enat } n < \text{llength } P P \subset n \in W \land \text{lset } (\text{ltake } (\text{enat } n ) P) \subseteq A \)
  - **Using** strategy-attracts-viaE [unfolded visits-via-def] **Assms** (1) by blast
  - **Have** \( n \neq 0 \) **Using** assms(2) \( n(2) \) by (metis P-0)
  - **Thus** ?thesis **Using** \( n \) not0-implies-Suc by blast

**Qed**

**Lemma (in vmc-path) strategy-attracts-via-lset:**

**Assumes** strategy-attracts-via \( p \sigma v0 A W \)

**Shows** \( \text{set } P \cap W \neq \{\} \)

**Using** assms[THEN strategy-attracts-viaE, unfolded visits-via-def]

by (meson disjoint iff-not-equal set-lnth-member subset-refl)

**Lemma** strategy-attracts-via-v0:

**Assumes** \( \sigma; \text{strategy } p \sigma \text{strategy-attracts-via } p \sigma v0 A W \)

and \( v0; v0 \in V \)

**Shows** \( v0 \in A \cup W \)

**Proof**

- **Obtain** \( P \) where \( \text{vmc-path } G P v0 p \sigma \) **Using** strategy-conforming-path-exists-single **Assms** by blast
  - **Then interpret** \( \text{vmc-path } G P v0 p \sigma \)
  - **Obtain** \( n \) where \( n. \text{enat } n < \text{llength } P P \subset n \in W \land \text{lset } (\text{ltake } (\text{enat } n ) P) \subseteq A \)
    - **Using** \( \sigma(2) \) [unfolded strategy-attracts-via-def visits-via-def] **Vmc-path-axioms** by blast
  - **Show** ?thesis **Proof** (cases \( n = 0 \))
    - **Case** True **Thus** ?thesis **Using** \( n \) (2) by simp
  - **Next**
    - **Case** False
      - **Hence** \( \text{lhd } (\text{ltake } (\text{enat } n ) P) = \text{lhd } P \) by (simp add: enat-0 iff (1))
      - \( \text{lhd } v0 \in \text{lset } (\text{ltake } (\text{enat } n ) P) \)
        - by (metis \( \langle n \neq 0 \rangle \); P-not-null P-v0 enat-0 iff (1) llist.set-set (1) ltake.disc (2))
      - **Thus** ?thesis **Using** \( n \) (3) by blast
  - **Qed**

**Qed**

**Corollary** strategy-attracts-not-outside:

\[ \forall v0 \in V - A - W; \text{strategy } p \sigma \] \( \Rightarrow \) \( \neg \text{strategy-attracts-via } p \sigma v0 A W \)

**Using** strategy-attracts-via-v0 by blast

**Lemma** strategy-attracts-viaI [intro]:

**Assumes** \( \forall P. \text{vmc-path } G P v0 p \sigma \rightarrow \text{visits-via } P A W \)

**Shows** strategy-attracts-via \( p \sigma v0 A W \)

**Unfolding** strategy-attracts-via-def **Using** **Assms** by blast

---

33
lemma strategy-attracts-via-no-deadends:
  assumes v ∈ V v ∈ A − W strategy-attracts-via p σ v A W
  shows ¬deadend v
proof
  assume deadend v
  define P where [simp]: P = LCons v LNil
interpret vmc-path G P v p σ proof
  show valid-path P using (v ∈ A − W) (v ∈ V) valid-path-base' by auto
  show maximal-path P using (deadend v) by (simp add: maximal-pathintros(2))
  show path-conforms-with-strategy p P σ by (simp add: path-conforms-LCons-LNil)
qed simp-all
have visits-via P A W using assms(3) strategy-attracts-viaE by blast
moreover have length P = eSuc 0 by simp
ultimately have P $ 0 ∈ W by (simp add: enat-0-iff(1) visits-via-def)
with v ∈ A − W show False by auto
qed

lemma attractor-strategy-on-extends:
  [strategy-attracts-via p σ v0 A W; A ⊆ A'] ⇒ strategy-attracts-via p σ v0 A' W
unfolding strategy-attracts-via-def using visits-via-monotone by blast

lemma strategy-attracts-via-trivial: v0 ∈ W ⇒ strategy-attracts-via p σ v0 A W
proof
  fix P assume v0 ∈ W vmc-path G P v0 p σ
  then interpret vmc-path G P v0 p σ by blast
  show visits-via P A W using visits-via-trivial using (v0 ∈ W) by blast
qed

lemma strategy-attracts-via-successor:
  assumes σ: strategy p σ strategy-attracts-via p σ v0 A W
  and w0: v0 ∈ A − W
  and w0: v0 → w0 v0 ∈ VV p ⇒ σ v0 = w0
  shows strategy-attracts-via p σ w0 A W
proof
  fix P assume vmc-path G P w0 p σ
  then interpret vmc-path G P w0 p σ
  define P' where [simp]: P' = LCons v0 P
  then interpret P': vmc-path G P' v0 p σ
  using extension-valid-maximal-conforming w0 by blast
interpret P': vmc-path-no-deadend G P' v0 p σ using (v0 → w0) by unfold-locales blast
have visits-via P' A W using σ(2) P'.strategy-attracts-viaE by blast
thus visits-via P A W using P'.visits-via-Ill v0 by simp
qed

lemma strategy-attracts-VVp:
  assumes σ: strategy p σ strategy-attracts-via p σ v0 A W
  and v: v0 ∈ A − W v0 ∈ VV p ¬deadend v0
  shows σ v0 ∈ A ∪ W
proof
  have v0 → σ v0 using σ(1)[unfoldd strategy-def] v(2,3) by blast
  hence strategy-attracts-via p σ (σ v0) A W
pro of
lemma strategy-attr acts-invalid-p ath
lemma strategy-attr acts-empty
(lemma strategy-attr actsE
lemma strategy-attr acts definitions
lemma strategy-attr acts-VVpstar
qed

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-attr acts :: Player $\Rightarrow$ 'a Strategy $\Rightarrow$ 'a set $\Rightarrow$ bool
strategy-attr acts $\sigma$ $A$ $W$ $\equiv \forall v \in A$. strategy-attr acts via $p$ $\sigma$ $v$ $A$ $W$

lemma (in vmc-path) strategy-attr actsE:
assumes strategy-attr acts $p$ $\sigma$ $A$ $W$ $v$ $0$ $\in$ $A$
shows visits-via $P$ $A$ $W$
using assms(1)[unfolded strategy-attr acts-def] assms(2) strategy-attr acts via $E$ by blast

lemma strategy-attr actsI [intro]:
assumes $\forall P. v. \{ v \in A; \text{vmc-path } G P \ v \ \sigma \ \} \implies \text{visits-via } P \ A \ W$
shows strategy-attr acts $p$ $\sigma$ $A$ $W$
unfolding strategy-attr acts-def using assms by blast

lemma (in vmc-path) strategy-attr acts-lset:
assumes strategy-attr acts $p$ $\sigma$ $A$ $W$ $v$ $0$ $\in$ $A$
shows $\text{lset } P \cap W \neq \{\}$
using assms(1)[unfolded strategy-attr acts-def] assms(2) strategy-attr acts via $\text{lset}(1)[\text{of } A \ W]$
by blast

lemma strategy-attr acts-empty [simp]: strategy-attr acts $p$ $\{\}$ $W$ by blast

lemma strategy-attr acts-invalid-path:
assumes $P$: $P = \text{LCons } v \ (\text{LCons } w \ p \ \sigma \ \) \ 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 $n$: $\text{enat } n < \text{lenght } P \ P \ \& \ n \in W \ \text{lset } \text{ltake } \text{(enat } n \ P \ \)} \subseteq A$
unfolding visits-via-def by blast
have $n \neq 0$ using ($v \in A \ W \ n(2)$ $P(1)$ DiffD2 by force
moreover have $n \neq \text{Suc } 0$ using ($w \notin A \cup W$ $n(2)$ $P(1)$) by auto
ultimately have $\text{Suc } (\text{Suc } 0) \leq n$ by presburger
hence lset (ltake (enat (Suc (Suc 0)))) $P$ $\subseteq A$ using $n(3)$
by (meson contra-subsetD enat-ord-simps(1) lset-ltake-prefix lset-lth-member lset-subset)
moreover have enat ($\text{Suc } 0$) $< \text{lenght } (\text{ltake } (\text{eSuc } (\text{Suc } 0) \ P)$ proof --
have $*$: enat ($\text{Suc } (\text{Suc } 0)$) $<$ length $P$
using ($\text{Suc } (\text{Suc } 0) \leq n$ $n(1)$ by (meson enat-ord-simps(2) le-less-linear less-le-trans not-iff)
have llength (ltake (enat (Suc (Suc 0)))) $P$ = min (enat (Suc (Suc 0))) (llength $P$) by simp
hence llength (ltake (enat (Suc (Suc 0)))) $P$ = enat (Suc (Suc 0))

35
Given an attracting strategy \( \sigma \),

\( \text{let } lset = \text{take} \)

\( \text{proof} \)

\( \text{lemma strategy-attracts-irrelevant-override:} \)

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

\( \text{and } v: v \rightarrow w \quad v \in A - W \quad w \notin A \cup W \)

\( \text{shows } v \in V V P \wedge \sigma v \neq w \)

\( \text{proof (rule contr) } \)

\( \text{assume contr: } \neg(v \in V V P \wedge \sigma v \neq w) \)

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

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

\( \text{then obtain } P \text{ where } P: \text{vmc2-path } G P v p \sigma \sigma' \)

\( \text{using } (v \rightarrow w) \text{ strategy-conforming-path-exists } \sigma' \text{ by blast} \)

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

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

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

\( \text{have } w = w0 \text{ using andm } \sigma' \text{-def } w0 \text{-conforms comp.w0-conforms by } (\text{cases } v \in V V P) \text{ auto} \)

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

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

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

\( \text{qed} \)

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

\( \text{lemma strategy-attracts-irrelevant-ignore:} \)

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

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

\( \text{proof (rule strategy-attractsI, rule contr) } \)

\( \text{fix } P \)

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

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

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

\( \text{assume } v \in A \)

\( \text{hence } P \$ 0 \in A \text{ using } (v \in A) \text{ by simp} \)

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

\( \text{ultimately have } P \$ 0 \in A - W \text{ unfolding visits-via-def by } (\text{meson DiffI } P-Len not-less0 lset-blake) \)

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

\( \text{assume } \text{lset } P \subseteq A - W \)

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

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

\( \text{using path-conforms-with-strategy-irrelevant-updates } \text{of } P \text{-conforms by blast} \)
lemma \textit{vmc-path} \(G P (P \not\emptyset) p \sigma\)
using \textit{conforms-to-another-strategy} \(P-0\) by \textit{blast}

thus \textit{False}
using contra \((P \not\emptyset) \in A\) \textit{assms}(I)
by (meson \textit{vmc-path} \textit{strategy-attracts}\(E\))
\textbf{qed}

\textbf{hence} \(\exists n. \text{enat} n < \text{llength} P \land P \not\emptyset n \notin A - W\) by (meson \textit{lset-subset})

\textbf{then obtain} \(n\) \textbf{where} \(n. \text{enat} n < \text{llength} P \land P \not\emptyset n \notin A - W\)
\(\land i. i < n \implies \neg(\text{enat} i < \text{llength} P \land P \not\emptyset i \notin A - W)\)
using \textit{ex-least-nat-le} [of \(\lambda n. \text{enat} n < \text{llength} P \land P \not\emptyset n \notin A - W\)] by \textit{blast}

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

\textbf{have} \(n \neq 0\) \textbf{using} \((P \not\emptyset 0) \in A - W\) \textit{assms}(I) by \textit{meson}

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

\textbf{hence} \(P \not\emptyset n' \in A - W\) \textbf{using} \(n\)-min by \textit{blast}

\textbf{moreover have} \(P \not\emptyset n' \rightarrow P \not\emptyset \text{Suc} n'\) \textbf{using} \textit{P-valid} \(n(I)\) \textit{valid-path-edges} by \textit{blast}

\textbf{moreover have} \(P \not\emptyset \text{Suc} n' \notin A \cup W\) \textbf{proof-}

\textbf{have} \(P \not\emptyset n \notin W\) \textbf{using} \textit{contra} \(n(I)\) \(n\)-min \textit{unfolding} \textit{visits-via-def}

by (meson \textit{Diff-subset} \textit{lset-ltake} \textit{subsetCE})

\textbf{thus} \textit{\(\exists\)thesis} \textbf{using} \(n(I)\) \textit{n'} \textit{by blast}
\textbf{qed}

\textbf{ultimately have} \(P \not\emptyset n' \in \textit{VV} P \land \sigma (P \not\emptyset n') \neq P \not\emptyset \text{Suc} n'\)
using \textit{strategy-attracts-does-not-leave} [of \(\sigma A W P \not\emptyset n' P \not\emptyset \text{Suc} n\)]
\textit{assms}(I,2) \textbf{by blast}

\textbf{thus} \textit{False}
using \(n(I)\) \textit{n'} \textit{vmc-path-conforms} \((P \not\emptyset n' \in A - W)\) \textbf{by (meson \textit{override-on-apply-in})}
\textbf{qed}

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

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

\textbf{lemma} (in \textit{vmc-path}) \textit{attracted-path}:
\textbf{assumes} \(W \subseteq V\)
\textbf{and} \(\sigma:: \textit{strategy-attracts} p \sigma A W\)
\textbf{and} \(P\)-\textit{hits-A}: \textit{lset} \(P \cap A \neq \{\}\)
\textbf{shows} \textit{lset} \(P \cap W \neq \{\}\)
\textbf{proof-}
\textbf{obtain} \(n\) \textbf{where} \(n. \text{enat} n < \text{llength} P P \not\emptyset n \in A\) \textbf{using} \textit{P-hits-A} \textbf{by (meson \textit{lset-intersect-lnth})}
\textbf{define} \(P'\) \textbf{where} \(P' = \textit{idropn} n P\)
\textbf{interpret} \textit{vmc-path} \(G P P \not\emptyset n p \sigma\) \textit{unfolding} \(P'\)-\textit{def} \textbf{using} \textit{vmc-path-idropn} \(n(I)\) \textbf{by blast}
\textbf{have} \textit{visits-via} \(P' A W\) \textbf{using} \(\sigma n(2)\) \textit{strategy-attracts}\(E\) \textbf{by blast}
\textbf{thus} \textit{\(\exists\)thesis} \textit{unfolding} \(P'\)-\textit{def} \textbf{using} \textit{visits-via-visits in-lset-idropnD} [of - \(n P\)] \textbf{by blast}
\textbf{qed}

\textbf{lemma} \textit{attracted-strategy-step}:
\textbf{assumes} \(\sigma:: \textit{strategy} p \sigma\) \textit{strategy-attracts} \(p \sigma A W\)
\textbf{and} \(v0:: \neg\textit{deadend} v0 v0 \in A - W v0 \in \textit{VV} p\)
\textbf{shows} \(v0 \in A \cup W\)
by (metis \textit{Diff1} \textit{strategy-attracts-VVp} \textit{assms} \textit{strategy-attracts-def})

\textbf{lemma} (in \textit{vmc-path-no-deadend}) \textit{attracted-path-step}:
assumes \( \sigma : \text{strategy-attracts} \ p \ \sigma \ A \ W \)
and \( v0 : v0 \in A - W \)
shows \( w0 \in A \cup W \)
by (metis (no-types) Diff1 P-LCons' \( \sigma \) strategy-attractsE strategy-attracts-invalid-path \( v0 \))

end — context ParityGame

end

6 Attractor Sets

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

context ParityGame begin

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

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

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

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

definition attractor :: \( \text{Player} \Rightarrow \text{a set} \Rightarrow \text{a set where} \)
  attractor \( p \ W = \text{lfp} \ (\text{attractor-step} \ p \ W) \)

6.1 directly-attracted

Show a few basic properties of \( \text{directly-attracted} \).

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

6.2 attractor-step

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

The definition of \text{attractor} uses \text{lfp}. For this to be well-defined, we need show that \text{attractor-step} is monotone.
\textbf{lemma} \textit{attractor-step-mono}: mono \textit{(attractor-step p W)}

\textbf{unfolding} \textit{directly-attracted-def} by \textit{(rule monoI)} auto

6.3 Basic Properties of an Attractor

\textbf{lemma} \textit{attractor-unfolding}: \textit{attractor p W = attractor-step p W (attractor p W)}

\textbf{unfolding} \textit{attractor-def} using \textit{attractor-step-mono lfp-unfold} by blast

\textbf{lemma} \textit{attractor-lowerbound}: \textit{attractor-step p W S \subseteq S \Rightarrow attractor p W \subseteq S}

\textbf{unfolding} \textit{attr actor-def} using \textit{attr actor-step-mono} by \textit{(simp add: lfp-lowerbound)}

\textbf{lemma} \textit{attractor-set-non-empty}: \textit{W \neq \{\}} \Rightarrow \textit{attractor p W \neq \{\}}

\textbf{and} \textit{attractor-set-base}: \textit{W \subseteq attractor p W}

\textbf{using} \textit{attractor-unfolding} by auto

\textbf{lemma} \textit{attractor-set-VVp}:

\textbf{assumes} \textit{v \in VV p v \rightarrow w w \in attractor p W}

\textbf{shows} \textit{v \in attractor p W}

\textbf{apply} \textit{(subst attractor-unfolding)} unfolding \textit{directly-attracted-def} using \textit{assms} by auto

\textbf{lemma} \textit{attractor-set-VVpstar}:

\textbf{assumes} \textit{\neg deadend v \land v \rightarrow w \Rightarrow w \in attractor p W}

\textbf{shows} \textit{v \in attractor p W}

\textbf{apply} \textit{(subst attractor-unfolding)} unfolding \textit{directly-attracted-def} using \textit{assms} by auto

6.4 Attractor Set Extensions

\textbf{lemma} \textit{attractor-set-VVp}:

\textbf{assumes} \textit{v \in VV p v \rightarrow w w \in attractor p W}

\textbf{shows} \textit{v \in attractor p W}

\textbf{apply} \textit{(subst attractor-unfolding)} unfolding \textit{directly-attracted-def} using \textit{assms} by auto

\textbf{lemma} \textit{attractor-set-VVpstar}:

\textbf{assumes} \textit{\neg deadend v \land v \rightarrow w \Rightarrow w \in attractor p W}

\textbf{shows} \textit{v \in attractor p W}

\textbf{apply} \textit{(subst attractor-unfolding)} unfolding \textit{directly-attracted-def} using \textit{assms} by auto

6.5 Removing an Attractor

\textbf{lemma} \textit{removing-attractor-induces-no-deadends}:

\textbf{assumes} \textit{v \in S - attractor p W v \rightarrow w w \in S \land \ [ v \in VV p**; v \rightarrow w ] \Rightarrow w \in S}

\textbf{shows} \textit{\exists w \in S - attractor p W. v \rightarrow w}

\textbf{proof} –

\textbf{have} \textit{v \in V using \langle v \rightarrow w \rangle} by blast

\textbf{thus} \textit{?thesis} \textbf{proof} \ (\textit{cases rule: VV-cases})

\textbf{assume} \textit{v \in VV p}

\textbf{thus} \textit{?thesis} \textbf{using} \textit{attractor-set-VVp} \textit{assms} by blast

\textbf{next}

\textbf{assume} \textit{v \in VV p**}

\textbf{thus} \textit{?thesis} \textbf{using} \textit{attractor-set-VVpstar} \textit{assms} by \textit{(metis Di-iff edges-are-in-V(2))}

\textbf{qed}

\textbf{qed}

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

\textbf{lemma} \textit{subgame-without-deadends}:

\textbf{assumes} \textit{\textit{V'}-def: V' = V - attractor p (deadends p** - attractor p** (deadends p****))
(is V' = V - ?A - ?B)

and v: v \in V_subgame V'}

\textbf{shows} \textit{\neg Digraph.deadend (subgame V') v}

\textbf{proof} \ (\textit{cases})

\textbf{assume} \textit{deadend v}

\textbf{have} \textit{v: v \in V - ?A - ?B using v unfolding V'-def subgame-def by simp}

39
\{ \text{fix } p' \text{assume } v \in V V p' \}\n\text{hence } v \in \text{attractor } p' (\text{deadends } p'\star\star)
\text{using } (\text{deadend } v) \text{attractor-set-base[of deadends } p'\star\star p'\}\n\text{unfolding deadends-def by blast}
\text{hence } \text{False using } v \text{by } (\text{cases } p' ; \text{cases } p) \text{auto}
\}\n\text{thus } \text{thesis using } v \text{by blast}

\text{next}
\text{assume } \neg \text{deadend } v
\text{have } v : v \in V = ?A \to ?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 } \text{subgame-ParityGame}.
\text{show } \text{thesis proof}
\text{assume } D = \text{Graph } \text{deadend } (\text{subgame } V') v
\text{hence } G'.\text{deadend } v \text{unfolding } G'\text{-def}.
\text{have all-in-attractor: } \forall w. v \rightarrow w \implies w \in ?A \lor w \in ?B \text{ proof (rule contr)}
\text{fix } w
\text{assume } v \rightarrow w \neg (w \in ?A \lor w \in ?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 \rightarrow w) \text{by auto}
\text{hence } v \rightarrow G' v \text{using } (v \rightarrow w) \text{assms}(2) \text{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' \text{assume } v \in V V p'\}\n\{ \text{assume } \exists w. v \rightarrow w \land w \in \text{attractor } p' (\text{deadends } p'\star\star)\}
\text{hence } v \in \text{attractor } p' (\text{deadends } p'\star\star) \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 \rightarrow w \implies w \in \text{attractor } p'\star\star (\text{deadends } p'\star\star\star)
\text{using all-in-attractor by } (\text{cases } p' ; \text{cases } p) \text{auto}
\text{hence } v \in \text{attractor } p'\star\star (\text{deadends } p'\star\star\star)
\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}

6.6 Attractor Set Induction

\text{lemma mono-restriction-is-mono: mono } f \implies mono (\lambda S. f (S \cap V))
\text{unfolding 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 https://lists.cam.ac.uk/pipermail/cl-isabelle-users/2015-October/msg00123.html

\text{lemma } \text{attractor-set-induction } \{ \text{consumes } 1, \text{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)
proof –

let \( ?P = \lambda S. P (S \cap V) \)

let \( ?f = \lambda S. \text{attractor-step} p W (S \cap V) \)

let \( ?A = \lambda p \; ?f \)

let \( ?B = \lambda p \; (\text{attractor-step} p W) \)

have \( f\text{-mono: mono} \; ?f \)

using \( \text{mono-restriction-is-mono[of attractor-step p W]} \)

attractor-step-mono by simp

have \( P\text{-A} \):

\( ?P \; ?A \)

proof (rule lfp-or-dinal-induct-set)

show \( \bigwedge S. ?P S \implies ?P (W \cup (S \cap V) \cup \text{directly-attracted} p (S \cap V)) \)

by (metis assms(1) attractor-step-bounded-by-V inf absorbs1 inf-le2 local step)

show \( \bigwedge M. \forall S \in M. ?P S \implies ?P (\bigcup M) \) – proof –

fix \( M \)

let \( ?M = \{ S \cap V \mid S. S \in M \} \)

assume \( \forall S \in M. ?P S \)

hence \( \forall S \in ?M. S \subseteq V \wedge P S \) by auto

hence \( \ast : P (\bigcup ?M) \) by (simp add: union)

have \( \bigcup ?M = (\bigcup M) \cap V \) by blast

thus \( ?P (\bigcup M) \) using \( \ast \) by auto

qed

qed (insert f-mono)

have \( \ast : W \cup (V \cap V) \cup \text{directly-attracted} p (V \cap V) \subseteq V \)

using \( W \subseteq V \)

attractor-step-bounded-by-V by auto

have \( ?A \subseteq V \) ?B \subseteq V using \( \ast \) by (simp add: lfp-lowerbound)

have \( ?A = ?f \; ?A \) using \( f\text{-mono lfp-unfold} \) by blast

hence \( ?A = W \cup (?A \cap V) \cup \text{directly-attracted} p (?A \cap V) \) using \( ?A \subseteq V \)

by simp

hence \( \ast : \text{attractor-step} p W ?A \subseteq ?A \) using \( ?A \subseteq V \)

inf absorbs1 by fastforce

have \( ?B = \text{attractor-step} p W ?B \) using \( \text{attractor-step-mono lfp-unfold} \) by blast

hence \( ?f \; ?B \subseteq ?B \) using \( ?B \subseteq V \)

by (metis (no-types, lifting) equalityD2 le-iff-inf)

have \( ?A = ?B \)

proof

show \( ?A \subseteq ?B \) using \( ?f \; ?B \subseteq ?B \)

by (simp add: lfp-lowerbound)

show \( ?B \subseteq ?A \) using \( \ast \)

by (simp add: lfp-lowerbound)

qed

hence \( ?P \; ?B \) using \( P\text{-A} \)

by (simp add: attractor-def)

thus \( \text{thesis} \) using \( ?B \subseteq V \)

by (simp add: attractor-def le-iff-inf)

qed

end — context ParityGame

end

7 Winning Strategies

theory WinningStrategy

imports

Main

Strategy

begin
context ParityGame begin

Here we define winning strategies.

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

**definition** winning-strategy :: Player \( \Rightarrow 'a \) Strategy \( \Rightarrow 'a \Rightarrow \text{bool} \) where

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

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

**assumes** \( \bigwedge P. \text{vmc-path } G P v_0 p \sigma \rightarrow \text{winning-path } P P \)

**shows** winning-strategy \( p \sigma v_0 \)

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

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

**assumes** \( \sigma \): winning-strategy \( p \sigma v \)

\( \text{and } v: v \in \text{lset } P \)

**shows** winning-path \( p P \)

**proof**

- **obtain** \( n \) **where** \( n: \text{enat } n < \text{llength } P P \$ n = v \) **using** \( v \) **by** \( \text{(meson in-lset-conv-lthh)} \)

- **interpret** \( P': \text{vmc-path } G \text{Idropn } n P v p \sigma \) **using** \( n \) **vmc-path-hdropn** **by** blast

- **have** winning-path \( P \) \( (\text{ldropn } n P) \) **using** \( \sigma \) **by** \( \text{(simp add: winning-strategy-def } P'.\text{vmc-path-axioms)} \)

- **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 \( p \sigma p \) \( \sigma' \) winning-strategy \( p \sigma v \)

\( \text{and } v: v \in V \)

**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 \( \sigma(2) \) \( v0-\text{set-P} \) **by** blast

- **moreover have** winning-path \( p \sigma \sigma' \)

- **using** comp-paths-hits-winning-strategy-is-winning \( \sigma'(2) \) \( v0-\text{set-P} \) **by** blast

- **ultimately show** False **using** \( P-\text{valid} \) \( \text{paths-are-winning-for-one-player} \) **by** blast

**qed**

7.1 Deadends

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

**assumes** \( v \in \text{VV } p \) \( \text{deadend } v \) strategy \( p \sigma \)

**shows** \( \neg \text{winning-strategy } p \sigma v \)

**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** \( \text{P-deadend-v0-LCons } \) \( \text{deadend } v \) **by** blast

- **hence** \( \neg \text{winning-path } p P \) **unfolding** \( \text{winning-path-def} \) **using** \( v \in \text{VV } p \) **by** auto

- **thus** ?thesis **using** \( \text{winning-strategy-def } \) \( \text{vmc-path-axioms} \) **by** blast

42
lemma winning-strategy-on-deadends:
assumes $v \in VV_p$ deadend $v$ strategy $p \sigma$
shows $\text{winning-strategy } p** \sigma v$

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

7.2 Extension Theorems

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

proof
fix $P$ assume $\text{vmc-path } G P (\sigma v0) p \sigma$
then interpret $\text{vmc-path } G P v0 p \sigma$.
have $v0 \rightarrow \sigma v0$ using $\sigma(1)$ strategy-def by blast
hence $\text{winning-path } p (LCons v0 P)$
using $\sigma(2)$ extension-valid-maximal-conforming $\text{winning-strategy-def}$ by blast
thus $\text{winning-path } p P$ using $\text{winning-path-ltl[of } p LCons v0 P]$ by simp
qed

lemma strategy-extends-VVp-star:
assumes $v0 : v0 \in VV_p** v0 \rightarrow w0$
and $\sigma : \text{strategy } p \sigma \text{ winning-strategy } p \sigma v0$
shows $\text{winning-strategy } p \sigma w0$

proof
fix $P$ assume $\text{vmc-path } G P w0 p \sigma$
then interpret $\text{vmc-path } G P w0 p \sigma$.
have $\text{winning-path } p (LCons v0 P)$
using $\text{extension-valid-maximal-conforming VV-impl1}$ $\sigma v0$ winning-strategy-def
by auto
thus $\text{winning-path } p P$ using $\text{winning-path-ltl[of } p LCons v0 P]$ by auto
qed

lemma strategy-extends-backwards-VVp-star:
assumes $v0 : v0 \in VV_p**$
and $\sigma : \text{strategy } p \sigma \oslash w. v0 \rightarrow w \Rightarrow \text{winning-strategy } p \sigma w$
shows $\text{winning-strategy } p \sigma v0$

proof
fix $P$ assume $\text{vmc-path } G P v0 p \sigma$
then interpret $\text{vmc-path } G P v0 p \sigma$.
show $\text{winning-path } p P$ proof (cases)
assume deadend $v0$
thus ?thesis using $\text{P-deadend-v0-LCons}$ winning-path-def $v0$ by auto
next

qed
assumes \neg \text{deadend } v0
then interpret \text{vmc-path-no-deadend } G P v0 p \sigma \text{ by unfold-locals}
interpret \text{ltlP: vmc-path } G \text{ ltl } P w0 p \sigma \text{ using vmc-path-lll } .
have \text{winning-path } p (\text{ ltl } P )
using \sigma(2) \text{ v0-edge-w0 } \text{vmc-path-lll winning-strategy-def by blast }
thus \text{winning-path } p P
using \text{winning-path-LCons by } (\text{metis P-LCons' ltlP.P-LCons ltlP.P-not-null})
qed

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

end — context ParityGame

8 Well-Ordered Strategy

theory WellOrderedStrategy
imports
  Main
  Strategy
begin

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

The following locale formalizes this idea.

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

locale WellOrderedStrategies = ParityGame +
fixes S :: 'a set
  and p :: Player
  — The set of good strategies on a node v
and good :: 'a ⇒ 'a Strategy set
  and r :: ('a Strategy × 'a Strategy) set
assumes S-V: S ⊆ V
  — r is a wellorder on the set of all strategies which are good somewhere.
and r-wo: well-order-on \{\sigma. \exists v \in S. \sigma \in \text{good } v}\ r
— Every node has a good strategy.
and good-ex: \(\forall v. v \in S \implies \exists \sigma. \sigma \in \text{good } v\)
— good strategies are well-formed strategies.
and good-strategies: \(\forall v. \sigma \in \text{good } v \implies \text{strategy } p \sigma\)
— A good strategy on \(v\) is also good on possible successors of \(v\).
and strategies-continue: \(\forall v w. \sigma. \sigma \in \text{good } v \implies \text{strategy } p \sigma w\)

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

The set of all strategies which are good somewhere.

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

definition minimal-good-strategy where
minimal-good-strategy \(v \sigma \equiv \sigma \in \text{good } v \land (\forall \sigma'. (\sigma', \sigma) \in r - Id \implies \sigma' \notin \text{good } v)\)

no-notation binomial (infix choose 65)

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

definition choose where
choose \(v \equiv \text{THE } \sigma. \text{minimal-good-strategy } v \sigma\)

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

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

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

lemma r-refl [simp]: refl-on Strategies \(r\)
  using r-wo unfolding well-order-on-def linear-order-on-def partial-order-on-def preorder-on-def by blast

lemma r-total [simp]: total-on Strategies \(r\)
  using r-wo unfolding well-order-on-def linear-order-on-def by blast

lemma r-trans [simp]: trans \(r\)
  using r-wo unfolding well-order-on-def linear-order-on-def partial-order-on-def preorder-on-def by blast

lemma r-wf [simp]: wf \((r - Id)\)
  using well-order-on-def r-wo by blast

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

lemma choose-works:
assumes \(v \in S\)
shows minimal-good-strategy \(v\) (choose \(v\))

proof –
  have \(wf: wf (r - Id)\) using well-order-on-def r-wo by blast
  obtain \(\sigma\) where minimal-good-strategy \(v \sigma\)
    unfolding minimal-good-strategy-def by (meson good-ex[of \(v \in S\)] \(wf\) wf-eq-minimal)
  hence \(\sigma: \sigma \in \text{good } v \land (\sigma', \sigma) \in r - Id \implies \sigma' \notin \text{good } v\)
  unfolding minimal-good-strategy-def by auto
  \{ fix \(\sigma'\) resume minimal-good-strategy \(v \sigma'\) \}
hence $\sigma' : \sigma' \in \text{good } v \land (\sigma, \sigma') \in r - Id \implies \sigma \notin \text{good } v$

unfolding minimal-good-strategy-def by auto
have $(\sigma, \sigma') \notin r - Id$ using $\sigma(1) \sigma'(2)$ by blast
moreover have $(\sigma', \sigma) \notin r - Id$ using $\sigma(2) \sigma'(1)$ by auto
moreover have $\sigma \in \text{Strategies}$ using $\sigma(1) \ (v \in S)$ by auto
moreover have $\sigma' \in \text{Strategies}$ using $\sigma'(1) \ (v \in S)$ by auto
ultimately have $\sigma' = \sigma$
using r-wo Linear-order-in-diff-Id well-order-on-Field well-order-on-def by fastforce
}
with $\sigma I$ have $\exists ! \sigma$. minimal-good-strategy $v \sigma$ by blast
thus ?thesis using theI $\{ \text{ of minimal-good-strategy } v, \text{ folded choose-def } \}$ by blast
qed

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

lemma well-ordered-strategy-valid: strategy $p$ well-ordered-strategy
proof -
{ 
  fix $v$ assume $v \in S \ v \in VV \ p \ \text{deadend } v$
  moreover have strategy $p \ (\text{choose } v)$
  using choose-works $\{ \text{ OF (v \in S) }$, unfolded minimal-good-strategy-def $\}$ good-strategies
  by blast
  ultimately have $v \rightarrow (\lambda v. \ \text{choose } v \ v) \ v$ using strategy-def by blast
}
thus ?thesis unfolding well-ordered-strategy-def using valid-strategy-updates-set by force
qed

8.1 Strategies on a Path
Maps a path to its strategies.
definition path-strategies $\triangleq \text{lmap choose}$

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

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

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

lemma path-strategies-monotone-Suc:
assumes $P$: lset $P \subseteq S$ valid-path $P$ path-conforms-with-strategy $p$ $P$ well-ordered-strategy
\[ \text{enat} \ (\text{Suc} \ n) < \text{llength} \ P \]
shows (path-strategies $P \subseteq \text{Suc} \ n$, path-strategies $P \subseteq n$) $\in r$
proof -
\begin{itemize}
  \item define $P'$ where $P' = \text{idropn} \ n \ P$
  \item hence $\text{enat} \ (\text{Suc} \ 0) < \text{llength} \ P' \ using \ P(4)$
    \begin{itemize}
      \item by (metis $\text{enat}$-llt-Suc $\text{idropn}$-Suc-$\text{idropn}$-Suc-$\text{conv}$-$\text{idropn}$-lmap-[\text{disc}(2)$ $\text{lnull}$-0-$\text{llength}$ $\text{llt}$-$\text{idropn}$])
    \end{itemize}
  \item then obtain $v \ w P$s where $w' = L\text{Cons} \ v \ (L\text{Cons} \ w \ P$s)
    \begin{itemize}
      \item by (metis $\text{idropn}$-$\text{idropn}$-$\text{Suc}$-$\text{conv}$-$\text{idropn}$-ldropn-$\text{lnull}$-$\text{lnull}$-$\text{llength}$)
    \end{itemize}
  \item moreover have lset $P' \subseteq S$ unfolding $P'$-def using $P(1)$ lset-$\text{idropn}$-subset[of $n$ $P$] by blast
  \item ultimately have $v \in S$ $w \in S$ by auto
  \item moreover have $v \mapsto w$ using valid-path-edges$[[v \ w P$s, folded $vw$] valid-$\text{drop}[[OF P(2)]]$ $P'$-def by blast
  \item hence $*:(\text{choose} \ v, \text{choose} \ w) \notin r$ $\text{Id}$ using choose-minimal $w \in S$ by blast
\end{itemize}

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

hence (path-strategies $P' \subseteq \text{Suc} \ 0$, path-strategies $P' \subseteq 0$) $\in r$
unfolding path-strategies-def using $vw$ by simp
thus $?\text{thesis}$ unfolding path-strategies-def $P'$-def
\begin{itemize}
  \item using lmap-$\text{idropn}$[OF $\text{Suc}$-$\text{llength}$[OF $P(4)$]], of choose]
    \item lmap-$\text{idropn}$-$\text{Suc}$(OF $P(4)$, of choose]
    \item by simp
\end{itemize}
qed

lemma path-strategies-monotone:
assumes $P$: lset $P \subseteq S$ valid-path $P$ path-conforms-with-strategy $p$ $P$ well-ordered-strategy
\[ n < m \ \text{enat} \ m < \text{llength} \ P \]
shows (path-strategies $P \subseteq m$, path-strategies $P \subseteq n$) $\in r$
using assms proof (induct $m - n \text{ arbitrary: n}$ $m$)
\begin{itemize}
  \item case (Suc $d$)
\end{itemize}

47
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 ≠ 0
  have m ≠ 0 using Suc.hyps(2) by linarith
  then obtain m' where m' = m using not0-implies-Suc by blast
  hence d = m' - n using Suc.hyps(2) by presburger
  moreover hence n < m' using d ≠ 0 by presburger
  ultimately have (path-strategies P $ m', path-strategies P $ n) ∈ r
    using Suc.hyps(1)[of m', OF - P(1,2,3)] Suc.prems(5) dual-order.strict-trans enat-ond-simps(2)
  m'
    by blast
  thus thesis
    using m' path-strategies-monotone-Suc[OF P(1,2,3)] by (metis (no-types) Suc.prems(5)
r-trans tvms-def)
qed
qed simp

lemma path-strategies-eventually-constant:
  assumes ¬finite P let P ⊆ S valid-path P path-conforms-with-strategy p P well-ordered-strategy
  shows ∃ n. ∀ m ≥ n. path-strategies P $ n = path-strategies P $ m
proof -
  define σ-set where σ-set = lset (path-strategies P)
  have ∃ σ. σ ∈ σ-set unfolding σ-set-def path-strategies-def
    using assms(1) finite-lmap lset-nth-member-inf by blast
  then obtain σ' where σ' ∈ σ-set \ τ. (τ, σ') ∈ r - Id ⟹ τ ∉ σ-set
    using wfE-min[of r - Id - σ-set] by auto
  obtain n where n: path-strategies P $ n = σ'
    using σ'(1) lset-lnth[of σ'] unfolding σ-set-def by blast
  { fix m assume n ≤ m
    have path-strategies P $ n = path-strategies P $ m proof (rule ccontr)
      assume *: path-strategies P $ n ≠ path-strategies P $ m
      with (n ≤ m) have n < m using le-imp-less-or-eq by blast
      with path-strategies-monotone have (path-strategies P $ m, path-strategies P $ n) ∈ r
        using assms by (simp add: infinite-small-length)
      with * have (path-strategies P $ m, path-strategies P $ n) ∈ r - Id by simp
      with σ'(2) n have path-strategies P $ m ∉ σ-set by blast
      thus False unfolding σ-set-def path-strategies-def
        using assms(1) finite-lmap lset-nth-member-inf by blast
      qed
    } thus thesis by blast
qed
8.2 Eventually One Strategy

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

\textbf{lemma} \quad \textit{path-eventually-conforms-to-σ-map-n}:
\begin{itemize}
    \item \textbf{assumes} \quad \textit{lset P} \subseteq \textit{S} \quad \textit{valid-path P} \textit{path-conforms-with-strategy} \textit{p P} \textit{well-ordered-strategy}
    \item \textbf{shows} \quad \exists \textit{n}. \quad \textit{path-conforms-with-strategy p (ldropn n P)} \quad \textit{(path-strategies P} \ \textit{§} \ \textit{n)}
\end{itemize}

\textbf{proof} \quad (\textit{cases})
\begin{itemize}
    \item \textbf{assume} \quad \textit{finite P}
        \textbf{then obtain} \quad \textit{n} \quad \textbf{where} \quad \textit{llength P} = \textit{enat n} \quad \textbf{using} \quad \textit{finite-length-enat} \quad \textbf{by} \quad \textit{blast}
    \item \textbf{hence} \quad \textit{ldropn n P} = \textit{LNil} \quad \textbf{by} \quad \textit{simp}
    \item \textbf{thus} \quad \textit{?thesis} \quad \textbf{by} \quad (\textit{metis path-conforms-LNil})
\end{itemize}

\textbf{next}
\begin{itemize}
    \item \textbf{assume} \quad \neg\textit{finite P}
        \textbf{then obtain} \quad \textit{n} \quad \textbf{where} \quad \textit{n}: \quad \land \textit{m}. \quad \textit{n} \leq \textit{m} \quad \rightarrow \quad \textit{path-strategies P} \ \textit{§} \ \textit{n} = \textit{path-strategies P} \ \textit{§} \ \textit{m}
    \item \quad \textbf{using} \quad \textit{path-strategies-eventually-constant} \quad \textbf{assms} \quad \textbf{by} \quad \textit{blast}
    \item \quad \textbf{let} \quad \textit{?σ} = \textit{well-ordered-strategy}
    \item \quad \textbf{define} \quad \textit{P' where} \quad \textit{P' = ldropn n P}
        \begin{itemize}
            \item \textbf{fix} \quad \textit{v assume} \quad \textit{v} \in \textit{lset P'}
                \textbf{hence} \quad \textit{v} \in \textit{S} \quad \textbf{using} \quad \langle \textit{lset P} \subseteq \textit{S} \rangle \quad \textit{P'}-def \quad \textbf{by} \quad \textit{fastforce}
            \item \quad \textbf{from} \quad \langle \textit{v} \in \textit{lset P'} \rangle \quad \textbf{obtain} \quad \textit{m} \quad \textbf{where} \quad \textit{m}: \quad \textit{enat m} < \textit{llength P'} \quad \textit{P'} \ \textit{§} \ \textit{m} = \textit{v} \quad \textbf{by} \quad (\textit{meson in-lset-conv-lth})
                \item \quad \textbf{hence} \quad \textit{P} \ \textit{§} \ \textit{m + n} = \textit{v} \quad \textbf{unfolding} \quad \textit{P'}-def \quad \textbf{by} \quad (\textit{simp add:} \quad \langle \neg\textit{finite P} \rangle \quad \textit{infinite-small-length})
            \item \quad \textbf{moreover have} \quad \langle \textit{?σ} \textit{v} = \textit{choose v v} \textit{unfolding} \quad \textit{well-ordered-strategy-def} \quad \textbf{using} \quad \langle \textit{v} \in \textit{S} \rangle \quad \textbf{by} \quad \textit{auto}
                \item \quad \textbf{ultimately have} \quad \langle \textit{?σ} \textit{v} = \textit{(path-strategies P} \ \textit{§} \ \textit{m + n}) \ \textit{v}\rangle
            \item \quad \textbf{unfolding} \quad \textit{path-strategies-def} \quad \textbf{using} \quad \textit{infinite-small-length} \quad \textbf{by} \quad \textit{simp}
                \item \quad \textbf{hence} \quad \langle \textit{?σ} \textit{v} = \textit{(path-strategies P} \ \textit{§} \ \textit{n}) \ \textit{v}\rangle \quad \textbf{using} \quad \langle \textit{of} \ \textit{m + n}\rangle \quad \textbf{by} \quad \textit{simp}
        \end{itemize}
    \item \quad \textbf{moreover have} \quad \textit{path-conforms-with-strategy p P' well-ordered-strategy}
        \textbf{unfolding} \quad \textit{P'}-def \quad \textbf{by} \quad (\textit{simp add:} \quad \textit{assms(3) path-conforms-with-strategy-drop})
    \item \quad \textbf{ultimately show} \quad \textit{?thesis}
        \textbf{using} \quad \textit{path-conforms-with-strategy-irrelevant-updates P'-def} \quad \textbf{by} \quad \textit{blast}
\end{itemize}

\textbf{qed}

\textbf{end} — \textbf{WellOrderedStrategies}

\textbf{end}

9 Winning Regions

\textbf{theory} \quad \textbf{WinningRegion}

\textbf{imports}
\begin{itemize}
    \item \textbf{Main}
    \item \textbf{WinningStrategy}
\end{itemize}

\textbf{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.

\textbf{context} \quad \textbf{ParityGame} \quad \textbf{begin}
**Definition**

\[ \text{winning-region } p \equiv \{ v \in V. \exists \sigma. \text{strategy } p \sigma \land \text{winning-strategy } p \sigma v \} \]

**Lemma** \text{winning-region-1} [intro]:

\begin{align*}
\text{assumes } & v \in V \text{ strategy } p \sigma \text{ winning-strategy } p \sigma v \\
\text{shows } & v \in \text{winning-region } p \\
\text{using } & \text{assms unfolding winning-region-def by blast}
\end{align*}

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

**Lemma** \text{winning-region-deadends}:

\begin{align*}
\text{assumes } & v \in VP \text{ deadend } v \\
\text{shows } & v \in \text{winning-region } p** \\
\text{proof } & \text{show } v \in V \text{ using } (v \in VVp) \text{ by blast} \\
& \text{show winning-strategy } p** \sigma -\text{arbitrary } v \text{ using } \text{assms winning-strategy-on-deadends by simp}
\end{align*}

**QED simp**

### 9.1 Paths in Winning Regions

**Lemma** \text{(in vmc-path) paths-stay-in-winning-region}:

\begin{align*}
\text{assumes } & \sigma': \text{strategy } p' \sigma' \text{ winning-strategy } p \sigma' v0 \\
& \text{and } \sigma: \bigwedge v. v \in \text{winning-region } p \implies \sigma' v = \sigma v \\
\text{shows } & \text{let } P \subseteq \text{winning-region } p \\
\text{proof } & \text{fix } x \\text{ assume } x \in \text{lset } P \\
& \text{thus } x \in \text{winning-region } p \text{ using } \text{assms vmc-path-axioms} \\
\text{proof } (\text{induct arbitrary: } v0 \text{ rule: list-set-induct}) \\
& \text{case } (\text{find } P v0) \\
& \text{interpret } \text{vmc-path } G P v0 p \sigma \text{ using find.prem(4)} \\
& \text{show } \text{case using } P-v0 \sigma'(1) \text{ find.prem(2) v0-V unfolding winning-region-def by blast}
\end{align*}

**Next**

\begin{align*}
& \text{case } (\text{step } P x v0) \\
& \text{interpret } \text{vmc-path } G P v0 p \sigma \text{ using step.prem(4)} \\
& \text{show } \text{case proof (cases)} \\
& \quad \text{assume } \text{null } (\text{ltl } P) \\
& \quad \text{thus } \text{thesis using } P-null-llt-LCons step.hyps(2) \text{ by auto}
\end{align*}

**Next**

\begin{align*}
& \quad \text{assume } \neg \text{null } (\text{ltl } P) \\
& \quad \text{then interpret } \text{vmc-path-no-deadend } G P v0 p \sigma \text{ using } P-no-deadend-v0 \text{ by unfold-locales} \\
& \quad \text{have } \text{winning-strategy } p \sigma' v0 \text{ proof (cases)} \\
& \quad \quad \text{assume } v0 \in VVp \\
& \quad \quad \text{hence } \text{winning-strategy } p \sigma' (v0) \\
& \quad \quad \text{using } \text{strategy-extends-VVp local.step(4) step.prem(2) v0-no-deadend by blast}
\end{align*}

**Moreover**

\begin{align*}
& \quad \text{moreover have } v0 = w0 \text{ using } v0\text{-conforms (v0 } \in VVp) \text{ by blast} \\
& \quad \text{moreover have } \sigma' v0 = \sigma v0 \\
& \quad \text{using } \sigma \text{ assms(1) step.prem(2) v0-V unfolding winning-region-def by blast}
\end{align*}

**Ultimately**

\begin{align*}
& \text{show } \text{thesis by simp} \\
& \text{next} \\
& \quad \text{assume } v0 \notin VVp \\
& \quad \text{thus } \text{thesis using v0-V strategy-extends-VVpstar step(4) step.prem(2) by simp}
\end{align*}

**QED**
thus \thesis using \step hyps(3) \step(4) \sigma \vmc-path-iltl by blast
qed
qed
qed


lemma \in \vmc-path \path-hits-winning-region-is-winning:
assumes \sigma': strategy \sigma' \land v. v \in \winning-region p \Rightarrow \winning-strategy p \sigma' v
and \sigma: \land v. v \in \winning-region p \Rightarrow \sigma' v = \sigma v
and \Pi: let P \cap \winning-region p \neq \emptyset
shows \winning-path p P

proof -

\obtain n where n: enat n < llength P P \notin n \in \winning-region p
using P by (meson lset-intersect-lnth)
de\fice P' where P' = ldrop n P
then interpret P': \vmc-path G P' P \notin n p \sigma
unfolding P'-def using \vmc-path-ldropn n(1) by blast
have winning-strategy p \sigma'(P \notin n) using \sigma'(2) n(2) by blast
hence let P' \subseteq \winning-region p
using P'\paths-stay-in-winning-region[OF \sigma'(1) - \sigma]
by blast
hence \land v. v \in lset P' \Rightarrow \sigma v = \sigma' v \\using \sigma by auto
hence path-conforms-with-strategy p P' \sigma'
using path-conforms-with-strategy-irrelevant-updates P'.P-conforms
by blast
then interpret P': \vmc-path G P' P \notin n p \sigma' \\using P'.conforms-to-another-strategy by blast
have winning-path p P' using \sigma'(2) n(2) P', \vmc-path-axioms winning-strategy-def by blast
thus winning-path p P unfolding P'-def using winning-path-drop-add n(1) P-valid by blast
qed

9.2 Irrelevant Updates

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

lemma winning-strategy-updates:
assumes \sigma: strategy \sigma \\\using \sigma v0
and v: v \notin \winning-region p \\\\using v \rightarrow w
shows winning-strategy p (\sigma(v := w)) v0

proof -
fix P assume \vmc-path G P v0 p (\sigma(v := w))
then interpret \vmc-path G P v0 p \sigma(v := w) .
have \land v'. v' \in \winning-region p \Rightarrow \sigma v' = (\sigma(v := w)) v' \\using v by auto
hence v \notin lset P using v \paths-stay-in-winning-region \sigma unfolding winning-region-def by blast
hence path-conforms-with-strategy p P \sigma
using P-conforms path-conforms-with-strategy-irrelevant' by blast
thus winning-path p P using conforms-to-another-strategy \sigma(2) \\using winning-strategy-def by blast
qed

9.3 Extending Winning Regions

lemma winning-region-extends-VVp:
assumes v: v \in VV p \\\\using v \rightarrow w and w: w \in \winning-region p
shows v \in \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(1) (v→w) by blast
  moreover hence winning-strategy p ?σ 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-VVp** 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-VVp** as soon as we have this.

end — context ParityGame

end

10 Uniform Strategies

Theorem 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: \( \forall v. v \in S \rightarrow \exists \sigma. \text{strategy } p \sigma \land \text{strategy-attracts-via } p \sigma v S W \)
  shows \( \exists \sigma. \text{strategy } p \sigma \land \text{strategy-attracts } p \sigma S W \)
proof-
  define good where good v = \{ \sigma. \text{strategy } p \sigma \land \text{strategy-attracts-via } p \sigma v S W \} \text{ for } v
  let ?G = \{ \sigma. \exists v \in S - W. \sigma \in 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 \( \forall v. v \in S - W \rightarrow \exists \sigma. \sigma \in good v \) unfolding good-def using strategies-ex by blast
next
  show \( \forall v. \sigma \in good v \rightarrow \text{strategy } p \sigma \) unfolding good-def by blast
next
  fix v w σ assume \( v. v \in S - W \rightarrow v \in V V p \rightarrow \sigma v = w \sigma \in good v \)
  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 \ \in \ S \ W \ \text{using} \ \text{strategy-attracts-\ via-\ successor} \ v \ \text{by} \ \text{blast}

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

\text{qed} \ (\text{insert \ } r)

\text{have} \ S-W-no-deadends: \ \forall \ v. \ v \in S - W \ \Rightarrow \ \neg \ \text{deadend} \ v

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

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

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

\text{hence} \ \text{bset} \ P \ \subseteq \ S - W \ \text{proof} \ (\text{induct rule: vmc-path-bset-induction})
\text{case base}
\text{show} \ v0 \in S - W \ \text{using} \ (v0 \in S) \ \text{contra \ visits-via-trivial \ by \ blast}
\text{next}
\text{case} \ (\text{step} \ P \ v0)
\text{interpret} \ \text{vmc-path-no-deadend} \ G \ P \ v0 \ p \ \text{well-ordered-strategy using} \ \text{step.hyps(1)} .
\text{have} \ \text{insert} \ v0 \ S = S \ \text{using} \ \text{step.hyps(2)} \ \text{by \ blast}
\text{hence} \ *: \ \neg \ \text{visits-via} \ (\text{ltl} \ P) \ S \ W

\text{using} \ \text{visits-via-LCons[of \ ltl \ P \ S \ W \ v0, \ folded \ P-LCons] \ step.hyps(3) \ by \ auto}

\text{hence} **: \ v0 \notin W \ \text{using} \ \text{vmc-path.visits-via-trivial[OF \ vmc-path-ltl]} \ \text{by \ blast}
\text{have} \ v0 \in S \cup W \ \text{proof} \ (\text{cases})
\text{assume} \ v0 \in VV p
\text{hence} \ \text{well-ordered-strategy} \ v0 = w0 \ \text{using} \ v0-conforms \ \text{by \ blast}
\text{hence} \ \text{choose} \ v0 = w0 = w0 \ \text{using} \ \text{step.hyps(2)} \ \text{well-ordered-strategy-def} \ \text{by \ auto}
\text{moreover have} \ \text{strategy-attracts-\ via} \ p \ \text{choose} \ v0 \ v0 \ S \ W

\text{using} \ \text{choose-good \ good-def \ step.hyps(2) \ by \ blast}
\text{ultimately show} \ ?\text{thesis}
\text{by} \ (\text{metis \ strategy-attracts-\ via-successor \ strategy-attracts-\ via-v0}
\text{strategy-attracts-\ via-successor}
\text{strategy-attracts-\ via-v0} \ \text{v0-edge-w0} \ w0-V)

\text{qed} \ (\text{metis \ D1D1 \ assms(2) \ step.hyps(2) \ strategy-attracts-\ via-successor}
\text{strategy-attracts-\ via-v0} \ \text{v0-edge-w0} \ w0-V)

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

\text{have} \ \neg \ \text{finite} \ P \ \text{proof}
\text{assume} \ \text{finite} \ P
\text{hence} \ \text{deadend} \ \text{(last} \ P) \ \text{using} \ \text{P-maximal \ P-not-null \ maximal-ends-on-deadend \ by \ blast}
\text{moreover have} \ \text{last} \ P \in S - W \ \text{using} \ \text{bset} \ P \\subseteq \ S - W \ \text{P-not-null \ (finite} \ P) \ \text{finite-bset}
\text{by \ blast}
\text{ultimately show} \ \text{False \ using} \ S-W-no-deadends \ \text{by \ blast}
\text{qed}

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

\text{using} \ \text{path-eventually-conforms-to-} \ \sigma - \text{map-n[OF \ lset} \ P \ \subseteq \ S - W \ \text{P-valid \ P-conforms]}
\text{by \ blast}
\text{define} \ \sigma' \ \text{where} \ \text{[simp]}: \ \sigma' = \ \text{path-strategies} \ P \ \text{S} \ n
\text{define} \ P' \ \text{where} \ \text{[simp]}: \ P' = \ \text{ldropn} \ n \ P

53
interpret \textit{vmc-path} $G \ P' \ \text{ld} \ P' \ p \ \sigma'$
proof
show $\neg\text{null } P' \ \text{unfolding } P'-\text{def}$
using ($\neg\text{finite } P' \ \text{finite-ldropn \ null-imp-finite \ by blast}$)
qed (simp-all add: n)
have strategy $p \ \sigma' \ \text{unfolding } \sigma'-\text{def}$
using path-strategies-strategy (let $P \subseteq S - W$) ($\neg\text{finite } P' \ \text{finite-small-length}$)
by blast
moreover have strategy-attracts-via $p \ \sigma' \ (\text{ld } P') \ S \ W$ proof --
have $P \ \sigma' \\ n \in S - W$ using (let $P \subseteq S - W$) ($\neg\text{finite } P' \ \text{finite-nth-member-inf \ by blast}$)
have $\sigma' \ \in \ \text{good } (P \ \sigma' \ n)$
using path-strategies-good $\sigma'-\text{def}$ ($\neg\text{finite } P' \ \text{finite-nth-member-inf \ by blast}$)
moreover from (let $P \subseteq S - W$) have lset $P' \subseteq S - W$
unfolding $P'-\text{def}$ using lset-ldropn-subset[of n P] by blast
ultimately show False using strategy-attracts-via-lset by blast
qed

thus ?thesis using well-ordered-strategy-valid by blast
qed

10.2 A Uniform Winning Strategy

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

lemma merge-winning-strategies:
shows $\exists \sigma. \ \text{strategy } p \ \sigma \ \text{and } (\forall v \in \text{winning-region } p. \ \text{winning-strategy } p \ \sigma \ \forall v)$
proof --
define good where good $v = \{ \sigma. \ \text{strategy } p \ \sigma \ \text{and } \text{winning-strategy } p \ \sigma \ \forall v \}$
let $?G = \{ \sigma. \ \exists v \in \text{winning-region } p. \ \sigma \ \in \ \text{good } v \}$
obtain $r$ where $r$: well-order-on $?G \ r \ \text{using well-order-on \ by blast}$

have no-\text{VVp-deadends}: $\forall v. \ [v \in \text{winning-region } p \ ; v \in \text{VV } p] \ \implies \neg\text{deadend } v$
using no-winning-strategy-on-deadends unfolding winning-region-def by blast

interpret WellOrderedStrategies $G$ winning-region $p$ good $r$
proof
show $\forall v. \ v \in \text{winning-region } p \ \implies \exists \sigma. \ \sigma \ \in \ \text{good } v$
unfolding good-def winning-region-def by blast
next
show $\forall \sigma. \ \sigma \ \in \ \text{good } v \ \implies \ \text{strategy } p \ \sigma \ \text{unfolding } \text{good-def \ by blast}$
next
fix $v \ \sigma$ assume $v: v \in \text{winning-region } p \ \implies v \in \text{VV } p \ \implies \sigma \ v = w \ \sigma \ \in \ \text{good } v$
hence $\sigma$: strategy $p \ \sigma$ winning-strategy $p \ \sigma$ unfolding good-def by simp-all
hence winning-strategy $p \ \sigma \ w$
proof (cases)
assume $*: v \in \text{VV } p$
hence $**: \sigma \ v = w$ using $v(3)$ by blast
have $\neg$deadend $v$ using no-\text{VVp-deadends} ($v \in \text{VV } p$) $v(1)$ by blast
with $**$ show ?thesis using strategy-extends-VVp $\sigma$ by blast
next
assume $v \notin \text{VV } p$
thus \textit{thesis using} strategy-extends-VV_{p\sigma} \langle v \rightarrow w \rangle \text{ by blast}\\
\textit{qed}\\
thus \sigma \in \text{good } w \text{ unfolding} \quad \textit{good-def using} \quad \sigma(1) \text{ by blast}\\
\textit{qed (insert winning-region-in-Vr)}

\{\\
\text{fix } v_0 \text{ assume } v_0 \in \text{winning-region } p\\
\text{fix } P \text{ assume } P: \text{vmc-path } G \ W \ v_0 \ W \ p \text{ well-ordered-strategy}\\
\text{then interpret } \text{vmc-path } G \ W \ v_0 \ W \ p \text{ well-ordered-strategy}.

\text{have lset } P \subseteq \text{winning-region } p \text{ proof (induct rule: vmc-path-lset-induction-simple)}\\
\text{case (step } P \ v_0)\\
\text{interpret } \text{vmc-path-no-deadend } G \ W \ v_0 \ W \ p \text{ well-ordered-strategy using step.hyps(1)}.
\{ \text{ assume } v_0 \in V V \ W p \\
\text{ hence well-ordered-strategy } v_0 = w_0 \text{ using } v_0\text{-conforms by blast} \\
\text{ hence choose } v_0 v_0 = w_0 \text{ by } (\text{simp add: step.hyps(2) well-ordered-strategy-def}) \}
\text{ hence choose } v_0 \in \text{good } w_0 \text{ using strategies-continue choose-good step.hyps(2) by simp}\\
\text{thus ?case unfolding good-def winning-region-def using } w_0\text{-V by blast}\\
\textit{qed (insert :} v_0 \in \text{winning-region } p)\\

\text{have winning-path } P \ W \ p \text{ proof (rule contr)}\\
\text{assume contra: } \neg \text{winning-path } P \ W \ p\\

\text{have } \neg \text{finite } P \text{ proof}\\
\text{assume } \text{finite } P \\
\text{ hence deadend (last } P) \text{ using maximal-ends-on-deadend by simp}\\
\text{ moreover have } \text{last } P \in \text{winning-region } p \\
\text{ using } (lset \subseteq \text{winning-region } p); \text{P-not-null } \text{finite } P; \text{finite-lset by blast}\\
\text{ moreover have } \text{last } P \in V V \ W \ p \\
\text{ using } \text{contr paths-are-winning-for-one-player } \text{finite } P; \\
\text{ unfolding winning-path-def by simp}\\
\text{ ultimately show } \text{False using } \neg \text{no-VVp-deadends by blast}\\
\textit{qed}\\

\text{obtain } n \text{ where } n: \text{path-conforms-with-strategy } p (\text{idropn } n \ P) (\text{path-strategies } P \ \text{\$ } n) \\
\text{ using } \text{path-eventually-conforms-to-\sigma-map-n}[OF } \text{lset } P \subseteq \text{winning-region } p; \text{P-valid } \text{P-conforms}] \\
\text{by blast}\\
\text{define } \sigma' \text{ where } [\text{simp}]: \sigma' = \text{path-strategies } P \ \text{\$ } n \\
\text{define } P' \text{ where } [\text{simp}]: \ P' = \text{idropn } n \ P \\
\text{interpret } P': \text{vmc-path } G \ W \ P' \text{ lhd } P' \ W \ \sigma' \text{ proof} \\
\text{ show } \neg \text{lnull } P' \text{ using (\neg } \text{finite } P; \text{unfolding } P'\text{-def} \\
\text{ using } \text{finite-idropn } \text{null-imp-finite by blast}\\
\textit{qed (simp-all add: } n)\\
\text{have } \text{strategy } P \ W \ \sigma' \text{ unfolding } \sigma'\text{-def} \\
\text{ using } \text{path-strategies-strategy } (\text{lset } P \subseteq \text{winning-region } p); (\neg \text{finite } P) \text{ by blast}\\
\text{ moreover have } \text{winning-strategy } P \ W \ \sigma' (\text{lhd } P') \text{ proof --} \\
\text{ have } P \ W \ n \in \text{winning-region } p \\
\text{ using } (\text{lset } P \subseteq \text{winning-region } p); (\neg \text{finite } P) \text{ lset-nth-member-inf by blast} \\
\text{ hence } \sigma' \in \text{good } (P \ W \ n) \\
\text{ using } \text{path-strategies-good choose-good } \sigma'\text{-def } (\neg \text{finite } P); (\text{lset } P \subseteq \text{winning-region } p) \\
\text{ by blast}
hence winning-strategy \( p \sigma' (P \notin n) \) unfolding good-def by blast
thus \( \exists \)thesis
  unfolding \( P'\)-def using \( P-0 \) (\( \neg \)finite \( P \)) by (simp add: infinite-small-length lhd-idropn)
qed
ultimately have winning-path \( p \) \( P' \) unfolding winning-strategy-def
  using \( P'\)-vmc-path-axioms by blast
moreover have \( \neg \)finite \( P' \) \( P'\)-def by simp
ultimately show False using contra winning-path-drop-add[OF \( P\)-valid] by auto
qed
}

thus \( \exists \)thesis unfolding winning-strategy-def using well-ordered-strategy-valid by auto
qed

10.3 Extending Winning Regions

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

lemma winning-region-extends-\( VVp\)star:
  assumes v: \( v \in VV p** \) and w: \( \land w. v \rightarrow w \implies w \in \text{winning-region} p \)
  shows v \( \in \) winning-region p
proof -
  obtain \( \sigma \) where \( \sigma \): strategy \( p \) \( \sigma \) \( \land v. v \in \text{winning-region} p \implies \text{winning-strategy} p \) \( \sigma \) \( v \)
    using merge-winning-strategies by blast
  have winning-strategy \( p \) \( \sigma \) \( v \) using strategy-extends-backwards-\( VVp\)star[OF \( v \sigma(1) \) \( \sigma(2) \) \( w \) by blast
  thus \( \exists \)thesis unfolding winning-region-def using \( v \) \( \sigma(1) \) by blast
qed

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

lemma removing-winning-region-induces-no-deadends:
  assumes v \( \in V \) - winning-region p \( \neg \)deadend v
  shows \( \exists w \in V \) - winning-region p. \( v \rightarrow w \)
  using assms winning-region-extends-\( VVp \) winning-region-extends-\( VVp\)star by blast

end — context ParityGame

end

11 Attractor Strategies

theory AttractorStrategy
imports
  Main
  Attractor UniformStrategy
begin

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

context ParityGame begin

lemma strategy-attracts-extends-\( VVp \):

assumes $\sigma$: strategy $p \sigma$ strategy-attracts $p \sigma S W$
and $v0: v0 \in VV p v0 \in \text{directly-attracted} p S v0 \notin S$
shows $\exists \sigma. \text{strategy } p \sigma \land \text{strategy-attracts-via } p \sigma v0 (\text{insert } v0 S) W$

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

fix $P$
assume $\text{vmc-path } G P v0 p ?\sigma$
then interpret $\text{vmc-path } G P v0 p ?\sigma$.
have $\neg \text{deadend } v0$ using $(v0 \rightarrow w)$ by blast
then interpret $\text{vmc-path-no-deadend } G P v0 p ?\sigma$ by unfold-locales

define $P''$ where $[\text{simp}]: P'' = \text{llt } P$
have $\nu d P'' = w$ using $v0(1) v0$-conforms $w0$-def by auto
hence $\text{vmc-path } G P'' w p ?\sigma$ using $\text{vmc-path-ltl}$ by $(\text{simp add: } w0$-def $)$

have $*: v0 \notin S \setminus W$ using $(v0 \notin S)$ by blast
have override-on $\langle \sigma(w := w) \rangle \sigma (S \setminus W) = ?\sigma$
by (rule ext) (metis $*$ $\text{fun-upd-def override-on-def}$)
hence $\text{strategy-attracts } p ?\sigma S W$
using $\text{strategy-attracts-irrelevant-override}\langle \text{OF } \sigma(2,1) \sigma \text{strategy } p ?\sigma \rangle$ by simp
hence $\text{strategy-attracts-via } p ?\sigma w S W \text{ unfolding } \text{strategy-attracts-def}$
using $(w \in S)$ by blast
hence $\text{visits-via } P'' S W \text{ unfolding } \text{strategy-attracts-via-def}$
using $\text{vmc-path } G P'' w p ?\sigma$ by blast
thus $\text{visits-via } P (\text{insert } v0 S) W$
using $\text{visits-via-LCons[of llt } P S W v0]\text{ P-LCons}$ by simp
qed
ultimately show $\neg \text{thesis}$ by blast
qed

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

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

lemma attractor-has-strategy-single:
assumes $W \subseteq V$
and $v0$-def: $v0 \in \text{attractor } p W (\text{is } - \in ?A)$
definition shows \( \exists \sigma. \text{strategy } p \sigma \land \text{strategy-attracts-via } p \sigma v0 \neq A W \)

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

case (step S)

have \( v0 \in W \implies \exists \sigma. \text{strategy } p \sigma \land \text{strategy-attracts-via } p \sigma v0 \} \) W

using strategy-attracts-via-trivial valid-arbitrary-strategy by blast

moreover {

assume \( s : v0 \in \text{directly-attracted } p S \) \( v0 \notin S \)

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 \( s \)

have \( \exists \sigma. \text{strategy } p \sigma \land \text{strategy-attracts-via } p \sigma v0 \) (insert \( v0 S \)) W

using strategy-attracts-extends-VVp strategy-attracts-extends-VVpstar by blast

ultimately show ?case

using step.prems step.hyps (2)

attractor-strategy-on-extends [of \( p - v0 \) insert \( v0 S \)] \( S W W \cup S \cup \text{directly-attracted } p S \)

attractor-strategy-on-extends [of \( p - v0 S \)] \( W W \cup S \cup \text{directly-attracted } p S \)

attractor-strategy-on-extends [of \( p - v0 \) \( \} \)] \( W W \cup S \cup \text{directly-attracted } p S \)

by blast

next

case (\( \cup \) union \( M \))

hence \( \exists S. S \in M \land v0 \in S \) by blast

thus ?case by (meson Union-upp 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 \( ?A = \text{attractor } p W \)

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
12.1 Induction Step

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

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

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

**Assumes** $v \in V$

**and no-deadends:** $\forall v. v \in V \implies \neg \text{deadend } v$

**and III:** $\forall (G :: (a, b) \text{ ParityGame-scheme}) v.$

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

$\forall v. v \in V_G \implies \neg \text{Digraph.deadend } G v ]$

$\implies \exists p. v \in \text{ParityGame, winning-region } G p$

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

**Proof**

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

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

**Have** $\exists p. \text{winning-priority } p \text{ min-prio by auto}$

**Then obtain** $p \text{ where } p: \text{winning-priority } p \text{ min-prio by blast}$

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** $W1 \text{ where } W1 = \text{winning-region } p^{**}$

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

**Define** $U \text{ where } U = V \setminus W1$

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

**Define** $V' \text{ where } V' = U \setminus \text{attr actor } p \cap K$

**Define** $G' \text{ where } [\text{simp}]: G' = \text{sub game } V'$

**Interpret** $G': \text{ParityGame } G' \text{ using } \text{sub game-ParityGame by simp}$

**Have** $U-\text{equiv} \forall v. v \in V \implies v \in U \iff v \notin \text{winning-region } p^{**}$

**Unfolding** $U-\text{def } W1-\text{def by blast}$

**Have** $V' \subseteq V \text{ unfolding } U-\text{def } V'-\text{def by blast}$

**Hence** $[\text{simp}]: V_G' = V' \text{ unfolding } G'-\text{def by simp}$

**Have** $V_G' \subseteq V \in G' \subseteq E \omega_G' = \omega \text{ unfolding } G'-\text{def by (simp-all add: sub game-}\omega)$

**Have** $G'.VVp = V' \cap VVp \text{ unfolding } G'-\text{def using } \text{sub game-VV by simp}$
have $V$-decomp: $V = \text{attractor } p K \cup V' \cup W1$ proof –
have $V \subseteq \text{attractor } p K \cup V' \cup W1$
unfolding $V'$-def $U$-def by blast
moreover have $\text{attractor } p K \subseteq V$
using $\text{attractor-in-}\{V[|K]\}$ unfolding $K$-def $U$-def by blast
ultimately show $?$thesis
unfolding $W1$-def winning-region-def using $\{V' \subseteq V\}$ by blast
qed

have $G'$-no-deadends: $\forall v. v \in V_G \Rightarrow \neg G'$-deadend $v$ proof –
fix $v$ assume $v \in V_G$

hence $\ast$: $v \in U \Rightarrow \text{attractor } p K$ using $\{V_G = V'\}$ $V'$-def by blast
moreover hence $\exists w \in U. v \rightarrow w$
using removing-winning-region-induces-no-deadends[of $v$ $p \ast\ast$] no-deadends $U$-equiv $U$-def by blast
moreover have $\exists w. [v \in VV p \ast\ast; v \rightarrow w] \Rightarrow w \in U$
using $\ast$ $U$-equiv winning-region-extends-$VVp$ by blast
ultimately have $\exists w \in V'. v \rightarrow w$
using $U$-equiv winning-region-extends-$VVp$ removing-attractor-induces-no-deadends $V'$-def by blast
thus $\neg G'$-deadend $v$ using $(v \in V_G) (V' \subseteq V)$ by simp
qed

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

obtain $\sigma W1$ where $\sigma W1$:
strategy $p \ast\ast \sigma W1$ $\forall v. v \in W1 \Rightarrow \text{winning-strategy } p \ast\ast \sigma W1 v$

unfolding $W1$-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: $\exists p. v \in G'$.winning-region $p$ proof –
have card $(\omega_G', V_G) < \text{card } (\omega ' V)$ proof –
{ assume min-prio $\in \omega_G'. V_G$
then obtain $v$ where $v: v \in V_G, \omega_G' v = \text{min-prio}$ by blast
hence $v \in \omega ' \{ \text{min-prio} \}$ using $\omega_G' = \omega$ by simp
hence False using $V'$-def $K$-def attractor-set-base $\langle V_G' = V'\rangle v(1)$
by (metis DiffD1 DiffD2 IntI contr-subsetD)
}

hence min-prio $\notin \omega_G', V_G'$ by blast
moreover have min-prio $\in \omega ' V$
unfolding min-prio-def using priorities-finite Min-in assms(1) by blast
moreover have $\omega_G', V_G' \subseteq \omega ' V$ unfolding $G'$-def by simp
ultimately show $?$thesis by (metis priorities-finite psubsetI psubset-card mono)
qed

thus $?$thesis using $\exists [v \in V_G \circ G'$-no-deadends $G'$.ParityGame-axioms by blast
qed

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

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

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

\[\text{moreover have } G',\text{maximal-path } P\]
using \langle \text{IsSet } P \subseteq V' \rangle \text{ subgame-maximal-path } \langle V' \subseteq V \rangle \text{ by simp}
moreover have \text{G'.path-conforms-with-strategy } p^{**} P \sigma \text{ proof -}
    have \text{G'.path-conforms-with-strategy } p^{**} P \sigma'
        \text{ using subgame-path-conforms-with-strategy } \langle V' \subseteq V \rangle \text{ IsSet } P \subseteq V'
        \text{ by simp}
moreover have \bigwedge v. v \in \text{IsSet } P \implies \sigma' v = \sigma v
    using \text{G'.path-conforms-with-strategy-irrelevant-updates by blast}
ultimately show ?thesis
    using \text{G'.path-conforms-with-strategy-irrelevant-updates by blast}

ultimately have \text{G'.winning-path } p^{**} P
    using \text{sigma(2) G'.winning-strategy-def G'.valid-maximal-conforming-path-0 P-0 P-not-null by blast}
moreover have \text{G'.VV } p^{***} \subseteq \text{VV } p^{***} \text{ using subgame-VV-subset G'-def by blast}
ultimately show False
    using \text{G'.winning-path-supergame[of p**] } \omega_{G'} = \omega
    \text{ by winning-path } p^{**} P \text{ ParityGame-axioms }
ultimately have \text{G'.winning-region } p = \text{VV G'}
    using recursion unfolding \text{G'.valid-maximal-conforming-path-0 P-0 P-not-null by blast}
thus False using \text{v } \in \text{V G'} \text{ using U-def V'-def } \text{V G'} = V' \text{ v } \in \text{V G'} \text{ by blast}

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

We compose a winning strategy for player \text{p} on \text{V} - \text{W1} out of three pieces.

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

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

Next, on \text{G'} we follow the winning strategy whose existence we proved earlier.

\text{have } \text{G'.winning-region } p = \text{V G'} \text{ using recursion unfolding G'.winning-region-def by blast}
then obtain \sigma 2
    where \sigma 2: \forall v. v \in \text{V G'} \implies \text{G'.strategy } p \sigma 2
        \forall v. v \in \text{V G'} \implies \text{G'.winning-strategy } p \sigma 2 v
        \text{using G'.merge-winning-strategies by blast}

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

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

Compose the three pieces.

\text{define } \sigma where \sigma = \text{override-on } (\text{override-on suc } \sigma 2 \text{ V'}) \sigma 1 \text{ (attractor } p \text{ K = K)}
\text{have attractor } p \text{ K } \cap \text{W1} = \{ \} \text{ proof (rule contr)}

62
assume attractor p K \cap W1 \neq \{\}
then obtain v where v: v \in attractor p K v \in W1 by blast
hence v \in V using W1-def winning-region-def by blast
obtain P where \text{vmc2-path} G P v p \sigma 1 \sigma W1
using strategy-conforming-path-exists \sigma W1(1) \sigma 1(1) (v \in V) by blast
then interpret \text{vmc2-path} G P v p \sigma 1 \sigma W1.

have strategy-attracts-via p \sigma 1 v (attractor p K) K using v(1) \sigma 1(2) strategy-attracts-def by blast
hence \text{set} P \cap K \neq \{\} using strategy-attracts-viaE visits-visits by blast
hence \neg \text{set} P \subseteq W1 unfolding K-def U-def by blast
thus False unfolding W1-def using comp.path-s-stay-in-winning-region \sigma W1 v(2) by auto
qed

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

have \sigma-1: \{w, v \in \text{attractor p K} K \implies \sigma v = \sigma 1 v\} unfolding \sigma-def by simp
have \sigma-2: \{w, v \in V' \implies \sigma v = \sigma 2 v\} unfolding \sigma-def V'-def by auto
have \sigma-K: \{w, v \in K \cap W1 \implies \sigma v = \text{suc} c v\} proof

fix v assume v: v \in K \cup W1
hence v \notin V' unfolding V'-def U-def using attractor-set-base by auto
with v show \sigma v = \text{suc} c v unfolding \sigma-def U-def using attractor p K \cap W1 = \{\} by (melis (mono-tags, lifting) Diff-iff Int1 UnE override-on-def override-on-emptyset)
qed

Show that \text{suc} c succeeds in avoiding entering W1.

\{ fix v assume v: v \in VV p
hence \neg \text{deadend} v using no-deadends 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 no-deadends \neg\text{deadend} w 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 \forall w. v \rightarrow w \implies \text{winning-strategy p**} \sigma W1 w using \sigma W1(2) by blast
    hence \text{winning-strategy p**} \sigma W1 v
      using strategy-extends-backwards-VVpath \sigma W1(1) (v \in VV p) by simp
      hence v \in W1 unfolding W1-def winning-region-def using \sigma W1(1) \neg\text{deadend} v by blast
     thus False using v \notin W1 by blast
   qed
   qed
   hence v \rightarrow \text{suc} c v v \in W1 \lor \text{suc} c v \notin W1 unfolding \text{suc-def}
     using some1-ex[\lambda w. v \rightarrow w \land (v \in W1 \lor w \notin W1)] by blast+
\} note suc-works = this

have strategy p \sigma
proof
  fix v assume v: v \in VV p \neg \text{deadend} v
  hence v \in \text{attractor p K} K \implies v \rightarrow \sigma v using \sigma-\sigma 1 \sigma 1(1) v unfolding strategy-def by auto
  moreover have v \in V' \implies v \rightarrow \sigma v proof-
    assume v \in V'
    moreover have v \in V G' using \forall v \in V' \langle V G' = V' \rangle by blast
moreover have \(v \in G' \land V \land p\) using \(G' \land V \land p = V' \land V \land p\) \((v \in V')\) \((v \in V \land p)\) by blast
moreover have \(\neg \text{Digraph.deadend } G' v\) using \(G'\text{-no-deadends} \ (v \in V \ G)\) by blast
ultimately have \(v \rightarrow G', \sigma 2 v\) using \(\sigma 2(1)\) \(G'\text{-strategy-def} [\text{of } p \sigma 2]\) by blast
with \((v \in V')\) show \(v \rightarrow \sigma v\) using \(\langle E_{G'} \subseteq E \rangle \) \(\sigma -\sigma 2\) by (metis subsetCE)

qed

moreover have \(v \in K \cup W I \implies v \rightarrow \sigma v\) using \(\text{suc-c-works}(1)\) \(v \sigma -K\) by auto
moreover have \(v \in V\) using \((v \in V \land p)\) by blast
ultimately show \(v \rightarrow \sigma v\) using \(\text{V-decomp}\) by blast

qed

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

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

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

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

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

ultimately show \(\text{thesis}\) by \(\text{simp}\)

qed

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

have \(\forall v \in V - W I . \text{winning-strategy } p v \sigma\) proof \((\text{intro ball1 winning-strategy2})\)

fix \(v P\) assume \(P: v \in V - W I \ \text{vmc-path } G P v p\) \(p\)

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

have \(\text{let } P \subseteq V - W I\)

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

fix \(v)\) assume \(v \in V - W I \ \neg \text{deadend } v v \in V \land p\)

show \(\sigma v \in V - W I \cup \{\}\) proof \((\text{rule ccontr})\)

assumee \(\text{thesis}\)

hence \(\sigma v \in W I\)

using \(\text{strategy } p \sigma\) \((\text{deadend } v) \ (v \in V \land p)\)

unfolding \text{strategy-def} by blast

hence \(v \notin K\) using \(\text{suc-c-works}(2)\) \((\text{OF } v \in V \land p)\) \((v \in V - W I)\) \(\sigma -K\) by auto

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

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

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

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

\text{attractor-set-base}

by blast

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

qed

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

assumee \(v \in V'\)

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

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

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

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

show \(v \in G' \land V \land p\) using \((v \in V \land p)\) \((G' \land V \land p = V' \land V \land p) \ (v \in V')\) by simp

qed

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

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

qed

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

next

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

show w ∈ V − W1 ∪ {}

proof (rule contr)

assume ¬thesis

hence w ∈ W1 using (v → w) by blast

let σ = σW1(v := w)

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

moreover have ¬(∃σ. strategy p∗∗ σ ∧ winning-strategy p∗∗ σ v)
using (v ∈ V − W1) unfolding W1-def winning-region-def by blast

ultimately have winning-strategy p∗∗ σ w using winning-strategy-updates of p∗∗ σ W1 w w σW1(1) (v → w)

unfolding winning-region-def by blast

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

ultimately have winning-strategy p∗∗ σ using (v ∈ VV p∗∗) (v → w)
by auto

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

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

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

qed

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

This concludes the proof of lset P ⊆ V − W1.

hence lset P ⊆ attractor p k ∪ V′ using V-decomp by blast

have ¬finite P

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

Every σ-conforming path starting in V − W1 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 ′ bad P ′ p σ2 proof

show ¬null P ′ unfolding P ′-def

using ¬finite P ′ lfinite 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: hence have thesis \( P' \subseteq V \setminus \{V' \subseteq V \} \) 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 thesis unfolding path-conforms-with-strategy-irrelevant-updates \( \{P \subseteq V \setminus \{V' \subseteq V \} \} \) G'-def by blast qed

qed simp have G'.winning-strategy p σ 2 (lhd P') unfolding \( \{P \text{ not-null } σ 2(2) \} \) \( \{V \} = V \) \( \{V \} \) \lset 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** \subseteq VV p** unfolding G'-def using subgame-VV by simp ultimately have winning-path p P' using G'.winning-path-supergame[of p P' G] (\( ω G' = ω \) ParityGame-axioms by blast thus thesis unfolding P'-def using infinite-small-length[of P P' G] (\( ω \) ParityGame-axioms by blast by blast

assumption asm: \( \not∃n. \text{ lset (ldropn n P) } \subseteq V' \)

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

have min-prio ∈ path-inf-priorities P unfolding path-inf-priorities-def proof (intro CollectI all1)

fix n obtain k1 where k1: ldropn n P \( k1 \) \( \notin V' \) using asm by (metis lset-lnth subsetI)
define k2 where k2 = k1 + n interpret vmc-path G ldropn k2 P P \( k2 \) p σ unfolding vmc-path-ldropn infinite-small-length (~{Lfinite P}) by blast have P \( k2 \notin V' \) unfolding k2-def using k1 lset-lnth infinite-small-length[of P (~{Lfinite P})] by simp hence P \( k2 \in \text{attractor p K using (~{Lfinite P}) \{set P \subseteq V \setminus W1\} \} \) by (metis DiffI U-def \( V' \)-def lset-lnth-member-inf)

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

using σ-attracts strategy-attractsE unfolding G'.visits-via-def by blast define k4 where k4 = k3 + k2 hence P \( k4 \in K \)

using k3 lset-lnth infinite-small-length[of P (~{Lfinite P})] by simp moreover have k4 ≥ n unfolding k4-def k2-def using le-add2 le-bounds by blast moreover have ldropn n P \( k4 \) \( = n = P \) \( (k4 - n) + n \)

using lset-lnth infinite-small-length (~{Lfinite P}) by blast ultimately have ldropn n P \( k4 \) \( = n \in K \) by simp

66
hence \( \text{lset} \ (\text{ldropn} \ n \ P) \cap K \neq \{\} \)
using \( \langle \sim \text{finite} \ P, \text{finite-ldropn in-lset-conv-th}[\text{of ldropn} \ n \ P \ \& \ k4 \ = \ n] \) by blast
thus \( \text{min-prio} \in \text{lset} \ (\text{ldropn} \ n \ (\text{inmap} \ \omega \ P)) \) unfolding \( K \)-def by auto
qed
thus \?thesis unfolding winning-path-def
using \( \text{path-inf-priorities-at-least-min-prio}[\text{OF} \ P\text{-valid}, \text{folded min-prio-def}] \)
\( \langle \text{winning-priority} \ \text{p} \ \text{min-prio} \sim \text{finite} \ P \rangle \) by blast
qed
qed
hence \( \forall v \in V, \exists p, \sigma. \text{strategy} \ \sigma \ \sigma \land \text{winning-strategy} \ \sigma \ \sigma \ v \)
unfolding \( W1\)-def winning-region-def using \( \text{strategy} \ \sigma \ \rangle \) by blast
hence \( \exists p, \sigma. \text{strategy} \ \sigma \ \sigma \land \text{winning-strategy} \ \sigma \ \sigma \ v \) using \( \langle v \in V \rangle \) by blast
thus \?thesis unfolding winning-region-def using \( \langle v \in V \rangle \) by blast
qed

12.2 Positional Determinacy without Deadends

theorem positional-strategy-exists-without-deadends:
assumes \( v \in V \land v \in V \implies \sim \text{deadend} \ v \)
shows \( \exists p, v \in \text{winning-region} \ p \)
proof -
{ fix \( p, v0 \in \text{winning-region} \ p \)
define \( A \) where \( A = \text{attractor} \ \tau0 \ \text{p} \ (\text{deadends} \ p** \ p) \)
assume \( v0\in\text{attractor} \ \tau0 \ \text{p} \ \text{deadends} \ p** \ p \)
then obtain \( \sigma \) where \( \sigma: \text{strategy} \ \sigma \ \sigma \ \text{strategy-attracts} \ \sigma \ \text{strategy} \ A \ (\text{deadends} \ p** \ p) \)
using \( \text{attractor-has-strategy}[\text{attractor} \ \tau0 \ \text{p} \ \text{deadends} \ p** \ p] \ A\)-def \( \text{deadends-in-V} \) by blast
have \( A \subseteq V \) using \( A\)-def using \( \text{attractor-in-V} \ \text{deadends-in-V} \) by blast
hence \( A \ = \ \text{deadends} \ p** \subseteq V \) by auto
have \( \text{winning-strategy} \ \sigma \ \tau0 \) proof (unfold \( \text{winning-strategy-def}, \ \text{intro} \ \text{allI} \ \text{implI} \)
fix \( \text{p} \) assume \( \text{vmc-path} \ G \ \text{p} \ \text{v0} \ \text{p} \ \sigma \)
then interpret \( \text{vmc-path} \ G \ \text{p} \ \text{v0} \ \text{p} \ \sigma \).
show \( \text{winning-path} \ \text{p} \ \text{P} \)
using \( \text{visits-deadend}[\text{of} \ \text{p**}] \ \sigma(2) \ \text{strategy-attracts-lset} \ \text{v0-in-attractor} \)
unfolding \( A\)-def by simp
qed
hence \( \exists p, \sigma. \text{strategy} \ \sigma \ \sigma \land \text{winning-strategy} \ \sigma \ \sigma \ v \) using \( \sigma \) by blast
}

12.3 Positional Determinacy with Deadends

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

theorem positional-strategy-exists:
assumes \( v0 \in V \)
shows \( \exists p, v0 \in \text{winning-region} \ p \)
proof -
{ fix \( p, v0 \in \text{winning-region} \ p \)
define \( A \) where \( A = \text{attractor} \ \tau0 \ \text{p} \ (\text{deadends} \ p** \ p) \)
assume \( v0\in\text{attractor} \ \tau0 \ \text{p} \ \text{deadends} \ p** \ p \)
then obtain \( \sigma \) where \( \sigma: \text{strategy} \ \sigma \ \sigma \ \text{strategy-attracts} \ \sigma \ \text{strategy} \ A \ (\text{deadends} \ p** \ p) \)
using \( \text{attractor-has-strategy}[\text{attractor} \ \tau0 \ \text{p} \ \text{deadends} \ p** \ p] \ A\)-def \( \text{deadends-in-V} \) by blast
have \( A \subseteq V \) using \( A\)-def using \( \text{attractor-in-V} \ \text{deadends-in-V} \) by blast
hence \( A \ = \ \text{deadends} \ p** \subseteq V \) by auto
have \( \text{winning-strategy} \ \sigma \ \tau0 \) proof (unfold \( \text{winning-strategy-def}, \ \text{intro} \ \text{allI} \ \text{implI} \)
fix \( \text{p} \) assume \( \text{vmc-path} \ G \ \text{p} \ \text{v0} \ \text{p} \ \sigma \)
then interpret \( \text{vmc-path} \ G \ \text{p} \ \text{v0} \ \text{p} \ \sigma \).
show \( \text{winning-path} \ \text{p} \ \text{P} \)
using \( \text{visits-deadend}[\text{of} \ \text{p**}] \ \sigma(2) \ \text{strategy-attracts-lset} \ \text{v0-in-attractor} \)
unfolding \( A\)-def by simp
qed
hence \( \exists p, \sigma. \text{strategy} \ \sigma \ \sigma \land \text{winning-strategy} \ \sigma \ \sigma \ v \) using \( \sigma \) by blast
}

note lemma-path-to-deadend = this
define A where A p = attractor p (deadends p **) for p

Remove the attractor sets of the sets of deadends.

define V' where V' = V - A Even - A Odd

hence V' \subseteq V by blast

show \reflectbox{thesis proof (cases)}

assume v0 \in V'

define G' where G' = subgame V'

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

have V' G' = V' unfolding G'-def using (V' \subseteq V) by simp

hence v0 \in V' G' using (v0 \in V') by simp

moreover have V'-no-deadends: \forall v. v \in V' G' \implies \neg G'.deadend v proof -

fix v assume v \in V' G'

moreover have V' = V - A Even - A Odd using V'-def by simp

ultimately show \neg G'.deadend v

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

qed

ultimately obtain p \sigma where \sigma: G'.strategy p \sigma G'.winning-strategy p \sigma v0

using G'.positional-strategy-exists-without-deadends

unfolding G'.winning-region-def by blast

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

fix v assume v \in V'

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

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

qed

obtain \sigma-attr

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

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

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

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

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

have strategy p \sigma'-proof -

have \sigma' = override-on \sigma-attr \sigma (UNIV - A Even - A Odd)

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

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

using valid-strategy-supergame \sigma-attr(1) \sigma(1) V'-no-deadends \langle V' G' = V' \rangle

unfolding G'-def by blast

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

qed

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

fix P assume vmc-path G P v0 p \sigma'

then interpret vmc-path G P v0 p \sigma' .

interpret vmc-path-no-deadend G P v0 p \sigma'

using V'-no-deadends' (v0 \in V') by unfold-locales

assume contra: \neg winning-path p P

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

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

next
fix $v w$ assume $v \in V' \neg-deadend v v \in VV p** v \rightarrow w$

have $w \notin A p**$ proof
assume $w \in A p**$
hence $v \in A p**$ unfolding $A$-def
using ($v \in VV p**$ ($v \rightarrow w$)) attractor-set-$VV p$ 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 $p$ ($override-on \sigma' \sigma$-attr ($A p - deadends p**$)) ($A p$) ($deadends p**$)
using strategy-attracts-irrelevant-override[OF $\sigma$-attr($2$) $\sigma$-attr($1$) ($strategy p \sigma'$)]
by blast
moreover have override-on $\sigma' \sigma$-attr ($A p - deadends p**$) = $\sigma'$
by (rule ext, unfold $\sigma'$-def, cases $p$) (simp-all add; override-on-def)
ultimately have strategy-attracts $p$ $\sigma'$ ($A p$) ($deadends p**$) by simp
hence lset $P \cap deadends p** \neq \{\}$
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 ($v0 \in V'$))

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

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

thus $\exists \theta \text{thesis unfolding winning-region-def}$

using $\text{lemma-path-to-deadend } (v_0 \in V)$ by blast

qed

qed

12.4 The Main Theorem: Positional Determinacy

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

theorem $\text{partition-into-winning-regions}$:

shows $V = \text{winning-region EVEN} \cup \text{winning-region ODD}$

and $\text{winning-region EVEN} \cap \text{winning-region ODD} = \emptyset$

proof

show $V \subseteq \text{winning-region EVEN} \cup \text{winning-region ODD}$

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

next

show $\text{winning-region EVEN} \cup \text{winning-region ODD} \subseteq V$

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

next

show $\text{winning-region EVEN} \cap \text{winning-region ODD} = \emptyset$

using $\text{winning-strategy-only-for-one-player[of EVEN]}$

unfolding $\text{winning-region-def}$ by auto

qed

end — context ParityGame

end

13 Defining the Attractor with inductive_set

theory $\text{AttractorInductive}$

imports

$\text{Main}$

$\text{Attractor}$

begin

custom ParityGame begin

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

13.1 $\text{attractor-inductive}$

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

inductive-set $\text{attractor-inductive} :: Player \Rightarrow 'a set \Rightarrow 'a set$

for $p :: \text{Player}$ and $W :: 'a set$ where
We show that the inductive definition and the definition via least fixed point are the same.

**Lemma** \( \text{attractor-inductive} = \text{attractor} \):

**Proof**

1. **Base** \([\text{intro}]\): \( v \in W \implies v \in \text{attractor-inductive} p W \)
2. **Case** \( v \in V V p; \exists w \cdot v \rightarrow w \land w \in \text{attractor-inductive} p W \)
   \( \implies v \in \text{attractor-inductive} p W \)
3. **Case** \( v \in V V p^{**}; \neg \text{deadend} v; \forall w \cdot v \rightarrow w \implies w \in \text{attractor-inductive} p W \)
   \( \implies v \in \text{attractor-inductive} p W \)

By (*proof*)
thus False using * by blast
qed
qed
}
ultimately show \(v \in \text{attractor-inductive } p\) \(W\) by (meson UnE)
qed
thus \(P \ (W \cup S \cup \text{directly-attracted } p \ S)\) using P-def by simp
qed (simp add: P-def Sup-least)
thus \(\text{thesis}\) using P-def by simp
qed
qed
end
end

14 Compatibility with the Graph Theory Package

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

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

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

14.1 To Graph Theory

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

definition to-pre-digraph :: \(\langle a, b \rangle\) Graph-scheme \(\Rightarrow \langle a, a \times a \rangle\) pre-digraph
where to-pre-digraph \(G \equiv \langle\rangle\)
  pre-digraph.verts = Graph.verts \(G\),
  pre-digraph.arcs = Graph.arcs \(G\),
  tail = fst,
  head = snd
\)

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

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

14.2 From Graph Theory

We can also convert in the other direction.

definition from-pre-digraph :: ('a, 'b) pre-digraph ⇒ 'a Gmph
where from-pre-digraph G ≡ ()
Gmph.verts = pre-digraph.verts G,
Gmph.arcs = arcs-ends G
()

context nomulti-digraph begin
interpretation is-Digraph: Digraph from-pre-digraph G proof –
{
  fix v w assume (v, w) ∈ E from-pre-digraph G
  then obtain e where e: e ∈ pre-digraph.arcs G tail G e = v head G e = w
    unfolding from-pre-digraph-def by auto
  hence (v, w) ∈ V from-pre-digraph G × V from-pre-digraph G
    unfolding from-pre-digraph-def by auto
}
thus Digraph (from-pre-digraph G) by (simp add: Digraph.intro subrelI)
qed
end

14.3 Isomorphisms

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

lemma graph-conversion-bij: G = from-pre-digraph (to-pre-digraph G)
unfolding to-pre-digraph-def from-pre-digraph-def arcs-ends-def arc-to-ends-def by auto

lemma (in nomulti-digraph) graph-conversion-bij2: digraph-iso G (to-pre-digraph (from-pre-digraph G))
proof –
define iso
  where iso = ()
  iso-verts = id :: 'a ⇒ 'a,
  iso-arcs = arc-to-ends G,
  iso-head = snd,
  iso-tail = fst
()
have inj-on (iso-verts iso) (pre-digraph.verts G) unfolding iso-def by auto
moreover have inj-on (iso-arcs iso) (pre-digraph.arcs G)
  unfolding iso-def arc-to-ends-def by (simp add: arc-to-ends-def inj-on1 no-multi-arcs)
moreover have $\forall a \in \text{pre-digraph.ares } G$.
iso-verts iso (tail $G a$) = iso-tail iso (iso-ares iso $a$)
\wedge iso-verts iso (head $G a$) = iso-head iso (iso-ares 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


