

# Galois Energy Games

Caroline Lemke

May 23, 2025

## Abstract

We provide a generic decision procedure for energy games with energy-bounded attacker and reachability objective, moving beyond vector-valued energies and vector-addition updates. All we demand is that energies form well-founded bounded join-semilattices, and that energy updates have upward-closed domains and can be undone through Galois-connected functions.

Offering a simple framework to construct decidable energy games we introduce the class of Galois energy games. We establish decidability of the (un)known initial credit problem for Galois energy games assuming energy-positional determinacy. For this we show correctness and termination of a simple algorithm relying on an inductive characterization of winning budgets and properties of Galois connections. Further, we prove that energy games over vectors of (extended) naturals with vector-addition and min-updates form a subclass of Galois energy games and are thus decidable.

## Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Energy Games</b>	<b>5</b>
2.1	Energy-positional Strategies . . . . .	11
2.2	Non-positional Strategies . . . . .	16
2.3	Attacker Winning Budgets . . . . .	19
<b>3</b>	<b>Galois Energy Games</b>	<b>64</b>
3.1	Properties of Galois connections . . . . .	65
3.2	Properties of the Partial Order . . . . .	69
3.3	Winning Budgets Revisited . . . . .	74
<b>4</b>	<b>Decidability of Galois Energy Games</b>	<b>88</b>
4.1	Minimal Attacker Winning Budgets as Pareto Fronts . . . . .	88
4.2	Proof of Decidability . . . . .	93
4.3	Applying Kleene’s Fixed Point Theorem . . . . .	171
<b>5</b>	<b>Vectors of (extended) Naturals as Energies</b>	<b>174</b>
<b>6</b>	<b>Bisping’s Updates</b>	<b>191</b>
6.1	Bisping’s Updates . . . . .	191
6.2	Bisping’s Inversion . . . . .	195
6.3	Relating Updates and “Inverse” Updates . . . . .	203

<b>7</b>	<b>Galois Energy Games over Naturals</b>	<b>221</b>
<b>8</b>	<b>References</b>	<b>225</b>
<b>A</b>	<b>Appendix</b>	<b>226</b>
A.1	List Lemmas . . . . .	226

# 1 Introduction

Building on Benjamin Bisping’s research[1], we study (multi-weighted) energy games with reachability winning conditions. These are zero-sum two-player games with perfect information played on directed graphs labelled by (multi-weighted) energy functions. Bisping [1] introduces a class of energy games, called *declining energy games* and provides an algorithm to compute minimal attacker winning budgets (i.e. Pareto fronts). He claims decidability of this class of energy games if the set of positions is finite. We substantiate this claim by providing a formal proof using a simplified and generalised version of that algorithm [5].

We abstract the necessary properties used in the proof and introduce a new class of energy games: Galois energy games. In such games updates can be undone through Galois connections, yielding a weakened form of inversion sufficient for an algorithm similar to standard shortest path algorithms. We establish decidability of the unknown and known initial credit problem for Galois energy games over well-founded bounded join-semilattices with a finite set of positions.

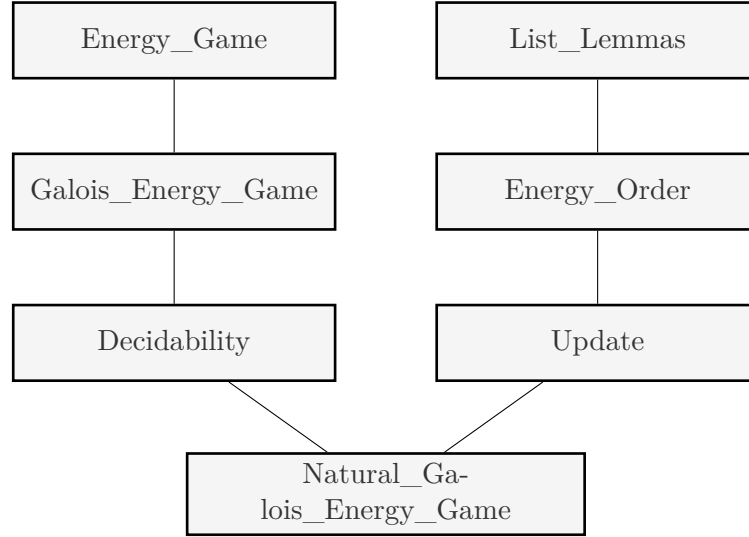
Galois energy games can be instantiated to common energy games, declining energy games [1], multi-weighted reachability games [2] and coverability on vector addition systems with states [4]. By confirming a subclass relationship (via sublocales) we conclude decidability of Galois energy games over vectors of (extended) naturals with the component-wise order. Finally, we show this in the case of vector-addition and min-updates only, subsuming the case of Bisping’s declining energy games.

For a broader perspective on the formalised results, including motivation, a high-level proof outline, complexity considerations, and connections to related work, we refer to the preprint [6].

## Theory Structure

We now give an overview of all our theories. In summary, we first formalise energy games with reachability winning conditions (in `Energy_Game.thy`), then formalise Galois energy games (in `Galois_Energy_Game.thy`) and prove decidability (in `Decidability.thy`). Finally, we formalise a superclass of Bisping’s declining energy games (in `Natural_Galois_Energy_Game.thy`) and conclude decidability.

The file structure is given by the following excerpt of the session graph, where the theories above are imported by the ones below.



Energy games are formalised as two-player zero-sum games with perfect information and reachability winning conditions played on labeled directed graphs in `Energy_Game.thy`. In particular, strategies and an inductive characterisation of winning budgets is discussed. (This corresponds to section 2.1 and 2.2 in the preprint [6].)

Galois energy games over well-founded bounded join-semilattices are formalized in `Galois_Energy_Game.thy`. (This corresponds to section 2.3 in the preprint [6].)

In `Decidability.thy` we formalise one iteration of a simplified and generalised version of Bisping’s algorithm. Using an order on possible Pareto fronts we are able to apply Kleene’s fixed point theorem. Assuming the game graph to be finite we then prove correctness of the algorithm. Further, we provide the key argument for termination, thus proving decidability of Galois energy games. (This corresponds to section 3.2 in the preprint [6].)

The file `List_Lemmas.thy` contains a few simple observations about lists, specifically when using `those`. This file’s contents can be found in the appendix.

In `Energy_Order.thy` we introduce the energies, i.e. vectors with entries in the extended natural numbers, and the component-wise order. There we establish that this order is a well-founded bounded join-semilattice.

In `Update.thy` we define a superset of Bisping’s updates. These are partial functions of energy vectors updating each component by subtracting or adding one, replacing it with the minimum of some components or not changing it. In particular, we observe that these functions are monotonic and have upward-closed domains. Further, we introduce a generalisation of Bisping’s inversion and relate it to the updates using Galois connections.

In `Natural_Galois_Energy_Game.thy` we formalise galois energy games over the previously defined with a fixed dimension. Afterwards, we formalise a subclass of such games where all edges of the game graph are labeled with a representation of the previously discussed updates (and thereby formalise Bisping’s declining energy games). Finally, we establish the subclass-relationships and thereby conclude decidability. (This corresponds to section 4.2 in the preprint [6].)

## 2 Energy Games

```
theory Energy_Game
  imports Coinductive.Coinductive_List Open_Induction.Restricted_Predicates
begin
```

Energy games are two-player zero-sum games with perfect information played on labeled directed graphs. The labels contain information on how each edge affects the current energy. We call the two players attacker and defender. In this theory we give fundamental definitions of plays, energy levels and (winning) attacker strategies. (This corresponds to section 2.1 and 2.2 in the preprint [6].)

```
locale energy_game =
  fixes attacker :: "'position set" and
    weight :: "'position  $\Rightarrow$  'position  $\Rightarrow$  'label option" and
    application :: "'label  $\Rightarrow$  'energy  $\Rightarrow$  'energy option"
begin

abbreviation "positions  $\equiv$  {g. g  $\in$  attacker  $\vee$  g  $\notin$  attacker}"
abbreviation "apply_w g g'  $\equiv$  application (the (weight g g'))"
```

### Plays

A play is a possibly infinite walk in the underlying directed graph.

```
coinductive valid_play :: "'position llist  $\Rightarrow$  bool" where
  "valid_play LNil" |
  "valid_play (LCons v LNil)" |
  "[weight v (lhd Ps)  $\neq$  None; valid_play Ps;  $\neg$ lnull Ps]
 $\implies$  valid_play (LCons v Ps)"
```

The following lemmas follow directly from the definition `valid_play`. In particular, a play is valid if and only if for each position there is an edge to its successor in the play. We show this using the coinductive definition by first establishing coinduction.

```
lemma valid_play_append:
  assumes "valid_play (LCons v Ps)" and "lfinite (LCons v Ps)" and
    "weight (llast (LCons v Ps)) v'  $\neq$  None" and "valid_play (LCons v' Ps')"
  shows "valid_play (lappend (LCons v Ps) (LCons v' Ps'))"
using assms proof(induction "list_of Ps" arbitrary: v Ps)
  case Nil
  then show ?case using valid_play.simps
  by (metis lappend_code(2) lappend_lnull1 lfinite_LCons lhd_LCons lhd_LCons_ltl
list.distinct(1) list_of_LCons llast_singleton llist.collapse(1) llist.disc(2))
next
  case (Cons a x)
  then show ?case using valid_play.simps
  by (smt (verit) lappend_code(2) lfinite_LCons lfinite_llist_of lhd_lappend list_of_llist_of
llast_LCons llist.discI(2) llist.distinct(1) llist_of.simps(2) llist_of_list_of
ltl_simps(2) valid_play.intros(3))
qed

lemma valid_play_coinduct:
  assumes "Q p" and
    " $\bigwedge v Ps. Q (LCons v Ps) \implies Ps \neq LNil \implies Q Ps \wedge \text{weight } v (lhd Ps) \neq \text{None}$ "
  shows "valid_play p"
using assms proof(coinduction arbitrary: p)
```

```

case valid_play
then show ?case
proof (cases "p = LNil")
  case True
  then show ?thesis by simp
next
case False
then show ?thesis
proof(cases "( $\exists v. p = LCons\ v\ LNil$ ")
  case True
  then show ?thesis by simp
next
case False
hence " $\exists v\ Ps. p = LCons\ v\ Ps \wedge \neg lnull\ Ps$ " using <math>\neg p = LNil>
  by (metis llist.collapse(1) not_lnull_conv)
from this obtain v Ps where "p = LCons v Ps  $\wedge \neg lnull Ps$ " by blast
hence " $Q\ Ps \wedge weight\ v\ (lhd\ Ps) \neq None$ " using valid_play
  using llist.disc(1) by blast
then show ?thesis using valid_play.simps valid_play
  using <math>p = LCons\ v\ Ps \wedge \neg lnull\ Ps> by blast
qed
qed
qed

lemma valid_play_nth_not_None:
  assumes "valid_play p" and "Suc i < llength p"
  shows "weight (lnth p i) (lnth p (Suc i))  $\neq None$ "
proof-
  have " $\exists prefix\ p'. p = lappend\ prefix\ p' \wedge llength\ prefix = Suc\ i \wedge weight\ (llast\ prefix)\ (lhd\ p') \neq None \wedge valid\_play\ p'$ "
  using assms proof(induct i)
  case 0
  hence " $\exists v\ Ps. p = LCons\ v\ Ps$ "
  by (metis llength_LNil neq_LNil_conv not_less_zero)
  from this obtain v Ps where "p = LCons v Ps" by auto
  hence "p = lappend (LCons v LNil) Ps"
  by (simp add: lappend_code(2))
  have "llength (LCons v LNil) = Suc 0" using one_eSuc one_enat_def by simp
  have "weight v (lhd Ps)  $\neq None$ " using 0 valid_play.simps <math>p = LCons\ v\ Ps>
  by (smt (verit) One_nat_def add.commute gen_llength_code(1) gen_llength_code(2)
less_numeral_extra(4) lhd_LCons llength_code llist.distinct(1) ltl_simps(2) one_enat_def
plus_1_eq_Suc)
  hence "p = lappend (LCons v LNil) Ps  $\wedge llength\ (LCons\ v\ LNil) = Suc\ 0 \wedge weight\ (llast\ (LCons\ v\ LNil))\ (lhd\ Ps) \neq None$ " using <math>p = LCons\ v\ Ps>
  using <math>p = lappend\ (LCons\ v\ LNil)\ Ps> <math>llength\ (LCons\ v\ LNil) = Suc\ 0>
  by simp
  hence "p = lappend (LCons v LNil) Ps  $\wedge llength\ (LCons\ v\ LNil) = Suc\ 0 \wedge weight\ (llast\ (LCons\ v\ LNil))\ (lhd\ Ps) \neq None \wedge valid\_play\ Ps$ " using valid_play.simps
  0
  by (metis (no_types, lifting) <math>p = LCons\ v\ Ps> llist.distinct(1) ltl_simps(2))

  then show ?case by blast
next
case (Suc 1)
hence " $\exists prefix\ p'. p = lappend\ prefix\ p' \wedge llength\ prefix = enat\ (Suc\ 1) \wedge weight\ (llast\ prefix)\ (lhd\ p') \neq None \wedge valid\_play\ p'$ "

```

```

    using Suc_ile_eq order_less_imp_le by blast
    from this obtain prefix p' where P: "p = lappend prefix p' ∧ llength prefix
= enat (Suc 1) ∧ weight (llast prefix) (lhd p') ≠ None ∧ valid_play p'" by auto
    have "p = lappend (lappend prefix (LCons (lhd p') LNil)) (ltl p') ∧ llength
(lappend prefix (LCons (lhd p') LNil)) = enat (Suc (Suc 1)) ∧ weight (llast (lappend
prefix (LCons (lhd p') LNil))) (lhd (ltl p')) ≠ None ∧ valid_play (ltl p')"
    proof
      show "p = lappend (lappend prefix (LCons (lhd p') LNil)) (ltl p')" using P
      by (metis Suc.prem2 enat_ord_simps2 lappend_LNil2 lappend_snocL1_conv_LCons2
lessI llist.exhaust_sel order.asym)
      show "llength (lappend prefix (LCons (lhd p') LNil)) = enat (Suc (Suc 1))"
    ∧
      weight (llast (lappend prefix (LCons (lhd p') LNil))) (lhd (ltl p')) ≠ None
    ∧ valid_play (ltl p')"
    proof
      have "llength (lappend prefix (LCons (lhd p') LNil)) = 1 + (llength prefix)"
      by (smt (verit, best) add.commute epred_1 epred_inject epred_llength llength_LNil
llength_eq_0 llength_lappend llist.disc2 ltl_simps2 zero_neq_one)
      thus "llength (lappend prefix (LCons (lhd p') LNil)) = enat (Suc (Suc 1))"
    using P
      by (simp add: one_enat_def)
      show "weight (llast (lappend prefix (LCons (lhd p') LNil))) (lhd (ltl p'))
≠ None ∧ valid_play (ltl p') "
    proof
      show "weight (llast (lappend prefix (LCons (lhd p') LNil))) (lhd (ltl
p')) ≠ None" using P valid_play.simps
      by (metis Suc.prem2 <llength (lappend prefix (LCons (lhd p') LNil))
= 1 + llength prefix> <llength (lappend prefix (LCons (lhd p') LNil)) = enat (Suc
(Suc 1))> <p = lappend (lappend prefix (LCons (lhd p') LNil)) (ltl p')> add.commute
enat_add_mono eq_LConsD lappend_LNil2 less_numeral_extra4 llast_lappend_LCons
llast_singleton llength_eq_enat_lfiniteD ltl_simps1)
      show "valid_play (ltl p')" using P valid_play.simps
      by (metis (full_types) energy_game.valid_play.intros1 ltl_simps1)
    ltl_simps2)
    qed
    qed
    qed
    then show ?case by blast
    qed
    thus ?thesis
    by (smt (z3) assms2 cancel_comm_monoid_add_class.diff_cancel eSuc_enat enat_ord_simps2)
lappend_eq_lappend_conv lappend_lnull2 lessI lhd_LCons_ltl linorder_neq_iff llast_conv_lnth
lnth_0 lnth_lappend the_enat.simps)
    qed

lemma valid_play_nth:
  assumes "∧i. enat (Suc i) < llength p
    → weight (lnth p i) (lnth p (Suc i)) ≠ None"
  shows "valid_play p"
  using assms proof (coinduction arbitrary: p rule: valid_play_coinduct)
  show "∧v Ps p.
    LCons v Ps = p ⇒
    ∀i. enat (Suc i) < llength p → weight (lnth p i) (lnth p (Suc i)) ≠ None
  ⇒
    Ps ≠ LNil ⇒
    (∃p. Ps = p ∧ (∀i. enat (Suc i) < llength p → weight (lnth p i) (lnth

```

```

p (Suc i))  $\neq$  None))  $\wedge$ 
  weight v (lhd Ps)  $\neq$  None"
proof-
  fix v Ps p
  show "LCons v Ps = p  $\implies$ 
     $\forall i. \text{enat } (\text{Suc } i) < \text{llength } p \longrightarrow \text{weight } (\text{lnth } p \ i) (\text{lnth } p \ (\text{Suc } i)) \neq \text{None}$ 
 $\implies$ 
    Ps  $\neq$  LNil  $\implies$ 
    ( $\exists p. \text{Ps} = p \wedge (\forall i. \text{enat } (\text{Suc } i) < \text{llength } p \longrightarrow \text{weight } (\text{lnth } p \ i) (\text{lnth } p \ (\text{Suc } i)) \neq \text{None})) \wedge$ 
    weight v (lhd Ps)  $\neq$  None"
  proof-
    assume "LCons v Ps = p"
    show " $\forall i. \text{enat } (\text{Suc } i) < \text{llength } p \longrightarrow \text{weight } (\text{lnth } p \ i) (\text{lnth } p \ (\text{Suc } i)) \neq \text{None} \implies$ 
      Ps  $\neq$  LNil  $\implies$ 
      ( $\exists p. \text{Ps} = p \wedge (\forall i. \text{enat } (\text{Suc } i) < \text{llength } p \longrightarrow \text{weight } (\text{lnth } p \ i) (\text{lnth } p \ (\text{Suc } i)) \neq \text{None})) \wedge$ 
      weight v (lhd Ps)  $\neq$  None"
    proof-
      assume A: " $\forall i. \text{enat } (\text{Suc } i) < \text{llength } p \longrightarrow \text{weight } (\text{lnth } p \ i) (\text{lnth } p \ (\text{Suc } i)) \neq \text{None}$ "
      show "Ps  $\neq$  LNil  $\implies$ 
        ( $\exists p. \text{Ps} = p \wedge (\forall i. \text{enat } (\text{Suc } i) < \text{llength } p \longrightarrow \text{weight } (\text{lnth } p \ i) (\text{lnth } p \ (\text{Suc } i)) \neq \text{None})) \wedge$ 
        weight v (lhd Ps)  $\neq$  None"
      proof-
        assume "Ps  $\neq$  LNil"
        show "( $\exists p. \text{Ps} = p \wedge (\forall i. \text{enat } (\text{Suc } i) < \text{llength } p \longrightarrow \text{weight } (\text{lnth } p \ i) (\text{lnth } p \ (\text{Suc } i)) \neq \text{None})) \wedge$ 
          weight v (lhd Ps)  $\neq$  None"
      proof
        show " $\exists p. \text{Ps} = p \wedge (\forall i. \text{enat } (\text{Suc } i) < \text{llength } p \longrightarrow \text{weight } (\text{lnth } p \ i) (\text{lnth } p \ (\text{Suc } i)) \neq \text{None})$ "
      proof
        have "( $\forall i. \text{enat } (\text{Suc } i) < \text{llength } \text{Ps} \longrightarrow \text{weight } (\text{lnth } \text{Ps } i) (\text{lnth } \text{Ps } (\text{Suc } i)) \neq \text{None})$ "
        proof
          fix i
          show " $\text{enat } (\text{Suc } i) < \text{llength } \text{Ps} \longrightarrow \text{weight } (\text{lnth } \text{Ps } i) (\text{lnth } \text{Ps } (\text{Suc } i)) \neq \text{None}$ "
          proof
            assume " $\text{enat } (\text{Suc } i) < \text{llength } \text{Ps}$ "
            hence " $\text{enat } (\text{Suc } (\text{Suc } i)) < \text{llength } (\text{LCons } v \ \text{Ps})$ "
            by (metis ldropsn_Suc_LCons ldropsn_eq_LNil linorder_not_le)
            have " $(\text{lnth } \text{Ps } i) = (\text{lnth } (\text{LCons } v \ \text{Ps}) (\text{Suc } i))$ " by simp
            have " $(\text{lnth } \text{Ps } (\text{Suc } i)) = (\text{lnth } (\text{LCons } v \ \text{Ps}) (\text{Suc } (\text{Suc } i)))$ " by
simp
            thus " $\text{weight } (\text{lnth } \text{Ps } i) (\text{lnth } \text{Ps } (\text{Suc } i)) \neq \text{None}$ "
            using A <( $\text{lnth } \text{Ps } i) = (\text{lnth } (\text{LCons } v \ \text{Ps}) (\text{Suc } i))$ >
            using <LCons v Ps = p> < $\text{enat } (\text{Suc } (\text{Suc } i)) < \text{llength } (\text{LCons } v \ \text{Ps})$ > by auto
          qed
        qed
      thus "Ps = Ps  $\wedge (\forall i. \text{enat } (\text{Suc } i) < \text{llength } \text{Ps} \longrightarrow \text{weight } (\text{lnth } \text{Ps } i) (\text{lnth } \text{Ps } (\text{Suc } i)) \neq \text{None})$ "

```



```

      by simp
    qed
  have "v = lnth (LCons v Ps) 0" by simp
  have "lhd Ps = lnth (LCons v Ps) (Suc 0)" using lnth_def <Ps ≠ LNil>
    by (metis llist.exhaust_sel lnth_0 lnth_Suc_LCons)
  thus "weight v (lhd Ps) ≠ None"
    using <v = lnth (LCons v Ps) 0> A
    by (metis <LCons v Ps = p> <Ps ≠ LNil> <∃p. Ps = p ∧ (∀i. enat
(Suc i) < llength p → weight (lnth p i) (lnth p (Suc i)) ≠ None)> gen_llength_code(1)
ldropn_0 ldropn_Suc_LCons ldropn_eq_LConsD llist.collapse(1) lnth_Suc_LCons not_lnull_conv)
  qed
  qed
  qed
  qed
  qed
  qed

```

## Energy Levels

The energy level of a play is calculated by repeatedly updating the current energy according to the edges in the play. The final energy level of a finite play is `energy_level e p (the_enat (llength p - 1))` where `e` is the initial energy.

```

fun energy_level:: "'energy ⇒ 'position llist ⇒ nat ⇒ 'energy option" where
  "energy_level e p 0 = (if p = LNil then None else Some e)" |
  "energy_level e p (Suc i) =
    (if (energy_level e p i) = None ∨ llength p ≤ (Suc i) then None
     else apply_w (lnth p i) (lnth p (Suc i)) (the (energy_level e p i)))"

```

We establish some (in)equalities to simplify later proofs.

```

lemma energy_level_cons:
  assumes "valid_play (LCons v Ps)" and "¬lnull Ps" and
    "apply_w v (lhd Ps) e ≠ None" and "enat i < (llength Ps)"
  shows "energy_level (the (apply_w v (lhd Ps) e)) Ps i
    = energy_level e (LCons v Ps) (Suc i)"
  using assms proof(induction i arbitrary: e Ps rule: energy_level.induct)
  case (1 e p)
  then show ?case using energy_level.simps
    by (smt (verit) ldropn_Suc_LCons ldropn_eq_LNil le_zero_eq lhd_conv_lnth llength_eq_0
llist.distinct(1) lnth_0 lnth_Suc_LCons lnull_def option.collapse option.discI option.sel
zero_enat_def)
  next
  case (2 e p n)
  hence "enat n < (llength Ps)"
    using Suc_ile_eq nless_le by blast
  hence IA: "energy_level (the (apply_w v (lhd Ps) e)) Ps n = energy_level e (LCons
v Ps) (Suc n)"
    using 2 by simp
  have "(llength Ps) > Suc n" using <enat (Suc n) < (llength Ps)>
    by simp
  hence "llength (LCons v Ps) > (Suc (Suc n))"
    by (metis ldropn_Suc_LCons ldropn_eq_LNil linorder_not_less)
  show "energy_level (the (apply_w v (lhd Ps) e)) Ps (Suc n) = energy_level e (LCons
v Ps) (Suc (Suc n))"
  proof(cases "energy_level e (LCons v Ps) (Suc (Suc n)) = None")
  case True

```

```

    hence "(energy_level e (LCons v Ps) (Suc n)) = None  $\vee$  llength (LCons v Ps)  $\leq$ 
(Suc (Suc n))  $\vee$  apply_w (lnth (LCons v Ps) (Suc n)) (lnth (LCons v Ps) (Suc (Suc
n))) (the (energy_level e (LCons v Ps) (Suc n))) = None "
    using energy_level.simps
    by metis
    hence none: "(energy_level e (LCons v Ps) (Suc n)) = None  $\vee$  apply_w (lnth (LCons
v Ps) (Suc n)) (lnth (LCons v Ps) (Suc (Suc n))) (the (energy_level e (LCons v Ps)
(Suc n))) = None "
    using <llength (LCons v Ps) > (Suc (Suc n))>
    by (meson linorder_not_less)
    show ?thesis
    proof(cases "(energy_level e (LCons v Ps) (Suc n)) = None")
    case True
    then show ?thesis using IA by simp
    next
    case False
    hence "apply_w (lnth (LCons v Ps) (Suc n)) (lnth (LCons v Ps) (Suc (Suc n)))
(the (energy_level e (LCons v Ps) (Suc n))) = None "
    using none by auto
    hence "apply_w (lnth (LCons v Ps) (Suc n)) (lnth (LCons v Ps) (Suc (Suc n)))
(the (energy_level (the (apply_w v (lhd Ps) e)) Ps n)) = None "
    using IA by auto
    then show ?thesis by (simp add: IA)
    qed
  next
  case False
  then show ?thesis using IA
  by (smt (verit) <enat (Suc n) < llength Ps> energy_level.simps(2) lnth_Suc_LCons
order.asym order_le_imp_less_or_eq)
  qed
qed

lemma energy_level_nth:
  assumes "energy_level e p m  $\neq$  None" and "Suc i  $\leq$  m"
  shows "apply_w (lnth p i) (lnth p (Suc i)) (the (energy_level e p i))  $\neq$  None
 $\wedge$  energy_level e p i  $\neq$  None"
using assms proof(induct "m - (Suc i)" arbitrary: i)
  case 0
  then show ?case using energy_level.simps
  by (metis diff_diff_cancel minus_nat.diff_0)
next
  case (Suc x)
  hence "x = m - Suc (Suc i)"
  by (metis add_Suc_shift diff_add_inverse2 diff_le_self le_add_diff_inverse)
  hence "apply_w (lnth p (Suc i)) (lnth p (Suc (Suc i))) (the (energy_level e p
(Suc i)))  $\neq$  None  $\wedge$  (energy_level e p (Suc i))  $\neq$  None" using Suc
  by (metis diff_is_0_eq nat.distinct(1) not_less_eq_eq)
  then show ?case using energy_level.simps by metis
qed

lemma energy_level_append:
  assumes "lfinite p" and "i < the_enat (llength p)" and
    "energy_level e p (the_enat (llength p) - 1)  $\neq$  None"
  shows "energy_level e p i = energy_level e (lappend p p') i"
proof-
  have A: " $\wedge$ i. i < the_enat (llength p)  $\implies$  energy_level e p i  $\neq$  None" using energy_level_nth

```

```

assms
  by (metis Nat.lessE diff_Suc_1 less_eq_Suc_le)
show ?thesis using assms A proof(induct i)
  case 0
  then show ?case using energy_level.simps
    by (metis LNil_eq_lappend_iff llength_lnull llist.disc(1) the_enat_0 verit_comp_simplify1)

next
  case (Suc i)
  hence "energy_level e p i = energy_level e (lappend p p') i"
    by simp
  have "Suc i < (llength p) ∧ energy_level e p i ≠ None" using Suc
    by (metis Suc_lessD enat_ord_simps(2) lfinite_conv_llength_enat the_enat.simps)
  hence "Suc i < (llength (lappend p p')) ∧ energy_level e (lappend p p') i ≠
None"
    using <energy_level e p i = energy_level e (lappend p p') i>
    by (metis dual_order.strict_trans1 enat_le_plus_same(1) llength_lappend)
  then show ?case unfolding energy_level.simps using <Suc i < (llength p) ∧ energy_level
e p i ≠ None> <energy_level e p i = energy_level e (lappend p p') i>
    by (smt (verit) Suc_ile_eq energy_level.elims le_zero_eq linorder_not_less lnth_lappend1
nle_le the_enat.simps zero_enat_def)
qed
qed

```

## Won Plays

All infinite plays are won by the defender. Further, the attacker is energy-bound and the defender wins if the energy level becomes None. Finite plays with an energy level that is not None are won by a player, if the other is stuck.

**abbreviation** "deadend g  $\equiv$  ( $\forall g'. \text{weight } g \ g' = \text{None}$ )"

**abbreviation** "attacker\_stuck p  $\equiv$  ( $\text{llast } p \in \text{attacker} \wedge \text{deadend } (\text{llast } p)$ )"

**definition** defender\_wins\_play:: "'energy  $\Rightarrow$  'position llist  $\Rightarrow$  bool" where  
 "defender\_wins\_play e p  $\equiv$  lfinite p  $\longrightarrow$   
 (energy\_level e p (the\_enat (llength p)-1) = None  $\vee$  attacker\_stuck p)"

## 2.1 Energy-positional Strategies

Energy-positional strategies map pairs of energies and positions to a next position. Further, we focus on attacker strategies, i.e. partial functions mapping attacker positions to successors.

**definition** attacker\_strategy:: "('energy  $\Rightarrow$  'position  $\Rightarrow$  'position option)  $\Rightarrow$  bool"  
 where  
 "attacker\_strategy s = ( $\forall g \ e. (g \in \text{attacker} \wedge \neg \text{deadend } g) \longrightarrow$   
 ( $s \ e \ g \neq \text{None} \wedge \text{weight } g \ (\text{the } (s \ e \ g)) \neq \text{None}$ ))"

We now define what it means for a play to be consistent with some strategy.

**coinductive** play\_consistent\_attacker:: "('energy  $\Rightarrow$  'position  $\Rightarrow$  'position option)  
 $\Rightarrow$  'position llist  $\Rightarrow$  'energy  $\Rightarrow$  bool" where  
 "play\_consistent\_attacker \_ LNil \_" |  
 "play\_consistent\_attacker \_ (LCons v LNil) \_" |  
 "[[play\_consistent\_attacker s Ps (the (apply\_w v (lhd Ps) e));  $\neg \text{lnull } Ps$ ;  
 v  $\in$  attacker  $\longrightarrow$  (s e v) = Some (lhd Ps)]]  
 $\implies$  play\_consistent\_attacker s (LCons v Ps) e"

The coinductive definition allows for coinduction.

```

lemma play_consistent_attacker_coinduct:
  assumes "Q s p e" and
    " $\bigwedge s v Ps e'. Q s (LCons v Ps) e' \wedge \neg \text{lnull } Ps \implies$ "
      Q s Ps (the (apply_w v (lhd Ps) e'))  $\wedge$ 
      (v  $\in$  attacker  $\longrightarrow$  s e' v = Some (lhd Ps))"
  shows "play_consistent_attacker s p e"
  using assms proof (coinduction arbitrary: s p e)
  case play_consistent_attacker
  then show ?case
  proof (cases "p = LNil")
  case True
  then show ?thesis by simp
  next
  case False
  hence " $\exists v Ps. p = LCons v Ps$ "
  by (meson llist.exhaust)
  from this obtain v Ps where "p = LCons v Ps" by auto
  then show ?thesis
  proof (cases "Ps = LNil")
  case True
  then show ?thesis using <p = LCons v Ps> by simp
  next
  case False
  hence "Q s Ps (the (apply_w v (lhd Ps) e))  $\wedge$  (v  $\in$  attacker  $\longrightarrow$  s e v = Some (lhd Ps))"
  using assms
  using <p = LCons v Ps> llist.collapse(1) play_consistent_attacker(1) by
blast
  then show ?thesis using play_consistent_attacker play_consistent_attacker.simps
  by (metis (no_types, lifting) <p = LCons v Ps> lnull_def)
  qed
  qed
  qed

```

Adding a position to the beginning of a consistent play is simple by definition. It is harder to see, when a position can be added to the end of a finite play. For this we introduce the following lemma.

```

lemma play_consistent_attacker_append_one:
  assumes "play_consistent_attacker s p e" and "lfinite p" and
    "energy_level e p (the_enat (llength p)-1)  $\neq$  None" and
    "valid_play (lappend p (LCons g LNil))" and "llast p  $\in$  attacker  $\longrightarrow$ "
      Some g = s (the (energy_level e p (the_enat (llength p)-1))) (llast p)"
  shows "play_consistent_attacker s (lappend p (LCons g LNil)) e"
  using assms proof (induct "the_enat (llength p)" arbitrary: p e)
  case 0
  then show ?case
  by (metis lappend_lnull1 length_list_of length_list_of_conv_the_enat llength_eq_0
play_consistent_attacker.simps zero_enat_def)
  next
  case (Suc x)
  hence " $\exists v Ps. p = LCons v Ps$ "
  by (metis Zero_not_Suc llength_LNil llist.exhaust the_enat_0)
  from this obtain v Ps where "p = LCons v Ps" by auto

```

```

have B: "play_consistent_attacker s (lappend Ps (LCons g LNil)) (the (apply_w
v (lhd (lappend Ps (LCons g LNil))))e))"
proof(cases "Ps=LNil")
  case True
  then show ?thesis
  by (simp add: play_consistent_attacker.intros(2))
next
  case False
  show ?thesis
  proof(rule Suc.hyps)
    show "valid_play (lappend Ps (LCons g LNil))"
    by (metis (no_types, lifting) LNil_eq_lappend_iff Suc.prem(4) <p = LCons
v Ps> lappend_code(2) llist.distinct(1) llist.inject valid_play.cases)
    show "x = the_enat (llength Ps)" using Suc <p = LCons v Ps>
    by (metis diff_add_inverse length_Cons length_list_of_conv_the_enat lfinite_ltl
list_of_LCons ltl_simps(2) plus_1_eq_Suc)
    show "play_consistent_attacker s Ps (the (apply_w v (lhd (lappend Ps (LCons
g LNil)))) e))"
    using False Suc.prem(1) <p = LCons v Ps> play_consistent_attacker.cases
by fastforce
    show "lfinite Ps" using Suc <p = LCons v Ps> by simp

    hence EL: "energy_level (the (apply_w v (lhd (lappend Ps (LCons g LNil))))
e)) Ps
(the_enat (llength Ps) - 1) = energy_level e (LCons v (lappend Ps (LCons g
LNil)))
(Suc (the_enat (llength Ps) - 1))"
    proof-
      have A: "valid_play (LCons v Ps) ∧ ¬ lnull Ps ∧ apply_w v (lhd Ps) e
≠ None ∧
enat (the_enat (llength Ps) - 1) < llength Ps"
      proof
        show "valid_play (LCons v Ps)" proof(rule valid_play_nth)
          fix i
          show "enat (Suc i) < llength (LCons v Ps) →
weight (lnth (LCons v Ps) i) (lnth (LCons v Ps) (Suc i)) ≠ None"
          proof
            assume "enat (Suc i) < llength (LCons v Ps)"
            hence "(lnth (LCons v Ps) i) = (lnth (lappend p (LCons g LNil)) i)"
using <p = LCons v Ps>
            by (metis Suc_ile_eq lnth_lappend1 order.strict_implies_order)
            have "(lnth (LCons v Ps) (Suc i)) = (lnth (lappend p (LCons g LNil))
(Suc i))" using <p = LCons v Ps> <enat (Suc i) < llength (LCons v Ps)>
            by (metis lnth_lappend1)

            from Suc have "valid_play (lappend p (LCons g LNil))" by simp
            hence "weight (lnth (lappend p (LCons g LNil)) i) (lnth (lappend p
(LCons g LNil)) (Suc i)) ≠ None"
            using <enat (Suc i) < llength (LCons v Ps)> valid_play_nth_not_None
            by (metis Suc.prem(2) <p = LCons v Ps> llist.disc(2) lstrict_prefix_lappend_c
lstrict_prefix_llength_less min.absorb4 min.strict_coboundedI1)

            thus "weight (lnth (LCons v Ps) i) (lnth (LCons v Ps) (Suc i)) ≠ None"
            using <(lnth (LCons v Ps) (Suc i)) = (lnth (lappend p (LCons g LNil))
(Suc i))> <(lnth (LCons v Ps) i) = (lnth (lappend p (LCons g LNil)) i)> by simp

```

```

      qed
    qed
    show "¬ lnull Ps ∧ apply_w v (lhd Ps) e ≠ None ∧ enat (the_enat (llength
Ps) - 1) < llength Ps"
    proof
      show "¬ lnull Ps" using False by auto
      show "apply_w v (lhd Ps) e ≠ None ∧ enat (the_enat (llength Ps) - 1)
< llength Ps"
      proof
        show "apply_w v (lhd Ps) e ≠ None" using Suc
          by (smc (verit, ccfv_threshold) One_nat_def <¬ lnull Ps> <lfinite
Ps> <p = LCons v Ps> <x = the_enat (llength Ps)> diff_add_inverse energy_level.simps(1)
energy_level_nth le_SucE le_add1 length_list_of length_list_of_conv_the_enat lhd_conv_lnth
llength_eq_0 llist.discI(2) lnth_0 lnth_ltl ltl_simps(2) option.sel plus_1_eq_Suc
zero_enat_def)
        show "enat (the_enat (llength Ps) - 1) < llength Ps" using False
          by (metis <¬ lnull Ps> <lfinite Ps> diff_Suc_1 enat_0_iff(2) enat_ord_simps(2)
gr0_conv_Suc lessI lfinite_llength_enat llength_eq_0 not_gr_zero the_enat.simps)
      qed
    qed
    qed

    have "energy_level (the (apply_w v (lhd (lappend Ps (LCons g LNil)))) e))
Ps
(the_enat (llength Ps) - 1) = energy_level (the (apply_w v (lhd Ps) e)) Ps
(the_enat (llength Ps) - 1)" using False
    by (simp add: lnull_def)
    also have "... = energy_level e (LCons v Ps) (Suc (the_enat (llength Ps)
- 1))"
    using energy_level_cons A by simp
    also have "... = energy_level e (LCons v (lappend Ps (LCons g LNil)))
(Suc (the_enat (llength Ps) - 1))" using energy_level_append
    by (metis False One_nat_def Suc.hyps(2) Suc.prems(2) Suc.prems(3) <lfinite
Ps> <p = LCons v Ps> <x = the_enat (llength Ps)> diff_Suc_less lappend_code(2)
length_list_of length_list_of_conv_the_enat less_SucE less_Suc_eq_0_disj llength_eq_0
llist.disc(1) llist.expand nat_add_left_cancel_less plus_1_eq_Suc zero_enat_def)

    finally show ?thesis .
  qed

  thus EL_notNone: "energy_level (the (apply_w v (lhd (lappend Ps (LCons g LNil))))
e)) Ps
(the_enat (llength Ps) - 1) ≠ None"
    using Suc
    by (metis False One_nat_def Suc_pred <p = LCons v Ps> <x = the_enat (llength
Ps)> diff_Suc_1' energy_level.simps(1) energy_level_append lappend_code(2) lessI
not_less_less_Suc_eq not_one_less_zero option.distinct(1) zero_less_Suc zero_less_diff)

  show "llast Ps ∈ attacker →
Some g = s (the (energy_level (the (apply_w v (lhd (lappend Ps (LCons g LNil))))
e)) Ps
(the_enat (llength Ps) - 1)) (llast Ps)"
  proof
    assume "llast Ps ∈ attacker"
    have "llast Ps = llast p" using False <p = LCons v Ps>

```

```

      by (simp add: llast_LCons lnull_def)
      hence "llast p ∈ attacker" using <llast Ps ∈ attacker> by simp
      hence "Some g = s (the (energy_level e p (the_enat (llength p) - 1))) (llast
p)" using Suc by simp
      hence "Some g = s (the (energy_level e (LCons v Ps) (the_enat (llength (LCons
v Ps)) - 1))) (llast Ps)" using <p = LCons v Ps> <llast Ps = llast p> by simp

      have "apply_w v (lhd Ps) e ≠ None" using Suc
      by (smt (verit, best) EL EL_notNone False One_nat_def energy_level.simps(1)
energy_level_nth le_add1 lhd_conv_lnth lhd_lappend llist.discI(2) llist.exhaust_sel
lnth_0 lnth_Suc_LCons lnull_lappend option.sel plus_1_eq_Suc)
      thus "Some g = s (the (energy_level (the (apply_w v (lhd (lappend Ps (LCons
g LNil)))) e)) Ps
(the_enat (llength Ps) - 1))) (llast Ps)" using EL
      by (metis (no_types, lifting) False Suc.hyps(2) Suc.prems(2) Suc.prems(3)
Suc_diff_Suc <Some g = s (the (energy_level e (LCons v Ps) (the_enat (llength (LCons
v Ps)) - 1))) (llast Ps)> <lfinite Ps> <p = LCons v Ps> <x = the_enat (llength
Ps)> cancel_comm_monoid_add_class.diff_cancel diff_Suc_1 energy_level_append lappend_code(2)
lessI lfinite.cases lfinite_conv_llength_enat linorder_neqE_nat llength_eq_0 llist.discI(2)
not_add_less1 plus_1_eq_Suc the_enat.simps zero_enat_def)
      qed
      qed
      qed

      have A: "¬ lnull (lappend Ps (LCons g LNil)) ∧ (v ∈ attacker → (s e v = Some
(lhd (lappend Ps (LCons g LNil)))))"
      proof
      show "¬ lnull (lappend Ps (LCons g LNil))" by simp
      show "v ∈ attacker →
s e v = Some (lhd (lappend Ps (LCons g LNil)))"
      proof
      assume "v ∈ attacker"
      show "s e v = Some (lhd (lappend Ps (LCons g LNil)))" using <v ∈ attacker>
Suc
      by (smt (verit) One_nat_def <p = LCons v Ps> diff_add_0 energy_game.energy_level.simps
eq_LConsD length_Conv length_list_of_conv_the_enat lfinite_ltl lhd_lappend list.size(3)
list_of_LCons list_of_LNil llast_singleton llist.disc(1) option.exhaust_sel option.inject
play_consistent_attacker.cases plus_1_eq_Suc)
      qed
      qed

      have "(lappend p (LCons g LNil)) = LCons v (lappend Ps (LCons g LNil))"
      by (simp add: <p = LCons v Ps>)
      thus ?case using play_consistent_attacker.simps A B
      by meson
      qed

```

We now define attacker winning strategies, i.e. attacker strategies where the defender does not win any consistent plays w.r.t some initial energy and a starting position.

```

fun attacker_winning_strategy:: "('energy ⇒ 'position ⇒ 'position option) ⇒ 'energy
⇒ 'position ⇒ bool" where
  "attacker_winning_strategy s e g = (attacker_strategy s ∧
(∀p. (play_consistent_attacker s (LCons g p) e ∧ valid_play (LCons g p))
→ ¬defender_wins_play e (LCons g p)))"

```

## 2.2 Non-positional Strategies

A non-positional strategy maps finite plays to a next position. We now introduce non-positional strategies to better characterise attacker winning budgets. These definitions closely resemble the definitions for energy-positional strategies.

```

definition attacker_nonpos_strategy:: "('position list  $\Rightarrow$  'position option)  $\Rightarrow$  bool"
where
  "attacker_nonpos_strategy s = ( $\forall$ list  $\neq$  []. ((last list)  $\in$  attacker
     $\wedge$   $\neg$ deadend (last list))  $\longrightarrow$  s list  $\neq$  None
     $\wedge$  (weight (last list) (the (s list))) $\neq$ None)"

```

We now define what it means for a play to be consistent with some non-positional strategy.

```

coinductive play_consistent_attacker_nonpos:: "('position list  $\Rightarrow$  'position option)
 $\Rightarrow$  ('position llist)  $\Rightarrow$  ('position list)  $\Rightarrow$  bool" where
  "play_consistent_attacker_nonpos s LNil _" |
  "play_consistent_attacker_nonpos s (LCons v LNil) []" |
  "(last (w#l)) $\notin$ attacker
 $\implies$  play_consistent_attacker_nonpos s (LCons v LNil) (w#l)" |
  "[[(last (w#l)) $\in$ attacker; the (s (w#l)) = v ]]"
 $\implies$  play_consistent_attacker_nonpos s (LCons v LNil) (w#l)" |
  "[[play_consistent_attacker_nonpos s Ps (l@[v]);  $\neg$ lnull Ps; v $\notin$ attacker]]
 $\implies$  play_consistent_attacker_nonpos s (LCons v Ps) l" |
  "[[play_consistent_attacker_nonpos s Ps (l@[v]);  $\neg$ lnull Ps; v $\in$ attacker;
    lhd Ps = the (s (l@[v]))]]
 $\implies$  play_consistent_attacker_nonpos s (LCons v Ps) l"

```

```

inductive_simps play_consistent_attacker_nonpos_cons_simp:
  "play_consistent_attacker_nonpos s (LCons x xs) []"

```

The definition allows for coinduction.

```

lemma play_consistent_attacker_nonpos_coinduct:
  assumes "Q s p l" and
    base: " $\bigwedge$ s v l. Q s (LCons v LNil) l  $\implies$  (l = []  $\vee$  (last l)  $\notin$  attacker
       $\vee$  ((last l) $\in$ attacker  $\wedge$  the (s l) = v))" and
    step: " $\bigwedge$ s v Ps l. Q s (LCons v Ps) l  $\wedge$  Ps $\neq$ LNil
       $\implies$  Q s Ps (l@[v])  $\wedge$  (v $\in$ attacker  $\longrightarrow$  lhd Ps = the (s (l@[v])))"
  shows "play_consistent_attacker_nonpos s p l"
  using assms proof(coinduction arbitrary: s p l)
  case play_consistent_attacker_nonpos
  then show ?case proof(cases "p=LNil")
    case True
    then show ?thesis by simp
  next
    case False
    hence " $\exists$ v p'. p = LCons v p'"
    by (simp add: neq_LNil_conv)
    from this obtain v p' where "p=LCons v p'" by auto
    then show ?thesis proof(cases "p'=LNil")
      case True
      then show ?thesis
        by (metis <p = LCons v p'> neq_Nil_conv play_consistent_attacker_nonpos(1)
          play_consistent_attacker_nonpos(2))
    next

```



```

      case False
      then show ?thesis
        using <p = LCons v p'> assms(3) llist.expand play_consistent_attacker_nonpos(1)
assms(2) by auto
    qed
  qed
qed

```

We now show that a position can be added to the end of a finite consistent play while remaining consistent.

```

lemma consistent_nonpos_append_defender:
  assumes "play_consistent_attacker_nonpos s (LCons v Ps) l" and
    "llast (LCons v Ps)  $\notin$  attacker" and "lfinite (LCons v Ps)"
  shows "play_consistent_attacker_nonpos s (lappend (LCons v Ps) (LCons g' LNil))
  l"
  using assms proof(induction "list_of Ps" arbitrary: v Ps l)
  case Nil
  hence v_append_Ps: "play_consistent_attacker_nonpos s (lappend (LCons v Ps) (LCons
  g' LNil)) l = play_consistent_attacker_nonpos s (LCons v (LCons g' LNil)) l"
    by (metis lappend_code(1) lappend_code(2) lfinite_LCons llist_of_eq_LNil_conv
    llist_of_list_of)

    from Nil.prem(1) have "play_consistent_attacker_nonpos s (LCons g' LNil) (l@[v])"
  using play_consistent_attacker_nonpos.intros Nil
    by (metis (no_types, lifting) lfinite_LCons list.exhaust_sel llast_singleton
    llist_of.simps(1) llist_of_list_of snoc_eq_iff_butlast)
  hence "play_consistent_attacker_nonpos s (LCons v (LCons g' LNil)) l" using play_consistent_a
  Nil
    by (metis lfinite_code(2) llast_singleton llist.disc(2) llist_of.simps(1) llist_of_list_of)

  then show ?case using v_append_Ps by simp
next
  case (Cons a x)
  hence v_append_Ps: "play_consistent_attacker_nonpos s (lappend (LCons v Ps) (LCons
  g' LNil)) l = play_consistent_attacker_nonpos s (LCons v (lappend Ps (LCons g' LNil)))
  l"
    by simp

  from Cons have " $\neg$ lnull Ps"
    by (metis list.discI list_of_LNil llist.collapse(1))

  have " $\neg$  lnull (lappend Ps (LCons g' LNil))" by simp

  have "x = list_of (ltl Ps)" using Cons.hyps(2)
    by (metis Cons.prem(3) lfinite_code(2) list.sel(3) tl_list_of)
  have "llast (LCons (lhd Ps) (ltl Ps))  $\notin$  attacker" using Cons.prem(2)
    by (simp add: < $\neg$  lnull Ps> llast_LCons)
  have "lfinite (LCons (lhd Ps) (ltl Ps))" using Cons.prem(3) by simp
  have "play_consistent_attacker_nonpos s (LCons (lhd Ps) (ltl Ps)) (l @ [v])" using
  Cons.prem(1) play_consistent_attacker_nonpos.simps
    by (smt (verit, best) < $\neg$  lnull Ps> eq_LConsD lhd_LCons lhd_LCons_ltl ltl_simps(2))
  hence "play_consistent_attacker_nonpos s (lappend Ps (LCons g' LNil)) (l @ [v])"
  using Cons.hyps <lfinite (LCons (lhd Ps) (ltl Ps))> <llast (LCons (lhd Ps) (ltl
  Ps))  $\notin$  attacker> <x = list_of (ltl Ps)>
    by (metis < $\neg$  lnull Ps> lhd_LCons_ltl)

```

```

    have "play_consistent_attacker_nonpos s (LCons v (lappend Ps (LCons g' LNil)))
1"
    proof(cases "v ∈ attacker")
      case True
        have "lhd Ps = the (s (1 @ [v]))" using True Cons.prem1 play_consistent_attacker_nonpos.
          by (smt (verit) <¬ lnull Ps> llist.distinct(1) llist.inject lnull_def)
        hence "lhd (lappend Ps (LCons g' LNil)) = the (s (1 @ [v]))" by (simp add: <¬
lnull Ps>)

        then show ?thesis using play_consistent_attacker_nonpos.intros(6) True <play_consistent_att
s (lappend Ps (LCons g' LNil)) (1 @ [v])> <lhd (lappend Ps (LCons g' LNil)) = the
(s (1 @ [v]))> <¬ lnull (lappend Ps (LCons g' LNil))>
          by simp
        next
          case False
            then show ?thesis using play_consistent_attacker_nonpos.intros(5) False <¬
lnull (lappend Ps (LCons g' LNil))> <play_consistent_attacker_nonpos s (lappend
Ps (LCons g' LNil)) (1 @ [v])>
              by simp
            qed
            then show ?case using v_append_Ps by simp
          qed

lemma consistent_nonpos_append_attacker:
  assumes "play_consistent_attacker_nonpos s (LCons v Ps) 1"
    and "llass (LCons v Ps) ∈ attacker" and "lfinite (LCons v Ps)"
  shows "play_consistent_attacker_nonpos s (lappend (LCons v Ps) (LCons (the (s
(l@list_of (LCons v Ps)))) LNil)) 1"
  using assms proof(induction "list_of Ps" arbitrary: v Ps 1)
    case Nil
      hence v_append_Ps: "play_consistent_attacker_nonpos s (lappend (LCons v Ps) (LCons
(the (s (l@list_of (LCons v Ps)))) LNil)) 1
        = play_consistent_attacker_nonpos s (LCons v (LCons (the (s (l@[v]))) LNil))
1"
      by (metis lappend_code(1) lappend_code(2) lfinite_code(2) list_of_LCons llist_of.simps(1)
llass_of_list_of)
      have "play_consistent_attacker_nonpos s (LCons v (LCons (the (s (l@[v]))) LNil))
1" using play_consistent_attacker_nonpos.intros Nil
        by (metis hd_Cons_tl lhd_LCons llist.disc(2))
      then show ?case using v_append_Ps by simp
    next
      case (Cons a x)
        have v_append_Ps: "play_consistent_attacker_nonpos s (lappend (LCons v Ps) (LCons
(the (s (1 @ list_of (LCons v Ps)))) LNil)) 1
          = play_consistent_attacker_nonpos s (LCons v (lappend Ps (LCons
(the (s (1 @ [v]@list_of Ps))) LNil))) 1"
        using Cons.prem3 by auto
        have "x = list_of (ltl Ps)" using Cons.hyps(2)
          by (metis Cons.prem3 lfinite_code(2) list.sel(3) tl_list_of)
        have "play_consistent_attacker_nonpos s (LCons (lhd Ps) (ltl Ps)) (l@[v])" using
Cons.prem1 play_consistent_attacker_nonpos.simps
          by (smt (verit) Cons.hyps(2) eq_LConsD lhd_LCons list.discI list_of_LNil ltl_simps(2))
        have "llass (LCons (lhd Ps) (ltl Ps)) ∈ attacker" using Cons.prem2
          by (metis Cons.hyps(2) lhd_LCons_ltl list.distinct(1) list_of_LNil llass_LCons
llass.collapse(1))
        have "lfinite (LCons (lhd Ps) (ltl Ps))" using Cons.prem3 by simp

```

```

  hence "play_consistent_attacker_nonpos s (lappend Ps (LCons (the (s ((l @[v]))@list_of
Ps))) LNil)) (l@[v])"
  using Cons.hyps <x = list_of (ltl Ps)> <play_consistent_attacker_nonpos s (LCons
(lhd Ps) (ltl Ps)) (l@[v])>
  <llast (LCons (lhd Ps) (ltl Ps)) ∈ attacker>
  by (metis llist.exhaust_sel ltl_simps(1) not_Conself2)
  hence "play_consistent_attacker_nonpos s (LCons v (lappend Ps (LCons (the (s ((l
@[v]))@list_of Ps)))) LNil))) 1"
  using play_consistent_attacker_nonpos_simps Cons
  by (smt (verit) lhd_LCons lhd_lappend list.discI list_of_LNil llist.distinct(1)
lnull_lappend ltl_simps(2))

  then show ?case using v_append_Ps by simp
qed

```

We now define non-positional attacker winning strategies, i.e. attacker strategies where the defender does not win any consistent plays w.r.t some initial energy and a starting position.

```

fun nonpos_attacker_winning_strategy:: "('position list ⇒ 'position option) ⇒
'energy ⇒ 'position ⇒ bool" where
"nonpos_attacker_winning_strategy s e g = (attacker_nonpos_strategy s ∧
(∀p. (play_consistent_attacker_nonpos s (LCons g p) []
  ∧ valid_play (LCons g p)) → ¬defender_wins_play e (LCons g p)))"

```

## 2.3 Attacker Winning Budgets

We now define attacker winning budgets utilising strategies.

```

fun winning_budget:: "'energy ⇒ 'position ⇒ bool" where
"winning_budget e g = (∃s. attacker_winning_strategy s e g)"

fun nonpos_winning_budget:: "'energy ⇒ 'position ⇒ bool" where
"nonpos_winning_budget e g = (∃s. nonpos_attacker_winning_strategy s e g)"

```

Note that `nonpos_winning_budget = winning_budget` holds but is not proven in this theory. Using this fact we can give an inductive characterisation of attacker winning budgets.

```

inductive winning_budget_ind:: "'energy ⇒ 'position ⇒ bool" where
defender: "winning_budget_ind e g" if
" g ∉ attacker ∧ (∀g'. weight g g' ≠ None → (apply_w g g' e ≠ None
  ∧ winning_budget_ind (the (apply_w g g' e)) g'))" |
attacker: "winning_budget_ind e g" if
" g ∈ attacker ∧ (∃g'. weight g g' ≠ None ∧ apply_w g g' e ≠ None
  ∧ winning_budget_ind (the (apply_w g g' e)) g'"

```

Before proving some correspondence of those definitions we first note that attacker winning budgets in monotonic energy games are upward-closed. We show this for two of the three definitions.

```

lemma upward_closure_wb_nonpos:
  assumes monotonic: "∧g g' e e'. weight g g' ≠ None
    ⇒ apply_w g g' e ≠ None ⇒ leq e e' ⇒ apply_w g g' e' ≠ None
    ∧ leq (the (apply_w g g' e)) (the (apply_w g g' e'))"
  and "leq e e'" and "nonpos_winning_budget e g"
  shows "nonpos_winning_budget e' g"
proof-

```

```

    from assms have "∃s. nonpos_attacker_winning_strategy s e g" using nonpos_winning_budget.simp
  by simp
    from this obtain s where S: "nonpos_attacker_winning_strategy s e g" by auto
    have "nonpos_attacker_winning_strategy s e' g" unfolding nonpos_attacker_winning_strategy.simp

  proof
    show "attacker_nonpos_strategy s" using S by simp
    show "∀p. play_consistent_attacker_nonpos s (LCons g p) [] ∧ valid_play (LCons
g p) → ¬ defender_wins_play e' (LCons g p)"
  proof
    fix p
    show "play_consistent_attacker_nonpos s (LCons g p) [] ∧ valid_play (LCons
g p) → ¬ defender_wins_play e' (LCons g p) "
  proof
    assume P: "play_consistent_attacker_nonpos s (LCons g p) [] ∧ valid_play
(LCons g p)"
    hence X: "lfinite (LCons g p) ∧ ¬ (energy_level e (LCons g p) (the_enat
(1length (LCons g p)) - 1) = None ∨ llast (LCons g p) ∈ attacker ∧ deadend (llast
(LCons g p)))"
    using S unfolding nonpos_attacker_winning_strategy.simps defender_wins_play_def
  by simp
    have "lfinite (LCons g p) ∧ ¬ (energy_level e' (LCons g p) (the_enat (1length
(LCons g p)) - 1) = None ∨ llast (LCons g p) ∈ attacker ∧ deadend (llast (LCons
g p)))"
  proof
    show "lfinite (LCons g p)" using P S unfolding nonpos_attacker_winning_strategy.simps
defender_wins_play_def by simp
    have "energy_level e' (LCons g p) (the_enat (1length (LCons g p)) - 1)
≠ None ∧ ¬(llast (LCons g p) ∈ attacker ∧ deadend (llast (LCons g p)))"
  proof
    have E: "energy_level e (LCons g p) (the_enat (1length (LCons g p))
- 1) ≠ None" using P S unfolding nonpos_attacker_winning_strategy.simps defender_wins_play_def
  by simp
    have "∧len. len ≤ the_enat (1length (LCons g p)) - 1 → energy_level
e' (LCons g p) len ≠ None ∧ (leq (the (energy_level e (LCons g p) len)) (the (energy_level
e' (LCons g p) len)))"
  proof
    fix len
    show "len ≤ the_enat (1length (LCons g p)) - 1 ⇒ energy_level e'
(LCons g p) len ≠ None ∧ leq (the (energy_level e (LCons g p) len)) (the (energy_level
e' (LCons g p) len))"
  proof(induct len)
    case 0
    then show ?case using energy_level.simps assms(2)
    by (simp add: llist.distinct(1) option.discI option.sel)
  next
    case (Suc len)
    hence "energy_level e' (LCons g p) len ≠ None" by simp
    have W: "weight (lnth (LCons g p) len)(lnth (LCons g p) (Suc len))
≠ None" using P Suc.prem1 valid_play.simps valid_play_nth_not_None
    by (smt (verit) <lfinite (LCons g p)> diff_Suc_1 enat_ord_simps(2)
le_less_Suc_eq less_imp_diff_less lfinite_1length_enat linorder_le_less_linear not_less_eq
the_enat.simps)
    have A: "apply_w (lnth (LCons g p) len) (lnth (LCons g p) (Suc len))
(the (energy_level e (LCons g p) len)) ≠ None"
    using E Suc.prem1 energy_level_nth by blast

```

```

      have "llength (LCons g p) > Suc len" using Suc.premss
      by (metis <lfinite (LCons g p)> diff_Suc_1 enat_ord_simps(2)
less_imp_diff_less lfinite_conv_llength_enat nless_le not_le_imp_less not_less_eq
the_enat.simps)
      hence "energy_level e' (LCons g p) (Suc len) = apply_w (lnth (LCons
g p) len)(lnth (LCons g p) (Suc len)) (the (energy_level e' (LCons g p) len))"
      using <energy_level e' (LCons g p) len ≠ None> energy_level.simps
      by (meson leD)
      then show ?case using A W Suc assms
      by (smt (verit) E Suc_leD energy_level.simps(2) energy_level_nth)
    qed
  qed
  thus "energy_level e' (LCons g p) (the_enat (llength (LCons g p)) -
1) ≠ None" by simp
  show "¬ (llast (LCons g p) ∈ attacker ∧ deadend (llast (LCons g p)))"
using P S unfolding nonpos_attacker_winning_strategy.simps defender_wins_play_def
by simp
  qed
  thus "¬ (energy_level e' (LCons g p) (the_enat (llength (LCons g p)) -
1) = None ∨ llast (LCons g p) ∈ attacker ∧ deadend (llast (LCons g p)))"
  by simp
  qed
  thus "¬ defender_wins_play e' (LCons g p)" unfolding defender_wins_play_def
by simp
  qed
  qed
  qed
  thus ?thesis using nonpos_winning_budget.simps by auto
qed

```

```

lemma upward_closure_wb_ind:
  assumes monotonic: "⋀g g' e e'. weight g g' ≠ None
    ⇒ apply_w g g' e ≠ None ⇒ leq e e' ⇒ apply_w g g' e' ≠ None
    ∧ leq (the (apply_w g g' e)) (the (apply_w g g' e'))"
    and "leq e e'" and "winning_budget_ind e g"
  shows "winning_budget_ind e' g"
proof-
  define P where "P ≡ λ e g. (∀e'. leq e e' → winning_budget_ind e' g)"
  have "P e g" using assms(3) proof (induct rule: winning_budget_ind.induct)
    case (defender g e)
    then show ?case using P_def
      using monotonic winning_budget_ind.defender by blast
  next
    case (attacker g e)
    then show ?case using P_def
      using monotonic winning_budget_ind.attacker by blast
  qed
  thus ?thesis using assms(2) P_def by blast
qed

```

Now we prepare the proof of the inductive characterisation. For this we define an order and a set allowing for a well-founded induction.

```

definition strategy_order:: "('energy ⇒ 'position ⇒ 'position option) ⇒
  'position × 'energy ⇒ 'position × 'energy ⇒ bool" where
  "strategy_order s ≡ λ(g1, e1)(g2, e2). Some e1 = apply_w g2 g1 e2 ∧

```

```

    (if g2 ∈ attacker then Some g1 = s e2 g2 else weight g2 g1 ≠ None)"

definition reachable_positions:: "('energy ⇒ 'position ⇒ 'position option) ⇒
'position ⇒ 'energy ⇒ ('position × 'energy) set" where
  "reachable_positions s g e = {(g',e') | g' e' .
    (∃p. lfinite p ∧ llast (LCons g p) = g' ∧ valid_play (LCons g p)
      ∧ play_consistent_attacker s (LCons g p) e
      ∧ Some e' = energy_level e (LCons g p) (the_enat (llength p))))}"

lemma strategy_order_well_founded:
  assumes "attacker_winning_strategy s e g"
  shows "wfp_on (strategy_order s) (reachable_positions s g e)"
  unfolding Restricted_Predicates.wfp_on_def
proof
  assume "∃f. ∀i. f i ∈ reachable_positions s g e ∧ strategy_order s (f (Suc i))
(f i)"
  from this obtain f where F: "∀i. f i ∈ reachable_positions s g e ∧ strategy_order
s (f (Suc i)) (f i)" by auto

  define p where "p = lmap (λi. fst (f i))(iterates Suc 0)"
  hence "∧i. lnth p i = fst (f i)"
    by simp

  from p_def have "¬lfinite p" by simp

  have "∧i. enat (Suc i) < llength p ⇒ weight (lnth p i) (lnth p (Suc i)) ≠ None"
  proof-
    fix i
    have "∃g1 e1 g2 e2. (f i) = (g2, e2) ∧ f (Suc i) = (g1, e1)" using F reachable_positions_d
  by simp
    from this obtain g1 e1 g2 e2 where "(f i) = (g2, e2)" and "f (Suc i) = (g1,
e1)"
      by blast
    assume "enat (Suc i) < llength p"

    have "weight g2 g1 ≠ None"
    proof(cases "g2 ∈ attacker")
      case True
      then show ?thesis
      proof(cases "deadend g2")
        case True
        have "(g2, e2) ∈ reachable_positions s g e" using F by (metis <f i = (g2,
e2)>)
        hence "(∃p'. (lfinite p' ∧ llast (LCons g p') = g2
          ∧ valid_play (LCons g p')
          ∧ play_consistent_attacker s
          ∧ (Some e2 = energy_level e
            (LCons g p') (the_enat (llength p')))))"
          using reachable_positions_def by simp
          from this obtain p' where P': "(lfinite p' ∧ llast (LCons g p') = g2
            ∧ valid_play (LCons g p')
            ∧ play_consistent_attacker s
            ∧ (Some e2 = energy_level e
              (LCons g p') (the_enat (llength p'))))" by auto

```

```

      have "¬defender_wins_play e (LCons g p')" using assms unfolding attacker_winning_strategy
using P' by auto
      have "llast (LCons g p') ∈ attacker ∧ deadend (llast (LCons g p'))" using
True <g2 ∈ attacker> P' by simp
      hence "defender_wins_play e (LCons g p')"
        unfolding defender_wins_play_def by simp
      hence "False" using <¬defender_wins_play e (LCons g p')> by simp
      then show ?thesis by simp
    next
      case False
      from True have "Some g1 = s e2 g2"
        using F unfolding strategy_order_def using <f (Suc i) = (g1, e1)> <(f
i) = (g2, e2)>
        by (metis (mono_tags, lifting) case_prod_conv)
      have "(∀g e. (g ∈ attacker ∧ ¬ deadend g) → (s e g ≠ None ∧ weight g
(the (s e g)) ≠ None))"
        using assms unfolding attacker_winning_strategy.simps attacker_strategy_def
        by simp
      hence "weight g2 (the (s e2 g2)) ≠ None" using False True
        by simp
      then show ?thesis using <Some g1 = s e2 g2>
        by (metis option.sel)
    qed
  next
    case False
    then show ?thesis using F unfolding strategy_order_def using <f (Suc i) =
(g1, e1)> <(f i) = (g2, e2)>
      by (metis (mono_tags, lifting) case_prod_conv)
    qed
    thus "weight (lnth p i) (lnth p (Suc i)) ≠ None"
      using p_def <f i = (g2, e2)> <f (Suc i) = (g1, e1)> by simp
    qed
  hence "valid_play p" using valid_play_nth
    by simp

  have "(f 0) ∈ reachable_positions s g e" using F by simp
  hence "∃g0 e0. f 0 = (g0,e0)" using reachable_positions_def by simp
  from this obtain g0 e0 where "f 0 = (g0,e0)" by blast
  hence "∃p'. (lfinite p' ∧ llast (LCons g p') = g0
                                                    ∧ valid_play (LCons g p')
                                                    ∧ play_consistent_attacker s
(LCons g p') e)
                                                    ∧ (Some e0 = energy_level e
(LCons g p') (the_enat (llength p'))))"
    using <(f 0) ∈ reachable_positions s g e> unfolding reachable_positions_def
    by auto
    from this obtain p' where P': "(lfinite p' ∧ llast (LCons g p') = g0
                                                    ∧ valid_play (LCons g p')
                                                    ∧ play_consistent_attacker s
(LCons g p') e)
                                                    ∧ (Some e0 = energy_level e
(LCons g p') (the_enat (llength p'))))" by auto

  have "∧i. strategy_order s (f (Suc i)) (f i)" using F by simp

```

```

hence "\i. Some (snd (f (Suc i))) = apply_w (fst (f i)) (fst (f (Suc i))) (snd
(f i))" using strategy_order_def
by (simp add: case_prod_beta)
hence "\i. (snd (f (Suc i))) = the (apply_w (fst (f i)) (fst (f (Suc i))) (snd
(f i)))"
by (metis option.sel)

have "\i. (energy_level e0 p i) = Some (snd (f i))"
proof-
fix i
show "(energy_level e0 p i) = Some (snd (f i))"
proof(induct i)
case 0
then show ?case using <f 0 = (g0,e0)> <¬ lfinite p> by auto
next
case (Suc i)
have "Some (snd (f (Suc i))) = (apply_w (fst (f i)) (fst (f (Suc i))) (snd
(f i)))"
using <\i. Some (snd (f (Suc i))) = apply_w (fst (f i)) (fst (f (Suc i)))
(snd (f i))> by simp
also have "... = (apply_w (fst (f i)) (fst (f (Suc i))) ( the (energy_level
e0 p i)))" using Suc by simp
also have "... = (apply_w (lnth p i) (lnth p (Suc i)) ( the (energy_level
e0 p i)))" using <\i. lnth p i = fst (f i)> by simp
also have "... = (energy_level e0 p (Suc i))" using energy_level.simps <¬
lfinite p> Suc
by (simp add: lfinite_conv_llength_enat)
finally show ?case
by simp
qed
qed

define Q where "Q ≡ λ s p e0. ¬lfinite p ∧ valid_play p ∧ (∀i. (energy_level
e0 p i) ≠ None ∧ ((lnth p i), the (energy_level e0 p i)) ∈ reachable_positions
s g e
      ∧ strategy_order s ((lnth p (Suc i)), the (energy_level e0 p (Suc i)))
      ((lnth p i), the (energy_level e0 p i)))"

have Q: "¬lfinite p ∧ valid_play p ∧ (∀i. (energy_level e0 p i) ≠ None ∧ ((lnth
p i), the (energy_level e0 p i)) ∈ reachable_positions s g e
      ∧ strategy_order s ((lnth p (Suc i)), the (energy_level e0 p (Suc i)))
      ((lnth p i), the (energy_level e0 p i)))"
proof
show "¬ lfinite p " using <¬lfinite p> .
show "valid_play p ∧
(∀i. energy_level e0 p i ≠ None ∧
(lnth p i, the (energy_level e0 p i)) ∈ reachable_positions s g e ∧
strategy_order s (lnth p (Suc i), the (energy_level e0 p (Suc i)))
(lnth p i, the (energy_level e0 p i)))"
proof
show "valid_play p" using <valid_play p> .
show "∀i. energy_level e0 p i ≠ None ∧
(lnth p i, the (energy_level e0 p i)) ∈ reachable_positions s g e ∧
strategy_order s (lnth p (Suc i), the (energy_level e0 p (Suc i)))
(lnth p i, the (energy_level e0 p i)) "
proof

```



```

    fix i
    show "energy_level e0 p i ≠ None ∧
      (lnth p i, the (energy_level e0 p i)) ∈ reachable_positions s g e ∧
      strategy_order s (lnth p (Suc i), the (energy_level e0 p (Suc i)))
      (lnth p i, the (energy_level e0 p i))"
    proof
      show "energy_level e0 p i ≠ None" using <^i. (energy_level e0 p i) =
Some (snd (f i))> by simp
      show "(lnth p i, the (energy_level e0 p i)) ∈ reachable_positions s g
e ∧
      strategy_order s (lnth p (Suc i), the (energy_level e0 p (Suc i)))
      (lnth p i, the (energy_level e0 p i)) "
      proof
        show "(lnth p i, the (energy_level e0 p i)) ∈ reachable_positions s
g e"
          using <^i. (energy_level e0 p i) = Some (snd (f i))> F <^i. lnth
p i = fst (f i)>
          by simp
          show "strategy_order s (lnth p (Suc i), the (energy_level e0 p (Suc
i)))
(lnth p i, the (energy_level e0 p i))"
            using <^i. strategy_order s (f (Suc i)) (f i)> <^i. lnth p i = fst
(f i)> <^i. (energy_level e0 p i) = Some (snd (f i))>
            by (metis option.sel split_pairs)
            qed
            qed
            qed
            qed
      hence "Q s p e0" using Q_def by simp

    have "∧s v Ps e'.
      (¬ lfinite (LCons v Ps) ∧
      valid_play (LCons v Ps) ∧
      (∀i. energy_level e' (LCons v Ps) i ≠ None ∧
      (lnth (LCons v Ps) i, the (energy_level e' (LCons v Ps) i)) ∈ reachable_positions
s g e ∧
      strategy_order s (lnth (LCons v Ps) (Suc i), the (energy_level e' (LCons
v Ps) (Suc i)))
      (lnth (LCons v Ps) i, the (energy_level e' (LCons v Ps) i)))) ∧
      ¬ lnull Ps ⇒
      (¬ lfinite Ps ∧
      valid_play Ps ∧
      (∀i. energy_level (the (apply_w v (lhd Ps) e')) Ps i ≠ None ∧
      (lnth Ps i, the (energy_level (the (apply_w v (lhd Ps) e')) Ps i))
      ∈ reachable_positions s g e ∧
      strategy_order s (lnth Ps (Suc i), the (energy_level (the (apply_w
v (lhd Ps) e')) Ps (Suc i)))
      (lnth Ps i, the (energy_level (the (apply_w v (lhd Ps) e')) Ps i))))
      ∧
      (v ∈ attacker → s e' v = Some (lhd Ps)) ∧ (apply_w v (lhd Ps) e') ≠ None"
    proof-
      fix s v Ps e'
      assume A: "(¬ lfinite (LCons v Ps) ∧ valid_play (LCons v Ps) ∧ (∀i. energy_level
e' (LCons v Ps) i ≠ None ∧

```

```

      (lnth (LCons v Ps) i, the (energy_level e' (LCons v Ps) i))
      ∈ reachable_positions s g e ∧
      strategy_order s (lnth (LCons v Ps) (Suc i), the (energy_level e' (LCons
v Ps) (Suc i)))
      (lnth (LCons v Ps) i, the (energy_level e' (LCons v Ps) i)))) ∧
      ¬ lnull Ps"

  show "(¬lfinite Ps ∧ valid_play Ps ∧ (∀i. energy_level (the (apply_w v (lhd
Ps) e'))) Ps i ≠ None ∧
      (lnth Ps i, the (energy_level (the (apply_w v (lhd Ps) e'))) Ps i))
      ∈ reachable_positions s g e ∧
      strategy_order s
      (lnth Ps (Suc i), the (energy_level (the (apply_w v (lhd Ps) e'))) Ps
(Suc i)))
      (lnth Ps i, the (energy_level (the (apply_w v (lhd Ps) e'))) Ps i)))
  ∧
  (v ∈ attacker → s e' v = Some (lhd Ps)) ∧ (apply_w v (lhd Ps) e') ≠ None"
proof
  show "(¬lfinite Ps ∧ valid_play Ps ∧ (∀i. energy_level (the (apply_w v (lhd
Ps) e'))) Ps i ≠ None ∧
      (lnth Ps i, the (energy_level (the (apply_w v (lhd Ps) e'))) Ps i))
      ∈ reachable_positions s g e ∧
      strategy_order s
      (lnth Ps (Suc i), the (energy_level (the (apply_w v (lhd Ps) e'))) Ps (Suc
i)))
      (lnth Ps i, the (energy_level (the (apply_w v (lhd Ps) e'))) Ps i)))"
proof
  show "¬ lfinite Ps" using A by simp
  show "valid_play Ps ∧
(∀i. energy_level (the (apply_w v (lhd Ps) e'))) Ps i ≠ None ∧
      (lnth Ps i, the (energy_level (the (apply_w v (lhd Ps) e'))) Ps i))
      ∈ reachable_positions s g e ∧
      strategy_order s
      (lnth Ps (Suc i), the (energy_level (the (apply_w v (lhd Ps) e'))) Ps (Suc
i)))
      (lnth Ps i, the (energy_level (the (apply_w v (lhd Ps) e'))) Ps i)))"
proof
  show "valid_play Ps" using A valid_play.simps
  by (metis llist.distinct(1) llist.inject)
  show "∀i. energy_level (the (apply_w v (lhd Ps) e'))) Ps i ≠ None ∧
      (lnth Ps i, the (energy_level (the (apply_w v (lhd Ps) e'))) Ps i))
      ∈ reachable_positions s g e ∧
      strategy_order s
      (lnth Ps (Suc i), the (energy_level (the (apply_w v (lhd Ps) e'))) Ps (Suc
i)))
      (lnth Ps i, the (energy_level (the (apply_w v (lhd Ps) e'))) Ps i))"
proof
  fix i
  show "energy_level (the (apply_w v (lhd Ps) e'))) Ps i ≠ None ∧
      (lnth Ps i, the (energy_level (the (apply_w v (lhd Ps) e'))) Ps i))
      ∈ reachable_positions s g e ∧
      strategy_order s
      (lnth Ps (Suc i), the (energy_level (the (apply_w v (lhd Ps) e'))) Ps (Suc
i)))
      (lnth Ps i, the (energy_level (the (apply_w v (lhd Ps) e'))) Ps i))"
proof

```

```

    from A have "energy_level e' (LCons v Ps) (Suc i) ≠ None" by blast
    from A have "valid_play (LCons v Ps) ∧ ¬ lnull Ps" by simp
    have "apply_w v (lhd Ps) e' ≠ None" using energy_level.simps
      by (metis A lhd_conv_lnth lnth_0 lnth_Suc_LCons option.sel)
    from A have "enat i < (llength Ps)"
      by (meson Suc_ile_eq <¬ lfinite Ps> enat_less_imp_le less_enatE
lfinite_conv_llength_enat)
    have EL: "energy_level (the (apply_w v (lhd Ps) e')) Ps i = energy_level
e' (LCons v Ps) (Suc i)"
      using energy_level_cons <valid_play (LCons v Ps) ∧ ¬ lnull Ps>
<apply_w v (lhd Ps) e' ≠ None>
      by (simp add: <enat i < llength Ps>)
    thus "energy_level (the (apply_w v (lhd Ps) e')) Ps i ≠ None"
      using <energy_level e' (LCons v Ps) (Suc i) ≠ None> by simp
    show "(lnth Ps i, the (energy_level (the (apply_w v (lhd Ps) e'))
Ps i)) ∈ reachable_positions s g e ∧
strategy_order s (lnth Ps (Suc i), the (energy_level (the (apply_w v (lhd Ps)
e')) Ps (Suc i)))
(lnth Ps i, the (energy_level (the (apply_w v (lhd Ps) e')) Ps i))"
      proof
        have "(lnth Ps i, the (energy_level (the (apply_w v (lhd Ps) e'))
Ps i)) = (lnth (LCons v Ps) (Suc i), the (energy_level e' (LCons v Ps) (Suc i)))"

          using EL by simp
          thus "(lnth Ps i, the (energy_level (the (apply_w v (lhd Ps) e'))
Ps i)) ∈ reachable_positions s g e"
            using A by metis
          have <enat (Suc i) < llength Ps>
            using <¬ lfinite Ps> enat_iless linorder_less_linear llength_eq_enat_lfinite
by blast
          hence "(lnth Ps (Suc i), the (energy_level (the (apply_w v (lhd
Ps) e')) Ps (Suc i)))
= (lnth (LCons v Ps) (Suc (Suc i)), the (energy_level e'
(LCons v Ps) (Suc (Suc i))))"
            using energy_level_cons <valid_play (LCons v Ps) ∧ ¬ lnull Ps>
<apply_w v (lhd Ps) e' ≠ None>
            by (metis lnth_Suc_LCons)
          thus "strategy_order s (lnth Ps (Suc i), the (energy_level (the
(apply_w v (lhd Ps) e')) Ps (Suc i)))
(lnth Ps i, the (energy_level (the (apply_w v (lhd Ps) e')) Ps i))" using
A
            by (metis EL lnth_Suc_LCons)
          qed
        qed
      qed
    qed
  show "(v ∈ attacker → s e' v = Some (lhd Ps)) ∧ (apply_w v (lhd Ps) e')
≠ None"
    proof
      show "v ∈ attacker → s e' v = Some (lhd Ps)"
        proof
          assume "v ∈ attacker"
          from A have "strategy_order s (lnth (LCons v Ps) (Suc 0), the (energy_level
e' (LCons v Ps) (Suc 0))) (lnth (LCons v Ps) 0, the (energy_level e' (LCons v Ps)
0))"

```

```

      by blast
      hence "strategy_order s ((lhd Ps), the (energy_level e' (LCons v Ps) (Suc
0))) (v, the (energy_level e' (LCons v Ps) 0))"
      by (simp add: A lnth_0_conv_lhd)
      hence "strategy_order s ((lhd Ps), the (energy_level e' (LCons v Ps) (Suc
0))) (v, e')" using energy_level.simps
      by simp
      hence "(if v ∈ attacker then Some (lhd Ps) = s e' v else weight v (lhd
Ps) ≠ None)" using strategy_order_def
      using split_beta split_pairs by auto
      thus "s e' v = Some (lhd Ps)" using <v ∈ attacker> by auto
    qed
    from A have "energy_level (the (apply_w v (lhd Ps) e')) Ps 0 ≠ None" by
auto
    show "apply_w v (lhd Ps) e' ≠ None"
    by (metis A energy_level.simps(1) energy_level.simps(2) eq_LConsD lnth_0
lnth_Suc_LCons not_lnull_conv option.sel)
  qed
  qed
  qed

  hence "( $\bigwedge$  s v Ps e'.
    Q s (LCons v Ps) e'  $\wedge$   $\neg$  lnull Ps  $\implies$  (apply_w v (lhd Ps) e') ≠ None  $\wedge$ 
    Q s Ps (the (apply_w v (lhd Ps) e'))  $\wedge$  (v ∈ attacker  $\longrightarrow$  s e' v = Some
(lhd Ps)))" using Q_def by blast

  hence "play_consistent_attacker s p e0"
  using <Q s p e0> play_consistent_attacker_coinduct
  by metis

  have "valid_play (lappend (LCons g p') (ltl p))  $\wedge$  play_consistent_attacker s (lappend
(LCons g p') (ltl p)) e"
  proof
    have "weight (llast (LCons g p')) (lhd (ltl p)) ≠ None" using P'
    by (metis < $\bigwedge$  i. lnth p i = fst (f i)> < $\neg$  lfinite p> <f 0 = (g0, e0)> <valid_play
p> fstI lfinite.simps lnth_0 ltl_simps(2) valid_play.cases)
    show "valid_play (lappend (LCons g p') (ltl p))" using valid_play_append P'
    by (metis (no_types, lifting) < $\neg$  lfinite p> <valid_play p> <weight (llast
(LCons g p')) (lhd (ltl p)) ≠ None> lfinite_LConsI lfinite_LNil llist.exhaust_sel
ltl_simps(2) valid_play.simps)

    have "energy_level e (LCons g p') (the_enat (llength p')) ≠ None"
    by (metis P' not_Some_eq)
    hence A: "lfinite p'  $\wedge$  llast (LCons g p') = lhd p  $\wedge$  play_consistent_attacker
s p (the (energy_level e (LCons g p') (the_enat (llength p'))))
       $\wedge$  play_consistent_attacker s (LCons g p') e  $\wedge$  valid_play (LCons g p')
 $\wedge$  energy_level e (LCons g p') (the_enat (llength p')) ≠ None"
    using P' <play_consistent_attacker s p e0> p_def <f 0 = (g0, e0)>
    by (metis < $\bigwedge$  i. lnth p i = fst (f i)> < $\neg$  lfinite p> fst_conv lhd_conv_lnth
lnull_imp_lfinite option.sel)

    show "play_consistent_attacker s (lappend (LCons g p') (ltl p)) e"
    using A proof(induct "the_enat (llength p')" arbitrary: p' g e)
    case 0
    hence "(lappend (LCons g p') (ltl p)) = p"
    by (metis < $\neg$  lfinite p> gen_llength_code(1) lappend_code(1) lappend_code(2)

```

```

lfinite_llength_enat lhd_LCons_ltl llast_singleton llength_LNil llength_code llength_eq_0
l1ist.collapse(1) the_enat.simps)
  have "the (energy_level e (LCons g p') (the_enat (llength p')))) = e" using
0 energy_level.simps by auto
  then show ?case using <(lappend (LCons g p') (ltl p)) = p> 0 by simp
next
  case (Suc x)
  hence "lhd p' = lhd (lappend (p') (ltl p))"
    using the_enat_0 by auto
  have "∃Ps. (lappend (LCons g p') (ltl p)) = LCons g Ps"
    by simp
  from this obtain Ps where "(lappend (LCons g p') (ltl p)) = LCons g Ps" by
auto
  hence "(lappend (p') (ltl p)) = Ps" by simp

  have "g ∈ attacker ⟶ s e g = Some (lhd Ps)"
  proof
    assume "g ∈ attacker"
    show "s e g = Some (lhd Ps)"
      using Suc
      by (metis Zero_not_Suc <g ∈ attacker> <lappend p' (ltl p) = Ps> <lhd
p' = lhd (lappend p' (ltl p))> lhd_LCons llength_LNil l1ist.distinct(1) ltl_simps(2)
play_consistent_attacker.cases the_enat_0)
    qed
  have "play_consistent_attacker s (lappend (LCons (lhd p') (ltl p')) (ltl p))
(the (apply_w g (lhd p') e))"
  proof-
    have "x = the_enat (llength (ltl p'))" using Suc
      by (metis One_nat_def diff_Suc_1' epred_enat epred_llength lfinite_conv_llength_enat
the_enat.simps)
    have "lfinite (ltl p') ∧
llast (LCons (lhd p') (ltl p')) = lhd p ∧
play_consistent_attacker s p
(the (energy_level (the (apply_w g (lhd p') e)) (LCons (lhd p') (ltl p')) (the_enat
(llength (ltl p'))))) ∧
play_consistent_attacker s (LCons (lhd p') (ltl p')) (the (apply_w g (lhd p')
e)))
  ∧ valid_play (LCons (lhd p') (ltl p')) ∧ energy_level (the (apply_w g (lhd
p') e)) (LCons (lhd p') (ltl p')) (the_enat (llength (ltl p')))) ≠ None"
    proof
      show "lfinite (ltl p'" using Suc lfinite_ltl by simp
      show "llast (LCons (lhd p') (ltl p')) = lhd p ∧
play_consistent_attacker s p
(the (energy_level (the (apply_w g (lhd p') e)) (LCons (lhd p') (ltl p'))
(the_enat (llength (ltl p'))))) ∧
play_consistent_attacker s (LCons (lhd p') (ltl p')) (the (apply_w g (lhd p')
e))) ∧
valid_play (LCons (lhd p') (ltl p')) ∧ energy_level (the (apply_w g (lhd p')
e)) (LCons (lhd p') (ltl p')) (the_enat (llength (ltl p')))) ≠ None"
    proof
      show "llast (LCons (lhd p') (ltl p')) = lhd p" using Suc
      by (metis (no_types, lifting) <x = the_enat (llength (ltl p'))> llast_LCons2
l1ist.exhaust_sel ltl_simps(1) n_not_Suc_n)
      show "play_consistent_attacker s p
(the (energy_level (the (apply_w g (lhd p') e)) (LCons (lhd p') (ltl p'))
(the_enat (llength (ltl p'))))) ∧

```

```

    play_consistent_attacker s (LCons (lhd p') (ltl p')) (the (apply_w g (lhd p')
e)) ∧
    valid_play (LCons (lhd p') (ltl p')) ∧ energy_level (the (apply_w g (lhd p')
e)) (LCons (lhd p') (ltl p')) (the_enat (llength (ltl p')))) ≠ None"
  proof
    have "energy_level e (LCons g p') (the_enat (llength p')) ≠ None"
using Suc
    by blast
  hence "apply_w g (lhd p') e ≠ None"
  by (smt (verit) Suc.hyps(2) Suc.leI <x = the_enat (llength (ltl
p'))> energy_level.simps(1) energy_level_nth llist.distinct(1) llist.exhaust_sel
lnth_0 lnth_Suc_LCons ltl_simps(1) n_not_Suc_n option.sel zero_less_Suc)
  hence cons_assms: "valid_play (LCons g p') ∧ ¬ lnull p' ∧ apply_w
g (lhd p') e ≠ None ∧ enat (the_enat (llength (ltl p')))) < llength p'"
  using Suc
  by (metis <x = the_enat (llength (ltl p'))> enat_ord_simps(2) lessI
lfinite_conv_llength_enat lnull_def ltl_simps(1) n_not_Suc_n the_enat.simps)

  have "(the (energy_level e (LCons g p') (the_enat (llength p'))))
=
    (the (energy_level e (LCons g p') (Suc (the_enat (llength (ltl
p'))))))"
  using Suc.hyps(2) <x = the_enat (llength (ltl p'))> by auto
  also have "... = (the (energy_level (the (apply_w g (lhd p') e)) p'
(the_enat (llength (ltl p')))))"
  using energy_level_cons cons_assms by simp
  finally have EL: "(the (energy_level e (LCons g p') (the_enat (llength
p')))) =
    (the (energy_level (the (apply_w g (lhd p') e)) (LCons (lhd
p') (ltl p')) (the_enat (llength (ltl p')))))"
  by (simp add: cons_assms)
  thus "play_consistent_attacker s p
(the (energy_level (the (apply_w g (lhd p') e)) (LCons (lhd p') (ltl p'))
(the_enat (llength (ltl p')))))"
  using Suc by argo
  show "play_consistent_attacker s (LCons (lhd p') (ltl p')) (the (apply_w
g (lhd p') e)) ∧
    valid_play (LCons (lhd p') (ltl p')) ∧ energy_level (the (apply_w
g (lhd p') e)) (LCons (lhd p') (ltl p')) (the_enat (llength (ltl p')))) ≠ None"
  proof
    show "play_consistent_attacker s (LCons (lhd p') (ltl p')) (the
(apply_w g (lhd p') e))"
  using Suc
  by (metis cons_assms lhd_LCons lhd_LCons_ltl llist.distinct(1)
ltl_simps(2) play_consistent_attacker.simps)
  show "valid_play (LCons (lhd p') (ltl p')) ∧ energy_level (the (apply_w
g (lhd p') e)) (LCons (lhd p') (ltl p')) (the_enat (llength (ltl p')))) ≠ None"
  proof
    show "valid_play (LCons (lhd p') (ltl p'))" using Suc
  by (metis llist.distinct(1) llist.exhaust_sel llist.inject ltl_simps(1)
valid_play.simps)
  show "energy_level (the (apply_w g (lhd p') e)) (LCons (lhd p')
(ltl p')) (the_enat (llength (ltl p')))) ≠ None"
  using EL Suc
  by (metis <x = the_enat (llength (ltl p'))> cons_assms energy_level_cons
lhd_LCons_ltl)

```

```

      qed
    qed
  qed
  qed
  qed
  thus ?thesis using <x = the_enat (llength (ltl p'))> Suc
    by blast
  qed
  hence "play_consistent_attacker s Ps (the (apply_w g (lhd p') e))"
    using <(lappend (p') (ltl p)) = Ps>
    by (metis Suc.hyps(2) diff_0_eq_0 diff_Suc_1 lhd_LCons_ltl llength_lnull
n_not_Suc_n the_enat_0)
  then show ?case using play_consistent_attacker.simps <g ∈ attacker → s
e g = Some (lhd Ps)> <(lappend (LCons g p') (ltl p)) = LCons g Ps>
    by (metis (no_types, lifting) Suc.prems <¬ lfinite p> <lappend p' (ltl
p) = Ps> <lhd p' = lhd (lappend p' (ltl p))> energy_level.simps(1) lappend_code(1)
lhd_LCons llast_singleton llength_LNil llist.distinct(1) lnull_lappend ltl_simps(2)
option.sel the_enat_0)
  qed
  qed

  hence "¬defender_wins_play e (lappend (LCons g p') (ltl p))" using assms unfolding
attacker_winning_strategy.simps using P'
    by simp

  have "¬lfinite (lappend p' p)" using p_def by simp
  hence "defender_wins_play e (lappend (LCons g p') (ltl p))" using defender_wins_play_def
by auto
  thus "False" using <¬defender_wins_play e (lappend (LCons g p') (ltl p))> by
simp
  qed

```

We now show that an energy-positional attacker winning strategy w.r.t. some energy  $e$  and position  $g$  guarantees that  $e$  is in the attacker winning budget of  $g$ .

```

lemma winning_budget_implies_ind:
  assumes "winning_budget e g"
  shows "winning_budget_ind e g"
proof-
  define wb where "wb ≡ λ(g,e). winning_budget_ind e g"

  from assms have "∃s. attacker_winning_strategy s e g" using winning_budget.simps
by auto
  from this obtain s where S: "attacker_winning_strategy s e g" by auto
  hence "wfp_on (strategy_order s) (reachable_positions s g e)"
    using strategy_order_well_founded by simp
  hence "inductive_on (strategy_order s) (reachable_positions s g e)"
    by (simp add: wfp_on_iff_inductive_on)

  hence "wb (g,e)"
proof(rule inductive_on_induct)
  show "(g,e) ∈ reachable_positions s g e"
    unfolding reachable_positions_def proof
  have "lfinite LNil ∧
      llast (LCons g LNil) = g ∧
      valid_play (LCons g LNil) ∧ play_consistent_attacker s (LCons g LNil)
e ∧

```

```

    Some e = energy_level e (LCons g LNil) (the_enat (llength LNil))"
    using valid_play.simps play_consistent_attacker.simps energy_level.simps
    by (metis lfinite_code(1) llast_singleton llength_LNil neq_LNil_conv the_enat_0)

  thus "∃g' e'.
    (g, e) = (g', e') ∧
    (∃p. lfinite p ∧
      llast (LCons g p) = g' ∧
      valid_play (LCons g p) ∧ play_consistent_attacker s (LCons g p) e ∧
      Some e' = energy_level e (LCons g p) (the_enat (llength p))))"
  by (metis lfinite_code(1) llast_singleton llength_LNil the_enat_0)
qed

show "∧y. y ∈ reachable_positions s g e ⇒
  (∧x. x ∈ reachable_positions s g e ⇒ strategy_order s x y ⇒ wb x)
⇒ wb y"
proof-
  fix y
  assume "y ∈ reachable_positions s g e"
  hence "∃e' g'. y = (g', e')" using reachable_positions_def by auto
  from this obtain e' g' where "y = (g', e')" by auto

  hence "(∃p. lfinite p ∧ llast (LCons g p) = g'
    ∧ valid_play (LCons g p)
    ∧ play_consistent_attacker s
    (LCons g p) e
    ∧ (Some e' = energy_level e
    (LCons g p) (the_enat (llength p))))"
    using <y ∈ reachable_positions s g e> unfolding reachable_positions_def
    by auto
  from this obtain p where P: "(lfinite p ∧ llast (LCons g p) = g'
    ∧ valid_play (LCons g p)
    ∧ play_consistent_attacker s
    (LCons g p) e
    ∧ (Some e' = energy_level e
    (LCons g p) (the_enat (llength p))))" by auto

  show "(∧x. x ∈ reachable_positions s g e ⇒ strategy_order s x y ⇒ wb
x) ⇒ wb y"
  proof-
    assume ind: "(∧x. x ∈ reachable_positions s g e ⇒ strategy_order s x
y ⇒ wb x)"
    have "winning_budget_ind e' g'"
    proof(cases "g' ∈ attacker")
      case True
      then show ?thesis
      proof(cases "deadend g'")
        case True
        hence "attacker_stuck (LCons g p)" using <g' ∈ attacker> P
        by (meson S defender_wins_play_def attacker_winning_strategy.elims(2))

        hence "defender_wins_play e (LCons g p)" using defender_wins_play_def
        by simp
      case False
      hence "-defender_wins_play e (LCons g p)" using P S by simp
      then show ?thesis using <defender_wins_play e (LCons g p)> by simp
    next

```



```

      case False
      hence "(s e' g') ≠ None ∧ (weight g' (the (s e' g')))) ≠ None" using S
attacker_winning_strategy.simps
      by (simp add: True attacker_strategy_def)

    define x where "x = (the (s e' g'), the (apply_w g' (the (s e' g'))
e'))"

    define p' where "p' = (lappend p (LCons (the (s e' g')) LNil))"
    hence "lfinite p'" using P by simp
    have "llast (LCons g p') = the (s e' g')" using p'_def <lfinite p'>
      by (simp add: llast_LCons)

    have "the_enat (llength p') > 0" using P
      by (metis LNil_eq_lappend_iff <lfinite p'> bot_nat_0.not_eq_extremum
enat_0_iff(2) lfinite_conv_llength_enat llength_eq_0 llist.collapse(1) llist.distinct(1)
p'_def the_enat.simps)
    hence "∃i. Suc i = the_enat (llength p')"
      using less_iff_Suc_add by auto
    from this obtain i where "Suc i = the_enat (llength p'" by auto
    hence "i = the_enat (llength p)" using p'_def P
      by (metis Suc_leI <lfinite p'> length_append_singleton length_list_of_conv_the_e
less_Suc_eq_le less_irrefl_nat lfinite_LConsI lfinite_LNil list_of_LCons list_of_LNil
list_of_lappend not_less_less_Suc_eq)
    hence "Some e' = (energy_level e (LCons g p) i)" using P by simp

    have A: "lfinite (LCons g p) ∧ i < the_enat (llength (LCons g p)) ∧
energy_level e (LCons g p) (the_enat (llength (LCons g p)) - 1) ≠ None"
    proof
      show "lfinite (LCons g p)" using P by simp
      show "i < the_enat (llength (LCons g p)) ∧ energy_level e (LCons g
p) (the_enat (llength (LCons g p)) - 1) ≠ None"
    proof
      show "i < the_enat (llength (LCons g p))" using <i = the_enat (llength
p)> P
      by (metis <lfinite (LCons g p)> length_Cons length_list_of_conv_the_enat
lessI list_of_LCons)
      show "energy_level e (LCons g p) (the_enat (llength (LCons g p))
- 1) ≠ None" using P <i = the_enat (llength p)>
      using S defender_wins_play_def by auto
    qed
  qed

    hence "Some e' = (energy_level e (LCons g p') i)" using p'_def energy_level_append
P <Some e' = (energy_level e (LCons g p) i)>
      by (metis lappend_code(2))
    hence "energy_level e (LCons g p') i ≠ None"
      by (metis option.distinct(1))

    have "enat (Suc i) = llength p'" using <Suc i = the_enat (llength p')>
      by (metis <lfinite p'> lfinite_conv_llength_enat the_enat.simps)
    also have "... < eSuc (llength p'"
      by (metis calculation illess_Suc_eq order_refl)
    also have "... = llength (LCons g p'" using <lfinite p'> by simp
    finally have "enat (Suc i) < llength (LCons g p')".

    have "(lnth (LCons g p) i) = g'" using <i = the_enat (llength p)> P

```

```

      by (metis lfinite_conv_llength_enat llast_conv_lnth llength_LCons
the_enat.simps)
      hence "(lnth (LCons g p') i) = g'" using p'_def
      by (metis P <i = the_enat (llength p)> enat_ord_simps(2) energy_level.elims
lessI lfinite_llength_enat lnth_0 lnth_Suc_LCons lnth_lappend1 the_enat.simps)

      have "energy_level e (LCons g p') (the_enat (llength p')) = energy_level
e (LCons g p') (Suc i)"
      using <Suc i = the_enat (llength p')> by simp
      also have "... = apply_w (lnth (LCons g p') i) (lnth (LCons g p') (Suc
i)) (the (energy_level e (LCons g p') i))"
      using energy_level.simps <enat (Suc i) < llength (LCons g p')> <energy_level
e (LCons g p') i ≠ None>
      by (meson leD)
      also have "... = apply_w (lnth (LCons g p') i) (lnth (LCons g p') (Suc
i)) e'" using <Some e' = (energy_level e (LCons g p') i)>
      by (metis option.sel)
      also have "... = apply_w (lnth (LCons g p') i) (the (s e' g')) e'"
using p'_def <enat (Suc i) = llength p'>
      by (metis <eSuc (llength p') = llength (LCons g p')> <llast (LCons
g p') = the (s e' g')> llast_conv_lnth)
      also have "... = apply_w g' (the (s e' g')) e'" using <(lnth (LCons
g p') i) = g'> by simp
      finally have "energy_level e (LCons g p') (the_enat (llength p')) =
apply_w g' (the (s e' g')) e'" .

      have P': "lfinite p' ∧
      llast (LCons g p') = (the (s e' g')) ∧
      valid_play (LCons g p') ∧ play_consistent_attacker s (LCons g p') e
      ∧
      Some (the (apply_w g' (the (s e' g')) e')) = energy_level e (LCons g
p') (the_enat (llength p'))"
      proof
        show "lfinite p'" using p'_def P by simp
        show "llast (LCons g p') = the (s e' g') ∧
      valid_play (LCons g p') ∧
      play_consistent_attacker s (LCons g p') e ∧
      Some (the (apply_w g' (the (s e' g')) e')) = energy_level e (LCons g p') (the_enat
      (llength p'))"
      proof
        show "llast (LCons g p') = the (s e' g')" using p'_def <lfinite
p'>
      by (simp add: llast_LCons)
        show "valid_play (LCons g p') ∧
      play_consistent_attacker s (LCons g p') e ∧
      Some (the (apply_w g' (the (s e' g')) e')) = energy_level e (LCons g p') (the_enat
      (llength p'))"
      proof
        show "valid_play (LCons g p')" using p'_def P
      using <s e' g' ≠ None ∧ weight g' (the (s e' g')) ≠ None> valid_play.intr
valid_play_append by auto
        show "play_consistent_attacker s (LCons g p') e ∧
      Some (the (apply_w g' (the (s e' g')) e')) = energy_level e (LCons g p') (the_enat
      (llength p'))"
      proof
        have "(LCons g p') = lappend (LCons g p) (LCons (the (s e' g'))"

```

```

LNil)" using p'_def
      by simp
      have "play_consistent_attacker s (lappend (LCons g p) (LCons
(the (s e' g')) LNil)) e"
      proof (rule play_consistent_attacker_append_one)
        show "play_consistent_attacker s (LCons g p) e"
          using P by auto
        show "lfinite (LCons g p)" using P by auto
        show "energy_level e (LCons g p) (the_enat (llength (LCons
g p)) - 1) ≠ None" using P
          using A by auto
        show "valid_play (lappend (LCons g p) (LCons (the (s e' g'))
LNil))"
          using <valid_play (LCons g p') > <(LCons g p') = lappend
(LCons g p) (LCons (the (s e' g')) LNil)> by simp
          show "llast (LCons g p) ∈ attacker →
Some (the (s e' g')) =
s (the (energy_level e (LCons g p) (the_enat (llength (LCons g p)) - 1))) (llast
(LCons g p))"
          proof
            assume "llast (LCons g p) ∈ attacker"
            show "Some (the (s e' g')) =
s (the (energy_level e (LCons g p) (the_enat (llength (LCons g p)) - 1))) (llast
(LCons g p))"
              using <llast (LCons g p) ∈ attacker> P
              by (metis One_nat_def <s e' g' ≠ None ∧ weight g' (the
(s e' g')) ≠ None> diff_Suc_1' eSuc_enat lfinite_llength_enat llength_LCons option.collapse
option.sel the_enat.simps)
            qed
            qed
            thus "play_consistent_attacker s (LCons g p') e" using <(LCons
g p') = lappend (LCons g p) (LCons (the (s e' g')) LNil)> by simp
            show "Some (the (apply_w g' (the (s e' g')) e')) = energy_level
e (LCons g p') (the_enat (llength p'))"
              by (metis <eSuc (llength p') = llength (LCons g p')> <enat
(Suc i) = llength p'> <energy_level e (LCons g p') (the_enat (llength p')) = apply_w
g' (the (s e' g')) e'> <play_consistent_attacker s (LCons g p') e> <valid_play
(LCons g p')> S defender_wins_play_def diff_Suc_1 eSuc_enat option.collapse attacker_winning_st
the_enat.simps)
            qed
            qed
            qed
            hence "x ∈ reachable_positions s g e" using reachable_positions_def
x_def by auto
            have "(apply_w g' (the (s e' g')) e') ≠ None" using P'
            by (metis <energy_level e (LCons g p') (the_enat (llength p')) = apply_w
g' (the (s e' g')) e'> option.distinct(1))
            have "Some (the (apply_w g' (the (s e' g')) e')) = apply_w g' (the (s
e' g')) e' ∧ (if g' ∈ attacker then Some (the (s e' g')) = s e' g' else weight g'
(the (s e' g')) ≠ None)"
            using <(s e' g') ≠ None ∧ (weight g' (the (s e' g')) ≠ None)> <(apply_w
g' (the (s e' g')) e') ≠ None> by simp

```

```

      hence "strategy_order s x y" unfolding strategy_order_def using x_def
    <y = (g', e')>
      by blast
      hence "wb x" using ind <x ∈ reachable_positions s g e> by simp
      hence "winning_budget_ind (the (apply_w g' (the (s e' g')) e')) (the
(s e' g'))" using wb_def x_def by simp
      then show ?thesis using <g' ∈ attacker> winning_budget_ind.simps
      by (metis (mono_tags, lifting) <s e' g' ≠ None ∧ weight g' (the (s
e' g')) ≠ None> <strategy_order s x y> <y = (g', e')> old.prod.case option.distinct(1)
strategy_order_def x_def)
    qed
  next
  case False
  hence "g' ∉ attacker ∧
(∀g''. weight g' g'' ≠ None →
apply_w g' g'' e' ≠ None ∧ winning_budget_ind (the (apply_w g' g'' e'))
g'')"
  proof
    show "∀g''. weight g' g'' ≠ None →
apply_w g' g'' e' ≠ None ∧ winning_budget_ind (the (apply_w g' g'' e'))
g''"
    proof
      fix g''
      show "weight g' g'' ≠ None →
apply_w g' g'' e' ≠ None ∧ winning_budget_ind (the (apply_w g' g'' e'))
g'' "
      proof
        assume "weight g' g'' ≠ None"
        show "apply_w g' g'' e' ≠ None ∧ winning_budget_ind (the (apply_w
g' g'' e')) g''"
        proof
          show "apply_w g' g'' e' ≠ None"
          proof
            assume "apply_w g' g'' e' = None"
            define p' where "p' ≡ (LCons g (lappend p (LCons g'' LNil)))"
            hence "lfinite p'" using P by simp
            have "∃i. llength p = enat i" using P
            by (simp add: lfinite_llength_enat)
            from this obtain i where "llength p = enat i" by auto
            hence "llength (lappend p (LCons g'' LNil)) = enat (Suc i)"
            by (simp add: <llength p = enat i> eSuc_enat iadd_Suc_right)
            hence "llength p' = eSuc (enat (Suc i))" using p'_def
            by simp
            hence "the_enat (llength p') = Suc (Suc i)"
            by (simp add: eSuc_enat)
            hence "the_enat (llength p') - 1 = Suc i"
            by simp
            hence "the_enat (llength p') - 1 = the_enat (llength (lappend
p (LCons g'' LNil)))"
            using <llength (lappend p (LCons g'' LNil)) = enat (Suc i)>
            by simp
            have "(lnth p' i) = g'" using p'_def <llength p = enat i> P
            by (smt (verit) One_nat_def diff_Suc_1' enat_ord_simps(2)
energy_level.elims lessI llast_conv_lnth llength_LCons lnth_0 lnth_LCons' lnth_lappend
the_enat.simps)

```

```

      have "(lnth p' (Suc i)) = g'" using p'_def <llength p = enat
i>
      by (metis <llength p' = eSuc (enat (Suc i))> lappend.disc(2)
llast_LCons llast_conv_lnth llast_lappend_LCons llength_eq_enat_lfiniteD llist.disc(1)
llist.disc(2))
      have "p' = lappend (LCons g p) (LCons g'' LNil)" using p'_def
by simp
      hence "the (energy_level e p' i) = the (energy_level e (lappend
(LCons g p) (LCons g'' LNil)) i)" by simp
      also have "... = the (energy_level e (LCons g p) i)" using <llength
p = enat i> energy_level_append P
      by (metis diff_Suc_1 eSuc_enat lessI lfinite_LConsI llength_LCons
option.distinct(1) the_enat.simps)
      also have "... = e'" using P
      by (metis <llength p = enat i> option.sel the_enat.simps)

      finally have "the (energy_level e p' i) = e'" .
      hence "apply_w (lnth p' i) (lnth p' (Suc i)) (the (energy_level
e p' i)) = None" using <apply_w g' g'' e'=None> <(lnth p' i) = g'> <(lnth p' (Suc
i)) = g''> by simp

      have "energy_level e p' (the_enat (llength p')) - 1 =
          energy_level e p' (the_enat (llength (lappend p (LCons
g'' LNil))))"
      using <the_enat (llength p') - 1 = the_enat (llength (lappend
p (LCons g'' LNil)))>
      by simp
      also have "... = energy_level e p' (Suc i)" using <llength (lappend
p (LCons g'' LNil)) = enat (Suc i)> by simp
      also have "... = (if energy_level e p' i = None  $\vee$  llength p'
 $\leq$  enat (Suc i) then None
          else apply_w (lnth p' i) (lnth p' (Suc i))
(the (energy_level e p' i)))" using energy_level.simps by simp
      also have "... = None" using <apply_w (lnth p' i) (lnth p'
(Suc i)) (the (energy_level e p' i)) = None>
      by simp
      finally have "energy_level e p' (the_enat (llength p')) - 1)
= None" .
      hence "defender_wins_play e p'" unfolding defender_wins_play_def
by simp

      have "valid_play p'"
      by (metis P <p' = lappend (LCons g p) (LCons g'' LNil)> <weight
g' g''  $\neq$  None> energy_game.valid_play.intros(2) energy_game.valid_play_append lfinite_LConsI)

      have "play_consistent_attacker s (lappend (LCons g p) (LCons
g'' LNil)) e"
      proof(rule play_consistent_attacker_append_one)
      show "play_consistent_attacker s (LCons g p) e"
      using P by simp
      show "lfinite (LCons g p)" using P by simp
      show "energy_level e (LCons g p) (the_enat (llength (LCons
g p)) - 1)  $\neq$  None"
      using P
      by (meson S defender_wins_play_def attacker_winning_strategy.elims(2))

```

```

      show "valid_play (lappend (LCons g p) (LCons g'' LNil))"
      using <valid_play p'> <p' = lappend (LCons g p) (LCons
g'' LNil)> by simp
      show "llast (LCons g p) ∈ attacker →
      Some g'' =
      s (the (energy_level e (LCons g p) (the_enat (llength (LCons g p)) - 1))) (llast
(LCons g p))"
      using False P by simp
      qed
      hence "play_consistent_attacker s p' e"
      using <p' = lappend (LCons g p) (LCons g'' LNil)> by simp
      hence "¬defender_wins_play e p'" using <valid_play p'> p'_def
S by simp
      thus "False" using <defender_wins_play e p'> by simp

      qed

      define x where "x = (g'', the (apply_w g' g'' e'))"
      have "wb x"
      proof(rule ind)
      have "(∃p. lfinite p ∧
      llast (LCons g p) = g'' ∧
      valid_play (LCons g p) ∧ play_consistent_attacker s (LCons g p) e ∧
      Some (the (apply_w g' g'' e')) = energy_level e (LCons g p) (the_enat
      (llength p)))"
      proof
      define p' where "p' = lappend p (LCons g'' LNil)"
      show "lfinite p' ∧
      llast (LCons g p') = g'' ∧
      valid_play (LCons g p') ∧ play_consistent_attacker s (LCons g p') e ∧
      Some (the (apply_w g' g'' e')) = energy_level e (LCons g p') (the_enat (llength
      p'))"
      proof
      show "lfinite p'" using P p'_def by simp
      show "llast (LCons g p') = g'' ∧
      valid_play (LCons g p') ∧
      play_consistent_attacker s (LCons g p') e ∧
      Some (the (apply_w g' g'' e')) = energy_level e (LCons g p') (the_enat (llength
      p'))"
      proof
      show "llast (LCons g p') = g'" using p'_def
      by (metis <lfinite p'> lappend.disc_iff(2) lfinite_lappend
      llast_LCons llast_lappend_LCons llast_singleton llist.discI(2))
      show "valid_play (LCons g p') ∧
      play_consistent_attacker s (LCons g p') e ∧
      Some (the (apply_w g' g'' e')) = energy_level e (LCons g p') (the_enat (llength
      p'))"
      proof
      show "valid_play (LCons g p')" using p'_def P
      using <weight g' g'' ≠ None> lfinite_LCons valid_play.intros(2)
      valid_play_append by auto
      show "play_consistent_attacker s (LCons g p') e ∧
      Some (the (apply_w g' g'' e')) = energy_level e (LCons g p') (the_enat (llength
      p'))"
      proof

```

```

p) (LCons g'' LNil)) e"
    have "play_consistent_attacker s (lappend (LCons g
    proof(rule play_consistent_attacker_append_one)
      show "play_consistent_attacker s (LCons g p) e"

      using P by simp
      show "lfinite (LCons g p)" using P by simp
      show "energy_level e (LCons g p) (the_enat (llength
(LCons g p)) - 1) ≠ None"

      using P
      by (meson S defender_wins_play_def attacker_winning_strategy.
      show "valid_play (lappend (LCons g p) (LCons g''
LNil))"

      using <valid_play (LCons g p')> p'_def by simp
      show "llast (LCons g p) ∈ attacker →
        Some g'' =
          s (the (energy_level e (LCons g p) (the_enat
(llength (LCons g p)) - 1))) (llast (LCons g p))"
      using False P by simp
      qed
      thus "play_consistent_attacker s (LCons g p') e" using
p'_def

      by (simp add: lappend_code(2))

      have "∃i. Suc i = the_enat (llength p')" using p'_def

      by (metis P length_append_singleton length_list_of_conv_the_ena
lfinite_LConsI lfinite_LNil list_of_LCons list_of_LNil list_of_lappend)
      from this obtain i where "Suc i = the_enat (llength
p')" by auto

      hence "i = the_enat (llength p)" using p'_def
      by (smt (verit) One_nat_def <lfinite p'> add.commute
add_Suc_shift add_right_cancel length_append length_list_of_conv_the_enat lfinite_LNil
lfinite_lappend list.size(3) list.size(4) list_of_LCons list_of_LNil list_of_lappend
plus_1_eq_Suc)

      hence "Suc i = llength (LCons g p)"
      using P eSuc_enat lfinite_llength_enat by fastforce
      have "(LCons g p') = lappend (LCons g p) (LCons g''
LNil)" using p'_def by simp

      have A: "lfinite (LCons g p) ∧ i < the_enat (llength
(LCons g p)) ∧ energy_level e (LCons g p) (the_enat (llength (LCons g p)) - 1)
≠ None"

      proof
        show "lfinite (LCons g p)" using P by simp
        show "i < the_enat (llength (LCons g p)) ∧
          energy_level e (LCons g p) (the_enat (llength (LCons g p)) - 1) ≠ None"
          proof
            have "(llength p') = llength (LCons g p)" using
p'_def

            by (metis P <lfinite p'> length_Cons length_append_singleton
length_list_of lfinite_LConsI lfinite_LNil list_of_LCons list_of_LNil list_of_lappend)

            thus "i < the_enat (llength (LCons g p))" using
<Suc i = the_enat (llength p')>

            using lessI by force
            show "energy_level e (LCons g p) (the_enat (llength

```

```

(LCons g p)) - 1) ≠ None" using P
    by (meson S energy_game.defender_wins_play_def
energy_game.play_consistent_attacker.intros(2) attacker_winning_strategy.simps)
    qed
    qed
    hence "energy_level e (LCons g p') i ≠ None"
    using energy_level_append
    by (smt (verit) Nat.lessE Suc_leI <LCons g p' =
lappend (LCons g p) (LCons g'' LNil)> diff_Suc_1 energy_level_nth)
    have "enat (Suc i) < llength (LCons g p')"
    using <Suc i = the_enat (llength p')>
    by (metis Suc_ile_eq <lfinite p'> ldropn_Suc_LCons
leI lfinite_conv_llength_enat lnull_ldropn nless_le the_enat.simps)
    hence el_premis: "energy_level e (LCons g p') i ≠
None ∧ llength (LCons g p') > enat (Suc i)" using <energy_level e (LCons g p')
i ≠ None> by simp

    have "(lnth (LCons g p') i) = lnth (LCons g p) i"

    unfolding <(LCons g p') = lappend (LCons g p) (LCons
g'' LNil)> using <i = the_enat (llength p)> lnth_lappend1
    by (metis A enat_ord_simps(2) length_list_of length_list_of_con
have "lnth (LCons g p) i = llast (LCons g p)" using
<Suc i = llength (LCons g p)>
    by (metis enat_ord_simps(2) lappend_LNil2 ldropn_LNil
ldropn_Suc_conv_ldropn ldropn_lappend lessI less_not_refl llast_ldropn llast_singleton)
    hence "(lnth (LCons g p') i) = g'" using P
    by (simp add: <lnth (LCons g p') i = lnth (LCons
g p) i>)

    have "(lnth (LCons g p') (Suc i)) = g'"
    using p'_def <Suc i = the_enat (llength p')>
    by (smt (verit) <enat (Suc i) < llength (LCons g
p')> <lfinite p'> <llast (LCons g p') = g''> lappend_snocL1_conv_LCons2 ldropn_LNil
ldropn_Suc_LCons ldropn_Suc_conv_ldropn ldropn_lappend2 lfinite_llength_enat llast_ldropn
llast_singleton the_enat.simps wlog_linorder_le)

    have "energy_level e (LCons g p) i = energy_level
e (LCons g p') i"
    using energy_level_append A <(LCons g p') = lappend
(LCons g p) (LCons g'' LNil)>
    by presburger
    hence "Some e' = (energy_level e (LCons g p') i)"

    using P <i = the_enat (llength p)>
    by argo

    have "energy_level e (LCons g p') (the_enat (llength
p')) = energy_level e (LCons g p') (Suc i)" using <Suc i = the_enat (llength p')>
by simp
    also have "... = apply_w (lnth (LCons g p') i) (lnth
(LCons g p') (Suc i)) (the (energy_level e (LCons g p') i))"
    using energy_level.simps el_premis
    by (meson leD)
    also have "... = apply_w g' g'' (the (energy_level
e (LCons g p') i))"
    using <(lnth (LCons g p') i) = g'> <(lnth (LCons

```



```

g p') (Suc i)) = g'' > by simp
      finally have "energy_level e (LCons g p') (the_enat
(llengeth p')) = (apply_w g' g'' e')"
      using <Some e' = (energy_level e (LCons g p') i)>
      by (metis option.sel)
      thus "Some (the (apply_w g' g'' e')) = energy_level
e (LCons g p') (the_enat (llengeth p'))"
      by (simp add: <apply_w g' g'' e' ≠ None>)
      qed
      qed
      qed
      qed
      qed

      thus "x ∈ reachable_positions s g e"
      using x_def reachable_positions_def
      by (simp add: mem_Collect_eq)

      have "Some (the (apply_w g' g'' e')) = apply_w g' g'' e' ∧
(if g' ∈ attacker then Some g'' = s e' g' else weight g' g'' ≠ None)"
      proof
      show "Some (the (apply_w g' g'' e')) = apply_w g' g'' e'"
      by (simp add: <apply_w g' g'' e' ≠ None>)
      show "(if g' ∈ attacker then Some g'' = s e' g' else weight
g' g'' ≠ None)"
      using False
      by (simp add: <weight g' g'' ≠ None>)
      qed
      thus "strategy_order s x y" using strategy_order_def x_def <y
= (g', e')>
      by simp
      qed

      thus "winning_budget_ind (the (apply_w g' g'' e')) g'' " using
x_def wb_def
      by force
      qed
      qed
      qed
      qed
      thus ?thesis using winning_budget_ind.intros by blast
      qed
      thus "wb y" using <y = (g', e')> wb_def by simp
      qed
      qed
      thus ?thesis using wb_def by simp
      qed

```

We now prepare the proof of `winning_budget_ind` characterising subsets of `winning_budget_nonpos` for all positions. For this we introduce a construction to obtain a non-positional attacker winning strategy from a strategy at a next position.

```

fun nonpos_strat_from_next:: "'position ⇒ 'position ⇒
('position list ⇒ 'position option) ⇒ ('position list ⇒ 'position option)"

where

```

```

"nonpos_strat_from_next g g' s [] = s []" |
"nonpos_strat_from_next g g' s (x#xs) = (if x=g then (if xs=[] then Some g'
                                     else s xs) else s (x#xs))"

lemma play_nonpos_consistent_next:
  assumes "play_consistent_attacker_nonpos (nonpos_strat_from_next g g' s) (LCons
g (LCons g' xs)) []"
    and "g ∈ attacker" and "xs ≠ LNil"
  shows "play_consistent_attacker_nonpos s (LCons g' xs) []"
proof-
  have X: "∧l. l≠[] ⇒ (((nonpos_strat_from_next g g' s) ([g] @ l)) = s l)" using
nonpos_strat_from_next.simps by simp
  have A1: "∧s v l. play_consistent_attacker_nonpos (nonpos_strat_from_next g g'
s) (LCons v LNil) ([g]@l) ⇒ (l = [] ∨ (last l) ∉ attacker ∨ ((last l)∈attacker
∧ the (s l) = v))"
  proof-
    fix s v l
    assume "play_consistent_attacker_nonpos (nonpos_strat_from_next g g' s) (LCons
v LNil) ([g] @ l)"
    show "l = [] ∨ last l ∉ attacker ∨ last l ∈ attacker ∧ the (s l) = v"
    proof(cases "l=[]")
      case True
      then show ?thesis by simp
    next
      case False
      hence "l ≠ []" .
      then show ?thesis proof(cases "last l ∉ attacker")
        case True
        then show ?thesis by simp
      next
        case False
        hence "the ((nonpos_strat_from_next g g' s) ([g] @ l)) = v"
          by (smt (verit) <play_consistent_attacker_nonpos (nonpos_strat_from_next
g g' s) (LCons v LNil) ([g] @ l)> append_is_Nil_conv assms(2) eq_LConsD last.simps
last_append lhd_LCons list.distinct(1) llist.disc(1) play_consistent_attacker_nonpos.simps)
        hence "the (s l) = v" using X <l ≠ []> by auto
        then show ?thesis using False by simp
      qed
    qed
  qed

  have A2: "∧s v Ps l. play_consistent_attacker_nonpos (nonpos_strat_from_next
g g' s) (LCons v Ps) ([g]@l) ∧ Ps≠LNil ⇒ play_consistent_attacker_nonpos (nonpos_strat_from_
g g' s) Ps ([g]@(l@[v])) ∧ (v∈attacker → lhd Ps = the (s (l@[v])))"
  proof-
    fix s v Ps l
    assume play_cons: "play_consistent_attacker_nonpos (nonpos_strat_from_next g
g' s) (LCons v Ps) ([g]@l) ∧ Ps≠LNil"
    show "play_consistent_attacker_nonpos (nonpos_strat_from_next g g' s) Ps ([g]@(l@[v]))
∧ (v∈attacker → lhd Ps = the (s (l@[v])))"
    proof
      show "play_consistent_attacker_nonpos (nonpos_strat_from_next g g' s) Ps ([g]@(l@[v]))"
using play_cons play_consistent_attacker_nonpos.simps
      by (smt (verit) append_assoc lhd_LCons llist.distinct(1) ltl_simps(2))
      show "v ∈ attacker → lhd Ps = the (s (l @ [v]))"

```

```

    proof
      assume "v ∈ attacker"
      hence "lhd Ps = the ((nonpos_strat_from_next g g' s) ([g]@[1 @ [v]]))" using
play_cons play_consistent_attacker_nonpos.simps
      by (smt (verit) append_assoc lhd_LCons llist.distinct(1) ltl_simps(2))
      thus "lhd Ps = the (s (1 @ [v]))" using X by auto
    qed
  qed
qed

have "play_consistent_attacker_nonpos s xs [g']" proof (rule play_consistent_attacker_nonpos_
show "play_consistent_attacker_nonpos (nonpos_strat_from_next g g' s) xs ([g]@[g'])"
using assms(1)
  by (metis A2 append_Cons append_Nil assms(3) llist.distinct(1) play_consistent_attacker_n

show "∧s v l.
  play_consistent_attacker_nonpos (nonpos_strat_from_next g g' s) (LCons v
LNil) ([g] @ 1) ⇒
  l = [] ∨ last l ∉ attacker ∨ last l ∈ attacker ∧ the (s l) = v" using A1
by auto
show "∧s v Ps l.
  play_consistent_attacker_nonpos (nonpos_strat_from_next g g' s) (LCons v
Ps) ([g] @ 1) ∧ Ps ≠ LNil ⇒
  play_consistent_attacker_nonpos (nonpos_strat_from_next g g' s) Ps ([g] @
1 @ [v]) ∧ (v ∈ attacker → lhd Ps = the (s (1 @ [v])))" using A2 by auto
qed

thus ?thesis
  by (metis A2 append.left_neutral append_Cons assms(1) llist.distinct(1) lnull_def
play_consistent_attacker_nonpos_cons_simp)
qed

```

We now introduce a construction to obtain a non-positional attacker winning strategy from a strategy at a previous position.

```

fun nonpos_strat_from_previous:: "'position ⇒ 'position ⇒
  ('position list ⇒ 'position option) ⇒ ('position list ⇒ 'position option)"

where
  "nonpos_strat_from_previous g g' s [] = s []" |
  "nonpos_strat_from_previous g g' s (x#xs) = (if x=g' then s (g#(g'#xs))
    else s (x#xs))"

lemma play_nonpos_consistent_previous:
  assumes "play_consistent_attacker_nonpos (nonpos_strat_from_previous g g' s) p
  ([g']@l)"
  and "g∈attacker ⇒ g'=the (s [g])"
  shows "play_consistent_attacker_nonpos s p ([g,g']@l)"
proof (rule play_consistent_attacker_nonpos_coinduct)
  show "play_consistent_attacker_nonpos (nonpos_strat_from_previous g g' s) p (tl([g,g']@l))
  ∧ length ([g,g']@l) > 1 ∧ hd ([g,g']@l) = g ∧ hd (tl ([g,g']@l)) = g'" using assms(1)
by simp
  have X: "∧l. nonpos_strat_from_previous g g' s ([g']@l) = s ([g,g']@l)" using
nonpos_strat_from_previous.simps by simp
  have Y: "∧l. hd l ≠ g' ⇒ nonpos_strat_from_previous g g' s l = s l" using nonpos_strat_fro
  by (metis list.sel(1) neq_Nil_conv)
  show "∧s v l.

```

```

      play_consistent_attacker_nonpos (nonpos_strat_from_previous g g' s) (LCons
v LNil) (tl l) ∧ 1 < length l ∧ hd l = g ∧ hd (tl l) = g' ⇒
      l = [] ∨ last l ∉ attacker ∨ last l ∈ attacker ∧ the (s l) = v"
proof-
  fix s v l
  assume A: "play_consistent_attacker_nonpos (nonpos_strat_from_previous g g'
s) (LCons v LNil) (tl l) ∧ 1 < length l ∧ hd l = g ∧ hd (tl l) = g'"
  show "l = [] ∨ last l ∉ attacker ∨ last l ∈ attacker ∧ the (s l) = v"
  proof(cases "last l ∈ attacker")
    case True
    hence "last (tl l) ∈ attacker"
    by (metis A hd_Cons_tl last_tl less_Suc0 remdups_adj.simps(2) remdups_adj_singleton
remdups_adj_singleton_iff zero_neq_one)
    hence "the (nonpos_strat_from_previous g g' s (tl l)) = v" using play_consistent_attacker
A
    by (smt (verit) length_tl less_numeral_extra(3) list.size(3) llist.disc(1)
llist.distinct(1) llist.inject zero_less_diff)
    hence "the (s l) = v" using X A
    by (smt (verit, del_insts) One_nat_def hd_Cons_tl length_Cons less_numeral_extra(4)
list.inject list.size(3) not_one_less_zero nonpos_strat_from_previous.elims)
    then show ?thesis by simp
  next
  case False
  then show ?thesis by simp
qed
qed
show "∧s v Ps l.
(play_consistent_attacker_nonpos (nonpos_strat_from_previous g g' s) (LCons
v Ps) (tl l) ∧
  1 < length l ∧ hd l = g ∧ hd (tl l) = g') ∧
  Ps ≠ LNil ⇒
  (play_consistent_attacker_nonpos (nonpos_strat_from_previous g g' s) Ps (tl
(l @ [v])) ∧
  1 < length (l @ [v]) ∧ hd (l @ [v]) = g ∧ hd (tl (l @ [v])) = g') ∧
  (v ∈ attacker → lhd Ps = the (s (l @ [v]))))"
proof-
  fix s v Ps l
  assume A: "(play_consistent_attacker_nonpos (nonpos_strat_from_previous g g'
s) (LCons v Ps) (tl l) ∧
  1 < length l ∧ hd l = g ∧ hd (tl l) = g') ∧ Ps ≠ LNil"
  show "(play_consistent_attacker_nonpos (nonpos_strat_from_previous g g' s) Ps
(tl (l @ [v]))) ∧
  1 < length (l @ [v]) ∧ hd (l @ [v]) = g ∧ hd (tl (l @ [v])) = g') ∧
  (v ∈ attacker → lhd Ps = the (s (l @ [v]))))"
  proof
    show "play_consistent_attacker_nonpos (nonpos_strat_from_previous g g' s)
Ps (tl (l @ [v])) ∧
  1 < length (l @ [v]) ∧ hd (l @ [v]) = g ∧ hd (tl (l @ [v])) = g'"
    proof
      show "play_consistent_attacker_nonpos (nonpos_strat_from_previous g g' s)
Ps (tl (l @ [v]))" using A play_consistent_attacker_nonpos.simps
      by (smt (verit) lhd_LCons list.size(3) llist.distinct(1) ltl_simps(2)
not_one_less_zero tl_append2)
      show "1 < length (l @ [v]) ∧ hd (l @ [v]) = g ∧ hd (tl (l @ [v])) = g'"
      using A
      by (metis Suc_eq_plus1 add.comm_neutral add commute append_Nil hd_append2

```

```

length_append_singleton less_numeral_extra(4) list.exhaust_sel list.size(3) tl_append2
trans_less_add2)
  qed
  show "v ∈ attacker → lhd Ps = the (s (l @ [v]))"
  proof
    assume "v ∈ attacker"
    hence "lhd Ps = the ((nonpos_strat_from_previous g g' s) (tl (l @ [v])))"
using A play_consistent_attacker_nonpos.simps
    by (smt (verit) lhd_LCons list.size(3) llist.distinct(1) ltl_simps(2)
not_one_less_zero tl_append2)
    thus "lhd Ps = the (s (l @ [v]))" using X A
    by (smt (verit, ccfv_SIG) One_nat_def Suc_lessD <play_consistent_attacker_nonpos
(nonpos_strat_from_previous g g' s) Ps (tl (l @ [v])) ∧ 1 < length (l @ [v]) ∧ hd
(l @ [v]) = g ∧ hd (tl (l @ [v])) = g'> butlast.simps(2) butlast_snoc hd_Cons_tl
length_greater_0_conv list.inject nonpos_strat_from_previous.elims)

  qed
  qed
  qed
  qed

```

With these constructions we can show that the winning budgets defined by non-positional strategies are a fixed point of the inductive characterisation.

```

lemma nonpos_winning_budget_implies_inductive:
  assumes "nonpos_winning_budget e g"
  shows "g ∈ attacker ⇒ (∃ g'. (weight g g' ≠ None) ∧ (apply_w g g' e) ≠ None
    ∧ (nonpos_winning_budget (the (apply_w g g' e)) g'))" and
    "g ∉ attacker ⇒ (∀ g'. (weight g g' ≠ None) → (apply_w g g' e) ≠ None
    ∧ (nonpos_winning_budget (the (apply_w g g' e)) g'))"
proof-
  from assms obtain s where S: "nonpos_attacker_winning_strategy s e g" unfolding
nonpos_winning_budget.simps by auto
  show "g ∈ attacker ⇒ (∃ g'. (weight g g' ≠ None) ∧ (apply_w g g' e) ≠ None ∧
(nonpos_winning_budget (the (apply_w g g' e)) g'))"
  proof-
    assume "g ∈ attacker"
    have finite: "lfinite (LCons g LNil)" by simp
    have play_cons_g: "play_consistent_attacker_nonpos s (LCons g LNil) []"
      by (simp add: play_consistent_attacker_nonpos.intros(2))
    have valid_play_g: "valid_play (LCons g LNil)"
      by (simp add: valid_play.intros(2))
    hence "¬defender_wins_play e (LCons g LNil)" using nonpos_attacker_winning_strategy.simps
S play_cons_g by auto
    hence "¬deadend g" using finite defender_wins_play_def
      by (simp add: <g ∈ attacker>)
    hence "s [g] ≠ None" using nonpos_attacker_winning_strategy.simps attacker_nonpos_strategy.
S
      by (simp add: <g ∈ attacker>)
    show "(∃ g'. (weight g g' ≠ None) ∧ (apply_w g g' e) ≠ None ∧ (nonpos_winning_budget
(the (apply_w g g' e)) g'))"
    proof
      show "weight g (the (s [g])) ≠ None ∧ apply_w g (the (s [g])) e ≠ None ∧
nonpos_winning_budget (the (apply_w g (the (s [g])) e)) (the (s [g]))"
      proof
        show "weight g (the (s [g])) ≠ None" using nonpos_attacker_winning_strategy.simps
attacker_nonpos_strategy_def S <¬deadend g>

```

```

    using <g ∈ attacker> by (metis last_ConsL not_Cons_self2)
  show "apply_w g (the (s [g])) e ≠ None ∧
        nonpos_winning_budget (the (apply_w g (the (s [g])) e)) (the (s [g]))"

  proof
    show "apply_w g (the (s [g])) e ≠ None"
  proof-
    have finite: "lfinite (LCons g (LCons (the (s [g])) LNil))" by simp
    have play_cons_g': "play_consistent_attacker_nonpos s (LCons g (LCons
(the (s [g])) LNil)) []" using play_cons_g play_consistent_attacker_nonpos.intros
    by (metis append_Nil lhd_LCons llist.disc(2))
    have valid_play_g': "valid_play (LCons g (LCons (the (s [g])) LNil))"
using valid_play.intros valid_play_g
    using <weight g (the (s [g])) ≠ None> by auto
    hence "-defender_wins_play e (LCons g (LCons (the (s [g])) LNil))" using
nonpos_attacker_winning_strategy.simps S play_cons_g' by auto
    hence notNone: "energy_level e (LCons g (LCons (the (s [g])) LNil))
1 ≠ None" using finite defender_wins_play_def
    by (metis One_nat_def diff_Suc_1 length_Cons length_list_of_conv_the_enat
lfinite_LConsI lfinite_LNil list.size(3) list_of_LCons list_of_LNil)
    hence "energy_level e (LCons g (LCons (the (s [g])) LNil)) 1 = apply_w
(lnth (LCons g (LCons (the (s [g])) LNil)) 0)(lnth (LCons g (LCons (the (s [g]))
LNil)) 1) (the (energy_level e (LCons g (LCons (the (s [g])) LNil)) 0))"
    using energy_level.simps by (metis One_nat_def)
    hence "energy_level e (LCons g (LCons (the (s [g])) LNil)) 1 = apply_w
g (the (s [g])) e" by simp
    thus "apply_w g (the (s [g])) e ≠ None" using notNone by simp
  qed

  show "nonpos_winning_budget (the (apply_w g (the (s [g])) e)) (the (s
[g]))"
    unfolding nonpos_winning_budget.simps proof
    show "nonpos_attacker_winning_strategy (nonpos_strat_from_previous g
(the (s [g])) s) (the (apply_w g (the (s [g])) e)) (the (s [g]))"
    unfolding nonpos_attacker_winning_strategy.simps proof
    show "attacker_nonpos_strategy (nonpos_strat_from_previous g (the
(s [g])) s)" using S nonpos_strat_from_previous.simps
    by (smt (verit) nonpos_strat_from_previous.elims nonpos_attacker_winning_strate
attacker_nonpos_strategy_def last.simps list.distinct(1))
    show "∀p. play_consistent_attacker_nonpos (nonpos_strat_from_previous
g (the (s [g])) s) (LCons (the (s [g])) p) [] ∧
        valid_play (LCons (the (s [g])) p) →
        ¬ defender_wins_play (the (apply_w g (the (s [g])) e)) (LCons
(the (s [g])) p) "
    proof
      fix p
      show "play_consistent_attacker_nonpos (nonpos_strat_from_previous
g (the (s [g])) s) (LCons (the (s [g])) p) [] ∧
        valid_play (LCons (the (s [g])) p) →
        ¬ defender_wins_play (the (apply_w g (the (s [g])) e)) (LCons
(the (s [g])) p) "
    proof
      assume A: "play_consistent_attacker_nonpos (nonpos_strat_from_previous
g (the (s [g])) s) (LCons (the (s [g])) p) [] ∧
        valid_play (LCons (the (s [g])) p)"

```

```

    hence play_cons: "play_consistent_attacker_nonpos s (LCons g (LCons
(the (s [g])) p)) []"
    proof(cases "p = LNil")
    case True
    then show ?thesis using nonpos_strat_from_previous.simps play_consistent_at
    by (smt (verit) lhd_LCons llist.discI(2) self_append_conv2)

    next
    case False
    hence "play_consistent_attacker_nonpos (nonpos_strat_from_previous
g (the (s [g])) s) p [(the (s [g]))]" using A play_consistent_attacker_nonpos.cases
    using eq_Nil_appendI lhd_LCons by fastforce
    have "(the (s [g])) ∈ attacker ⇒ lhd p = the ((nonpos_strat_from_previous
g (the (s [g])) s) [(the (s [g]))])" using A play_consistent_attacker_nonpos.cases
    by (simp add: False play_consistent_attacker_nonpos_cons_simp)
    hence "(the (s [g])) ∈ attacker ⇒ lhd p = the (s [g],(the (s
[g]))])" using nonpos_strat_from_previous.simps by simp
    then show ?thesis using play_nonpos_consistent_previous
    by (smt (verit, del_insts) False <play_consistent_attacker_nonpos
(nonpos_strat_from_previous g (the (s [g])) s) p [the (s [g])]> append_Cons lhd_LCons
llist.collapse(1) play_consistent_attacker_nonpos.intros(5) play_consistent_attacker_nonpos.int
play_consistent_attacker_nonpos_cons_simp self_append_conv2)
    qed

    from A have "valid_play (LCons g (LCons (the (s [g])) p))"
    using <weight g (the (s [g])) ≠ None> valid_play.intros(3)
by auto
    hence not_won: "¬ defender_wins_play e (LCons g (LCons (the (s
[g])) p))" using S play_cons by simp
    hence "lfinite (LCons g (LCons (the (s [g])) p))" using defender_wins_play_def
by simp
    hence finite: "lfinite (LCons (the (s [g])) p)" by simp

    from not_won have no_deadend: "¬(llast (LCons (the (s [g])) p)
∈ attacker ∧ deadend (llast (LCons (the (s [g])) p)))"
    by (simp add: defender_wins_play_def)

    have suc: "Suc (the_enat (llength (LCons (the (s [g])) p)) - 1)
= (the_enat (llength (LCons g (LCons (the (s [g])) p))) - 1)" using finite
    by (smt (verit, ccfv_SIG) Suc_length_conv diff_Suc_1 length_list_of_conv_th
lfinite_LCons list_of_LCons)
    have "the_enat (llength (LCons (the (s [g])) p)) - 1 < the_enat
(llength (LCons (the (s [g])) p))" using finite
    by (metis (no_types, lifting) diff_less lfinite_llength_enat
llength_eq_0 llist.disc(2) not_less_less_Suc_eq the_enat.simps zero_enat_def zero_less_Suc
zero_less_one)
    hence cons_e_1:"valid_play (LCons g (LCons (the (s [g])) p))
∧ lfinite (LCons (the (s [g])) p) ∧ ¬ lnull (LCons (the (s [g])) p) ∧ apply_w
g (lhd (LCons (the (s [g])) p)) e ≠ None ∧ the_enat (llength (LCons (the (s [g]))
p)) - 1 < the_enat (llength (LCons (the (s [g])) p))"
    using <valid_play (LCons g (LCons (the (s [g])) p))> finite
<apply_w g (the (s [g])) e ≠ None> by simp

    from not_won have "energy_level e (LCons g (LCons (the (s [g]))
p)) (the_enat (llength (LCons g (LCons (the (s [g])) p))) - 1) ≠ None"
    by (simp add: defender_wins_play_def)

```

```

      hence "energy_level (the (apply_w g (the (s [g]))) e)) (LCons (the
(s [g])) p) (the_enat (llength (LCons (the (s [g])) p)) - 1) ≠ None"
      using energy_level_cons cons_e_1 suc
      by (metis enat_ord_simps(2) eq_LConsD length_list_of length_list_of_conv_th

      thus "¬ defender_wins_play (the (apply_w g (the (s [g]))) e)) (LCons
(the (s [g])) p) " using finite no_deadend defender_wins_play_def by simp
      qed
      qed
      qed
      qed
      qed
      qed
      qed
      qed
      show "g ∉ attacker ⇒ (∀ g'. (weight g g' ≠ None) → (apply_w g g' e) ≠ None
      ∧ (nonpos_winning_budget (the (apply_w g g' e)) g'))"
      proof-
      assume "g ∉ attacker"
      show "(∀ g'. (weight g g' ≠ None) → (apply_w g g' e) ≠ None ∧ (nonpos_winning_budget
(the (apply_w g g' e)) g'))"
      proof
      fix g'
      show "(weight g g' ≠ None) → (apply_w g g' e) ≠ None ∧ (nonpos_winning_budget
(the (apply_w g g' e)) g'"
      proof
      assume "(weight g g' ≠ None)"
      show "(apply_w g g' e) ≠ None ∧ (nonpos_winning_budget (the (apply_w g g'
e)) g'))"
      proof
      have "valid_play (LCons g (LCons g' LNil))" using <(weight g g' ≠ None)>
      by (simp add: valid_play.intros(2) valid_play.intros(3))
      have "play_consistent_attacker_nonpos s (LCons g' LNil) [g]" using play_consistent_at
      by (simp add: <g ∉ attacker>)
      hence "play_consistent_attacker_nonpos s (LCons g (LCons g' LNil)) []"
      using <g ∉ attacker> play_consistent_attacker_nonpos.intros(5) by simp
      hence "¬ defender_wins_play e (LCons g (LCons g' LNil))" using <valid_play
(LCons g (LCons g' LNil))> S by simp
      hence "energy_level e (LCons g (LCons g' LNil)) (the_enat (llength (LCons
g (LCons g' LNil))) - 1) ≠ None" using defender_wins_play_def by simp
      hence "energy_level e (LCons g (LCons g' LNil)) 1 ≠ None"
      by (metis One_nat_def diff_Suc_1 length_Cons length_list_of_conv_the_enat
lfinite_LConsI lfinite_LNil list.size(3) list_of_LCons list_of_LNil)
      thus "apply_w g g' e ≠ None" using energy_level.simps
      by (metis One_nat_def lnth_0 lnth_Suc_LCons option.sel)

      show "(nonpos_winning_budget (the (apply_w g g' e)) g'"
      unfolding nonpos_winning_budget.simps proof
      show "nonpos_attacker_winning_strategy (nonpos_strat_from_previous g
g' s) (the (apply_w g g' e)) g'"
      unfolding nonpos_attacker_winning_strategy.simps proof
      show "attacker_nonpos_strategy (nonpos_strat_from_previous g g' s)"
      using S
      by (smt (verit, del_insts) nonpos_strat_from_previous.elims nonpos_attacker_winning
attacker_nonpos_strategy_def last_ConsR list.distinct(1))
      show "∀ p. play_consistent_attacker_nonpos (nonpos_strat_from_previous

```



```

g g' s) (LCons g' p) [] ∧ valid_play (LCons g' p) →
  ¬ defender_wins_play (the (apply_w g g' e)) (LCons g' p)"
  proof
    fix p
    show "play_consistent_attacker_nonpos (nonpos_strat_from_previous
g g' s) (LCons g' p) [] ∧ valid_play (LCons g' p) →
  ¬ defender_wins_play (the (apply_w g g' e)) (LCons g' p) "
  proof
    assume A: "play_consistent_attacker_nonpos (nonpos_strat_from_previous
g g' s) (LCons g' p) [] ∧ valid_play (LCons g' p)"
    hence "valid_play (LCons g (LCons g' p))"
      using <weight g g' ≠ None> valid_play.intros(3) by auto

    from A have "play_consistent_attacker_nonpos (nonpos_strat_from_previous
g g' s) p [g']"
      using play_consistent_attacker_nonpos.intros(1) play_consistent_attacker_no
by auto
    hence "play_consistent_attacker_nonpos s p [g,g']" using play_nonpos_consiste
<g≠attacker>
      by fastforce
    hence "play_consistent_attacker_nonpos s (LCons g (LCons g' p))
[] "
      by (smt (verit) A Cons_eq_appendI <play_consistent_attacker_nonpos
s (LCons g (LCons g' LNil)) []> eq_Nil_appendI lhd_LCons llist.discI(2) llist.distinct(1)
ltl_simps(2) play_consistent_attacker_nonpos.simps nonpos_strat_from_previous.simps(2))
    hence not_won: "¬defender_wins_play e (LCons g (LCons g' p))"
using S <valid_play (LCons g (LCons g' p))> by simp
    hence finite: "lfinite (LCons g' p)"
      by (simp add: defender_wins_play_def)

    from not_won have no_deadend: "¬(llast (LCons g' p) ∈ attacker
∧ deadend (llast (LCons g' p)))" using defender_wins_play_def by simp

    have suc: "Suc ((the_enat (llength (LCons g' p)) - 1)) = (the_enat
(llength (LCons g (LCons g' p))) - 1)"
      using finite
      by (smt (verit, ccfv_SIG) Suc_length_conv diff_Suc_1 length_list_of_conv_th
lfinite_LCons list_of_LCons)
    from not_won have "energy_level e (LCons g (LCons g' p)) (the_enat
(llength (LCons g (LCons g' p))) - 1) ≠ None" using defender_wins_play_def by simp
    hence "energy_level (the (apply_w g g' e)) (LCons g' p) (the_enat
(llength (LCons g' p)) - 1) ≠ None"
      using suc energy_level_cons
      by (smt (verit, best) One_nat_def Suc_diff_Suc Suc_lessD <apply_w
g g' e ≠ None> <valid_play (LCons g (LCons g' p))> diff_zero enat_ord_simps(2)
energy_level.elims lessI lfinite_conv_llength_enat lhd_LCons llist.discI(2) llist.distinct(1)
local.finite option.collapse the_enat.simps zero_less_Suc zero_less_diff)
    thus "¬ defender_wins_play (the (apply_w g g' e)) (LCons g'
p)" using defender_wins_play_def finite no_deadend by simp
  qed
qed
qed
qed
qed
qed

```

```

qed
qed
qed

lemma inductive_implies_nonpos_winning_budget:
  shows "g ∈ attacker ⇒ (∃g'. (weight g g' ≠ None) ∧ (apply_w g g' e) ≠ None
    ∧ (nonpos_winning_budget (the (apply_w g g' e)) g'))
    ⇒ nonpos_winning_budget e g"
  and "g ∉ attacker ⇒ (∀g'. (weight g g' ≠ None)
    ⇒ (apply_w g g' e) ≠ None
    ∧ (nonpos_winning_budget (the (apply_w g g' e)) g'))
    ⇒ nonpos_winning_budget e g"

proof-
  assume "g ∈ attacker"
  assume "(∃g'. (weight g g' ≠ None) ∧ (apply_w g g' e) ≠ None ∧ (nonpos_winning_budget
    (the (apply_w g g' e)) g'))"

  from this obtain g' where A1: "(weight g g' ≠ None) ∧ (apply_w g g' e) ≠ None
    ∧ (nonpos_winning_budget (the (apply_w g g' e)) g'" by auto
  hence "∃s. nonpos_attacker_winning_strategy s (the (apply_w g g' e)) g'" using
    nonpos_winning_budget.simps by auto
  from this obtain s where s_winning: "nonpos_attacker_winning_strategy s (the (apply_w
    g g' e)) g'" by auto
  have "nonpos_attacker_winning_strategy (nonpos_strat_from_next g g' s) e g" unfolding
    nonpos_attacker_winning_strategy.simps
  proof
    show "attacker_nonpos_strategy (nonpos_strat_from_next g g' s)"
      unfolding attacker_nonpos_strategy_def proof
        fix list
        show "list ≠ [] ⇒
          last list ∈ attacker ∧ ¬ deadend (last list) ⇒
          nonpos_strat_from_next g g' s list ≠ None ∧ weight (last list) (the (nonpos_strat_from_
            g g' s list)) ≠ None"
        proof
          assume "list ≠ []"
          show "last list ∈ attacker ∧ ¬ deadend (last list) ⇒
            nonpos_strat_from_next g g' s list ≠ None ∧ weight (last list) (the
              (nonpos_strat_from_next g g' s list)) ≠ None"
          proof
            assume "last list ∈ attacker ∧ ¬ deadend (last list)"
            show "nonpos_strat_from_next g g' s list ≠ None ∧ weight (last list)
              (the (nonpos_strat_from_next g g' s list)) ≠ None"
            proof
              from s_winning have "attacker_nonpos_strategy s" by auto
              thus "nonpos_strat_from_next g g' s list ≠ None" using nonpos_strat_from_next.simps
                <list ≠ []> <last list ∈ attacker ∧ ¬ deadend (last list)>
              by (smt (verit) nonpos_strat_from_next.elims attacker_nonpos_strategy_def
                last_ConsR option.discI)
              show "weight (last list) (the (nonpos_strat_from_next g g' s list))
                ≠ None" using nonpos_strat_from_next.simps(2) <list ≠ []> <last list ∈ attacker
                ∧ ¬ deadend (last list)>
              by (smt (verit) A1 <attacker_nonpos_strategy s> nonpos_strat_from_next.elims
                attacker_nonpos_strategy_def last_ConsL last_ConsR option.sel)
            qed
          qed
        qed
      qed
    qed
  qed

```

```

qed
show "∀p. play_consistent_attacker_nonpos (nonpos_strat_from_next g g' s) (LCons
g p) [] ∧ valid_play (LCons g p) →
  ¬ defender_wins_play e (LCons g p) "
proof
  fix p
  show "play_consistent_attacker_nonpos (nonpos_strat_from_next g g' s) (LCons
g p) [] ∧ valid_play (LCons g p) →
  ¬ defender_wins_play e (LCons g p)"
  proof
    assume A: "play_consistent_attacker_nonpos (nonpos_strat_from_next g g'
s) (LCons g p) [] ∧ valid_play (LCons g p)"
    hence "play_consistent_attacker_nonpos s p []"
    proof(cases "p=LNil")
      case True
      then show ?thesis
      by (simp add: play_consistent_attacker_nonpos.intros(1))
    next
      case False
      hence "∃v p'. p=LCons v p'"
      by (meson llist.exhaust)
      from this obtain v p' where "p= LCons v p'" by auto
      then show ?thesis
      proof(cases "p'=LNil")
        case True
        then show ?thesis
        by (simp add: <p = LCons v p'> play_consistent_attacker_nonpos.intros(2))
      next
        case False
        from <p= LCons v p'> have "v=g'" using A nonpos_strat_from_next.simps
play_nonpos_consistent_previous <g ∈ attacker>
        by (simp add: play_consistent_attacker_nonpos_cons_simp)
        then show ?thesis using <p= LCons v p'> A nonpos_strat_from_next.simps
play_nonpos_consistent_next
        using False <g ∈ attacker> by blast
      qed
    qed

    have "valid_play p" using A valid_play.simps
    by (metis eq_LConsD)
    hence notNil: "p≠LNil ⇒ ¬ defender_wins_play (the (apply_w g g' e)) p"
    using s_winning <play_consistent_attacker_nonpos s p []> nonpos_attacker_winning_strategy.elim
    by (metis A <g ∈ attacker> lhd_LCons not_lnull_conv option.sel play_consistent_attac
nonpos_strat_from_next.simps(2))
    show "¬ defender_wins_play e (LCons g p)"
    proof(cases "p=LNil")
      case True
      hence "lfinite (LCons g p)" by simp
      have "llast (LCons g p) = g" using True by simp
      hence not_deadend: "¬ deadend (llast (LCons g p))" using A1 by auto
      have "energy_level e (LCons g p) (the_enat (llength (LCons g p)) - 1)
≠ None" using True
      by (simp add: gen_llength_code(1) gen_llength_code(2) llength_code)
      then show ?thesis using defender_wins_play_def not_deadend <lfinite (LCons
g p)> by simp

```

```

    next
    case False
    hence "¬ defender_wins_play (the (apply_w g g' e)) p" using notNil by
simp
    hence not: "lfinite p ∧ energy_level (the (apply_w g g' e)) p (the_enat
(1length p) - 1) ≠ None ∧ ¬(l1ast p ∈ attacker ∧ deadend (l1ast p))" using defender_wins_play
    by simp
    hence "lfinite (LCons g p)" by simp

    from False have "l1ast (LCons g p) = l1ast p"
    by (meson l1ast_LCons llist.collapse(1))
    hence "¬(l1ast (LCons g p) ∈ attacker ∧ deadend (l1ast (LCons g p)))"
using not by simp

    from <lfinite (LCons g p)> have "the_enat (1length (LCons g p)) = Suc
(the_enat (1length p))"
    by (metis eSuc_enat lfinite_LCons lfinite_conv_1length_enat 1length_LCons
the_enat.simps)
    hence E: "(the_enat (1length (LCons g p)) - 1) = Suc (the_enat (1length
p) - 1)" using <lfinite (LCons g p)> False
    by (metis diff_Suc_1 diff_self_eq_0 gr0_implies_Suc i0_less less_enatE
less_imp_diff_less lfinite_1length_enat 1length_eq_0 llist.collapse(1) not the_enat.simps)

    from False have "lhd p = g'" using A nonpos_strat_from_next.simps play_nonpos_consist
<g∈attacker>
    by (simp add: play_consistent_attacker_nonpos_cons_simp)
    hence "energy_level e (LCons g p) (the_enat (1length (LCons g p)) - 1)
= energy_level (the (apply_w g g' e)) p (the_enat (1length p) - 1)"
    using energy_level_cons A not False A1 E
    by (metis <the_enat (1length (LCons g p)) = Suc (the_enat (1length p))>
diff_Suc_1 enat_ord_simps(2) lessI lfinite_conv_1length_enat play_consistent_attacker_nonpos_co
the_enat.simps)
    hence "energy_level e (LCons g p) (the_enat (1length (LCons g p)) - 1)
≠ None" using not by auto
    then show ?thesis using defender_wins_play_def <lfinite (LCons g p)> <¬(l1ast
(LCons g p) ∈ attacker ∧ deadend (l1ast (LCons g p)))> by auto
    qed
    qed
    qed
    qed
    thus "nonpos_winning_budget e g" using nonpos_winning_budget.simps by auto
next
    assume "g ∉ attacker"
    assume all: "(∀g'. (weight g g' ≠ None) → (apply_w g g' e) ≠ None ∧ (nonpos_winning_budget
(the (apply_w g g' e)) g'))"

    have valid: "attacker_nonpos_strategy (λlist. (case list of
    [] ⇒ None |
    [x] ⇒ (if x ∈ attacker ∧ ¬deadend x then Some (SOME y. weight x
y ≠ None) else None) |
    (x#(g'#xs)) ⇒ (if (x=g ∧ weight x g' ≠ None) then ((SOME s. nonpos_attacker_winn
s (the (apply_w g g' e)) g' ) (g'#xs))
    else (if (last (x#(g'#xs))) ∈ attacker ∧ ¬deadend
(last (x#(g'#xs))) then Some (SOME y. weight (last (x#(g'#xs))) y ≠ None) else None))))"
    unfolding attacker_nonpos_strategy_def proof

```

```

fix list
show "list  $\neq$  []  $\longrightarrow$ 
  last list  $\in$  attacker  $\wedge \neg$  deadend (last list)  $\longrightarrow$ 
  (case list of []  $\Rightarrow$  None | [x]  $\Rightarrow$  if x  $\in$  attacker  $\wedge \neg$  deadend x then Some
(SOME y. weight x y  $\neq$  None) else None
  | x # g' # xs  $\Rightarrow$ 
    if (x=g  $\wedge$  weight x g'  $\neq$  None) then (SOME s. nonpos_attacker_winning_strategy
s (the (apply_w g g' e)) g') (g'#xs)
    else if last (x # g' # xs)  $\in$  attacker  $\wedge \neg$  deadend (last (x # g' # xs))
      then Some (SOME y. weight (last (x # g' # xs)) y  $\neq$  None) else None)
 $\neq$ 
  None  $\wedge$ 
  weight (last list)
  (the (case list of []  $\Rightarrow$  None | [x]  $\Rightarrow$  if x  $\in$  attacker  $\wedge \neg$  deadend x then
Some (SOME y. weight x y  $\neq$  None) else None
    | x # g' # xs  $\Rightarrow$ 
      if (x=g  $\wedge$  weight x g'  $\neq$  None) then (SOME s. nonpos_attacker_winning_strategy
s (the (apply_w g g' e)) g') (g'#xs)
      else if last (x # g' # xs)  $\in$  attacker  $\wedge \neg$  deadend (last (x #
g' # xs))
        then Some (SOME y. weight (last (x # g' # xs)) y  $\neq$  None)
else None))  $\neq$ 
  None"

proof
  assume "list  $\neq$  []"
  show "last list  $\in$  attacker  $\wedge \neg$  deadend (last list)  $\longrightarrow$ 
    (case list of []  $\Rightarrow$  None | [x]  $\Rightarrow$  if x  $\in$  attacker  $\wedge \neg$  deadend x then Some (SOME
y. weight x y  $\neq$  None) else None
    | x # g' # xs  $\Rightarrow$ 
      if (x=g  $\wedge$  weight x g'  $\neq$  None) then (SOME s. nonpos_attacker_winning_strategy
s (the (apply_w g g' e)) g') (g'#xs)
      else if last (x # g' # xs)  $\in$  attacker  $\wedge \neg$  deadend (last (x # g' # xs))
        then Some (SOME y. weight (last (x # g' # xs)) y  $\neq$  None) else None)
 $\neq$ 
    None  $\wedge$ 
    weight (last list)
    (the (case list of []  $\Rightarrow$  None | [x]  $\Rightarrow$  if x  $\in$  attacker  $\wedge \neg$  deadend x then
Some (SOME y. weight x y  $\neq$  None) else None
      | x # g' # xs  $\Rightarrow$ 
        if (x=g  $\wedge$  weight x g'  $\neq$  None) then (SOME s. nonpos_attacker_winning_strategy
s (the (apply_w g g' e)) g') (g'#xs)
        else if last (x # g' # xs)  $\in$  attacker  $\wedge \neg$  deadend (last (x # g'
# xs))
          then Some (SOME y. weight (last (x # g' # xs)) y  $\neq$  None) else
None))  $\neq$ 
    None"

  proof
    assume "last list  $\in$  attacker  $\wedge \neg$  deadend (last list)"
    show "(case list of []  $\Rightarrow$  None | [x]  $\Rightarrow$  if x  $\in$  attacker  $\wedge \neg$  deadend x then
Some (SOME y. weight x y  $\neq$  None) else None
      | x # g' # xs  $\Rightarrow$ 
        if (x=g  $\wedge$  weight x g'  $\neq$  None) then (SOME s. nonpos_attacker_winning_strategy
s (the (apply_w g g' e)) g') (g'#xs)
        else if last (x # g' # xs)  $\in$  attacker  $\wedge \neg$  deadend (last (x # g' # xs))
          then Some (SOME y. weight (last (x # g' # xs)) y  $\neq$  None) else None)
 $\neq$ 
    None"

```

```

None ∧
weight (last list)
(the (case list of [] ⇒ None | [x] ⇒ if x ∈ attacker ∧ ¬ deadend x then
Some (SOME y. weight x y ≠ None) else None
| x # g' # xs ⇒
if (x=g ∧ weight x g' ≠ None) then (SOME s. nonpos_attacker_winning_strategy
s (the (apply_w g g' e)) g') (g'#xs)
else if last (x # g' # xs) ∈ attacker ∧ ¬ deadend (last (x # g'
# xs))
then Some (SOME y. weight (last (x # g' # xs)) y ≠ None) else
None)) ≠
None"
proof
show "(case list of [] ⇒ None |
[x] ⇒ if x ∈ attacker ∧ ¬ deadend x then Some (SOME y. weight
x y ≠ None) else None
| x # g' # xs ⇒
if (x=g ∧ weight x g' ≠ None) then (SOME s. nonpos_attacker_winning_strategy
s (the (apply_w g g' e)) g') (g'#xs)
else if last (x # g' # xs) ∈ attacker ∧ ¬ deadend (last (x # g'
# xs))
then Some (SOME y. weight (last (x # g' # xs)) y ≠ None) else
None) ≠ None"
proof(cases "length list = 1")
case True
then show ?thesis
by (smt (verit) One_nat_def <last list ∈ attacker ∧ ¬ deadend (last
list)> append_butlast_last_id append_eq_Cons_conv butlast_snoc length_0_conv length_Suc_conv_re
list.simps(4) list.simps(5) option.discI)
next
case False
hence "∃x y xs. list = x # (y # xs)"
by (metis One_nat_def <list ≠ []> length_Cons list.exhaust list.size(3))
from this obtain x y xs where "list = x # (y # xs)" by auto
then show ?thesis proof(cases "(x=g ∧ weight x y ≠ None)")
case True
hence A: "(case list of [] ⇒ None |
[x] ⇒ if x ∈ attacker ∧ ¬ deadend x then Some (SOME y. weight
x y ≠ None) else None
| x # g' # xs ⇒
if (x=g ∧ weight x g' ≠ None) then (SOME s. nonpos_attacker_winning_strategy
s (the (apply_w g g' e)) g') (g'#xs)
else if last (x # g' # xs) ∈ attacker ∧ ¬ deadend (last (x # g'
# xs))
then Some (SOME y. weight (last (x # g' # xs)) y ≠ None) else
None) = (SOME s. nonpos_attacker_winning_strategy s (the (apply_w g y e)) y) (y#xs)"
using <list = x # y # xs> list.simps(5) by fastforce

from all True have "∃s. nonpos_attacker_winning_strategy s (the (apply_w
g y e)) y" by auto
hence "nonpos_attacker_winning_strategy (SOME s. nonpos_attacker_winning_strategy
s (the (apply_w g y e)) y) (the (apply_w g y e)) y"
using some_eq_ex by metis
hence "attacker_nonpos_strategy (SOME s. nonpos_attacker_winning_strategy
s (the (apply_w g y e)) y)"
by (meson nonpos_attacker_winning_strategy.simps)

```

```

      hence "(SOME s. nonpos_attacker_winning_strategy s (the (apply_w g
y e)) y) (y#xs) ≠ None"
      using <last list ∈ attacker ∧ ¬ deadend (last list)> <list = x
# (y # xs)>
      by (simp add: list.distinct(1) attacker_nonpos_strategy_def)

    then show ?thesis using A by simp
  next
    case False
    hence "(case list of [] ⇒ None |
[x] ⇒ if x ∈ attacker ∧ ¬ deadend x then Some (SOME y. weight
x y ≠ None) else None
| x # g' # xs ⇒
if (x=g ∧ weight x g' ≠ None) then (SOME s. nonpos_attacker_winning_strategy
s (the (apply_w g g' e)) g') (g'#xs)
else if last (x # g' # xs) ∈ attacker ∧ ¬ deadend (last (x # g'
# xs))
then Some (SOME y. weight (last (x # g' # xs)) y ≠ None) else
None) =
Some (SOME z. weight (last (x # y # xs)) z ≠ None)"
    using <last list ∈ attacker ∧ ¬ deadend (last list)> <list = x
# y # xs> by auto
    then show ?thesis by simp
  qed
qed

  show "weight (last list)
(the (case list of [] ⇒ None | [x] ⇒ if x ∈ attacker ∧ ¬ deadend
x then Some (SOME y. weight x y ≠ None) else None
| x # g' # xs ⇒
if (x=g ∧ weight x g' ≠ None) then (SOME s. nonpos_attacker_winning_strategy
s (the (apply_w g g' e)) g') (g'#xs)
else if last (x # g' # xs) ∈ attacker ∧ ¬ deadend (last
(x # g' # xs))
then Some (SOME y. weight (last (x # g' # xs)) y ≠ None)
else None)) ≠ None"
  proof(cases "length list =1")
    case True
    hence "the (case list of [] ⇒ None | [x] ⇒ if x ∈ attacker ∧ ¬ deadend
x then Some (SOME y. weight x y ≠ None) else None
| x # g' # xs ⇒
if (x=g ∧ weight x g' ≠ None) then (SOME s. nonpos_attacker_winning_strategy
s (the (apply_w g g' e)) g') (g'#xs)
else if last (x # g' # xs) ∈ attacker ∧ ¬ deadend (last (x # g'
# xs))
then Some (SOME y. weight (last (x # g' # xs)) y ≠ None) else
None) = (SOME y. weight (last list) y ≠ None)"
    using <last list ∈ attacker ∧ ¬ deadend (last list)>
    by (smt (verit) Eps_cong One_nat_def <(case list of [] ⇒ None | [x]
⇒ if x ∈ attacker ∧ ¬ deadend x then Some (SOME y. weight x y ≠ None) else None
| x # g' # xs ⇒ if (x=g ∧ weight x g' ≠ None) then (SOME s. nonpos_attacker_winning_strategy
s (the (apply_w g g' e)) g') (g' # xs) else if last (x # g' # xs) ∈ attacker ∧ ¬
deadend (last (x # g' # xs)) then Some (SOME y. weight (last (x # g' # xs)) y ≠
None) else None) ≠ None> last_snoc length_0_conv length_Suc_conv_rev list.case_eq_if
list.sel(1) list.sel(3) option.sel self_append_conv2)
    then show ?thesis

```

```

      by (smt (verit, del_insts) <last list ∈ attacker ∧ ¬ deadend (last
list)> some_eq_ex)
    next
      case False
      hence "∃ x y xs. list = x # (y # xs)"
      by (metis One_nat_def <list ≠ []> length_Cons list.exhaust list.size(3))
      from this obtain x y xs where "list = x # (y # xs)" by auto
      then show ?thesis proof(cases "(x=g ∧ weight x y ≠ None)")
        case True
        hence "(case list of [] ⇒ None |
[x] ⇒ if x ∈ attacker ∧ ¬ deadend x then Some (SOME y. weight
x y ≠ None) else None
| x # g' # xs ⇒
if (x=g ∧ weight x g' ≠ None) then (SOME s. nonpos_attacker_winning_strategy
s (the (apply_w g g' e)) g') (g'#xs)
else if last (x # g' # xs) ∈ attacker ∧ ¬ deadend (last (x # g'
# xs))
then Some (SOME y. weight (last (x # g' # xs)) y ≠ None) else
None) = (SOME s. nonpos_attacker_winning_strategy s (the (apply_w g y e)) y) (y#xs)"
        using <list = x # y # xs> list.simps(5) by fastforce

        from all True have "∃ s. nonpos_attacker_winning_strategy s (the (apply_w
g y e)) y" by auto
        hence "nonpos_attacker_winning_strategy (SOME s. nonpos_attacker_winning_strategy
s (the (apply_w g y e)) y) (the (apply_w g y e)) y"
        using some_eq_ex by metis
        then show ?thesis
        by (smt (verit) <(case list of [] ⇒ None | [x] ⇒ if x ∈ attacker
∧ ¬ deadend x then Some (SOME y. weight x y ≠ None) else None | x # g' # xs ⇒
if x = g ∧ weight x g' ≠ None then (SOME s. nonpos_attacker_winning_strategy s
(the (apply_w g g' e)) g') (g' # xs) else if last (x # g' # xs) ∈ attacker ∧ ¬
deadend (last (x # g' # xs)) then Some (SOME y. weight (last (x # g' # xs)) y ≠
None) else None) = (SOME s. nonpos_attacker_winning_strategy s (the (apply_w g y
e)) y) (y # xs)> <last list ∈ attacker ∧ ¬ deadend (last list)> <list = x # y
# xs> attacker_nonpos_strategy_def nonpos_attacker_winning_strategy.elims(1) last_ConsR
list.distinct(1))
        next
          case False
          hence "(case list of [] ⇒ None |
[x] ⇒ if x ∈ attacker ∧ ¬ deadend x then Some (SOME y. weight
x y ≠ None) else None
| x # g' # xs ⇒
if (x=g ∧ weight x g' ≠ None) then (SOME s. nonpos_attacker_winning_strategy
s (the (apply_w g g' e)) g') (g'#xs)
else if last (x # g' # xs) ∈ attacker ∧ ¬ deadend (last (x # g'
# xs))
then Some (SOME y. weight (last (x # g' # xs)) y ≠ None) else
None) =
Some (SOME z. weight (last (x # y # xs)) z ≠ None)"
          using <last list ∈ attacker ∧ ¬ deadend (last list)> <list = x
# y # xs> by auto
          then show ?thesis
          by (smt (verit, del_insts) <last list ∈ attacker ∧ ¬ deadend (last
list)> <list = x # y # xs> option.sel verit_sko_ex_indirect)
        qed
      qed

```



qed  
qed  
qed  
qed

```

have winning: "(∀p. (play_consistent_attacker_nonpos (λlist. (case list of []
⇒ None |
      [x] ⇒ if x ∈ attacker ∧ ¬ deadend x then Some (SOME y. weight
x y ≠ None) else None
      | x # g' # xs ⇒
        if (x=g ∧ weight x g' ≠ None) then (SOME s. nonpos_attacker_winning_strategy
s (the (apply_w g g' e)) g') (g'#xs)
        else if last (x # g' # xs) ∈ attacker ∧ ¬ deadend (last (x # g'
# xs))
          then Some (SOME y. weight (last (x # g' # xs)) y ≠ None) else
None)) (LCons g p) []
      ∧ valid_play (LCons g p)) → ¬ defender_wins_play e (LCons g p))"

proof
  fix p
  show "(play_consistent_attacker_nonpos (λlist. (case list of [] ⇒ None |
      [x] ⇒ if x ∈ attacker ∧ ¬ deadend x then Some (SOME y. weight
x y ≠ None) else None
      | x # g' # xs ⇒
        if (x=g ∧ weight x g' ≠ None) then (SOME s. nonpos_attacker_winning_strategy
s (the (apply_w g g' e)) g') (g'#xs)
        else if last (x # g' # xs) ∈ attacker ∧ ¬ deadend (last (x # g'
# xs))
          then Some (SOME y. weight (last (x # g' # xs)) y ≠ None) else
None)) (LCons g p) []
      ∧ valid_play (LCons g p)) → ¬ defender_wins_play e (LCons g p)"

  proof
    assume A: "(play_consistent_attacker_nonpos (λlist. (case list of [] ⇒ None
|
      [x] ⇒ if x ∈ attacker ∧ ¬ deadend x then Some (SOME y. weight
x y ≠ None) else None
      | x # g' # xs ⇒
        if (x=g ∧ weight x g' ≠ None) then (SOME s. nonpos_attacker_winning_strategy
s (the (apply_w g g' e)) g') (g'#xs)
        else if last (x # g' # xs) ∈ attacker ∧ ¬ deadend (last (x # g'
# xs))
          then Some (SOME y. weight (last (x # g' # xs)) y ≠ None) else
None)) (LCons g p) []
      ∧ valid_play (LCons g p))"
    show "¬ defender_wins_play e (LCons g p)"

    proof(cases "p = LNil")
      case True
      hence "lfinite (LCons g p)"
      by simp
      from True have "llength (LCons g p) = enat 1"
      by (simp add: gen_llength_code(1) gen_llength_code(2) llength_code)
      hence "the_enat (llength (LCons g p))-1 = 0" by simp
      hence "energy_level e (LCons g p) (the_enat (llength (LCons g p))-1) = Some
e" using energy_level.simps

```

```

    by simp
  thus ?thesis using <g ∉ attacker> <lfinite (LCons g p)> defender_wins_play_def
    by (simp add: True)
next
  case False
  hence "weight g (lhd p) ≠ None" using A
    using llist.distinct(1) valid_play.cases by auto

  hence "∃s. (nonpos_attacker_winning_strategy s (the (apply_w g (lhd p) e))
    (lhd p)) ∧ play_consistent_attacker_nonpos s p []"
  proof-
    have "∃s. (nonpos_attacker_winning_strategy s (the (apply_w g (lhd p)
    e)) (lhd p))" using <weight g (lhd p) ≠ None> all by simp
    hence a_win: "nonpos_attacker_winning_strategy (SOME s. nonpos_attacker_winning_strat
    s (the (apply_w g (lhd p) e)) (lhd p)) (the (apply_w g (lhd p) e)) (lhd p)"
      by (smt (verit, del_insts) list.simps(9) nat.case_distrib nat.disc_eq_case(1)
    neq_Nil_conv take_Suc take_eq_Nil2 tfl_some verit_sko_forall')

    define strat where Strat: "strat ≡ (SOME s. nonpos_attacker_winning_strategy
    s (the (apply_w g (lhd p) e)) (lhd p))"
    define strategy where Strategy: "strategy ≡ (λlist. (case list of
      [] ⇒ None |
      [x] ⇒ (if x ∈ attacker ∧ ¬deadend x then Some (SOME
    y. weight x y ≠ None) else None) |
      (x#(g'#xs)) ⇒ (if (x=g ∧ weight x g' ≠ None) then ((SOME
    s. nonpos_attacker_winning_strategy s (the (apply_w g g' e)) g' ) (g'#xs))
      else (if (last (x#(g'#xs))) ∈ attacker ∧ ¬deadend
    (last (x#(g'#xs))) then Some (SOME y. weight (last (x#(g'#xs))) y ≠ None) else None)))))"

    hence "play_consistent_attacker_nonpos strategy (LCons g p) []" using
    A by simp
    hence strategy_cons: "play_consistent_attacker_nonpos strategy (ltl p)
    [g, lhd p]" using play_consistent_attacker_nonpos.simps
      by (smt (verit) False butlast.simps(2) last_ConsL last_ConsR lhd_LCons
    list.distinct(1) ltl_simps(2) play_consistent_attacker_nonpos_cons_simp snoc_eq_iff_butlast)

    have tail: "∧p'. strategy (g#((lhd p)#p')) = strat ((lhd p)#p'" unfolding
    Strategy Strat
      by (simp add: <weight g (lhd p) ≠ None>)

    define Q where Q: "∧s P l. Q s P l ≡ play_consistent_attacker_nonpos
    strategy P (g#l)
      ∧ l ≠ [] ∧ (∀p'. strategy (g#((hd
    l)#p')) = s ((hd l)#p'))"

    have "play_consistent_attacker_nonpos strat (ltl p) [lhd p]"
    proof(rule play_consistent_attacker_nonpos_coinduct)
      show "Q strat (ltl p) [lhd p]"
        unfolding Q using tail strategy_cons False play_consistent_attacker_nonpos_cons_s
    by auto

    show "∧s v l. Q s (LCons v LNil) l ⇒ l = [] ∨ last l ∉ attacker ∨
    last l ∈ attacker ∧ the (s l) = v"
    proof-
      fix s v l
      assume "Q s (LCons v LNil) l"

```

```

have "l ≠ [] ∧ last l ∈ attacker ⇒ the (s l) = v"
proof-
  assume "l ≠ [] ∧ last l ∈ attacker"
  hence "(∀ p'. strategy (g#((hd l)#p')) = s ((hd l)#p'))" using <Q
s (LCons v LNil) l> Q by simp
  hence "s l = strategy (g#l)"
  by (metis <l ≠ [] ∧ last l ∈ attacker> list.exhaust list.sel(1))

  from <l ≠ [] ∧ last l ∈ attacker> have "last (g#l) ∈ attacker" by
simp
  from <Q s (LCons v LNil) l> have "the (strategy (g#l)) = v" unfolding
Q using play_consistent_attacker_nonpos.simps <last (g#l) ∈ attacker>
  using eq_LConsD list.discI llist.disc(1) by blast
  thus "the (s l) = v" using <s l = strategy (g#l)> by simp
qed
thus "l = [] ∨ last l ∉ attacker ∨ last l ∈ attacker ∧ the (s l)
= v" by auto
qed

show "∧ s v Ps l. Q s (LCons v Ps) l ∧ Ps ≠ LNil ⇒ Q s Ps (l @ [v])
∧ (v ∈ attacker → lhd Ps = the (s (l @ [v])))"
proof-
  fix s v Ps l
  assume "Q s (LCons v Ps) l ∧ Ps ≠ LNil"
  hence A: "play_consistent_attacker_nonpos strategy (LCons v Ps) (g#l)
  ∧ l ≠ [] ∧ (∀ p'. strategy (g#((hd
l)#p')) = s ((hd l)#p'))" unfolding Q by simp

  show "Q s Ps (l @ [v]) ∧ (v ∈ attacker → lhd Ps = the (s (l @ [v])))"
proof
  show "Q s Ps (l @ [v])"
  unfolding Q proof
    show "play_consistent_attacker_nonpos strategy Ps (g # l @ [v])"
    using A play_consistent_attacker_nonpos.simps
    by (smt (verit) Cons_eq_appendI lhd_LCons llist.distinct(1)
ltl_simps(2))
  have "(∀ p'. strategy (g # hd (l @ [v]) # p') = s (hd (l @ [v])
# p'))" using A by simp
  thus "l @ [v] ≠ [] ∧ (∀ p'. strategy (g # hd (l @ [v]) # p') =
s (hd (l @ [v]) # p'))" by auto
qed

  show "(v ∈ attacker → lhd Ps = the (s (l @ [v])))"
proof
  assume "v ∈ attacker"
  hence "the (strategy (g#(l@[v]))) = lhd Ps" using A play_consistent_attacker_
  by (smt (verit) Cons_eq_appendI <Q s (LCons v Ps) l ∧ Ps ≠
LNil> lhd_LCons llist.distinct(1) ltl_simps(2))

  have "s (l @ [v]) = strategy (g#(l@[v]))" using A
  by (metis (no_types, lifting) hd_Cons_tl hd_append2 snoc_eq_iff_butlast)

  thus "lhd Ps = the (s (l @ [v]))" using <the (strategy (g#(l@[v])))
= lhd Ps> by simp

```

```

      qed
    qed
  qed
  qed
  hence "play_consistent_attacker_nonpos strat p []" using play_consistent_attacker_nonpos
    by (smt (verit) False <g ∉ attacker> <play_consistent_attacker_nonpos
strategy (LCons g p) []> append_butlast_last_id butlast.simps(2) last_ConsL last_ConsR
lhd_LCons lhd_LCons_ltl list.distinct(1) ltl_simps(2) play_consistent_attacker_nonpos_cons_simp
tail)

    thus ?thesis using Strat a_win by blast
  qed

  from this obtain s where S: "(nonpos_attacker_winning_strategy s (the (apply_w
g (lhd p) e)) (lhd p)) ∧ play_consistent_attacker_nonpos s p []" by auto
  have "valid_play p" using A
    by (metis llist.distinct(1) ltl_simps(2) valid_play.simps)
  hence "¬defender_wins_play (the (apply_w g (lhd p) e)) p" using S
    by (metis False nonpos_attacker_winning_strategy.elims(2) lhd_LCons llist.collapse(1)
not_lnull_conv)
  hence P: "lfinite p ∧ (energy_level (the (apply_w g (lhd p) e)) p (the_enat
(llength p)-1)) ≠ None ∧ ¬ ((llast p) ∈ attacker ∧ deadend (llast p))"
    using defender_wins_play_def by simp

  hence "∃n. llength p = enat (Suc n)" using False
    by (metis lfinite_llength_enat llength_eq_0 lnull_def old.nat.exhaust
zero_enat_def)
  from this obtain n where "llength p = enat (Suc n)" by auto
  hence "llength (LCons g p) = enat (Suc (Suc n))"
    by (simp add: eSuc_enat)
  hence "Suc (the_enat (llength p)-1) = (the_enat (llength (LCons g p))-1)"
using <llength p = enat (Suc n)> by simp

  from <weight g (lhd p) ≠ None> have "(apply_w g (lhd p) e) ≠ None"
    by (simp add: all)
  hence "energy_level (the (apply_w g (lhd p) e)) p (the_enat (llength p)-1)
= energy_level e (LCons g p) (the_enat (llength (LCons g p))-1)"
    using P energy_level_cons <Suc (the_enat (llength p)-1) = (the_enat (llength
(LCons g p))-1)> A
    by (metis (no_types, lifting) False <∃n. llength p = enat (Suc n)> diff_Suc_1
enat_ord_simps(2) lessI llist.collapse(1) the_enat.simps)
  hence "(energy_level e (LCons g p) (the_enat (llength (LCons g p))-1)) ≠
None"
    using P by simp
  then show ?thesis using P
    by (simp add: False energy_game.defender_wins_play_def llast_LCons lnull_def)

  qed
  qed
  qed

  show "nonpos_winning_budget e g" using nonpos_winning_budget.simps nonpos_attacker_winning_st
winning valid
    by blast
  qed

lemma winning_budget_ind_implies_nonpos:

```

```

assumes "winning_budget_ind e g"
shows "nonpos_winning_budget e g"
proof-
  define f where "f = (λp x1 x2.
    (∃ g e. x1 = e ∧
      x2 = g ∧
      g ∉ attacker ∧
      (∀ g'. weight g g' ≠ None →
        apply_w g g' e ≠ None ∧ p (the (apply_w g g' e)) g'))
  ∨
    (∃ g e. x1 = e ∧
      x2 = g ∧
      g ∈ attacker ∧
      (∃ g'. weight g g' ≠ None ∧
        apply_w g g' e ≠ None ∧ p (the (apply_w g g' e)) g'))))"

  have "f nonpos_winning_budget = nonpos_winning_budget"
    unfolding f_def
  proof
    fix e0
    show "(λx2. (∃ g e. e0 = e ∧
      x2 = g ∧
      g ∉ attacker ∧
      (∀ g'. weight g g' ≠ None →
        apply_w g g' e ≠ None ∧
        nonpos_winning_budget (the (apply_w g g' e)) g')) ∨
      (∃ g e. e0 = e ∧
        x2 = g ∧
        g ∈ attacker ∧
        (∃ g'. weight g g' ≠ None ∧
          apply_w g g' e ≠ None ∧
          nonpos_winning_budget (the (apply_w g g' e)) g'))))
    =
      nonpos_winning_budget e0"
  proof
    fix g0
    show "((∃ g e. e0 = e ∧
      g0 = g ∧
      g ∉ attacker ∧
      (∀ g'. weight g g' ≠ None →
        apply_w g g' e ≠ None ∧
        nonpos_winning_budget (the (apply_w g g' e)) g')) ∨
      (∃ g e. e0 = e ∧
        g0 = g ∧
        g ∈ attacker ∧
        (∃ g'. weight g g' ≠ None ∧
          apply_w g g' e ≠ None ∧
          nonpos_winning_budget (the (apply_w g g' e)) g')))) =
      nonpos_winning_budget e0 g0"
  proof
    assume " (∃ g e. e0 = e ∧
      g0 = g ∧
      g ∉ attacker ∧
      (∀ g'. weight g g' ≠ None →
        apply_w g g' e ≠ None ∧ nonpos_winning_budget (the (apply_w g
g' e)) g')) ∨

```

```

      (∃ g e. e0 = e ∧
        g0 = g ∧
        g ∈ attacker ∧
        (∃ g'. weight g g' ≠ None ∧
          apply_w g g' e ≠ None ∧ nonpos_winning_budget (the (apply_w g
g' e)) g'))"
      thus "nonpos_winning_budget e0 g0" using inductive_implies_nonpos_winning_budget
      by blast
    next
      assume "nonpos_winning_budget e0 g0"
      thus " (∃ g e. e0 = e ∧
        g0 = g ∧
        g ∉ attacker ∧
        (∀ g'. weight g g' ≠ None →
          apply_w g g' e ≠ None ∧ nonpos_winning_budget (the (apply_w g
g' e)) g')) ∨
      (∃ g e. e0 = e ∧
        g0 = g ∧
        g ∈ attacker ∧
        (∃ g'. weight g g' ≠ None ∧
          apply_w g g' e ≠ None ∧ nonpos_winning_budget (the (apply_w g
g' e)) g'))"
      using nonpos_winning_budget_implies_inductive
      by meson
    qed
  qed
qed
hence "lfp f ≤ nonpos_winning_budget "
  using lfp_lowerbound
  by (metis order_refl)
hence "winning_budget_ind ≤ nonpos_winning_budget"
  using f_def HOL.nitpick_unfold(211) by simp

  thus ?thesis using assms
  by blast
qed

```

Finally, we can state the inductive characterisation of attacker winning budgets assuming energy-positional determinacy.

```

lemma inductive_winning_budget:
  assumes "nonpos_winning_budget = winning_budget"
  shows "winning_budget = winning_budget_ind"
proof
  fix e
  show "winning_budget e = winning_budget_ind e"
proof
  fix g
  show "winning_budget e g = winning_budget_ind e g"
proof
    assume "winning_budget e g"
    thus "winning_budget_ind e g"
      using winning_budget_implies_ind winning_budget.simps by auto
  next
    assume "winning_budget_ind e g"
    hence "nonpos_winning_budget e g"
      using winning_budget_ind_implies_nonpos by simp
  end
end

```

```
      thus "winning_budget e g" using assms by simp
    qed
  qed
end
end
```

### 3 Galois Energy Games

```
theory Galois_Energy_Game
  imports Energy_Game Well_Quasi_Orders.Well_Quasi_Orders
begin
```

We now define Galois energy games over well-founded bounded join-semilattices. We do this by building on a previously defined `energy_game`. In particular, we add a set of energies `energies` with an order `order` and a supremum mapping `energy_sup`. Then, we assume the set to be partially ordered in `energy_order`, the order to be well-founded in `energy_wqo`, the supremum to map finite sets to the least upper bound `bounded_join_semilattice` and the set to be upward-closed w.r.t the order in `upward_closed_energies`. Further, we assume the updates to actually map energies (elements of the set `energies`) to energies with `upd_well_defined` and assume the inversion to map updates to total functions between the set of energies and the domain of the update in `inv_well_defined`. The latter is assumed to be upward-closed in `domain_upw_closed`. Finally, we assume the updates to be Galois-connected with their inverse in `galois`. (This corresponds to section 2.3 in the preprint [6].)

```
locale galois_energy_game = energy_game attacker weight application
  for   attacker :: "'position set" and
        weight  :: "'position  $\Rightarrow$  'position  $\Rightarrow$  'label option" and
        application :: "'label  $\Rightarrow$  'energy  $\Rightarrow$  'energy option" and
        inverse_application :: "'label  $\Rightarrow$  'energy  $\Rightarrow$  'energy option"
+
  fixes energies :: "'energy set" and
        order    :: "'energy  $\Rightarrow$  'energy  $\Rightarrow$  bool" (infix "e $\leq$ " 80) and
        energy_sup :: "'energy set  $\Rightarrow$  'energy"
  assumes
    energy_order: "ordering order ( $\lambda e e'.$  order e e'  $\wedge e \neq e'$ )" and
    energy_wqo: "wqo_on order energies" and
    bounded_join_semilattice: " $\bigwedge$  set s'. set  $\subseteq$  energies  $\implies$  finite set
 $\implies$  energy_sup set  $\in$  energies
 $\wedge (\forall s. s \in \text{set} \longrightarrow \text{order } s (\text{energy\_sup } \text{set}))$ 
 $\wedge (s' \in \text{energies} \wedge (\forall s. s \in \text{set} \longrightarrow \text{order } s s') \longrightarrow \text{order } (\text{energy\_sup } \text{set}) s')$ " and
    upward_closed_energies: " $\bigwedge e e'.$  e  $\in$  energies  $\implies$  e e $\leq$  e'  $\implies$  e'  $\in$  energies"
  and
    upd_well_defined: " $\bigwedge p p' e.$  weight p p'  $\neq$  None
 $\implies$  application (the (weight p p')) e  $\neq$  None  $\implies$  e  $\in$  energies
 $\implies$  (the (application (the (weight p p')) e))  $\in$  energies" and
    inv_well_defined: " $\bigwedge p p' e.$  weight p p'  $\neq$  None  $\implies$  e  $\in$  energies
 $\implies$  (inverse_application (the (weight p p')) e)  $\neq$  None
 $\wedge$  (the (inverse_application (the (weight p p')) e))  $\in$  energies
 $\wedge$  application (the (weight p p')) (the (inverse_application (the (weight
p p')) e))  $\neq$  None" and
    domain_upw_closed: " $\bigwedge p p' e e'.$  weight p p'  $\neq$  None  $\implies$  order e e'
 $\implies$  application (the (weight p p')) e  $\neq$  None
 $\implies$  application (the (weight p p')) e'  $\neq$  None" and
    galois: " $\bigwedge p p' e e'.$  weight p p'  $\neq$  None
 $\implies$  application (the (weight p p')) e'  $\neq$  None
 $\implies$  e  $\in$  energies  $\implies$  e'  $\in$  energies
 $\implies$  order (the (inverse_application (the (weight p p')) e)) e' = order e
(the (application (the (weight p p')) e'))"
begin
```



```

abbreviation "upd u e  $\equiv$  the (application u e)"
abbreviation "inv_upd u e  $\equiv$  the (inverse_application u e)"

abbreviation energy_l:: "'energy  $\Rightarrow$  'energy  $\Rightarrow$  bool" (infix "e<" 80) where
  "energy_l e e'  $\equiv$  e e $\leq$  e'  $\wedge$  e  $\neq$  e'"

```

### 3.1 Properties of Galois connections

The following properties are described by Ern  et al. [3].

```

lemma galois_properties:
  shows upd_inv_increasing:
    " $\bigwedge p p' e. \text{weight } p p' \neq \text{None} \implies e \in \text{energies}$ 
 $\implies \text{order } e (\text{the } (\text{application } (\text{the } (\text{weight } p p')) (\text{the } (\text{inverse\_application}$ 
 $(\text{the } (\text{weight } p p')) e))))"$ 
    and inv_upd_decreasing:
    " $\bigwedge p p' e. \text{weight } p p' \neq \text{None} \implies e \in \text{energies}$ 
 $\implies \text{application } (\text{the } (\text{weight } p p')) e \neq \text{None}$ 
 $\implies \text{the } (\text{inverse\_application } (\text{the } (\text{weight } p p')) (\text{the } (\text{application } (\text{the } (\text{weight } p p')) e))) e \leq e"$ 
    and updates_monotonic:
    " $\bigwedge p p' e e'. \text{weight } p p' \neq \text{None} \implies e \in \text{energies} \implies e e \leq e'$ 
 $\implies \text{application } (\text{the } (\text{weight } p p')) e \neq \text{None}$ 
 $\implies \text{the } (\text{application } (\text{the } (\text{weight } p p')) e) e \leq \text{the } (\text{application } (\text{the } (\text{weight } p p')) e')"$ 
    and inverse_monotonic:
    " $\bigwedge p p' e e'. \text{weight } p p' \neq \text{None} \implies e \in \text{energies} \implies e e \leq e'$ 
 $\implies \text{inverse\_application } (\text{the } (\text{weight } p p')) e \neq \text{None}$ 
 $\implies \text{the } (\text{inverse\_application } (\text{the } (\text{weight } p p')) e) e \leq \text{the } (\text{inverse\_application } (\text{the } (\text{weight } p p')) e')"$ 
  proof-
    show upd_inv_increasing: " $\bigwedge p p' e. \text{weight } p p' \neq \text{None} \implies e \in \text{energies}$ 
 $\implies \text{order } e (\text{the } (\text{application } (\text{the } (\text{weight } p p')) (\text{the } (\text{inverse\_application}$ 
 $(\text{the } (\text{weight } p p')) e))))"$ 
    proof-
      fix p p' e
      assume "weight p p'  $\neq$  None"
      define u where "u = the (weight p p')"
      show "e  $\in$  energies  $\implies \text{order } e (\text{the } (\text{application } (\text{the } (\text{weight } p p')) (\text{the } (\text{inverse\_application}$ 
 $(\text{the } (\text{weight } p p')) e))))"$ 
    proof-
      assume "e  $\in$  energies"
      have "order (inv_upd u e) (inv_upd u e)"
        by (meson local.energy_order ordering.eq_iff)

      define e' where "e' = inv_upd u e"
      have "(inv_upd u e e $\leq$  e') = e e $\leq$  upd u e'"
        unfolding u_def using <weight p p'  $\neq$  None> proof(rule galois)
          show "apply_w p p' e'  $\neq$  None"
            using <e  $\in$  energies> <weight p p'  $\neq$  None> e'_def inv_well_defined u_def
        by presburger
      show "e  $\in$  energies" using <e  $\in$  energies>.
      show "e'  $\in$  energies" unfolding e'_def
        using <e  $\in$  energies> <weight p p'  $\neq$  None> inv_well_defined u_def
      by blast
    qed
  hence "e e $\leq$  upd u (inv_upd u e)"

```

```

    using <inv_upd u e e ≤ inv_upd u e> e'_def by auto
    thus "order e (the (application (the (weight p p')) (the (inverse_application
(the (weight p p')) e))))"
    using u_def by auto
qed
qed

show inv_upd_decreasing: "∧p p' e. weight p p' ≠ None ⇒ e ∈ energies
⇒ application (the (weight p p')) e ≠ None
⇒ the (inverse_application (the (weight p p')) (the (application (the (weight
p p')) e))) e ≤ e"
proof-
  fix p p' e
  assume "weight p p' ≠ None"
  define u where "u = the (weight p p')"
  show "e ∈ energies ⇒ application (the (weight p p')) e ≠ None ⇒ the (inverse_application
(the (weight p p')) (the (application (the (weight p p')) e))) e ≤ e"
  proof-
    assume "e ∈ energies" and "application (the (weight p p')) e ≠ None"
    define e' where "e' = upd u e"
    have "(inv_upd u e' e ≤ e) = e' e ≤ upd u e"
    unfolding u_def using <weight p p' ≠ None> <application (the (weight p
p')) e ≠ None> proof(rule galois)
      show <e ∈ energies> using <e ∈ energies> .
      show <e' ∈ energies> unfolding e'_def using <e ∈ energies>
      using <apply_w p p' e ≠ None> <weight p p' ≠ None> u_def upd_well_defined
by auto
    qed
    hence "inv_upd u (upd u e) e ≤ e" using e'_def
    by (meson energy_order ordering.eq_iff)
    thus "the (inverse_application (the (weight p p')) (the (application (the
(weight p p')) e))) e ≤ e"
    using u_def by simp
    qed
  qed

show updates_monotonic: "∧p p' e e'. weight p p' ≠ None ⇒ e ∈ energies ⇒ e ≤
e'
⇒ application (the (weight p p')) e ≠ None
⇒ the (application (the (weight p p')) e) e ≤ the (application (the (weight p
p')) e' )"
proof-
  fix p p' e e'
  assume "weight p p' ≠ None" and "e ∈ energies" and "e ≤ e'" and "application
(the (weight p p')) e ≠ None"
  define u where "u = the (weight p p')"
  define e'' where "e'' = upd u e"
  have "inv_upd u (upd u e) e ≤ e' = (upd u e) e ≤ upd u e'"
  unfolding u_def using <weight p p' ≠ None> proof(rule galois)
    show "apply_w p p' e' ≠ None"
    using <application (the (weight p p')) e ≠ None> <e ≤ e'> domain_upw_closed
    using <weight p p' ≠ None> by blast
    show "(upd (the (weight p p')) e) ∈ energies"
    using <e ∈ energies> <weight p p' ≠ None> upd_well_defined
    using <apply_w p p' e ≠ None> by blast
    show "e' ∈ energies"

```

```

    using <e∈energies> <e e≤ e'> upward_closed_energies by auto
qed

have "inv_upd u (upd u e) e≤ e"
  unfolding u_def using <weight p p' ≠ None> <e∈energies> <application (the
(weight p p')) e ≠ None>
  proof(rule inv_upd_decreasing)
  qed

hence "inv_upd u (upd u e) e≤ e'" using <e e≤ e'> energy_order ordering_def
  by (metis (mono_tags, lifting) partial_preordering.trans)
hence "upd u e e≤ upd u e'"
  using <inv_upd u (upd u e) e≤ e' = (upd u e) e≤ upd u e'> by auto
thus "the( application (the (weight p p')) e) e≤ the (application (the (weight
p p')) e'"
  using u_def by auto
qed

show inverse_monotonic: "∧p p' e e'. weight p p' ≠ None ⇒ e∈energies ⇒ e
e≤ e'
⇒ inverse_application (the (weight p p')) e ≠ None
⇒ the( inverse_application (the (weight p p')) e) e≤ the (inverse_application
(the (weight p p')) e'"
  proof-
    fix p p' e e'
    assume "weight p p' ≠ None"
    define u where "u = the (weight p p'"
    show "e∈energies ⇒ e e≤ e' ⇒ inverse_application (the (weight p p')) e
≠ None ⇒ the( inverse_application (the (weight p p')) e) e≤ the (inverse_application
(the (weight p p')) e'"
      proof-
        assume "e∈energies" and " e e≤ e'" and " inverse_application (the (weight
p p')) e ≠ None"

        define e'' where "e'' = inv_upd u e'"
        have "inv_upd u e e≤ e'' = e e≤ upd u e'"
          unfolding u_def using <weight p p' ≠ None> proof(rule galois)
          show "apply_w p p' e'' ≠ None"
            unfolding e''_def using <inverse_application (the (weight p p')) e ≠
None>
            using <e ∈ energies> <e e≤ e'> <weight p p' ≠ None> inv_well_defined
u_def upward_closed_energies by blast
          show "e∈energies" using <e∈energies>.
          hence "e'∈energies"
            using <e e≤ e'> energy_order ordering_def
            using upward_closed_energies by blast
          thus "e''∈energies"
            unfolding e''_def
            using <weight p p' ≠ None> inv_well_defined u_def by blast
        qed

        have "e' e≤ upd u e'"
          unfolding e''_def u_def using <weight p p' ≠ None>
          proof(rule upd_inv_increasing)
          from <e∈energies> show "e'∈energies"
            using <e e≤ e'> energy_order ordering_def

```

```

    using upward_closed_energies by blast
qed

hence "inv_upd u e e ≤ inv_upd u e'"
  using <inv_upd u e e ≤ e' = e e ≤ upd u e'> e'_def
  using <e e ≤ e'> energy_order ordering_def
  using upward_closed_energies
  by (metis (no_types, lifting) partial_preordering.trans)
thus "the( inverse_application (the (weight p p')) e) e ≤ the (inverse_application
(the (weight p p')) e'"
  using u_def by auto
qed
qed
qed

```

Galois connections compose. In particular, the “inverse” of  $u_g$  composed with that of  $u_p$  is the “inverse” of  $u_p \circ u_g$ . This forms a Galois connection between the set of energies and the reverse image under  $u_g$  of the domain of  $u_p$ , i.e.  $u_g^{-1}(\text{dom}(u_p))$

```

lemma galois_composition:
  assumes "weight g g' ≠ None" and "weight p p' ≠ None"
  shows "∃ inv. ∀ e ∈ energies. ∀ e' ∈ energies. (application (the (weight g g'))
e' ≠ None
    ∧ application (the (weight p p')) ((upd (the (weight g g')) e')) ≠ None)
    → (order (inv e) e') = (order e (upd (the (weight p p')) ((upd (the
(weight g g')) e')))))"
proof
  define inv where "inv ≡ λx. inv_upd (the (weight g g')) (inv_upd (the (weight
p p')) x)"
  show "∀ e ∈ energies. ∀ e' ∈ energies. apply_w g g' e' ≠ None ∧ apply_w p p' (upd
(the (weight g g')) e') ≠ None → inv e e ≤ e' = e e ≤ upd (the (weight p p')) (upd
(the (weight g g')) e'"
  proof
    fix e
    assume E: "e ∈ energies"
    show "∀ e' ∈ energies. apply_w g g' e' ≠ None ∧ apply_w p p' (upd (the (weight
g g')) e') ≠ None → inv e e ≤ e' = e e ≤ upd (the (weight p p')) (upd (the (weight
g g')) e'"
    proof
      fix e'
      assume E': "e' ∈ energies"
      show "apply_w g g' e' ≠ None ∧ apply_w p p' (upd (the (weight g g')) e')
≠ None → inv e e ≤ e' = e e ≤ upd (the (weight p p')) (upd (the (weight g g'))
e'"
    proof
      assume dom: "apply_w g g' e' ≠ None ∧ apply_w p p' (upd (the (weight g
g')) e') ≠ None"
      define x where "x = inv_upd (the (weight p p')) e"
      have "inv_upd (the (weight g g')) x e ≤ e' = x e ≤ upd (the (weight g g'))
e'"
    proof(rule galois)
      show "weight g g' ≠ None" using assms by simp
      show "apply_w g g' e' ≠ None" using dom by simp
      show "x ∈ energies"
        unfolding x_def using dom

```

```

        using E assms(2) inv_well_defined by blast
        show "e' ∈ energies" using E' .
      qed
      hence A1: "inv e e ≤ e' = inv_upd (the (weight p p')) e e ≤ upd (the (weight
g g')) e'"
        unfolding inv_def x_def .

      define y where "y = (upd (the (weight g g')) e'"
      have "inv_upd (the (weight p p')) e e ≤ y = e e ≤ upd (the (weight p p'))
y"
        proof(rule galois)
          show "weight p p' ≠ None" using assms by simp
          show "apply_w p p' y ≠ None" unfolding y_def using dom by simp
          show "e ∈ energies" using E .
          show "y ∈ energies" unfolding y_def
            using E' assms(1) dom upd_well_defined by auto
        qed
        hence A2: "inv_upd (the (weight p p')) e e ≤ upd (the (weight g g')) e' =
e e ≤ upd (the (weight p p')) (upd (the (weight g g')) e'"
          unfolding inv_def y_def .
        show "inv e e ≤ e' = e e ≤ upd (the (weight p p')) (upd (the (weight g g'))
e'"
          using A1 A2 by simp
        qed
      qed
    qed
  qed

```

### 3.2 Properties of the Partial Order

We now establish some properties of the partial order focusing on the set of minimal elements.

```

definition energy_Min:: "'energy set ⇒ 'energy set" where
  "energy_Min A = {e ∈ A . ∀ e' ∈ A. e ≠ e' → ¬ (e' e ≤ e)}"

```

```

fun enumerate_arbitrary :: "'a set ⇒ nat ⇒ 'a" where
  "enumerate_arbitrary A 0 = (SOME a. a ∈ A)" |
  "enumerate_arbitrary A (Suc n)
    = enumerate_arbitrary (A - {enumerate_arbitrary A 0}) n"

```

```

lemma enumerate_arbitrary_in:
  shows "infinite A ⇒ enumerate_arbitrary A i ∈ A"
proof(induct i arbitrary: A)
  case 0
  then show ?case using enumerate_arbitrary.simps finite.simps some_in_eq by auto

```

```

next
  case (Suc i)
  hence "infinite (A - {enumerate_arbitrary A 0})" using infinite_remove by simp
  hence "enumerate_arbitrary (A - {enumerate_arbitrary A 0}) i ∈ (A - {enumerate_arbitrary
A 0})" using Suc.hyps by blast
  hence "enumerate_arbitrary A (Suc i) ∈ (A - {enumerate_arbitrary A 0})" using
enumerate_arbitrary.simps by simp
  then show ?case by auto
qed

```

```

lemma enumerate_arbitrary_neq:
  shows "infinite A  $\implies$  i < j
         $\implies$  enumerate_arbitrary A i  $\neq$  enumerate_arbitrary A j"
proof(induct i arbitrary: j A)
  case 0
  then show ?case using enumerate_arbitrary.simps
  by (metis Diff_empty Diff_iff enumerate_arbitrary_in finite_Diff_insert gr0_implies_Suc
insert_iff)
next
  case (Suc i)
  hence " $\exists j'. j = \text{Suc } j'$ "
  by (simp add: not0_implies_Suc)
  from this obtain j' where "j = Suc j'" by auto
  hence "i < j'" using Suc by simp
  from Suc have "infinite (A - {enumerate_arbitrary A 0})" using infinite_remove
  by simp
  hence "enumerate_arbitrary (A - {enumerate_arbitrary A 0}) i  $\neq$  enumerate_arbitrary
(A - {enumerate_arbitrary A 0}) j'" using Suc <i < j'>
  by force
  then show ?case using enumerate_arbitrary.simps
  by (simp add: <j = Suc j'>)
qed

lemma energy_Min_finite:
  assumes "A  $\subseteq$  energies"
  shows "finite (energy_Min A)"
proof-
  have "wqo_on order (energy_Min A)" using energy_wqo assms energy_Min_def wqo_on_subset
  by (metis (no_types, lifting) mem_Collect_eq subsetI)
  hence wqoMin: "( $\forall f \in \text{SEQ } (\text{energy\_Min } A). (\exists i j. i < j \wedge \text{order } (f i) (f j))$ )"
  unfolding wqo_on_def almost_full_on_def good_def by simp
  have " $\neg$  finite (energy_Min A)  $\implies$  False"
  proof-
    assume " $\neg$  finite (energy_Min A)"
    hence "infinite (energy_Min A)"
    by simp

    define f where "f  $\equiv$  enumerate_arbitrary (energy_Min A)"
    have fneq: " $\bigwedge i j. f i \in \text{energy\_Min } A \wedge (j \neq i \longrightarrow f j \neq f i)$ "
    proof
      fix i j
      show "f i  $\in$  energy_Min A" unfolding f_def using enumerate_arbitrary_in <infinite
(energy_Min A)> by auto
      show "j  $\neq$  i  $\longrightarrow$  f j  $\neq$  f i" proof
        assume "j  $\neq$  i"
        show "f j  $\neq$  f i" proof(cases "j < i")
          case True
          then show ?thesis unfolding f_def using enumerate_arbitrary_neq <infinite
(energy_Min A)> by auto
          next
          case False
          hence "i < j" using <j  $\neq$  i> by auto
          then show ?thesis unfolding f_def using enumerate_arbitrary_neq <infinite
(energy_Min A)>
          by metis
        qed
      qed
    qed
  qed

```

```

      qed
    qed
  qed
  hence "∃ i j. i < j ∧ order (f i) (f j)" using wqoMin SEQ_def by simp
  thus "False" using energy_Min_def fneq by force
qed
thus ?thesis by auto
qed

fun enumerate_decreasing :: "'energy set ⇒ nat ⇒ 'energy" where
  "enumerate_decreasing A 0 = (SOME a. a ∈ A)" |
  "enumerate_decreasing A (Suc n)
    = (SOME x. (x ∈ A ∧ x < enumerate_decreasing A n))"

lemma energy_Min_not_empty:
  assumes "A ≠ {}" and "A ⊆ energies"
  shows "energy_Min A ≠ {}"
proof
  have "wqo_on order A" using energy_wqo assms wqo_on_subset
    by (metis (no_types, lifting))
  hence wqoA: "(∀ f ∈ SEQ A. (∃ i j. i < j ∧ (f i) ≤ (f j)))" unfolding wqo_on_def
    almost_full_on_def good_def by simp
  assume "energy_Min A = {}"
  have seq: "enumerate_decreasing A ∈ SEQ A"
    unfolding SEQ_def proof
      show "∀ i. enumerate_decreasing A i ∈ A"
      proof
        fix i
        show "enumerate_decreasing A i ∈ A"
        proof (induct i)
          case 0
          then show ?case using assms
            by (simp add: some_in_eq)
        next
          case (Suc i)
          show ?case
          proof (rule ccontr)
            assume "enumerate_decreasing A (Suc i) ∉ A"
            hence "{x. (x ∈ A ∧ x < enumerate_decreasing A i)} = {}" unfolding enumerate_decreasing
              by (metis (no_types, lifting) empty_Collect_eq someI_ex)
            thus "False"
              using Suc <energy_Min A = {}> energy_Min_def by auto
          qed
        qed
      qed
    qed
  qed
  qed
  have "¬(∃ i j. i < j ∧ (enumerate_decreasing A i) ≤ (enumerate_decreasing A j))"
  proof-
    have "∀ i j. ¬(i < j ∧ (enumerate_decreasing A i) ≤ (enumerate_decreasing A
j))"
    proof
      fix i
      show "∀ j. ¬(i < j ∧ (enumerate_decreasing A i) ≤ (enumerate_decreasing A
j))"
    proof

```

```

fix j
have leq: "i < j  $\implies$  (enumerate_decreasing A j) e< (enumerate_decreasing
A i)"
proof(induct "j-i" arbitrary: j i)
  case 0
  then show ?case
    using <i < j> by linarith
next
  case (Suc x)

  have suc_i: "enumerate_decreasing A (Suc i) e< enumerate_decreasing A
i"

  proof-
    have "{x. (x  $\in$  A  $\wedge$  x e< enumerate_decreasing A i)} $\neq$ {}"
    proof
      assume "{x  $\in$  A. x e< enumerate_decreasing A i} = {}"
      hence "enumerate_decreasing A i  $\in$  energy_Min A" unfolding energy_Min_def
        using seq by auto
      thus "False" using <energy_Min A = {}> by auto
    qed
    thus ?thesis unfolding enumerate_decreasing.simps
      by (metis (mono_tags, lifting) empty_Collect_eq verit_sko_ex')
  qed

  have "j - (Suc i) = x" using Suc
    by (metis Suc_diff_Suc nat.inject)
  then show ?case proof(cases "j = Suc i")
    case True
    then show ?thesis using suc_i
      by simp
  next
    case False
    hence "enumerate_decreasing A j e< enumerate_decreasing A (Suc i)"
      using Suc <j - (Suc i) = x>
      using Suc_lessI by blast
    then show ?thesis using suc_i energy_order ordering_def
      by (metis (no_types, lifting) ordering_axioms_def partial_preordering.trans)

  qed
qed

hence "i < j  $\implies$   $\neg$ (enumerate_decreasing A i) e $\leq$  (enumerate_decreasing A
j)"

proof-
  assume "i < j"
  hence "(enumerate_decreasing A j) e< (enumerate_decreasing A i)" using
leq by auto
  hence leq: "(enumerate_decreasing A j) e $\leq$  (enumerate_decreasing A i)"
by simp
  have neq: "(enumerate_decreasing A j)  $\neq$  (enumerate_decreasing A i)"
    using <(enumerate_decreasing A j) e< (enumerate_decreasing A i)>
    by simp
  show " $\neg$ (enumerate_decreasing A i) e $\leq$  (enumerate_decreasing A j)"
  proof
    assume "(enumerate_decreasing A i) e $\leq$  (enumerate_decreasing A j)"
    hence "(enumerate_decreasing A i) = (enumerate_decreasing A j)" using

```



```

leq leq energy_order ordering_def
  by (simp add: ordering.antisym)
  thus "False" using neq by simp
qed
qed
thus "¬(i < j ∧ (enumerate_decreasing A i) e ≤ (enumerate_decreasing A j))"
by auto
  qed
  qed
  thus ?thesis
  by simp
  qed
  thus "False" using seq wqoA
  by blast
qed

lemma energy_Min_contains_smaller:
  assumes "a ∈ A" and "A ⊆ energies"
  shows "∃b ∈ energy_Min A. b e ≤ a"
proof-
  define set where "set ≡ {e. e ∈ A ∧ e e ≤ a}"
  hence "a ∈ set" using energy_order ordering_def
    using assms ordering.eq_iff by fastforce
  hence "set ≠ {}" by auto
  have "∧s. s ∈ set ⇒ s ∈ energies" using energy_order set_def assms
    by auto
  hence "energy_Min set ≠ {}" using <set ≠ {}> energy_Min_not_empty
    by (simp add: subsetI)
  hence "∃b. b ∈ energy_Min set" by auto
  from this obtain b where "b ∈ energy_Min set" by auto
  hence "∧b'. b' ∈ A ⇒ b' ≠ b ⇒ ¬(b' e ≤ b)"
  proof-
    fix b'
    assume "b' ∈ A"
    assume "b' ≠ b"
    show "¬(b' e ≤ b)"
  proof
    assume "(b' e ≤ b)"
    hence "b' e ≤ a" using <b ∈ energy_Min set> energy_Min_def energy_order ordering_def
      by (metis (no_types, lifting) local.set_def mem_Collect_eq partial_preordering.trans)

    hence "b' ∈ set" using <b' ∈ A> set_def by simp
    thus "False" using <b ∈ energy_Min set> energy_Min_def <b' e ≤ b> <b' ≠
b> by auto
    qed
    qed
    hence "b ∈ energy_Min A" using energy_Min_def
      using <b ∈ energy_Min set> local.set_def by auto
    thus ?thesis using <b ∈ energy_Min set> energy_Min_def set_def by auto
  qed

lemma energy_sup_leq_energy_sup:
  assumes "A ≠ {}" and "∧a. a ∈ A ⇒ ∃b ∈ B. order a b" and
    "∧a. a ∈ A ⇒ a ∈ energies" and "finite A" and "finite B" and "B ⊆ energies"
  shows "order (energy_sup A) (energy_sup B)"
proof-

```

```

have A: "⋀s'. energy_sup A ∈ energies ∧ (∀s. s ∈ A → s e≤ energy_sup A) ∧
(s' ∈ energies ∧ (∀s. s ∈ A → s e≤ s') → energy_sup A e≤ s')"
proof(rule bounded_join_semilattice)
  fix s'
  show "finite A" using assms by simp
  show "A ⊆ energies" using assms
    by (simp add: subsetI)
qed

have B: "⋀s'. energy_sup B ∈ energies ∧ (∀s. s ∈ B → s e≤ energy_sup B) ∧
(s' ∈ energies ∧ (∀s. s ∈ B → s e≤ s') → energy_sup B e≤ s')"
proof(rule bounded_join_semilattice)
  fix s'
  show "finite B" using assms by simp
  show "B ⊆ energies"
    using assms by simp
qed

have "energy_sup B ∈ energies ∧ (∀s. s ∈ A → s e≤ energy_sup B)"
proof
  show "energy_sup B ∈ energies"
    using B by simp
  show "∀s. s ∈ A → s e≤ energy_sup B"
  proof
    fix s
    show "s ∈ A → s e≤ energy_sup B"
    proof
      assume "s ∈ A"
      from this obtain b where "s e≤ b" and "b ∈ B" using assms
        by blast
      hence "b e≤ energy_sup B" using B by auto
      thus "s e≤ energy_sup B" using <s e≤ b> energy_order ordering_def
        by (metis (mono_tags, lifting) partial_preordering.trans)
    qed
  qed
qed
qed
thus ?thesis using A by auto
qed

```

### 3.3 Winning Budgets Revisited

We now redefine attacker winning budgets to only include energies in the set `energies`.

```

inductive winning_budget_len::"'energy ⇒ 'position ⇒ bool" where
  defender: "winning_budget_len e g" if "e∈energies ∧ g ∉ attacker
    ∧ (∀g'. (weight g g' ≠ None) →
      ((application (the (weight g g')) e) ≠ None
      ∧ (winning_budget_len (the (application (the (weight g g'))
e))) g'))" |
  attacker: "winning_budget_len e g" if "e∈energies ∧ g ∈ attacker
    ∧ (∃g'. (weight g g' ≠ None)
      ∧ (application (the (weight g g')) e) ≠ None
      ∧ (winning_budget_len (the (application (the (weight g g'))
e)) g'))"

```

We first restate the upward-closure of winning budgets.

```

lemma upwards_closure_wb_len:

```

```

    assumes "winning_budget_len e g" and "e ≤ e'"
    shows "winning_budget_len e' g"
using assms proof (induct arbitrary: e' rule: winning_budget_len.induct)
  case (defender e g)
  have "(∀g'. weight g g' ≠ None →
    application (the (weight g g')) e' ≠ None ∧
    winning_budget_len (the (application (the (weight g g')) e')) g'")
  proof
    fix g'
    show "weight g g' ≠ None →
      application (the (weight g g')) e' ≠ None ∧
      winning_budget_len (the (application (the (weight g g')) e')) g'"
  proof
    assume "weight g g' ≠ None"
    hence A: "application (the (weight g g')) e ≠ None ∧
      winning_budget_len (the (application (the (weight g g')) e)) g'" using
    assms(1) winning_budget_len.simps defender by blast
    show "application (the (weight g g')) e' ≠ None ∧
      winning_budget_len (the (application (the (weight g g')) e')) g'"
  proof
    show "application (the (weight g g')) e' ≠ None" using domain_upw_closed
    assms(2) A defender <weight g g' ≠ None> by blast
    have "order (the (application (the (weight g g')) e)) (the (application
      (the (weight g g')) e'))" using assms A updates_monotonic
      using <weight g g' ≠ None> defender.hyps defender.premis by presburger

    thus "winning_budget_len (the (application (the (weight g g')) e')) g'"
using defender <weight g g' ≠ None> by blast
  qed
  qed
  qed
thus ?case using winning_budget_len.intros(1) defender
  by (meson upward_closed_energies)
next
  case (attacker e g)
  from this obtain g' where G: "weight g g' ≠ None ∧
    application (the (weight g g')) e ≠ None ∧
    winning_budget_len (the (application (the (weight g g')) e)) g' ∧
    (∀x. order (the (application (the (weight g g')) e)) x → winning_budget_len
x g'") by blast
  have "weight g g' ≠ None ∧
    application (the (weight g g')) e' ≠ None ∧
    winning_budget_len (the (application (the (weight g g')) e')) g'"
  proof
    show "weight g g' ≠ None" using G by auto
    show "application (the (weight g g')) e' ≠ None ∧ winning_budget_len (the (application
      (the (weight g g')) e')) g' "
  proof
    show "application (the (weight g g')) e' ≠ None" using G domain_upw_closed
    assms attacker by blast
    have "order (the (application (the (weight g g')) e)) (the (application (the
      (weight g g')) e'))" using assms G updates_monotonic
      using attacker.hyps attacker.premis by blast
    thus "winning_budget_len (the (application (the (weight g g')) e')) g' " using
    G by blast
  qed
  qed

```

```

qed
thus ?case using winning_budget_len.intros(2) attacker
using upward_closed_energies by blast
qed

```

We now show that this definition is consistent with our previous definition of winning budgets. We show this by well-founded induction.

**abbreviation** "reachable\_positions\_len s g e  $\equiv \{(g', e') \in \text{reachable\_positions } s \mid g \leq e' \wedge e' \in \text{energies}\}$ "

```

lemma winning_budget_len_is_wb:
  assumes "nonpos_winning_budget = winning_budget"
  shows "winning_budget_len e g = (winning_budget e g  $\wedge$  e  $\in$  energies)"
proof
  assume "winning_budget_len e g"
  show "winning_budget e g  $\wedge$  e  $\in$  energies"
proof
  have "winning_budget_ind e g"
  using <winning_budget_len e g> proof(rule winning_budget_len.induct)
  show " $\bigwedge e \ g. \ e \in \text{energies} \wedge$ 
    g  $\notin$  attacker  $\wedge$ 
    ( $\forall g'. \text{weight } g \ g' \neq \text{None} \longrightarrow$ 
      apply_w g g' e  $\neq$  None  $\wedge$ 
      winning_budget_len (upd (the (weight g g')) e) g'  $\wedge$ 
      winning_budget_ind (upd (the (weight g g')) e) g')  $\implies$ 
      winning_budget_ind e g"
  using winning_budget_ind.simps
  by meson
  show " $\bigwedge e \ g. \ e \in \text{energies} \wedge$ 
    g  $\in$  attacker  $\wedge$ 
    ( $\exists g'. \text{weight } g \ g' \neq \text{None} \wedge$ 
      apply_w g g' e  $\neq$  None  $\wedge$ 
      winning_budget_len (upd (the (weight g g')) e) g'  $\wedge$ 
      winning_budget_ind (upd (the (weight g g')) e) g')  $\implies$ 
      winning_budget_ind e g"
  using winning_budget_ind.simps
  by meson
qed
thus "winning_budget e g" using assms inductive_winning_budget
by fastforce
show "e  $\in$  energies" using <winning_budget_len e g> winning_budget_len.simps
by blast
qed
next
show "winning_budget e g  $\wedge$  e  $\in$  energies  $\implies$  winning_budget_len e g"
proof-
  assume A: "winning_budget e g  $\wedge$  e  $\in$  energies"
  hence "winning_budget_ind e g" using assms inductive_winning_budget by fastforce
  show "winning_budget_len e g"
proof-
  define wb where "wb  $\equiv \lambda(g, e). \text{winning\_budget\_len } e \ g"$ 

  from A have " $\exists s. \text{attacker\_winning\_strategy } s \ e \ g$ " using winning_budget.simps
by blast
  from this obtain s where S: "attacker_winning_strategy s e g" by auto

```

```

have "reachable_positions_len s g e  $\subseteq$  reachable_positions s g e" by auto
hence "wfp_on (strategy_order s) (reachable_positions_len s g e)"
  using strategy_order_well_founded S
  using Restricted_Predicates.wfp_on_subset by blast
hence "inductive_on (strategy_order s) (reachable_positions_len s g e)"
  by (simp add: wfp_on_iff_inductive_on)

hence "wb (g,e)"
proof(rule inductive_on_induct)
  show "(g,e)  $\in$  reachable_positions_len s g e"
    unfolding reachable_positions_def proof-
      have "lfinite LNil  $\wedge$ 
        llast (LCons g LNil) = g  $\wedge$ 
        valid_play (LCons g LNil)  $\wedge$  play_consistent_attacker s (LCons g LNil)
e  $\wedge$ 
        Some e = energy_level e (LCons g LNil) (the_enat (llength LNil))"
      using valid_play.simps play_consistent_attacker.simps energy_level.simps
      by (metis lfinite_code(1) llast_singleton llength_LNil neq_LNil_conv
the_enat_0)
      thus "(g, e)  $\in$  {(g', e')}."
      (g', e')
       $\in$  {(g', e') | g' e'.
         $\exists$ p. lfinite p  $\wedge$ 
          llast (LCons g p) = g'  $\wedge$ 
          valid_play (LCons g p)  $\wedge$ 
          play_consistent_attacker s (LCons g p) e  $\wedge$ 
          Some e' = energy_level e (LCons g p) (the_enat (llength p))}  $\wedge$ 
          e'  $\in$  energies}" using A

      by blast
qed

show " $\bigwedge y. y \in \text{reachable\_positions\_len } s \text{ g e} \implies$ 
  ( $\bigwedge x. x \in \text{reachable\_positions\_len } s \text{ g e} \implies \text{strategy\_order } s \text{ x y} \implies$ 
wb x)  $\implies$  wb y"
proof-
  fix y
  assume "y  $\in$  reachable_positions_len s g e"
  hence " $\exists e' g'. y = (g', e')$ " using reachable_positions_def by auto
  from this obtain e' g' where "y = (g', e')" by auto

  hence y_len: "( $\exists p. \text{lfinite } p \wedge \text{llast (LCons g p) = g'}$ 
     $\wedge \text{valid\_play (LCons g p)}$ 
     $\wedge \text{play\_consistent\_attacker } s$ 
(LCons g p) e
     $\wedge$  (Some e' = energy_level e
(LCons g p) (the_enat (llength p))))
     $\wedge e' \in \text{energies}"$ 
    using <y  $\in$  reachable_positions_len s g e> unfolding reachable_positions_def
    by auto
  from this obtain p where P: "(lfinite p  $\wedge$  llast (LCons g p) = g'
     $\wedge \text{valid\_play (LCons g p)}$ 
     $\wedge \text{play\_consistent\_attacker } s$ 
(LCons g p) e)
     $\wedge$  (Some e' = energy_level e

```

```

(LCons g p) (the_enat (llength p)))" by auto

show "( $\bigwedge x. x \in \text{reachable\_positions\_len } s \text{ g e} \implies \text{strategy\_order } s \text{ x y}$ 
 $\implies \text{wb } x) \implies \text{wb } y$ "
proof-
  assume ind: "( $\bigwedge x. x \in \text{reachable\_positions\_len } s \text{ g e} \implies \text{strategy\_order}$ 
  s x y  $\implies \text{wb } x$ )"
  have "winning_budget_len e' g'"
  proof(cases "g'  $\in$  attacker")
    case True
    then show ?thesis
    proof(cases "deadend g'")
      case True
      hence "attacker_stuck (LCons g p)" using <g'  $\in$  attacker> P
      by (meson A defender_wins_play_def attacker_winning_strategy.elims(2))

      hence "defender_wins_play e (LCons g p)" using defender_wins_play_def
    by simp

    have " $\neg$ defender_wins_play e (LCons g p)" using P A S by simp
    then show ?thesis using <defender_wins_play e (LCons g p)> by simp
  next
    case False
    hence "(s e' g')  $\neq$  None  $\wedge$  (weight g' (the (s e' g'))) $\neq$ None" using
    S attacker_winning_strategy.simps
    by (simp add: True attacker_strategy_def)

    define x where "x = (the (s e' g'), the (apply_w g' (the (s e' g'))
    e'))"

    define p' where "p' = (lappend p (LCons (the (s e' g')) LNil))"
    hence "lfinite p'" using P by simp
    have "llast (LCons g p') = the (s e' g')" using p'_def <lfinite
    p'>

    by (simp add: llast_LCons)

    have "the_enat (llength p') > 0" using P
    by (metis LNil_eq_lappend_iff <lfinite p'> bot_nat_0.not_eq_extremum
    enat_0_iff(2) lfinite_conv_llength_enat llength_eq_0 llist.collapse(1) llist.distinct(1)
    p'_def the_enat.simps)
    hence " $\exists i. \text{Suc } i = \text{the\_enat } (\text{llength } p')$ "
    using less_iff_Suc_add by auto
    from this obtain i where "Suc i = the_enat (llength p')" by auto
    hence "i = the_enat (llength p)" using p'_def P
    by (metis Suc_leI <lfinite p'> length_append_singleton length_list_of_conv_t
    less_Suc_eq_le less_irrefl_nat lfinite_LConsI lfinite_LNil list_of_LCons list_of_LNil
    list_of_lappend not_less_less_Suc_eq)
    hence "Some e' = (energy_level e (LCons g p) i)" using P by simp

    have A: "lfinite (LCons g p)  $\wedge$  i < the_enat (llength (LCons g p))
     $\wedge$  energy_level e (LCons g p) (the_enat (llength (LCons g p)) - 1)  $\neq$  None"
    proof
      show "lfinite (LCons g p)" using P by simp
      show "i < the_enat (llength (LCons g p))  $\wedge$  energy_level e (LCons
      g p) (the_enat (llength (LCons g p)) - 1)  $\neq$  None"
      proof
        show "i < the_enat (llength (LCons g p))" using <i = the_enat
        (llength p)> P

```

```

      by (metis <lfinite (LCons g p)> length_Cons length_list_of_conv_the_enat
lessI list_of_LCons)
      show "energy_level e (LCons g p) (the_enat (llength (LCons g
p)) - 1) ≠ None" using P <i = the_enat (llength p)>
      using S defender_wins_play_def by auto
      qed
      qed

      hence "Some e' = (energy_level e (LCons g p') i)" using p'_def energy_level_app
P <Some e' = (energy_level e (LCons g p) i)>
      by (metis lappend_code(2))
      hence "energy_level e (LCons g p') i ≠ None"
      by (metis option.distinct(1))

      have "enat (Suc i) = llength p'" using <Suc i = the_enat (llength
p')>
      by (metis <lfinite p'> lfinite_conv_llength_enat the_enat.simps)
      also have "... < eSuc (llength p')"
      by (metis calculation iless_Suc_eq order_refl)
      also have "... = llength (LCons g p')" using <lfinite p'> by simp
      finally have "enat (Suc i) < llength (LCons g p')".

      have "(lnth (LCons g p) i) = g'" using <i = the_enat (llength p)>
P
      by (metis lfinite_conv_llength_enat llast_conv_lnth llength_LCons
the_enat.simps)
      hence "(lnth (LCons g p') i) = g'" using p'_def
      by (metis P <i = the_enat (llength p)> enat_ord_simps(2) energy_level.elims
lessI lfinite_llength_enat lnth_0 lnth_Suc_LCons lnth_lappend1 the_enat.simps)

      have "energy_level e (LCons g p') (the_enat (llength p')) = energy_level
e (LCons g p') (Suc i)"
      using <Suc i = the_enat (llength p')> by simp
      also have "... = apply_w (lnth (LCons g p') i) (lnth (LCons g p')
(Suc i)) (the (energy_level e (LCons g p') i))"
      using energy_level.simps <enat (Suc i) < llength (LCons g p')>
<energy_level e (LCons g p') i ≠ None>
      by (meson leD)
      also have "... = apply_w (lnth (LCons g p') i) (lnth (LCons g p')
(Suc i)) e'" using <Some e' = (energy_level e (LCons g p') i)>
      by (metis option.sel)
      also have "... = apply_w (lnth (LCons g p') i) (the (s e' g'))
e'" using p'_def <enat (Suc i) = llength p'>
      by (metis <eSuc (llength p') = llength (LCons g p')> <llast (LCons
g p') = the (s e' g')> llast_conv_lnth)
      also have "... = apply_w g' (the (s e' g')) e'" using <(lnth (LCons
g p') i) = g'> by simp
      finally have "energy_level e (LCons g p') (the_enat (llength p'))
= apply_w g' (the (s e' g')) e'" .

      have P': "lfinite p' ∧
llast (LCons g p') = (the (s e' g')) ∧
valid_play (LCons g p') ∧ play_consistent_attacker s (LCons g p') e
∧
      Some (the (apply_w g' (the (s e' g')) e')) = energy_level e (LCons g
p') (the_enat (llength p'))"

```

```

      proof
        show "lfinite p'" using p'_def P by simp
        show "llast (LCons g p') = the (s e' g') ∧
valid_play (LCons g p') ∧
play_consistent_attacker s (LCons g p') e ∧
Some (the (apply_w g' (the (s e' g')) e')) = energy_level e (LCons g p') (the_enat
(llength p')))"
      proof
        show "llast (LCons g p') = the (s e' g')" using p'_def <lfinite
p'>
        by (simp add: llast_LCons)
        show "valid_play (LCons g p') ∧
play_consistent_attacker s (LCons g p') e ∧
Some (the (apply_w g' (the (s e' g')) e')) = energy_level e (LCons g p') (the_enat
(llength p')))"
      proof
        show "valid_play (LCons g p')" using p'_def P
        using <s e' g' ≠ None ∧ weight g' (the (s e' g')) ≠ None>
valid_play.intros(2) valid_play_append by auto
        show "play_consistent_attacker s (LCons g p') e ∧
Some (the (apply_w g' (the (s e' g')) e')) = energy_level e (LCons g p') (the_enat
(llength p')))"
      proof
        have "(LCons g p') = lappend (LCons g p) (LCons (the (s
e' g')) LNil)" using p'_def
        by simp
        have "play_consistent_attacker s (lappend (LCons g p) (LCons
(the (s e' g')) LNil)) e"
        proof (rule play_consistent_attacker_append_one)
          show "play_consistent_attacker s (LCons g p) e"
          using P by auto
          show "lfinite (LCons g p)" using P by auto
          show "energy_level e (LCons g p) (the_enat (llength (LCons
g p)) - 1) ≠ None" using P
          using A by auto
          show "valid_play (lappend (LCons g p) (LCons (the (s e'
g')) LNil))"
          using <valid_play (LCons g p')> <(LCons g p') = lappend
(LCons g p) (LCons (the (s e' g')) LNil)> by simp
          show "llast (LCons g p) ∈ attacker →
Some (the (s e' g')) =
s (the (energy_level e (LCons g p) (the_enat (llength (LCons g p)) - 1))) (llast
(LCons g p))"
          proof
            assume "llast (LCons g p) ∈ attacker"
            show "Some (the (s e' g')) =
s (the (energy_level e (LCons g p) (the_enat (llength (LCons g p)) - 1))) (llast
(LCons g p))"
            using <llast (LCons g p) ∈ attacker> P
            by (metis One_nat_def <s e' g' ≠ None ∧ weight g'
(the (s e' g')) ≠ None> diff_Suc_1' eSuc_enat lfinite_llength_enat llength_LCons
option.collapse option.sel the_enat.simps)
          qed
        qed
        thus "play_consistent_attacker s (LCons g p') e" using <(LCons
g p') = lappend (LCons g p) (LCons (the (s e' g')) LNil)> by simp

```



```

      show "Some (the (apply_w g' (the (s e' g'))) e')) = energy_level
e (LCons g p') (the_enat (llength p'))"
      by (metis <eSuc (llength p') = llength (LCons g p')> <enat
(Suc i) = llength p'> <energy_level e (LCons g p') (the_enat (llength p')) = apply_w
g' (the (s e' g'))) e'> <play_consistent_attacker s (LCons g p') e> <valid_play
(LCons g p')> S defender_wins_play_def diff_Suc_1 eSuc_enat option.collapse attacker_winning_st
the_enat.simps)

      qed
      qed
      qed
      qed

      have x_len: "(upd (the (weight g' (the (s e' g')))) e') ∈ energies"
using y_len
      by (metis P' <energy_level e (LCons g p') (the_enat (llength p'))
= apply_w g' (the (s e' g'))) e'> <s e' g' ≠ None ∧ weight g' (the (s e' g')) ≠
None> option.distinct(1) upd_well_defined)
      hence "x ∈ reachable_positions_len s g e" using P' reachable_positions_def
x_def by auto

      have "(apply_w g' (the (s e' g'))) e' ≠ None" using P'
      by (metis <energy_level e (LCons g p') (the_enat (llength p'))
= apply_w g' (the (s e' g'))) e'> option.distinct(1))

      have "Some (the (apply_w g' (the (s e' g'))) e')) = apply_w g' (the
(s e' g'))) e' ∧ (if g' ∈ attacker then Some (the (s e' g')) = s e' g' else weight
g' (the (s e' g'))) ≠ None)"
      using <(s e' g') ≠ None ∧ (weight g' (the (s e' g')))) ≠ None> <(apply_w
g' (the (s e' g'))) e'> ≠ None> by simp
      hence "strategy_order s x y" unfolding strategy_order_def using
x_def <y = (g', e')>
      by blast
      hence "wb x" using ind <x ∈ reachable_positions_len s g e> by simp
      hence "winning_budget_len (the (apply_w g' (the (s e' g'))) e'))
(the (s e' g'))" using wb_def x_def by simp
      then show ?thesis using <g' ∈ attacker> winning_budget_ind.simps
      by (meson <apply_w g' (the (s e' g'))) e' ≠ None> <s e' g' ≠
None ∧ weight g' (the (s e' g')) ≠ None> winning_budget_len.attacker_y_len)
      qed
    next
      case False
      hence "g' ∉ attacker ∧
(∀ g''. weight g' g'' ≠ None →
apply_w g' g'' e' ≠ None ∧ winning_budget_len (the (apply_w g' g'' e'))
g'')"
      proof
        show "∀ g''. weight g' g'' ≠ None →
apply_w g' g'' e' ≠ None ∧ winning_budget_len (the (apply_w g' g'' e'))
g'""
          proof
            fix g''
            show "weight g' g'' ≠ None →
apply_w g' g'' e' ≠ None ∧ winning_budget_len (the (apply_w g' g'' e'))
g'""
              proof

```

```

    assume "weight g' g'' ≠ None"
    show "apply_w g' g'' e' ≠ None ∧ winning_budget_len (the (apply_w
g' g'' e')) g'"
  proof
    show "apply_w g' g'' e' ≠ None"
    proof
      assume "apply_w g' g'' e' = None"
      define p' where "p' ≡ (LCons g (lappend p (LCons g'' LNil)))"
      hence "lfinite p'" using P by simp
      have "∃i. llength p = enat i" using P
        by (simp add: lfinite_llength_enat)
      from this obtain i where "llength p = enat i" by auto
      hence "llength (lappend p (LCons g'' LNil)) = enat (Suc
i)"

        by (simp add: <llength p = enat i> eSuc_enat iadd_Suc_right)
      hence "llength p' = eSuc (enat (Suc i))" using p'_def
        by simp
      hence "the_enat (llength p') = Suc (Suc i)"
        by (simp add: eSuc_enat)
      hence "the_enat (llength p') - 1 = Suc i"
        by simp
      hence "the_enat (llength p') - 1 = the_enat (llength (lappend
p (LCons g'' LNil)))"
        using <llength (lappend p (LCons g'' LNil)) = enat (Suc
i)>
        by simp

      have "(lnth p' i) = g'" using p'_def <llength p = enat i>
P
        by (smt (verit) One_nat_def diff_Suc_1' enat_ord_simps(2)
energy_level.elims lessI llast_conv_lnth llength_LCons lnth_0 lnth_LCons' lnth_lappend
the_enat.simps)
      have "(lnth p' (Suc i)) = g'" using p'_def <llength p =
enat i>
        by (metis <llength p' = eSuc (enat (Suc i))> lappend.disc(2)
llast_LCons llast_conv_lnth llast_lappend_LCons llength_eq_enat_lfiniteD llist.disc(1)
llist.disc(2))
      have "p' = lappend (LCons g p) (LCons g'' LNil)" using p'_def
by simp
      hence "the (energy_level e p' i) = the (energy_level e (lappend
(LCons g p) (LCons g'' LNil)) i)" by simp
      also have "... = the (energy_level e (LCons g p) i)" using
<llength p = enat i> energy_level_append P
        by (metis diff_Suc_1 eSuc_enat lessI lfinite_LConsI llength_LCons
option.distinct(1) the_enat.simps)
      also have "... = e'" using P
        by (metis <llength p = enat i> option.sel the_enat.simps)

      finally have "the (energy_level e p' i) = e'" .
      hence "apply_w (lnth p' i) (lnth p' (Suc i)) (the (energy_level
e p' i)) = None" using <apply_w g' g'' e'=None> <(lnth p' i) = g'> <(lnth p' (Suc
i)) = g''> by simp

      have "energy_level e p' (the_enat (llength p') - 1) =
        energy_level e p' (the_enat (llength (lappend p (LCons
g'' LNil))))"

```

```

    using <the_enat (llength p') - 1 = the_enat (llength (lappend
p (LCons g'' LNil)))>
    by simp
    also have "... = energy_level e p' (Suc i)" using <llength
(lappend p (LCons g'' LNil)) = enat (Suc i)> by simp
    also have "... = (if energy_level e p' i = None ∨ llength
p' ≤ enat (Suc i) then None
    else apply_w (lnth p' i) (lnth p' (Suc i))
(the (energy_level e p' i)))" using energy_level.simps by simp
    also have "... = None" using <apply_w (lnth p' i) (lnth
p' (Suc i)) (the (energy_level e p' i)) = None>
    by simp
    finally have "energy_level e p' (the_enat (llength p') -
1) = None" .
    hence "defender_wins_play e p'" unfolding defender_wins_play_def
by simp

    have "valid_play p'"
    by (metis P <p' = lappend (LCons g p) (LCons g'' LNil)>
<weight g' g'' ≠ None> energy_game.valid_play.intros(2) energy_game.valid_play_append
lfinite_LConsI)

    have "play_consistent_attacker s (lappend (LCons g p) (LCons
g'' LNil)) e"
    proof(rule play_consistent_attacker_append_one)
    show "play_consistent_attacker s (LCons g p) e"
    using P by simp
    show "lfinite (LCons g p)" using P by simp
    show "energy_level e (LCons g p) (the_enat (llength (LCons
g p)) - 1) ≠ None"
    using P
    by (meson S defender_wins_play_def attacker_winning_strategy.elims(
show "valid_play (lappend (LCons g p) (LCons g'' LNil))"
using <valid_play p'> <p' = lappend (LCons g p) (LCons
g'' LNil)> by simp
    show "llast (LCons g p) ∈ attacker →
    Some g'' =
    s (the (energy_level e (LCons g p) (the_enat (llength (LCons g p)) - 1))) (llast
(LCons g p))"
    using False P by simp
    qed
    hence "play_consistent_attacker s p' e"
    using <p' = lappend (LCons g p) (LCons g'' LNil)> by
simp
    hence "¬defender_wins_play e p'" using <valid_play p'>
p'_def S by simp
    thus "False" using <defender_wins_play e p'> by simp
    qed

    define x where "x = (g'', the (apply_w g' g'' e'))"
    have "wb x"
    proof(rule ind)
    have X: "(∃p. lfinite p ∧
llast (LCons g p) = g'' ∧
valid_play (LCons g p) ∧ play_consistent_attacker s (LCons g p) e ∧

```

```

    Some (the (apply_w g' g'' e')) = energy_level e (LCons g p) (the_enat
(1length p)))"

    proof
      define p' where "p' = lappend p (LCons g'' LNil)"
      show "lfinite p' ^
        llast (LCons g p') = g'' ^
        valid_play (LCons g p') ^ play_consistent_attacker s (LCons g p') e ^
        Some (the (apply_w g' g'' e')) = energy_level e (LCons g p') (the_enat (1length
p')))"

    proof
      show "lfinite p'" using P p'_def by simp
      show "llast (LCons g p') = g'' ^
        valid_play (LCons g p') ^
        play_consistent_attacker s (LCons g p') e ^
        Some (the (apply_w g' g'' e')) = energy_level e (LCons g p') (the_enat (1length
p')))"

    proof
      show "llast (LCons g p') = g''" using p'_def
        by (metis <lfinite p'> lappend.disc_iff(2) lfinite_lappend
llast_LCons llast_lappend_LCons llast_singleton llist.discI(2))
      show "valid_play (LCons g p') ^
        play_consistent_attacker s (LCons g p') e ^
        Some (the (apply_w g' g'' e')) = energy_level e (LCons g p') (the_enat (1length
p')))"

    proof
      show "valid_play (LCons g p'" using p'_def P
        using <weight g' g'' ≠ None> lfinite_LCons valid_play.intros

valid_play_append by auto

      show "play_consistent_attacker s (LCons g p') e
^
  Some (the (apply_w g' g'' e')) = energy_level e (LCons g p') (the_enat (1length
p'))"

    proof

      have "play_consistent_attacker s (lappend (LCons
g p) (LCons g'' LNil)) e"

      proof(rule play_consistent_attacker_append_one)
        show "play_consistent_attacker s (LCons g p)

          using P by simp
          show "lfinite (LCons g p)" using P by simp
          show "energy_level e (LCons g p) (the_enat (1length
(LCons g p)) - 1) ≠ None"

          using P
          by (meson S defender_wins_play_def attacker_winning_strat
show "valid_play (lappend (LCons g p) (LCons
g'' LNil))"

          using <valid_play (LCons g p')> p'_def by

simp

        show "llast (LCons g p) ∈ attacker →
          Some g'' =
          s (the (energy_level e (LCons g p) (the_enat
(1length (LCons g p)) - 1))) (llast (LCons g p))"
          using False P by simp
      qed
      thus "play_consistent_attacker s (LCons g p')

```

```

e" using p'_def
by (simp add: lappend_code(2))

have "∃ i. Suc i = the_enat (llength p')" using
p'_def <lfinite p'>
by (metis P length_append_singleton length_list_of_conv_the
lfinite_LConsI lfinite_LNil list_of_LCons list_of_LNil list_of_lappend)
from this obtain i where "Suc i = the_enat (llength
p')" by auto

hence "i = the_enat (llength p)" using p'_def
by (smt (verit) One_nat_def <lfinite p'> add.commute
add_Suc_shift add_right_cancel length_append length_list_of_conv_the_enat lfinite_LNil
lfinite_lappend list.size(3) list.size(4) list_of_LCons list_of_LNil list_of_lappend
plus_1_eq_Suc)

hence "Suc i = llength (LCons g p)"
using P eSuc_enat lfinite_llength_enat by fastforce
have "(LCons g p') = lappend (LCons g p) (LCons
g'' LNil)" using p'_def by simp

have A: "lfinite (LCons g p) ∧ i < the_enat (llength
(LCons g p)) ∧ energy_level e (LCons g p) (the_enat (llength (LCons g p)) - 1)
≠ None"

proof
show "lfinite (LCons g p)" using P by simp
show " i < the_enat (llength (LCons g p)) ∧
energy_level e (LCons g p) (the_enat (llength (LCons g p)) - 1) ≠ None "
proof
have "(llength p') = llength (LCons g p)"

using p'_def
by (metis P <lfinite p'> length_Cons length_append_singleton
length_list_of lfinite_LConsI lfinite_LNil list_of_LCons list_of_LNil list_of_lappend)

thus "i < the_enat (llength (LCons g p))"

using <Suc i = the_enat (llength p')>
using lessI by force
show "energy_level e (LCons g p) (the_enat
(llength (LCons g p)) - 1) ≠ None" using P
by (meson S energy_game.defender_wins_play_def
energy_game.play_consistent_attacker.intros(2) attacker_winning_strategy.simps)
qed
qed
hence "energy_level e (LCons g p') i ≠ None"
using energy_level_append
by (smt (verit) Nat.lessE Suc_leI <LCons g p'
= lappend (LCons g p) (LCons g'' LNil)> diff_Suc_1 energy_level_nth)
have "enat (Suc i) < llength (LCons g p')"
using <Suc i = the_enat (llength p')>
by (metis Suc_ile_eq <lfinite p'> ldropsn_Suc_LCons
leI lfinite_conv_llength_enat lnull_ldropsn nless_le the_enat.simps)
hence el_premis: "energy_level e (LCons g p')
i ≠ None ∧ llength (LCons g p') > enat (Suc i)" using <energy_level e (LCons g
p') i ≠ None> by simp

have "(lnth (LCons g p') i) = lnth (LCons g p)
i"

unfolding <(LCons g p') = lappend (LCons g p)
(LCons g'' LNil)> using <i = the_enat (llength p)> lnth_lappend1

```

```

length_list_of_conv_the_enat)
    by (metis A enat_ord_simps(2) length_list_of
length_list_of_conv_the_enat)
    have "lnth (LCons g p) i = llast (LCons g p)"
using <Suc i = llength (LCons g p)>
    by (metis enat_ord_simps(2) lappend_LNil2 ldropsn_LNil
ldropsn_Suc_conv_ldropsn ldropsn_lappend lessI less_not_refl llast_ldropsn llast_singleton)
    hence "(lnth (LCons g p') i) = g'" using P
    by (simp add: <lnth (LCons g p') i = lnth (LCons
g p) i>)
    have "(lnth (LCons g p') (Suc i)) = g'"
    using p'_def <Suc i = the_enat (llength p')>
    by (smt (verit) <enat (Suc i) < llength (LCons
g p')> <lfinite p'> <llast (LCons g p') = g'>> lappend_snocL1_conv_LCons2 ldropsn_LNil
ldropsn_Suc_LCons ldropsn_Suc_conv_ldropsn ldropsn_lappend2 lfinite_llength_enat llast_ldropsn
llast_singleton the_enat.simps wlog_linorder_le)

    have "energy_level e (LCons g p) i = energy_level
e (LCons g p') i"
    using energy_level_append A <(LCons g p') =
lappend (LCons g p) (LCons g'' LNil)>
    by presburger
    hence "Some e' = (energy_level e (LCons g p')
i)"
    using P <i = the_enat (llength p)>
    by argo

    have "energy_level e (LCons g p') (the_enat (llength
p')) = energy_level e (LCons g p') (Suc i)" using <Suc i = the_enat (llength p')>
by simp
    also have "... = apply_w (lnth (LCons g p') i)
(lnth (LCons g p') (Suc i)) (the (energy_level e (LCons g p') i))"
    using energy_level.simps el_prem
    by (meson leD)
    also have "... = apply_w g' g'' (the (energy_level
e (LCons g p') i))"
    using <(lnth (LCons g p') i) = g'> <(lnth (LCons
g p') (Suc i)) = g''> by simp
    finally have "energy_level e (LCons g p') (the_enat
(llength p')) = (apply_w g' g'' e)"
    using <Some e' = (energy_level e (LCons g p')
i)>
    by (metis option.sel)
    thus "Some (the (apply_w g' g'' e)) = energy_level
e (LCons g p') (the_enat (llength p'))"
    using <apply_w g' g'' e' ≠ None> by auto
qed
qed
qed
qed
qed

    have x_len: "(upd (the (weight g' g'')) e') ∈ energies" using
y_len
    using <apply_w g' g'' e' ≠ None> <weight g' g'' ≠ None>
upd_well_defined by auto

```

```

      thus "x ∈ reachable_positions_len s g e"
      using X x_def reachable_positions_def
      by (simp add: mem_Collect_eq)

      have "Some (the (apply_w g' g'' e')) = apply_w g' g'' e'"
    ^
      (if g' ∈ attacker then Some g'' = s e' g' else weight g' g'' ≠ None)"
    proof
      show "Some (the (apply_w g' g'' e')) = apply_w g' g'"
    e'"

      using <apply_w g' g'' e' ≠ None> by auto
      show "(if g' ∈ attacker then Some g'' = s e' g' else weight
g' g'' ≠ None)"

      using False
      by (simp add: <weight g' g'' ≠ None>)
    qed
    thus "strategy_order s x y" using strategy_order_def x_def
    <y = (g', e')>
      by simp
    qed

    thus "winning_budget_len (the (apply_w g' g'' e')) g'' " using
x_def wb_def
      by force
    qed
    qed
    qed
    qed
    thus ?thesis using winning_budget_len.intros y_len by blast
    qed
    thus "wb y" using <y = (g', e')> wb_def by simp
    qed
    qed
    qed
    thus ?thesis using wb_def by simp
    qed
    qed
    qed
  end
end

```

## 4 Decidability of Galois Energy Games

```
theory Decidability
  imports Galois_Energy_Game Complete_Non_Orders.Kleene_Fixed_Point
begin
```

In this theory we give a proof of decidability for Galois energy games (over vectors of naturals). We do this by providing a proof of correctness of the simplified version of Bisping's Algorithm to calculate minimal attacker winning budgets. We further formalise the key argument for its termination. (This corresponds to section 3.2 in the preprint [6].)

```
locale galois_energy_game_decidable = galois_energy_game attacker weight application
  inverse_application energies order energy_sup
  for attacker :: "'position set" and
    weight :: "'position ⇒ 'position ⇒ 'label option" and
    application :: "'label ⇒ 'energy ⇒ 'energy option" and
    inverse_application :: "'label ⇒ 'energy ⇒ 'energy option" and
    energies :: "'energy set" and
    order :: "'energy ⇒ 'energy ⇒ bool" (infix "e≤" 80) and
    energy_sup :: "'energy set ⇒ 'energy"
+
  assumes nonpos_eq_pos: "nonpos_winning_budget = winning_budget" and
    finite_positions: "finite positions"
begin
```

### 4.1 Minimal Attacker Winning Budgets as Pareto Fronts

We now prepare the proof of decidability by introducing minimal winning budgets.

```
abbreviation minimal_winning_budget :: "'energy ⇒ 'position ⇒ bool" where
  "minimal_winning_budget e g ≡ e ∈ energy_Min {e. winning_budget_len e g}"
abbreviation "a_win g ≡ {e. winning_budget_len e g}"
abbreviation "a_win_min g ≡ energy_Min (a_win g)"
```

Since the component-wise order on energies is well-founded, we can conclude that minimal winning budgets are finite.

```
lemma minimal_winning_budget_finite:
  shows "∧g. finite (a_win_min g)"
proof(rule energy_Min_finite)
  fix g
  show "a_win g ⊆ energies" using nonpos_eq_pos winning_budget_len.cases
    by blast
qed
```

We now introduce the set of mappings from positions to possible Pareto fronts, i.e. incomparable sets of energies.

```
definition possible_pareto :: "('position ⇒ 'energy set) set" where
  "possible_pareto ≡ {F. ∀g. F g ⊆ {e. e∈energies}
    ∧ (∀e e'. (e ∈ F g ∧ e' ∈ F g ∧ e ≠ e')
      → (¬ e e≤ e' ∧ ¬ e' e≤ e))}"
```

By definition minimal winning budgets are possible Pareto fronts.

```
lemma a_win_min_in_pareto:
  shows "a_win_min ∈ possible_pareto"
  unfolding energy_Min_def possible_pareto_def proof
```



```

show "∀g. {e ∈ a_win g. ∀e'∈a_win g. e ≠ e' → ¬ e' e ≤ e} ⊆ {e. e∈energies}
^
  (∀e e'.
    e ∈ {e ∈ a_win g. ∀e'∈a_win g. e ≠ e' → ¬ e' e ≤ e} ∧
    e' ∈ {e ∈ a_win g. ∀e'∈a_win g. e ≠ e' → ¬ e' e ≤ e} ∧ e ≠ e' →
    incomparable (e ≤) e e') "
proof
  fix g
  show "{e ∈ a_win g. ∀e'∈a_win g. e ≠ e' → ¬ e' e ≤ e} ⊆ {e. e∈energies}
^
  (∀e e'.
    e ∈ {e ∈ a_win g. ∀e'∈a_win g. e ≠ e' → ¬ e' e ≤ e} ∧
    e' ∈ {e ∈ a_win g. ∀e'∈a_win g. e ≠ e' → ¬ e' e ≤ e} ∧ e ≠ e' →
    incomparable (e ≤) e e') "
proof
  show "{e ∈ a_win g. ∀e'∈a_win g. e ≠ e' → ¬ e' e ≤ e} ⊆ {e. e∈energies}"
  using winning_budget_len.simps
  by (smt (verit) Collect_mono_iff mem_Collect_eq)
show " ∀e e'.
  e ∈ {e ∈ a_win g. ∀e'∈a_win g. e ≠ e' → ¬ e' e ≤ e} ∧
  e' ∈ {e ∈ a_win g. ∀e'∈a_win g. e ≠ e' → ¬ e' e ≤ e} ∧ e ≠ e' →
  incomparable (e ≤) e e' "
proof
  fix e
  show "∀e'. e ∈ {e ∈ a_win g. ∀e'∈a_win g. e ≠ e' → ¬ e' e ≤ e} ∧
    e' ∈ {e ∈ a_win g. ∀e'∈a_win g. e ≠ e' → ¬ e' e ≤ e} ∧ e ≠ e'
→
    incomparable (e ≤) e e'"
proof
  fix e'
  show "e ∈ {e ∈ a_win g. ∀e'∈a_win g. e ≠ e' → ¬ e' e ≤ e} ∧
    e' ∈ {e ∈ a_win g. ∀e'∈a_win g. e ≠ e' → ¬ e' e ≤ e} ∧ e ≠ e' →
    incomparable (e ≤) e e'"
proof
  assume " e ∈ {e ∈ a_win g. ∀e'∈a_win g. e ≠ e' → ¬ e' e ≤ e} ∧
    e' ∈ {e ∈ a_win g. ∀e'∈a_win g. e ≠ e' → ¬ e' e ≤ e} ∧ e ≠ e'"
  thus "incomparable (e ≤) e e'"
  by auto
qed
qed
qed
qed
qed
qed
qed

```

We define a partial order on possible Pareto fronts.

```

definition pareto_order:: "('position ⇒ 'energy set) ⇒ ('position ⇒ 'energy set)
⇒ bool" (infix "⊆" 80) where
  "pareto_order F F' ≡ (∀g e. e ∈ F(g) → (∃e'. e' ∈ F'(g) ∧ e' e ≤ e))"

lemma pareto_partial_order_vanilla:
  shows reflexivity: "⋀F. F ∈ possible_pareto ⇒ F ⊆ F" and
  transitivity: "⋀F F' F''. F ∈ possible_pareto ⇒ F' ∈ possible_pareto
    ⇒ F'' ∈ possible_pareto ⇒ F ⊆ F' ⇒ F' ⊆ F''
    ⇒ F ⊆ F'' " and
  antisymmetry: "⋀F F'. F ∈ possible_pareto ⇒ F' ∈ possible_pareto

```

```

       $\implies F \preceq F' \implies F' \preceq F \implies F = F'$ 
proof-
  fix F F' F''
  assume "F ∈ possible_pareto" and "F' ∈ possible_pareto" and "F'' ∈ possible_pareto"
  show "F  $\preceq$  F'"
    unfolding pareto_order_def energy_order ordering_def
    by (meson energy_order ordering.eq_iff)
  show "F  $\preceq$  F'  $\implies$  F'  $\preceq$  F''  $\implies$  F  $\preceq$  F'' "
proof-
  assume "F  $\preceq$  F'" and "F'  $\preceq$  F'"
  show " F  $\preceq$  F'' "
    unfolding pareto_order_def proof
    show " $\bigwedge g. \forall e. e \in F g \longrightarrow (\exists e'. e' \in F'' g \wedge e' \leq e)$ "
    proof
      fix g e
      show "e ∈ F g  $\longrightarrow$  ( $\exists e'. e' \in F'' g \wedge e' \leq e$ )"
      proof
        assume "e ∈ F g"
        hence "( $\exists e'. e' \in F' g \wedge e' \leq e$ )" using <F  $\preceq$  F'> unfolding pareto_order_def
      by simp
        from this obtain e' where "e' ∈ F' g  $\wedge$  e' ≤ e" by auto
        hence "( $\exists e''. e'' \in F'' g \wedge e'' \leq e$ )" using <F'  $\preceq$  F''> unfolding pareto_order_def
      by simp
        from this obtain e'' where "e'' ∈ F'' g  $\wedge$  e'' ≤ e" by auto
        hence "e'' ∈ F'' g  $\wedge$  e'' ≤ e" using <e' ∈ F' g  $\wedge$  e' ≤ e> energy_order
      ordering_def
        by (metis (mono_tags, lifting) partial_preordering.trans)
        thus " $\exists e'. e' \in F'' g \wedge e' \leq e$ " by auto
      qed
    qed
  qed
  show "F  $\preceq$  F'  $\implies$  F'  $\preceq$  F  $\implies$  F = F'"
proof-
  assume "F  $\preceq$  F'" and "F'  $\preceq$  F"
  show "F = F'"
  proof
    fix g
    show "F g = F' g"
    proof
      show "F g  $\subseteq$  F' g"
      proof
        fix e
        assume "e ∈ F g"
        hence " $\exists e'. e' \in F' g \wedge e' \leq e$ " using <F  $\preceq$  F'> unfolding pareto_order_def
      by auto
        from this obtain e' where "e' ∈ F' g  $\wedge$  e' ≤ e" by auto
        hence " $\exists e''. e'' \in F g \wedge e'' \leq e$ " using <F'  $\preceq$  F> unfolding pareto_order_def
      by auto
        from this obtain e'' where "e'' ∈ F g  $\wedge$  e'' ≤ e" by auto
        hence "e'' = e  $\wedge$  e' = e" using possible_pareto_def <F ∈ possible_pareto>
      energy_order ordering_def
        by (smt (verit, ccfv_SIG) <e ∈ F g> <e' ∈ F' g  $\wedge$  e' ≤ e> mem_Collect_eq
      ordering.antisym partial_preordering_def)
        thus "e ∈ F' g" using <e' ∈ F' g  $\wedge$  e' ≤ e> by auto
      qed
    qed
  qed

```

```

    show "F' g  $\subseteq$  F g"
  proof
    fix e
    assume "e  $\in$  F' g"
    hence " $\exists e'. e' \in F g \wedge e' \leq e$ " using <F'  $\preceq$  F> unfolding pareto_order_def
  by auto
    from this obtain e' where "e'  $\in$  F g  $\wedge$  e'  $\leq$  e" by auto
    hence " $\exists e''. e'' \in F' g \wedge e'' \leq e$ " using <F  $\preceq$  F'> unfolding pareto_order_def
  by auto
    from this obtain e'' where "e''  $\in$  F' g  $\wedge$  e''  $\leq$  e'" by auto
    hence "e'' = e  $\wedge$  e' = e" using possible_pareto_def <F'  $\in$  possible_pareto>
  energy_order ordering_def
    by (smt (verit, best) <F g  $\subseteq$  F' g> <e  $\in$  F' g> <e'  $\in$  F g  $\wedge$  e'  $\leq$  e>)
  in_mono mem_Collect_eq
    thus "e  $\in$  F g" using <e'  $\in$  F g  $\wedge$  e'  $\leq$  e> by auto
  qed
qed
qed
qed
qed

```

```

lemma pareto_partial_order:
  shows "reflp_on possible_pareto ( $\preceq$ )" and
    "transp_on possible_pareto ( $\preceq$ )" and
    "antisympt_on possible_pareto ( $\preceq$ )"
proof-
  show "reflp_on possible_pareto ( $\preceq$ )"
    using reflexivity
  by (simp add: reflt_onI)
  show "transp_on possible_pareto ( $\preceq$ )"
    using transitivity
    using transp_onI by blast
  show "antisympt_on possible_pareto ( $\preceq$ )"
    using antisymmetry
    using antisympt_onI by auto
qed

```

By defining a supremum, we show that the order is directed-complete bounded join-semilattice.

```

definition pareto_sup:: "('position  $\Rightarrow$  'energy set) set  $\Rightarrow$  ('position  $\Rightarrow$  'energy set)" where
  "pareto_sup P g = energy_Min {e.  $\exists F. F \in P \wedge e \in F g$ }"

```

```

lemma pareto_sup_is_sup:
  assumes "P  $\subseteq$  possible_pareto"
  shows "pareto_sup P  $\in$  possible_pareto" and
    " $\bigwedge F. F \in P \implies F \preceq$  pareto_sup P" and
    " $\bigwedge Fs. Fs \in possible\_pareto \implies (\bigwedge F. F \in P \implies F \preceq Fs) \implies$ 
      pareto_sup P  $\preceq$  Fs"
proof-
  show "pareto_sup P  $\in$  possible_pareto" unfolding pareto_sup_def possible_pareto_def
  energy_Min_def
    by (smt (verit, ccfv_threshold) Ball_Collect assms mem_Collect_eq possible_pareto_def)

  show " $\bigwedge F. F \in P \implies F \preceq$  pareto_sup P"
proof-

```

```

fix F
assume "F ∈ P"
show "F ≼ pareto_sup P"
  unfolding pareto_order_def proof
  show "∧g. ∀e. e ∈ F g → (∃e'. e' ∈ pareto_sup P g ∧ e' ≤ e)"
  proof
    fix g e
    show "e ∈ F g → (∃e'. e' ∈ pareto_sup P g ∧ e' ≤ e)"
    proof
      have in_energy: "{e. ∃F. F ∈ P ∧ e ∈ F g} ⊆ energies"
      using assms possible_pareto_def by force
      assume "e ∈ F g"
      hence "e ∈ {(e::'energy). (∃F. F ∈ P ∧ e ∈ (F g))}" using <F ∈ P> by auto
      hence "∃e'. e' ∈ energy_Min {(e::'energy). (∃F. F ∈ P ∧ e ∈ (F g))} ∧
e' ≤ e"
      using energy_Min_contains_smaller in_energy
      by meson
      thus "∃e'. e' ∈ pareto_sup P g ∧ e' ≤ e" unfolding pareto_sup_def by
simp
      qed
    qed
  qed
  qed
  show "∧Fs. Fs ∈ possible_pareto ⇒ (∧F. F ∈ P ⇒ F ≼ Fs) ⇒ pareto_sup P
≼ Fs"
  proof-
    fix Fs
    assume "Fs ∈ possible_pareto" and "(∧F. F ∈ P ⇒ F ≼ Fs)"
    show "pareto_sup P ≼ Fs"
      unfolding pareto_order_def proof
      show "∧g. ∀e. e ∈ pareto_sup P g → (∃e'. e' ∈ Fs g ∧ e' ≤ e) "
      proof
        fix g e
        show "e ∈ pareto_sup P g → (∃e'. e' ∈ Fs g ∧ e' ≤ e)"
        proof
          assume "e ∈ pareto_sup P g"
          hence "e ∈ {e. ∃F. F ∈ P ∧ e ∈ F g}" unfolding pareto_sup_def using energy_Min_def
by simp
          from this obtain F where "F ∈ P ∧ e ∈ F g" by auto
          thus "∃e'. e' ∈ Fs g ∧ e' ≤ e" using <(∧F. F ∈ P ⇒ F ≼ Fs)> pareto_order_def
by auto
          qed
        qed
      qed
    qed
  qed
  qed

lemma pareto_directed_complete:
  shows "directed_complete possible_pareto (≼)"
  unfolding directed_complete_def
proof-
  show "(λX r. directed X r ∧ X ≠ {})-complete possible_pareto (≼)"
  unfolding complete_def
  proof
    fix P
    show "P ⊆ possible_pareto →

```

```

    directed P ( $\preceq$ )  $\wedge$  P  $\neq$  {}  $\longrightarrow$  ( $\exists$ s. extreme_bound possible_pareto ( $\preceq$ ) P
s)"
  proof
    assume "P  $\subseteq$  possible_pareto"
    show "directed P ( $\preceq$ )  $\wedge$  P  $\neq$  {}  $\longrightarrow$  ( $\exists$ s. extreme_bound possible_pareto ( $\preceq$ )
P s)"
    proof
      assume "directed P ( $\preceq$ )  $\wedge$  P  $\neq$  {}"
      show " $\exists$ s. extreme_bound possible_pareto ( $\preceq$ ) P s"
      proof
        show "extreme_bound possible_pareto ( $\preceq$ ) P (pareto_sup P)"
        unfolding extreme_bound_def
        proof
          show "pareto_sup P  $\in$  {b  $\in$  possible_pareto. bound P ( $\preceq$ ) b}"
          using pareto_sup_is_sup <P  $\subseteq$  possible_pareto> <directed P ( $\preceq$ )  $\wedge$ 
P  $\neq$  {}>
          by blast
          show " $\bigwedge$ x. x  $\in$  {b  $\in$  possible_pareto. bound P ( $\preceq$ ) b}  $\implies$  pareto_sup
P  $\preceq$  x"
          proof-
            fix x
            assume "x  $\in$  {b  $\in$  possible_pareto. bound P ( $\preceq$ ) b}"
            thus "pareto_sup P  $\preceq$  x"
            using pareto_sup_is_sup <P  $\subseteq$  possible_pareto> <directed P ( $\preceq$ )
 $\wedge$  P  $\neq$  {}>
            by auto
          qed
        qed
      qed
    qed
  qed
qed
qed
qed
qed
qed
qed
qed

lemma pareto_minimal_element:
  shows " $(\lambda g. \{\}) \preceq F$ "
  unfolding pareto_order_def by simp

```

## 4.2 Proof of Decidability

Using Kleene's fixed point theorem we now show, that the minimal attacker winning budgets are the least fixed point of the algorithm. For this we first formalise one iteration of the algorithm.

```

definition iteration:: "('position  $\Rightarrow$  'energy set)  $\Rightarrow$  ('position  $\Rightarrow$  'energy set)"
where
  "iteration F g  $\equiv$  (if g  $\in$  attacker
    then energy_Min {inv_upd (the (weight g g')) e' | e' g'.
      e'  $\in$  energies  $\wedge$  weight g g'  $\neq$  None  $\wedge$  e'  $\in$  F g'}
    else energy_Min {energy_sup
      {inv_upd (the (weight g g')) (e_index g') | g'.
        weight g g'  $\neq$  None} | e_index.  $\forall$ g'. weight g g'  $\neq$  None
 $\longrightarrow$  (e_index g')  $\in$  energies  $\wedge$  e_index g'  $\in$  F g'}})"

```

We now show that iteration is a Scott-continuous functor of possible Pareto fronts.

```

lemma iteration_pareto_functor:

```

```

assumes "F ∈ possible_pareto"
shows "iteration F ∈ possible_pareto"
unfolding possible_pareto_def
proof
  show "∀g. iteration F g ⊆ {e. e∈energies} ∧
    (∀e e'. e ∈ iteration F g ∧ e' ∈ iteration F g ∧ e ≠ e' → incomparable
    (e≤) e e'))"
  proof
    fix g
    show "iteration F g ⊆ {e. e∈energies} ∧
      (∀e e'. e ∈ iteration F g ∧ e' ∈ iteration F g ∧ e ≠ e' → incomparable
      (e≤) e e'))"
    proof
      show "iteration F g ⊆ {e. e∈energies}"
      proof
        fix e
        assume "e ∈ iteration F g"
        show "e ∈ {e. e∈energies}"
        proof
          show "e∈energies"
          proof(cases "g ∈ attacker")
            case True
            hence "e ∈ energy_Min {inv_upd (the (weight g g')) e' | e' g'. e'∈energies
            ∧ weight g g' ≠ None ∧ e' ∈ F g'})"
            using <e ∈ iteration F g> iteration_def by auto
            then show ?thesis using assms energy_Min_def
            using inv_well_defined by force
          next
            case False
            hence "e ∈ energy_Min {energy_sup {inv_upd (the (weight g g')) (e_index
            g') | g'. weight g g' ≠ None} | e_index. (∀g'. weight g g' ≠ None → ((e_index g')∈energies
            ∧ e_index g' ∈ F g'))}"
            using <e ∈ iteration F g> iteration_def by auto
            hence "e ∈ {energy_sup {inv_upd (the (weight g g')) (e_index g') | g'.
            weight g g' ≠ None} | e_index. (∀g'. weight g g' ≠ None → ((e_index g')∈energies
            ∧ e_index g' ∈ F g'))}"
            using energy_Min_def
            by simp
            from this obtain e_index where E: "e = energy_sup {inv_upd (the (weight
            g g')) (e_index g') | g'. weight g g' ≠ None}" and A: "(∀g'. weight g g' ≠ None
            → ((e_index g')∈energies ∧ e_index g' ∈ F g'))"
            by blast
            have fin: "finite {inv_upd (the (weight g g')) (e_index g') | g'. g'
            ∈ positions}" using finite_positions
            proof -
              have "finite {p. p ∈ positions}"
              using finite_positions by auto
              then show ?thesis
              using finite_image_set by fastforce
            qed
            have "{inv_upd (the (weight g g')) (e_index g') | g'. weight g g' ≠ None}
            ⊆ {inv_upd (the (weight g g')) (e_index g') | g'. g' ∈ positions}"
            by blast
            hence fin: "finite {inv_upd (the (weight g g')) (e_index g') | g'. weight
            g g' ≠ None}" using fin
            by (meson finite_subset)
          
```

```

      have "{inv_upd (the (weight g g')) (e_index g')) | g'. weight g g' ≠ None}
    ⊆ energies"
      proof
        fix x
        assume "x ∈ {inv_upd (the (weight g g')) (e_index g')) | g'. weight
g g' ≠ None}"
        from this obtain g' where "x=inv_upd (the (weight g g')) (e_index
g')" and "weight g g' ≠ None" by auto
        hence "(e_index g') ∈ energies ∧ e_index g' ∈ F g'" using A
        by blast
        thus "x ∈ energies" using inv_well_defined
        using <weight g g' ≠ None> <x = inv_upd (the (weight g g')) (e_index
g')> by blast
      qed
      then show ?thesis using bounded_join_semilattice fin E
      by meson
    qed
  qed
  qed
  show "(∀ e e'. e ∈ iteration F g ∧ e' ∈ iteration F g ∧ e ≠ e' → incomparable
(e ≤) e e')"
    using possible_pareto_def iteration_def energy_Min_def
    by (smt (verit) mem_Collect_eq)
  qed
  qed
  qed

lemma iteration_monotonic:
  assumes "F ∈ possible_pareto" and "F' ∈ possible_pareto" and "F ≤ F'"
  shows "iteration F ≤ iteration F'"
  unfolding pareto_order_def
proof
  fix g
  show "∀ e. e ∈ iteration F g → (∃ e'. e' ∈ iteration F' g ∧ e' ≤ e)"
proof
  fix e
  show "e ∈ iteration F g → (∃ e'. e' ∈ iteration F' g ∧ e' ≤ e)"
proof
  assume "e ∈ iteration F g"
  show "(∃ e'. e' ∈ iteration F' g ∧ e' ≤ e)"
  proof (cases "g ∈ attacker")
    case True
    hence "e ∈ energy_Min {inv_upd (the (weight g g')) e' | e' g'. e' ∈ energies
  ∧ weight g g' ≠ None ∧ e' ∈ F g'}"
    using iteration_def <e ∈ iteration F g> by simp
    from this obtain e' g' where E: "e = inv_upd (the (weight g g')) e' ∧ e'
  ∈ energies ∧ weight g g' ≠ None ∧ e' ∈ F g'"
    using energy_Min_def by auto
    hence "∃ e''. e'' ∈ F' g' ∧ e'' ≤ e'" using pareto_order_def assms by simp
    from this obtain e'' where "e'' ∈ F' g' ∧ e'' ≤ e'" by auto

    have "F' g' ⊆ {e. e ∈ energies}" using assms(2) unfolding possible_pareto_def
    by simp
    hence E'': "e'' ∈ energies" using <e'' ∈ F' g' ∧ e'' ≤ e'>
    by auto

```

```

      have uE: "inv_upd (the (weight g g')) e'' e ≤ inv_upd (the (weight g g'))
e'"
      proof(rule inverse_monotonic)
        show "weight g g' ≠ None"
          by (simp add: E)
        show "e'' e ≤ e'" using <e'' ∈ F' g' ∧ e'' e ≤ e'> by simp
        show "e'' ∈ energies" using E''.
        thus "inverse_application (the (weight g g')) e'' ≠ None"
          using <weight g g' ≠ None> inv_well_defined
          by auto
      qed
      hence "inv_upd (the (weight g g')) e'' ∈ {inv_upd (the (weight g g')) e'
| e' g'. e' ∈ energies ∧ weight g g' ≠ None ∧ e' ∈ F' g'}"
        using E'' <e'' ∈ F' g' ∧ e'' e ≤ e'> E
        by auto
      hence "∃e'''. e''' ∈ energy_Min {inv_upd (the (weight g g')) e' | e' g'.
e' ∈ energies ∧ weight g g' ≠ None ∧ e' ∈ F' g'} ∧ e''' e ≤ inv_upd (the (weight
g g')) e'"
        using energy_Min_contains_smaller
        by (smt (verit, del_insts) inv_well_defined mem_Collect_eq subset_iff)
      hence "∃e'''. e''' ∈ iteration F' g ∧ e''' e ≤ inv_upd (the (weight g g'))
e'"
        unfolding iteration_def using True by simp
      from this obtain e''' where E''': "e''' ∈ iteration F' g ∧ e''' e ≤ inv_upd
(the (weight g g')) e'" by auto
      hence "e''' e ≤ e" using E uE energy_order
        by (smt (verit, ccfv_threshold) E'' assms(2) energy_wqo galois_energy_game_decidable.
galois_energy_game_decidable_axioms in_mono inv_well_defined iteration_pareto_functor
mem_Collect_eq transp_onD wqo_on_imp_transp_on)
      then show ?thesis using E''' by auto
    next
      case False
      hence "e ∈ (energy_Min {energy_sup {inv_upd (the (weight g g')) (e_index
g') | g'. weight g g' ≠ None} | e_index. (∀g'. weight g g' ≠ None → ((e_index
g') ∈ energies ∧ e_index g' ∈ F' g'))))"
        using iteration_def <e ∈ iteration F' g> by simp
      from this obtain e_index where E: "e = energy_sup {inv_upd (the (weight g
g')) (e_index g') | g'. weight g g' ≠ None}" and "(∀g'. weight g g' ≠ None → ((e_index
g') ∈ energies ∧ e_index g' ∈ F' g'))"
        using energy_Min_def by auto
      hence "∧g'. weight g g' ≠ None ⇒ ∃e'. e' ∈ F' g' ∧ e' e ≤ e_index g'"
        using assms(3) pareto_order_def by force
      define e_index' where "e_index' ≡ (λg'. (SOME e'. (e' ∈ F' g' ∧ e' e ≤
e_index g')))"
      hence E': "∧g'. weight g g' ≠ None ⇒ e_index' g' ∈ F' g' ∧ e_index'
g' e ≤ e_index g'"
        using <∧g'. weight g g' ≠ None ⇒ ∃e'. e' ∈ F' g' ∧ e' e ≤ e_index
g'> some_eq_ex
        by (metis (mono_tags, lifting))
      hence "∧g'. weight g g' ≠ None ⇒ inv_upd (the (weight g g')) (e_index'
g') e ≤ inv_upd (the (weight g g')) (e_index g')"
        using inverse_monotonic
        using <∀g'. weight g g' ≠ None → (e_index g') ∈ energies ∧ e_index
g' ∈ F' g'>
        using inv_well_defined energy_order
        by (smt (verit) Collect_mem_eq assms(2) galois_energy_game_decidable.possible_pareto_

```



```

galois_energy_game_decidable_axioms mem_Collect_eq subsetD)
  hence leq: "\a. a ∈ {inv_upd (the (weight g g')) (e_index' g') | g'. weight
g g' ≠ None} ⇒ ∃b. b ∈ {inv_upd (the (weight g g')) (e_index g') | g'. weight
g g' ≠ None} ∧ a ≤ b"
  by blast
  have len: "\a. a ∈ {inv_upd (the (weight g g')) (e_index' g') | g'. weight
g g' ≠ None} ⇒ a ∈ energies"
  using E' E inv_well_defined
  using <∀g'. weight g g' ≠ None → (e_index g') ∈ energies ∧ e_index
g' ∈ F g'> energy_order
  using assms(2) galois_energy_game_decidable.possible_pareto_def galois_energy_game_de
in_mono by blast
  hence leq: "energy_sup {inv_upd (the (weight g g')) (e_index' g') | g'. weight
g g' ≠ None} ≤ energy_sup {inv_upd (the (weight g g')) (e_index g') | g'. weight
g g' ≠ None}"
  proof (cases "{g'. weight g g' ≠ None} = {}")
    case True
      hence "{inv_upd (the (weight g g')) (e_index' g') | g'. weight g g' ≠ None}
= {} ∧ {inv_upd (the (weight g g')) (e_index g') | g'. weight g g' ≠ None} = {}"
      by simp
      then show ?thesis
      by (simp add: bounded_join_semilattice)
    next
      case False
        have in_energy: "{inv_upd (the (weight g g')) (e_index g') | g'. weight
g g' ≠ None} ⊆ energies"
        using <∀g'. weight g g' ≠ None → e_index g' ∈ energies ∧ e_index
g' ∈ F g'> inv_well_defined by blast
        have fin: "finite {inv_upd (the (weight g g')) (e_index' g') | g'. weight
g g' ≠ None} ∧ finite {inv_upd (the (weight g g')) (e_index g') | g'. weight g g'
≠ None}"
        proof
          have "{inv_upd (the (weight g g')) (e_index' g') | g'. weight g g' ≠
None} ⊆ {inv_upd (the (weight g g')) (e_index' g') | g'. g' ∈ positions}"
          by auto
          thus "finite {inv_upd (the (weight g g')) (e_index' g') | g'. weight
g g' ≠ None}"
          using finite_positions
          using rev_finite_subset by fastforce
          have "{inv_upd (the (weight g g')) (e_index g') | g'. weight g g' ≠ None}
⊆ {inv_upd (the (weight g g')) (e_index g') | g'. g' ∈ positions}"
          by auto
          thus "finite {inv_upd (the (weight g g')) (e_index g') | g'. weight g
g' ≠ None}"
          using finite_positions
          using rev_finite_subset by fastforce
        qed
        from False have "{inv_upd (the (weight g g')) (e_index' g') | g'. weight
g g' ≠ None} ≠ {}" by simp
        then show ?thesis using energy_sup_leq_energy_sup len leq fin in_energy
        by meson
      qed
  qed

```

```

      have "\g'. weight g g' ≠ None ⇒ (e_index' g') ∈ energies" using E' <∀g'.
weight g g' ≠ None ⇒ (e_index g') ∈ energies ∧ e_index g' ∈ F g'>
      using assms(2) galois_energy_game_decidable.possible_pareto_def galois_energy_game_de
in_mono by blast
      hence "energy_sup {inv_upd (the (weight g g')) (e_index' g') | g'. weight
g g' ≠ None} ∈ {energy_sup {inv_upd (the (weight g g')) (e_index g') | g'. weight
g g' ≠ None} | e_index. (∀g'. weight g g' ≠ None ⇒ ((e_index g') ∈ energies ∧
e_index g' ∈ F' g'))}"
      using E'
      by blast
      hence "∃e'. e' ∈ energy_Min {energy_sup {inv_upd (the (weight g g')) (e_index
g') | g'. weight g g' ≠ None} | e_index. (∀g'. weight g g' ≠ None ⇒ ((e_index g')
∈ energies ∧ e_index g' ∈ F' g'))}
      ∧ e' ≤ energy_sup {inv_upd (the (weight g g')) (e_index' g') | g'.
weight g g' ≠ None}"
      using energy_Min_contains_smaller
      proof -
        obtain ee :: "'energy ⇒ 'energy set ⇒ 'energy" and eea :: "'energy ⇒
'energy set ⇒ 'energy" where
          f1: "∀e E. ee e E e ≤ e ∧ ee e E ∈ energy_Min E ∨ ¬ E ⊆ energies ∨
e ∉ E"
        using energy_Min_contains_smaller by moura
        have "finite ({}::'energy set)"
        by blast
        have in_energy: "\f. ∀p. weight g p ≠ None ⇒ f p ∈ energies ∧ f p
∈ F' p ⇒ {inv_upd (the (weight g p)) (f p) | p. weight g p ≠ None} ⊆ energies"
        using inv_well_defined by blast
        have "\f. ∀p. weight g p ≠ None ⇒ f p ∈ energies ∧ f p ∈ F' p ⇒
finite {inv_upd (the (weight g p)) (f p) | p. weight g p ≠ None}"
        proof-
          fix f
          have "{inv_upd (the (weight g p)) (f p) | p. weight g p ≠ None} ⊆ {inv_upd
(the (weight g p)) (f p) | p. p ∈ positions}" by auto
          thus "∀p. weight g p ≠ None ⇒ f p ∈ energies ∧ f p ∈ F' p ⇒ finite
{inv_upd (the (weight g p)) (f p) | p. weight g p ≠ None}" using finite_positions
          by (simp add: rev_finite_subset)
        qed
        then have "{energy_sup {inv_upd (the (weight g p)) (f p) | p. weight g
p ≠ None} | f. ∀p. weight g p ≠ None ⇒ f p ∈ energies ∧ f p ∈ F' p} ⊆ energies"
        using in_energy bounded_join_semilattice
        by force
        then show ?thesis
        using f1 <energy_sup {inv_upd (the (weight g g')) (e_index' g') | g'.
weight g g' ≠ None} ∈ {energy_sup {inv_upd (the (weight g g')) (e_index g') | g'.
weight g g' ≠ None} | e_index. ∀g'. weight g g' ≠ None ⇒ e_index g' ∈ energies
∧ e_index g' ∈ F' g'}> by blast
      qed
      hence "∃e'. e' ∈ iteration F' g ∧ e' ≤ energy_sup {inv_upd (the (weight
g g')) (e_index' g') | g'. weight g g' ≠ None} "
      unfolding iteration_def using False by auto
      from this obtain e' where "e' ∈ iteration F' g" and "e' ≤ energy_sup {inv_upd
(the (weight g g')) (e_index' g') | g'. weight g g' ≠ None} " by auto
      hence "e' ≤ energy_sup {inv_upd (the (weight g g')) (e_index g') | g'.
weight g g' ≠ None}"
      using leq_energy_order ordering_def
      by (metis (no_types, lifting) partial_preordering.trans)

```



```

qed
qed

lemma iteration_scott_continuous_vanilla:
  assumes "P ⊆ possible_pareto" and
    " $\bigwedge F F'. F \in P \implies F' \in P \implies \exists F''. F'' \in P \wedge F \preceq F'' \wedge F' \preceq F''$ " and
    "P ≠ {}"
  shows "iteration (pareto_sup P) = pareto_sup {iteration F | F. F ∈ P}"
proof(rule antisymmetry)
  from assms have "(pareto_sup P) ∈ possible_pareto" using assms pareto_sup_is_sup
  by simp
  thus A: "iteration (pareto_sup P) ∈ possible_pareto" using iteration_pareto_functor
  by simp

  have B: "{iteration F | F. F ∈ P} ⊆ possible_pareto"
  proof
    fix F
    assume "F ∈ {iteration F | F. F ∈ P}"
    from this obtain F' where "F = iteration F'" and "F' ∈ P" by auto
    thus "F ∈ possible_pareto" using iteration_pareto_functor
    using assms by auto
  qed
  thus "pareto_sup {iteration F | F. F ∈ P} ∈ possible_pareto" using pareto_sup_is_sup
  by simp

  show "iteration (pareto_sup P) ≤ pareto_sup {iteration F | F. F ∈ P}"
  unfolding pareto_order_def proof
    fix g
    show "∀ e. e ∈ iteration (pareto_sup P) g ⟶
      (∃ e'. e' ∈ pareto_sup {iteration F | F. F ∈ P} g ∧ e' ≤ e)"
  proof
    fix e
    show "e ∈ iteration (pareto_sup P) g ⟶
      (∃ e'. e' ∈ pareto_sup {iteration F | F. F ∈ P} g ∧ e' ≤ e)"
  proof
    assume "e ∈ iteration (pareto_sup P) g"
    show "∃ e'. e' ∈ pareto_sup {iteration F | F. F ∈ P} g ∧ e' ≤ e"
    proof(cases "g ∈ attacker")
      case True
      hence "e ∈ energy_Min {inv_upd (the (weight g g')) e' | e' g'. e' ∈ energies
        ∧ weight g g' ≠ None ∧ e' ∈ (pareto_sup P) g'}"
      using iteration_def <e ∈ iteration (pareto_sup P) g> by auto
      from this obtain e' g' where "e = inv_upd (the (weight g g')) e'" and
        "e' ∈ energies ∧ weight g g' ≠ None ∧ e' ∈ (pareto_sup P) g'"
      using energy_Min_def by auto
      hence "∃ F. F ∈ P ∧ e' ∈ F g'" using pareto_sup_def energy_Min_def by simp
      from this obtain F where "F ∈ P ∧ e' ∈ F g'" by auto
      hence E: "e ∈ {inv_upd (the (weight g g')) e' | e' g'. e' ∈ energies
        ∧ weight g g' ≠ None ∧ e' ∈ F g'}" using <e = inv_upd (the (weight g g')) e'>
      using <e' ∈ energies ∧ weight g g' ≠ None ∧ e' ∈ pareto_sup P g'>
    by blast

    have "{inv_upd (the (weight g g')) e' | e' g'. e' ∈ energies ∧ weight g
      g' ≠ None ∧ e' ∈ F g'} ⊆ energies"
    using inv_well_defined by blast
    hence "∃ e''. e'' ∈ energy_Min {inv_upd (the (weight g g')) e' | e' g'.

```

```

e' ∈ energies ∧ weight g g' ≠ None ∧ e' ∈ F g' ∧ e'' ≤ e"
  using energy_Min_contains_smaller E
  by meson
hence "∃e''. e'' ∈ iteration F g ∧ e'' ≤ e" using True iteration_def
by simp
  from this obtain e'' where "e'' ∈ iteration F g ∧ e'' ≤ e" by auto
  have "∃e''' ∈ pareto_sup {iteration F | F. F ∈ P} g. e''' ≤ e'"
    unfolding pareto_sup_def proof(rule energy_Min_contains_smaller)
    show "e'' ∈ {e. ∃F. F ∈ {iteration F | F. F ∈ P} ∧ e ∈ F g}"
      using <e'' ∈ iteration F g ∧ e'' ≤ e>
      using <F ∈ P ∧ e' ∈ F g'> by blast
    show "{e. ∃F. F ∈ {iteration F | F. F ∈ P} ∧ e ∈ F g} ⊆ energies"
    proof
      fix x
      assume X: "x ∈ {e. ∃F. F ∈ {iteration F | F. F ∈ P} ∧ e ∈ F g}"
      from this obtain F where "F ∈ {iteration F | F. F ∈ P} ∧ x ∈ F g"
by auto
      from this obtain F' where "F = iteration F'" and "F' ∈ P" by auto
      hence "F ∈ possible_pareto" using assms
        using iteration_pareto_functor by auto
      thus "x ∈ energies" unfolding possible_pareto_def using X
        using <F ∈ {iteration F | F. F ∈ P} ∧ x ∈ F g> by blast
    qed
  qed
  then show ?thesis
    using <e'' ∈ iteration F g ∧ e'' ≤ e> energy_order ordering_def
    by (metis (mono_tags, lifting) partial_preordering_def)
next
  case False
  hence "e ∈ energy_Min {energy_sup {inv_upd (the (weight g g')) (e_index
g') | g'. weight g g' ≠ None} | e_index. (∀g'. weight g g' ≠ None → ((e_index g')
∈ energies ∧ e_index g' ∈ (pareto_sup P) g'))}"
    using iteration_def <e ∈ iteration (pareto_sup P) g> by auto
  from this obtain e_index where "e = energy_sup {inv_upd (the (weight g
g')) (e_index g') | g'. weight g g' ≠ None}" and "(∀g'. weight g g' ≠ None → (
(e_index g') ∈ energies ∧ e_index g' ∈ (pareto_sup P) g'))"
    using energy_Min_def by auto
  hence "∧g'. weight g g' ≠ None ⇒ e_index g' ∈ (pareto_sup P) g'" by
auto
  hence "∧g'. weight g g' ≠ None ⇒ ∃F'. F' ∈ P ∧ e_index g' ∈ F' g'"
using pareto_sup_def energy_Min_def
  by (simp add: mem_Collect_eq)
  define F_index where "F_index ≡ ∧g'. SOME F'. F' ∈ P ∧ e_index g' ∈ F'
g'"
  hence Fg: "∧g'. weight g g' ≠ None ⇒ F_index g' ∈ P ∧ e_index g' ∈
F_index g' g'"
    using <∧g'. weight g g' ≠ None ⇒ ∃F'. F' ∈ P ∧ e_index g' ∈ F'
g'> some_eq_ex
    by (smt (verit))

  have "∃F'. F' ∈ P ∧ (∀F. F ∈ {F_index g' | g'. weight g g' ≠ None} →
F ≤ F')"
  proof(rule finite_directed_set_upper_bound)
    show "∧F F'. F ∈ P ⇒ F' ∈ P ⇒ ∃F''. F'' ∈ P ∧ F ≤ F'' ∧ F' ≤
F''" using assms by simp
    show "P ≠ {}" using assms by simp

```

```

show "{F_index g' | g'. weight g g' ≠ None} ⊆ P"
  using Fg
  using subsetI by auto
have "finite {g'. weight g g' ≠ None}" using finite_positions
  by (metis Collect_mono finite_subset)
thus "finite {F_index g' | g'. weight g g' ≠ None}" by auto
show "P ⊆ possible_pareto" using assms by simp
qed
from this obtain F where F: "F ∈ P ∧ (∀g'. weight g g' ≠ None → F_index
g' ≤ F)" by auto
hence "F ∈ possible_pareto" using assms by auto
have "∧g'. weight g g' ≠ None ⇒ ∃e'. e' ∈ F g' ∧ e' e ≤ e_index g'"
proof-
  fix g'
  assume "weight g g' ≠ None"
  hence "e_index g' ∈ F_index g' g'" using Fg by auto
  have "F_index g' ≤ F" using F <weight g g' ≠ None> by auto
  thus "∃e'. e' ∈ F g' ∧ e' e ≤ e_index g'" unfolding pareto_order_def
    using <e_index g' ∈ F_index g' g'> by fastforce
qed

define e_index' where "e_index' ≡ λg'. SOME e'. e' ∈ F g' ∧ e' e ≤ e_index
g'"
hence "∧g'. weight g g' ≠ None ⇒ e_index' g' ∈ F g' ∧ e_index' g' e ≤
e_index g'"
  using <∧g'. weight g g' ≠ None ⇒ ∃e'. e' ∈ F g' ∧ e' e ≤ e_index
g'> some_eq_ex by (smt (verit))
hence "energy_sup {inv_upd (the (weight g g')) (e_index' g') | g'. weight
g g' ≠ None} e ≤ energy_sup {inv_upd (the (weight g g')) (e_index g') | g'. weight
g g' ≠ None}"
proof(cases "{g'. weight g g' ≠ None} = {}")
case True
hence "{inv_upd (the (weight g g')) (e_index' g') | g'. weight g g' ≠
None} = {}" by simp
have "{inv_upd (the (weight g g')) (e_index g') | g'. weight g g' ≠ None}
= {}" using True by simp
then show ?thesis unfolding energy_order using <{inv_upd (the (weight
g g')) (e_index' g') | g'. weight g g' ≠ None} = {}>
  using energy_order ordering.eq_iff by fastforce
next
case False
show ?thesis
proof(rule energy_sup_leq_energy_sup)
show "{inv_upd (the (weight g g')) (e_index' g') | g'. weight g g'
≠ None} ≠ {}"
  using False by simp
show "∧a. a ∈ {inv_upd (the (weight g g')) (e_index' g') | g'. weight
g g' ≠ None} ⇒
  ∃b∈{inv_upd (the (weight g g')) (e_index g') | g'. weight g
g' ≠ None}. a e ≤ b"
  proof-
    fix a
    assume "a ∈ {inv_upd (the (weight g g')) (e_index' g') | g'. weight
g g' ≠ None}"
    from this obtain g' where "a=inv_upd (the (weight g g')) (e_index'
g')" and "weight g g' ≠ None" by auto

```

```

      have "(e_index' g') e ≤ (e_index' g')"
      using <weight g g' ≠ None> <∧g'. weight g g' ≠ None ⇒ e_index'
g' ∈ F g' ∧ e_index' g' e ≤ e_index g'>
      by (meson energy_order ordering.eq_iff)
      have "(e_index' g') ∈ energies"
      using <∧g'. weight g g' ≠ None ⇒ e_index' g' ∈ F g' ∧ e_index'
g' e ≤ e_index g'> possible_pareto_def <weight g g' ≠ None> F assms
      by blast
      hence "a e ≤ inv_upd (the (weight g g')) (e_index' g')"
      using <a=inv_upd (the (weight g g')) (e_index' g')> <(e_index'
g') e ≤ (e_index' g')> inverse_monotonic <weight g g' ≠ None>
      using inv_well_defined by presburger
      hence "a e ≤ inv_upd (the (weight g g')) (e_index g')"
      using <∧g'. weight g g' ≠ None ⇒ e_index' g' ∈ F g' ∧ e_index'
g' e ≤ e_index g'>
      using <a = inv_upd (the (weight g g')) (e_index' g')> <e_index'
g' ∈ energies> <weight g g' ≠ None> inv_well_defined inverse_monotonic by blast
      thus "∃b∈{inv_upd (the (weight g g')) (e_index g') | g'. weight
g g' ≠ None}. a e ≤ b"
      using <weight g g' ≠ None> by blast
    qed
    show "∧a. a ∈ {inv_upd (the (weight g g')) (e_index' g') | g'. weight
g g' ≠ None} ⇒
      a ∈ energies"
  proof-
    fix a
    assume "a ∈ {inv_upd (the (weight g g')) (e_index' g') | g'. weight
g g' ≠ None}"
    from this obtain g' where "a=inv_upd (the (weight g g')) (e_index'
g')" and "weight g g' ≠ None" by auto
    hence "e_index' g' ∈ F g'" using <∧g'. weight g g' ≠ None ⇒ e_index'
g' ∈ F g' ∧ e_index' g' e ≤ e_index g'>
    by simp
    hence "(e_index' g') ∈ energies" using <F ∈ possible_pareto> possible_pareto_def
    by blast
    thus "a ∈ energies" using <a=inv_upd (the (weight g g')) (e_index'
g')> <weight g g' ≠ None>
    using inv_well_defined by blast
  qed
  have "{inv_upd (the (weight g g')) (e_index' g') | g'. weight g g'
≠ None} ⊆ {inv_upd (the (weight g g')) (e_index' g') | g'. g' ∈ positions}" by auto
  thus "finite {inv_upd (the (weight g g')) (e_index' g') | g'. weight
g g' ≠ None}"
    using finite_positions
    using rev_finite_subset by fastforce
  have "{inv_upd (the (weight g g')) (e_index g') | g'. weight g g' ≠
None} ⊆ {inv_upd (the (weight g g')) (e_index g') | g'. g' ∈ positions}" by auto
  thus "finite {inv_upd (the (weight g g')) (e_index g') | g'. weight
g g' ≠ None}"
    using finite_positions
    using rev_finite_subset by fastforce
  show "{inv_upd (the (weight g g')) (e_index g') | g'. weight g g' ≠
None} ⊆ energies"
    using <∀g'. weight g g' ≠ None → e_index g' ∈ energies ∧ e_index
g' ∈ pareto_sup P g'> inv_well_defined by blast
  qed

```

```

qed
hence leq: "energy_sup {inv_upd (the (weight g g')) (e_index' g') | g'.
weight g g' ≠ None} e ≤ e"
using <e= energy_sup {inv_upd (the (weight g g')) (e_index g') | g'.
weight g g' ≠ None}> by simp

have in_energies: "{energy_sup {inv_upd (the (weight g g')) (e_index g')
|g'. weight g g' ≠ None} |e_index. ∀g'. weight g g' ≠ None → e_index g' ∈ energies
∧ e_index g' ∈ F g'} ⊆ energies"
proof
fix x
assume "x ∈ {energy_sup {inv_upd (the (weight g g')) (e_index g') |g'.
weight g g' ≠ None} |e_index. ∀g'. weight g g' ≠ None → e_index g' ∈ energies
∧ e_index g' ∈ F g'}"
from this obtain e_index where X: "x = energy_sup {inv_upd (the (weight
g g')) (e_index g') |g'. weight g g' ≠ None}" and "∀g'. weight g g' ≠ None →
e_index g' ∈ energies ∧ e_index g' ∈ F g'" by auto
have "{inv_upd (the (weight g g')) (e_index g') |g'. weight g g' ≠ None}
⊆ {inv_upd (the (weight g g')) (e_index g') |g'. g' ∈ positions}" by auto
hence fin: "finite {inv_upd (the (weight g g')) (e_index g') |g'. weight
g g' ≠ None}" using finite_positions
using rev_finite_subset by fastforce
have "{inv_upd (the (weight g g')) (e_index g') |g'. weight g g' ≠ None}
⊆ energies"
using <∀g'. weight g g' ≠ None → e_index g' ∈ energies ∧ e_index
g' ∈ F g'> inv_well_defined by force
thus "x ∈ energies" unfolding X using bounded_join_semilattice fin
by meson
qed
have in_energies2: "{e. ∃F. (F ∈ {iteration F |F. F ∈ P} ∧ e ∈ F g)}
⊆ energies"
using assms unfolding possible_pareto_def
by (smt (verit) B mem_Collect_eq possible_pareto_def subset_iff)
have "∧g'. weight g g' ≠ None ⇒ e_index' g' ∈ F g'" using <∧g'. weight
g g' ≠ None ⇒ e_index' g' ∈ F g' ∧ e_index' g' e ≤ e_index g'>
by simp
hence "∧g'. weight g g' ≠ None ⇒ (e_index' g') ∈ energies" using <F
∈ possible_pareto> possible_pareto_def
by blast
hence "(energy_sup {inv_upd (the (weight g g')) (e_index' g') | g'. weight
g g' ≠ None}) ∈ {energy_sup
{inv_upd (the (weight g g')) (e_index g') |g'. weight g g' ≠ None} |
e_index.
∀g'. weight g g' ≠ None → (e_index g') ∈ energies ∧ e_index g' ∈
F g'}"
using <∧g'. weight g g' ≠ None ⇒ e_index' g' ∈ F g' ∧ e_index' g'
e ≤ e_index g'> by auto
hence "∃e'. e' ∈ iteration F g ∧ e' e ≤ (energy_sup {inv_upd (the (weight
g g')) (e_index' g') | g'. weight g g' ≠ None})"
unfolding iteration_def using energy_Min_contains_smaller False in_energies
by meson
from this obtain e' where E': "e' ∈ iteration F g ∧ e' e ≤ (energy_sup
{inv_upd (the (weight g g')) (e_index' g') | g'. weight g g' ≠ None})"
by auto
hence "e' ∈ {(e::'energy). (∃F. F ∈ {iteration F |F. F ∈ P} ∧ e ∈ (F
g))}" using F by auto

```



```

      hence "∃a. a ∈ pareto_sup {iteration F | F. F ∈ P} g ∧ a ≤ e'"
      unfolding pareto_sup_def using energy_Min_contains_smaller in_energies2
by meson
      from this obtain a where "a ∈ pareto_sup {iteration F | F. F ∈ P} g ∧
a ≤ e'" by auto
      hence "a ≤ e" using E' leq energy_order ordering_def
      by (metis (no_types, lifting) partial_preordering.trans)
      then show ?thesis using <a ∈ pareto_sup {iteration F | F. F ∈ P} g ∧ a
e ≤ e'> by auto
      qed
      qed
      qed
      qed

show "pareto_sup {iteration F | F. F ∈ P} ≤ iteration (pareto_sup P)"
proof(rule pareto_sup_is_sup(3))
  show "{iteration F | F. F ∈ P} ⊆ possible_pareto" using B by simp
  show "iteration (pareto_sup P) ∈ possible_pareto" using A by simp
  show "∧F. F ∈ {iteration F | F. F ∈ P} ⇒ F ≤ iteration (pareto_sup P)"
  proof-
    fix F
    assume "F ∈ {iteration F | F. F ∈ P}"
    from this obtain F' where "F = iteration F'" and "F' ∈ P" by auto
    hence "F' ≤ pareto_sup P" using pareto_sup_is_sup
    by (simp add: assms)
    thus "F ≤ iteration (pareto_sup P)" using <F = iteration F'> iteration_monotonic
assms
    by (simp add: <F' ∈ P> <pareto_sup P ∈ possible_pareto> subsetD)
    qed
    qed
    qed

lemma iteration_scott_continuous:
  shows "scott_continuous possible_pareto (≤) possible_pareto (≤) iteration"
proof
  show "iteration ' possible_pareto ⊆ possible_pareto"
  using iteration_pareto_functor
  by blast
  show "∧X s. directed_set X (≤) ⇒
X ≠ {} ⇒
X ⊆ possible_pareto ⇒
extreme_bound possible_pareto (≤) X s ⇒
extreme_bound possible_pareto (≤) (iteration ' X) (iteration s)"
  proof-
    fix P s
    assume A1: "directed_set P (≤)" and A2: "P ≠ {}" and A3: "P ⊆ possible_pareto"
and
      A4: "extreme_bound possible_pareto (≤) P s"
    hence A4: "s = pareto_sup P" unfolding extreme_bound_def using pareto_sup_is_sup
    by (metis (no_types, lifting) A4 antisymmetry extreme_bound_iff)

    from A1 have A1: "∧F F'. F ∈ P ⇒ F' ∈ P ⇒ ∃F''. F'' ∈ P ∧ F ≤ F'' ∧ F'
≤ F''"
    unfolding directed_set_def
    by (metis A1 directedD2)

```

```

hence "iteration s = pareto_sup {iteration F | F. F ∈ P}"
  using iteration_scott_continuous_vanilla A2 A3 A4 finite_positions
  by blast

show "extreme_bound possible_pareto (≤) (iteration ' P) (iteration s)"
  unfolding <iteration s = pareto_sup {iteration F | F. F ∈ P}> extreme_bound_def
proof
  have A3: "{iteration F | F. F ∈ P} ⊆ possible_pareto"
    using iteration_pareto_functor A3
    by auto

  thus "pareto_sup {iteration F | F. F ∈ P} ∈ {b ∈ possible_pareto. bound (iteration
' P) (≤) b}"
    using pareto_sup_is_sup
    by (simp add: Setcompr_eq_image bound_def)

  show "∧x. x ∈ {b ∈ possible_pareto. bound (iteration ' P) (≤) b} ⇒
    pareto_sup {iteration F | F. F ∈ P} ≤ x"
    using A3 pareto_sup_is_sup
    by (smt (verit, del_insts) bound_def image_eqI mem_Collect_eq)
qed
qed
qed

We now show that a_win_min is a fixed point of iteration.

lemma a_win_min_is_fp:
  shows "iteration a_win_min = a_win_min"
proof
  have minimal_winning_budget_attacker: "∧g e. g ∈ attacker ⇒ minimal_winning_budget
e g = (e ∈ energy_Min {e. ∃g' e'. weight g g' ≠ None ∧ minimal_winning_budget e'
g' ∧ e=(the (inverse_application (the (weight g g')) e'))})"
  proof-
    fix g e
    assume "g ∈ attacker" <g ∈ attacker>
    have attacker_inv_in_winning_budget: "∧g g' e'. g ∈ attacker ⇒ weight g g'
≠ None ⇒ winning_budget_len e' g' ⇒ winning_budget_len (inv_upd (the (weight
g g')) e') g"
    proof-
      fix g g' e'
      assume A1: "g ∈ attacker" and A2: "weight g g' ≠ None" and A3: "winning_budget_len
e' g'"
      show "winning_budget_len (inv_upd (the (weight g g')) e') g"
      proof
        from A3 have "e' ∈ energies" using winning_budget_len.simps
        by blast
        show "(the (inverse_application (the (weight g g')) e')) ∈ energies ∧ g
∈ attacker ∧
          (∃g'a. weight g g'a ≠ None ∧
            application (the (weight g g'a)) (the (inverse_application (the (weight
g g')) e')) ≠ None ∧
            winning_budget_len (the (application (the (weight g g'a)) (the (inverse_application
(the (weight g g')) e')))) g'a)"
        proof
          show "(the (inverse_application (the (weight g g')) e')) ∈ energies" using

```

```

<e' ∈ energies> A2
  using inv_well_defined by blast
  show "g ∈ attacker ∧
    (∃g'a. weight g g'a ≠ None ∧
      application (the (weight g g'a)) (the (inverse_application (the (weight
g g')) e')) ≠ None ∧
        winning_budget_len (the (application (the (weight g g'a)) (the (inverse_application
(the (weight g g')) e')))) g'a) "
  proof
    show "g ∈ attacker" using A1 .
    show "∃g'a. weight g g'a ≠ None ∧
      application (the (weight g g'a)) (the (inverse_application (the (weight
g g')) e')) ≠ None ∧
        winning_budget_len (the (application (the (weight g g'a)) (the (inverse_application
(the (weight g g')) e')))) g'a"
  proof
    show "weight g g' ≠ None ∧
      application (the (weight g g')) (the (inverse_application (the (weight
g g')) e')) ≠ None ∧
        winning_budget_len (the (application (the (weight g g')) (the (inverse_applicat
(the (weight g g')) e')))) g'"
  proof
    show "weight g g' ≠ None" using A2 .
    show "application (the (weight g g')) (the (inverse_application
(the (weight g g')) e')) ≠ None ∧
      winning_budget_len (the (application (the (weight g g')) (the
(inverse_application (the (weight g g')) e')))) g'"
  proof
    from A1 A2 show "application (the (weight g g')) (the (inverse_application
(the (weight g g')) e')) ≠ None" using inv_well_defined
    by (simp add: <e' ∈ energies>)
    have "order e' (the (application (the (weight g g')) (the (inverse_application
(the (weight g g')) e'))))" using upd_inv_increasing
    using A2 <e' ∈ energies> by blast
    thus "winning_budget_len (the (application (the (weight g g'))
(the (inverse_application (the (weight g g')) e')))) g'" using upwards_closure_wb_len
    using A3 by auto
  qed
qed
qed
qed
qed
qed
qed
qed

  have min_winning_budget_is_inv_a: "∧e g. g ∈ attacker ⇒ minimal_winning_budget
e g ⇒ ∃g' e'. weight g g' ≠ None ∧ winning_budget_len e' g' ∧ e = (inv_upd (the
(weight g g')) e'"
  proof-
    fix e g
    assume A1: "g ∈ attacker" and A2: "minimal_winning_budget e g"
    show "∃g' e'. weight g g' ≠ None ∧ winning_budget_len e' g' ∧ e = (inv_upd
(the (weight g g')) e'"
  proof-
    from A1 A2 have "winning_budget_len e g" using energy_Min_def by simp
    hence <e ∈ energies> using winning_budget_len.simps by blast

```

```

    from A1 A2 <winning_budget_len e g> have " ( $\exists g'. (\text{weight } g \ g' \neq \text{None}) \wedge$ 
(application (the (weight g g')) e) $\neq$  None  $\wedge$  (winning_budget_len (the (application
(the (weight g g')) e)) g') )"
    using winning_budget_len.simps
    by blast
    from this obtain g' where G: "(weight g g'  $\neq$  None)  $\wedge$  (application (the
(weight g g')) e) $\neq$  None  $\wedge$  (winning_budget_len (the (application (the (weight g
g')) e)) g'))" by auto
    hence "(the (application (the (weight g g')) e))  $\in$  energies"
    using <e  $\in$  energies> upd_well_defined by blast
    hence W: "winning_budget_len (the (inverse_application (the (weight g g'))
(the (application (the (weight g g')) e)))) g" using G attacker_inv_in_winning_budget
    by (meson A1)
    have "order (the (inverse_application (the (weight g g')) (the (application
(the (weight g g')) e)))) e" using inv_upd_decreasing
    using G
    using <e  $\in$  energies> by blast
    hence E: "e = (the (inverse_application (the (weight g g')) (the (application
(the (weight g g')) e))))" using W A1 A2 energy_Min_def
    by auto
    show ?thesis
    proof
    show " $\exists e'. \text{weight } g \ g' \neq \text{None} \wedge \text{winning\_budget\_len } e' \ g' \wedge e = \text{the (inverse\_applicat$ 
(the (weight g g')) e'))"
    proof
    show "weight g g'  $\neq$  None  $\wedge$  winning_budget_len (the (application (the
(weight g g')) e)) g'  $\wedge$  e = the (inverse_application (the (weight g g')) (the (application
(the (weight g g')) e))))"
    using G E by simp
    qed
    qed
    qed
    qed

    have min_winning_budget_a_iff_energy_Min: "minimal_winning_budget e g
 $\longleftrightarrow$  e  $\in$  energy_Min {e.  $\exists g' \ e'. \text{weight } g \ g' \neq \text{None} \wedge \text{winning\_budget\_len } e' \ g' \wedge$ 
e=(inv_upd (the (weight g g')) e'))}"
    proof-
    have len: " $\bigwedge e. e \in \{e. \exists g' \ e'. \text{weight } g \ g' \neq \text{None} \wedge \text{winning\_budget\_len } e' \ g' \wedge$ 
e=(the (inverse_application (the (weight g g')) e'))}  $\implies$  e  $\in$  energies"
    proof-
    fix e
    assume "e  $\in$  {e.  $\exists g' \ e'. \text{weight } g \ g' \neq \text{None} \wedge \text{winning\_budget\_len } e' \ g' \wedge$ 
e=(the (inverse_application (the (weight g g')) e'))}"
    hence " $\exists g' \ e'. \text{weight } g \ g' \neq \text{None} \wedge \text{winning\_budget\_len } e' \ g' \wedge$  e=(the
(inverse_application (the (weight g g')) e'))" by auto
    from this obtain g' e' where eg: "weight g g'  $\neq$  None  $\wedge$  winning_budget_len
e' g'  $\wedge$  e=(the (inverse_application (the (weight g g')) e'))" by auto
    hence "weight g g'  $\neq$  None" by auto
    from eg have "e'  $\in$  energies" using winning_budget_len.simps by blast
    thus "e  $\in$  energies" using eg <e'  $\in$  energies>
    using inv_well_defined by blast
    qed

    show ?thesis
    proof

```

```

    assume "minimal_winning_budget e g"
    hence A: "winning_budget_len e g  $\wedge$  ( $\forall e'. e' \neq e \longrightarrow e' \leq e \longrightarrow \neg$  winning_budget_len
e' g)" using energy_Min_def by auto
    hence E: "e  $\in$  {e.  $\exists g' e'. \text{weight } g \ g' \neq \text{None} \wedge \text{winning\_budget\_len } e' \ g'$ 
 $\wedge$  e=(the (inverse_application (the (weight g g')) e'))}"
    using min_winning_budget_is_inv_a <g  $\in$  attacker>
    by (simp add: <minimal_winning_budget e g>)

    have " $\wedge x. x \in$  {e.  $\exists g' e'. \text{weight } g \ g' \neq \text{None} \wedge \text{winning\_budget\_len } e' \ g'$ 
 $\wedge$  e=(the (inverse_application (the (weight g g')) e'))}  $\wedge$  order x e  $\implies$  e=x"
    proof-
      fix x
      assume X: "x  $\in$  {e.  $\exists g' e'. \text{weight } g \ g' \neq \text{None} \wedge \text{winning\_budget\_len } e' \ g'$ 
 $\wedge$  e=(the (inverse_application (the (weight g g')) e'))}  $\wedge$  order x e"
      have "winning_budget_len x g"
      proof
        show "x  $\in$  energies  $\wedge$ 
g  $\in$  attacker  $\wedge$ 
( $\exists g'. \text{weight } g \ g' \neq \text{None} \wedge$ 
application (the (weight g g')) x  $\neq \text{None} \wedge \text{winning\_budget\_len}$  (the
(application (the (weight g g')) x)) g'"
        proof
          show "x  $\in$  energies" using len X by blast
          show "g  $\in$  attacker  $\wedge$ 
( $\exists g'. \text{weight } g \ g' \neq \text{None} \wedge$ 
application (the (weight g g')) x  $\neq \text{None} \wedge \text{winning\_budget\_len}$  (the
(application (the (weight g g')) x)) g'"
          proof
            show "g  $\in$  attacker" using <g  $\in$  attacker>.

            from X have " $\exists g' e'. \text{weight } g \ g' \neq \text{None} \wedge$ 
winning_budget_len e' g'  $\wedge$  x = inv_upd (the (weight g g')) e'"
            by blast
            from this obtain g' e' where X: "weight g g'  $\neq \text{None} \wedge$ 
winning_budget_len e' g'  $\wedge$  x = inv_upd (the (weight g g')) e'" by
auto

            show " $\exists g'. \text{weight } g \ g' \neq \text{None} \wedge$ 
apply_w g g' x  $\neq \text{None} \wedge \text{winning\_budget\_len}$  (upd (the (weight g g')) x)
g'"
            proof
              show "weight g g'  $\neq \text{None} \wedge$ 
apply_w g g' x  $\neq \text{None} \wedge \text{winning\_budget\_len}$  (upd (the (weight g g')) x)
g'"
              proof
                show "weight g g'  $\neq \text{None}$ " using X by simp
                show "apply_w g g' x  $\neq \text{None} \wedge \text{winning\_budget\_len}$  (upd (the
(weight g g')) x) g'"
                proof
                  have "e'  $\leq$  (upd (the (weight g g')) x)"
                  using X upd_inv_increasing
                  by (metis winning_budget_len.simps)
                  have "winning_budget_len (inv_upd (the (weight g g')) e')
g"

```

```

        using X attacker_inv_in_winning_budget <weight g g' ≠ None>
    <g ∈ attacker>

    by blast
    thus "winning_budget_len (upd (the (weight g g')) x) g'"
    using <e' e ≤ (upd (the (weight g g')) x)> upwards_closure_wb_len

X by blast

    have "inverse_application (the (weight g g')) e' ≠ None"
    using inv_well_defined <weight g g' ≠ None>
    by (metis X winning_budget_len.simps)
    thus "apply_w g g' x ≠ None"
    using X inv_well_defined
    using nonpos_eq_pos winning_budget_len_is_wb by blast
    qed
    qed
    qed
    qed
    qed
    qed
    thus "e=x" using X A
    by metis
    qed
    thus "e ∈ energy_Min {e. ∃g' e'. weight g g' ≠ None ∧ winning_budget_len
e' g' ∧ e=(the (inverse_application (the (weight g g')) e'))}"
    using E energy_Min_def
    by (smt (verit, del_insts) mem_Collect_eq)
    next
    assume "e ∈ energy_Min {e. ∃g' e'. weight g g' ≠ None ∧ winning_budget_len
e' g' ∧ e=(the (inverse_application (the (weight g g')) e'))}"
    hence E: "e ∈ {e. ∃g' e'. weight g g' ≠ None ∧ winning_budget_len e' g'
    ∧ e=(the (inverse_application (the (weight g g')) e'))}"
    using energy_Min_def by auto
    have "winning_budget_len e g ∧ (∀e'. e' ≠ e → order e' e → ¬ winning_budget_len
e' g)"
    proof
    show W: "winning_budget_len e g" using len E <g ∈ attacker> winning_budget_len.intro
    by (smt (verit, ccfv_SIG) attacker_inv_in_winning_budget mem_Collect_eq)

    from W have "e ∈ {e''. order e'' e ∧ winning_budget_len e'' g}" using energy_order
    ordering_def
    by (metis (no_types, lifting) mem_Collect_eq partial_preordering_def)
    hence notempty: "{e''} ≠ {e''. order e'' e ∧ winning_budget_len e'' g}"
    by auto
    have "∧e''. e'' ∈ {e''. order e'' e ∧ winning_budget_len e'' g} ⇒ e''
    ∈ energies"
    using winning_budget_len.simps by blast
    hence "{e''} ≠ energy_Min {e''. order e'' e ∧ winning_budget_len e'' g}"
    using energy_Min_not_empty notempty
    by (metis (no_types, lifting) subsetI)
    hence "∃e''. e'' ∈ energy_Min {e''. order e'' e ∧ winning_budget_len
e'' g}" by auto
    from this obtain e'' where "e'' ∈ energy_Min {e''. order e'' e ∧ winning_budget_len
e'' g}" by auto
    hence X: "order e'' e ∧ winning_budget_len e'' g ∧ (∀e'. e' ∈ {e''. order
e'' e ∧ winning_budget_len e'' g} → e'' ≠ e' → ¬ order e' e'')"
    using energy_Min_def by simp

```

```

have "( $\forall e' \neq e''$ . order e' e''  $\longrightarrow$   $\neg$  winning_budget_len e' g)"
proof
  fix e'
  show "e'  $\neq$  e''  $\longrightarrow$  order e' e''  $\longrightarrow$   $\neg$  winning_budget_len e' g"
  proof
    assume "e'  $\neq$  e'"
    show "order e' e''  $\longrightarrow$   $\neg$  winning_budget_len e' g"
    proof
      assume "order e' e'"
      from <order e' e''> have "order e' e" using X energy_order ordering_def
        by (metis (no_types, lifting) partial_preordering_def)
      show " $\neg$  winning_budget_len e' g"
      proof
        assume "winning_budget_len e' g"
        hence "e'  $\in$  {e''. order e'' e  $\wedge$  winning_budget_len e'' g}" using
<order e' e> by auto
        hence " $\neg$  order e' e'" using X <e'  $\neq$  e''> by simp
        thus "False" using <order e' e''> by simp
      qed
    qed
  qed
  hence E: "order e'' e  $\wedge$  winning_budget_len e'' g  $\wedge$  ( $\forall e' \neq e''$ . order
e' e''  $\longrightarrow$   $\neg$  winning_budget_len e' g)" using X
    by meson
  hence "order e'' e  $\wedge$  minimal_winning_budget e'' g" using energy_Min_def
by auto
  hence " $\exists g' e'$ . weight g g'  $\neq$  None  $\wedge$  winning_budget_len e' g'  $\wedge$  e''=(the
(inverse_application (the (weight g g'))) e'))"
    using min_winning_budget_is_inv_a X <g  $\in$  attacker> by simp
  hence "e''  $\in$  {e.  $\exists g' e'$ . weight g g'  $\neq$  None  $\wedge$  winning_budget_len e' g'
 $\wedge$  e=(the (inverse_application (the (weight g g'))) e'))}" by auto
  hence "e=e'" using <g  $\in$  attacker> X energy_Min_def E
    by (smt (verit, best) <e  $\in$  energy_Min {e.  $\exists g' e'$ . weight g g'  $\neq$  None
 $\wedge$  winning_budget_len e' g'  $\wedge$  e = the (inverse_application (the (weight g g'))) e'})>
mem_Collect_eq)
  thus "( $\forall e'$ . e'  $\neq$  e  $\longrightarrow$  order e' e  $\longrightarrow$   $\neg$  winning_budget_len e' g)" using
E by auto
  qed
  thus "minimal_winning_budget e g" using energy_Min_def by auto
  qed
qed

have min_winning_budget_is_minimal_inv_a: " $\bigwedge e$  g. g  $\in$  attacker  $\implies$  minimal_winning_budget
e g  $\implies \exists g' e'$ . weight g g'  $\neq$  None  $\wedge$  minimal_winning_budget e' g'  $\wedge$  e=(inv_upd
(the (weight g g'))) e'"
proof-
  fix e g
  assume A1: "g  $\in$  attacker" and A2: "minimal_winning_budget e g"
  show " $\exists g' e'$ . weight g g'  $\neq$  None  $\wedge$  minimal_winning_budget e' g'  $\wedge$  e=(inv_upd
(the (weight g g'))) e'"
  proof-
    from A1 A2 have "winning_budget_len e g" using energy_Min_def by simp
    from A1 A2 have " $\forall e' \neq e$ . order e' e  $\longrightarrow$   $\neg$  winning_budget_len e' g" using
energy_Min_def

```

```

      using mem_Collect_eq by auto
      hence "∃ g' e'. weight g g' ≠ None ∧ winning_budget_len e' g' ∧ e=(the
(inverse_application (the (weight g g')) e'))"
      using min_winning_budget_is_inv_a A1 A2 <winning_budget_len e g> by auto
      from this obtain g' e' where G: "weight g g' ≠ None ∧ winning_budget_len
e' g' ∧ e=(the (inverse_application (the (weight g g')) e'))" by auto
      hence "e' ∈ {e. winning_budget_len e g' ∧ order e e'}" using energy_order
ordering_def
      using partial_preordering.refl by fastforce
      have "∧ e'. e' ∈ {e. winning_budget_len e g' ∧ order e e'} ⇒ e' ∈ energies"
using winning_budget_len.simps by blast
      hence "energy_Min {e. winning_budget_len e g' ∧ order e e'} ≠ {}" using
<e' ∈ {e. winning_budget_len e g' ∧ order e e'}> energy_Min_not_empty
      by (metis (mono_tags, lifting) empty_iff energy_order mem_Collect_eq ordering.eq_iff
subsetI)
      hence "∃ e''. e'' ∈ energy_Min {e. winning_budget_len e g' ∧ order e e'}"
by auto
      from this obtain e'' where "e'' ∈ energy_Min {e. winning_budget_len e g'
∧ order e e'}" by auto
      have <minimal_winning_budget e'' g'>
      unfolding energy_Min_def proof
      show "e'' ∈ a_win g' ∧ (∀ e' ∈ a_win g'. e'' ≠ e' → ¬ e' e ≤ e'')"
      proof
      have "winning_budget_len e'' g' ∧ order e'' e'"
      using <e'' ∈ energy_Min {e. winning_budget_len e g' ∧ order e e'}>
energy_Min_def by auto
      thus "e'' ∈ a_win g'" by auto
      show "(∀ e' ∈ a_win g'. e'' ≠ e' → ¬ e' e ≤ e'')"
      proof
      fix e
      assume "e ∈ a_win g'"
      show "e'' ≠ e → ¬ e e ≤ e''"
      proof
      assume "e'' ≠ e"
      show "¬ e e ≤ e''"
      proof
      assume "e e ≤ e''"
      hence "e e ≤ e''" using <winning_budget_len e'' g' ∧ order e''
e'> energy_order ordering_def
      by (metis (no_types, lifting) partial_preordering_def)
      hence "winning_budget_len e g' ∧ order e e'"
      using <e ∈ a_win g'> by auto
      hence "e ∈ {e. winning_budget_len e g' ∧ order e e'} ∧ e'' ≠
e ∧ e e ≤ e''"
      by (simp add: <e e ≤ e''> <e'' ≠ e>)
      thus "False"
      using <e'' ∈ energy_Min {e. winning_budget_len e g' ∧ order
e e'}> energy_Min_def
      by auto
      qed
      qed
      qed
      qed
      qed
      from <e'' ∈ energy_Min {e. winning_budget_len e g' ∧ order e e'}> have

```



```

"e'' ∈ {e. winning_budget_len e g' ∧ order e e'}" using energy_Min_def by auto
hence "winning_budget_len e'' g' ∧ order e'' e'" by simp

have "order e'' e'" using <e'' ∈ energy_Min {e. winning_budget_len e g'
∧ order e e'}> energy_Min_def by auto
hence "order (the (inverse_application (the (weight g g'))) e'')) (the (inverse_applicat
(the (weight g g'))) e'))"
using inverse_monotonic
using G inv_well_defined energy_order nonpos_eq_pos winning_budgnet_len_is_wb
using <winning_budget_len e'' g' ∧ e'' e ≤ e'> by presburger
hence "order (the (inverse_application (the (weight g g'))) e'')) e" using
G by auto
hence "e=(the (inverse_application (the (weight g g'))) e''))" using <winning_budget_len
e'' g' ∧ order e'' e'> <∀e' ≠ e. order e' e → ¬ winning_budget_len e' g'>
by (metis A1 G attacker_inv_in_winning_budget)
thus ?thesis using G <minimal_winning_budget e'' g'> by auto
qed
qed

show "minimal_winning_budget e g = (e ∈ energy_Min {e. ∃g' e'. weight g g'
≠ None ∧ minimal_winning_budget e' g' ∧ e=(the (inverse_application (the (weight
g g'))) e')))"
proof
assume "minimal_winning_budget e g"
show "(e ∈ energy_Min {e. ∃g' e'. weight g g' ≠ None ∧ minimal_winning_budget
e' g' ∧ e=(the (inverse_application (the (weight g g'))) e')))"
proof-
from <g ∈ attacker> have exist: "∃g' e'. weight g g' ≠ None ∧ minimal_winning_budget
e' g' ∧ e = inv_upd (the (weight g g'))) e'"
using <minimal_winning_budget e g> min_winning_budget_is_minimal_inv_a
by simp
have "∧e''. e'' e ≤ e ∧ e ≠ e'' ⇒ e'' ∉ {e. ∃g' e'. weight g g' ≠ None
∧ minimal_winning_budget e' g' ∧ e=(the (inverse_application (the (weight g g')))
e')))"
proof-
fix e''
show "e'' e ≤ e ∧ e ≠ e'' ⇒ e'' ∉ {e. ∃g' e'. weight g g' ≠ None ∧
minimal_winning_budget e' g' ∧ e=(the (inverse_application (the (weight g g'))) e')))"
proof-
assume "e'' e ≤ e ∧ e ≠ e'' "
show "e'' ∉ {e. ∃g' e'. weight g g' ≠ None ∧ minimal_winning_budget
e' g' ∧ e=(the (inverse_application (the (weight g g'))) e')))"
proof
assume "e'' ∈ {e. ∃g' e'. weight g g' ≠ None ∧ minimal_winning_budget
e' g' ∧ e=(the (inverse_application (the (weight g g'))) e')))"
hence "∃g' e'. weight g g' ≠ None ∧ minimal_winning_budget e' g'
∧ e''=(the (inverse_application (the (weight g g'))) e'"))
by auto
from this obtain g' e' where EG: "weight g g' ≠ None ∧ minimal_winning_budget
e' g' ∧ e''=(the (inverse_application (the (weight g g'))) e'")) by auto
hence "winning_budget_len e' g'" using energy_Min_def by simp
hence "winning_budget_len e'' g'" using EG winning_budget_len_simps
by (metis <g ∈ attacker> attacker_inv_in_winning_budget)
then show "False" using <e'' e ≤ e ∧ e ≠ e''> <minimal_winning_budget
e g> energy_Min_def by auto
qed

```

```

qed
qed
thus "(e ∈ energy_Min {e. ∃g' e'. weight g g' ≠ None ∧ minimal_winning_budget
e' g' ∧ e=(the (inverse_application (the (weight g g')) e'))}"
using exist energy_Min_def
by (smt (verit) mem_Collect_eq)
qed
next
assume emin: "(e ∈ energy_Min {e. ∃g' e'. weight g g' ≠ None ∧ minimal_winning_budget
e' g' ∧ e=(the (inverse_application (the (weight g g')) e'))}"
show "minimal_winning_budget e g"
proof-
from emin have "∃g' e'. weight g g' ≠ None ∧ minimal_winning_budget e'
g' ∧ e=(the (inverse_application (the (weight g g')) e'))" using energy_Min_def
by auto
hence "∃g' e'. weight g g' ≠ None ∧ winning_budget_len e' g' ∧ e=(the
(inverse_application (the (weight g g')) e'))" using energy_Min_def
by (metis (no_types, lifting) mem_Collect_eq)
hence element_of: "e∈{e. ∃g' e'.
weight g g' ≠ None ∧
winning_budget_len e' g' ∧ e = inv_upd (the (weight g g')) e'}"
by auto

have "∧e''. e'' e< e ⇒ e'' ∉ {e. ∃g' e'.
weight g g' ≠ None ∧
winning_budget_len e' g' ∧ e = inv_upd (the (weight g g')) e'}"

proof
fix e''
assume "e'' e< e"
assume "e'' ∈ {e. ∃g' e'.
weight g g' ≠ None ∧
winning_budget_len e' g' ∧ e = inv_upd (the (weight g g')) e'}"
hence "∃g' e'.
weight g g' ≠ None ∧
winning_budget_len e' g' ∧ e'' = inv_upd (the (weight g g'))
e'" by auto
from this obtain g' e' where E'G': "weight g g' ≠ None ∧
winning_budget_len e' g' ∧ e'' = inv_upd (the (weight g g'))
e'" by auto
hence "e' ∈ {e. winning_budget_len e g'}" by simp
hence "∃e_min. minimal_winning_budget e_min g' ∧ e_min e≤ e'"
using energy_Min_contains_smaller
by (metis mem_Collect_eq nonpos_eq_pos subsetI winning_budgget_len_is_wb)
from this obtain e_min where "minimal_winning_budget e_min g' ∧ e_min
e≤ e'" by auto
have "inv_upd (the (weight g g')) e_min e≤ inv_upd (the (weight g g'))
e'"
proof(rule inverse_monotonic)
show "weight g g' ≠ None"
using <weight g g' ≠ None ∧ winning_budget_len e' g' ∧ e'' = inv_upd
(the (weight g g')) e'> by simp
show "e_min e≤ e'" using <minimal_winning_budget e_min g' ∧ e_min e≤
e'>
by auto
hence "e_min ∈ energies" using winning_budget_len.simps
by (metis (no_types, lifting) <minimal_winning_budget e_min g' ∧

```

```

e_min e ≤ e' > energy_Min_def mem_Collect_eq)
  thus "inverse_application (the (weight g g')) e_min ≠ None"
    using inv_well_defined <weight g g' ≠ None> by auto
  show "e_min ∈ energies"
    by (simp add: <e_min ∈ energies>)
qed
hence "inv_upd (the (weight g g')) e_min e < e" using <e'' e < e> E'G'
  using energy_order ordering_def
  by (metis (no_types, lifting) ordering.antisym partial_preordering.trans)

  have "inv_upd (the (weight g g')) e_min ∈ {e. ∃ g' e'. weight g g' ≠ None
    ∧ minimal_winning_budget e' g' ∧ e = (the (inverse_application (the (weight g g'))
    e'))}"
    using <minimal_winning_budget e_min g' ∧ e_min e ≤ e'> E'G'
    by blast
  thus "False" using <inv_upd (the (weight g g')) e_min e < e> energy_Min_def
emin
    by (smt (verit) mem_Collect_eq)
qed

hence "e ∈ energy_Min
  {e. ∃ g' e'.
    weight g g' ≠ None ∧
    winning_budget_len e' g' ∧ e = inv_upd (the (weight g g')) e'}"

  using energy_Min_def element_of
  by (smt (verit, ccfv_threshold) mem_Collect_eq)
then show ?thesis using min_winning_budget_a_iff_energy_Min <g ∈ attacker>
by simp
  qed
  qed
  qed

  have minimal_winning_budget_defender: "∧ g e. g ∉ attacker ⇒ minimal_winning_budget
e g = (e ∈ energy_Min {e''. ∃ strat. (∀ g'. weight g g' ≠ None → strat g' ∈ {the
(inverse_application (the (weight g g')) e) | e. minimal_winning_budget e g'})
  ∧ e'' = (energy_sup {strat g' | g'. weight g g' ≠ None}))}"
proof-
  fix g e
  assume "g ∉ attacker"
  have sup_inv_in_winning_budget: "∧ (strat:: 'position ⇒ 'energy) g. g ∉ attacker
⇒ ∀ g'. weight g g' ≠ None → strat g' ∈ {inv_upd (the (weight g g')) e | e.
winning_budget_len e g' } ⇒ winning_budget_len (energy_sup {strat g' | g'. weight
g g' ≠ None}) g"
proof-
  fix strat g
  assume A1: "g ∉ attacker" and "∀ g'. weight g g' ≠ None → strat g' ∈ {inv_upd
(the (weight g g')) e | e. winning_budget_len e g' }"
  hence A2: "∧ g'. weight g g' ≠ None ⇒ strat g' ∈ {inv_upd (the (weight
g g')) e | e. winning_budget_len e g' }"
  by simp
  show "winning_budget_len (energy_sup {strat g' | g'. weight g g' ≠ None}) g"
proof (rule winning_budget_len.intros(1))
  have A: "(∀ g'. weight g g' ≠ None →"

```

```

    application (the (weight g g')) (energy_sup {strat g' |g'. weight g g'
≠ None})) ≠ None ∧
    winning_budget_len (the (application (the (weight g g')) (energy_sup {strat
g' |g'. weight g g' ≠ None}))) g') "
  proof
    fix g'
    show "weight g g' ≠ None →
    application (the (weight g g')) (energy_sup {strat g' |g'. weight g g'
≠ None})) ≠ None ∧
    winning_budget_len (the (application (the (weight g g')) (energy_sup {strat
g' |g'. weight g g' ≠ None}))) g'"
    proof
      assume "weight g g' ≠ None"
      hence "strat g' ∈ {the (inverse_application (the (weight g g')) e) |
e. winning_budget_len e g' }" using A2 by simp
      hence "∃e. strat g' = the (inverse_application (the (weight g g')) e)
∧ winning_budget_len e g'" by blast
      from this obtain e where E: "strat g' = the (inverse_application (the
(weight g g')) e) ∧ winning_budget_len e g'" by auto

      hence "e ∈ energies" using winning_budget_len.simps by blast
      hence "inverse_application (the (weight g g')) e ≠ None" using inv_well_defined
<weight g g' ≠ None> by simp

      have "{strat g' |g'. weight g g' ≠ None} ⊆ energies ∧ finite {strat
g' |g'. weight g g' ≠ None}"
      proof
        show "{strat g' |g'. weight g g' ≠ None} ⊆ energies"
          by (smt (verit, best) A2 inv_well_defined mem_Collect_eq nonpos_eq_pos
subsetI winning_budget_len_is_wb)
        have "{strat g' |g'. weight g g' ≠ None} ⊆ {strat g' |g'. g' ∈ positions}"
          by auto
        thus "finite {strat g' |g'. weight g g' ≠ None}"
          using finite_positions
          using rev_finite_subset by fastforce
      qed
      hence leq: "order (strat g') (energy_sup {strat g' |g'. weight g g'
≠ None}))"
        using bounded_join_semilattice <weight g g' ≠ None>
        by (metis (mono_tags, lifting) mem_Collect_eq)

      show "application (the (weight g g')) (energy_sup {strat g' |g'. weight
g g' ≠ None})) ≠ None ∧
      winning_budget_len (the (application (the (weight g g')) (energy_sup
{strat g' |g'. weight g g' ≠ None}))) g'"
      proof
        have "application (the (weight g g')) (strat g') = application (the
(weight g g')) (the (inverse_application (the (weight g g')) e))" using E
        by simp
        also have "... ≠ None" using <inverse_application (the (weight g
g')) e ≠ None> inv_well_defined
        using <e ∈ energies> <weight g g' ≠ None> by presburger
        finally have "application (the (weight g g')) (strat g') ≠ None" .
        thus "application (the (weight g g')) (energy_sup {strat g' |g'. weight
g g' ≠ None})) ≠ None"
          using leq domain_upw_closed

```

```

    using <weight g g' ≠ None> by blast

    have "order e (the (application (the (weight g g')) (strat g')))"
using upd_inv_increasing
    by (metis <application (the (weight g g')) (strat g') = application
(the (weight g g')) (the (inverse_application (the (weight g g')) e))> <e ∈ energies>
<weight g g' ≠ None>)
    hence W: "winning_budget_len (the (application (the (weight g g'))
(strat g')) g'" using E upwards_closure_wb_len
    by blast
    have "order (the (application (the (weight g g')) (strat g')) (the
(application (the (weight g g')) (energy_sup {strat g' | g'. weight g g' ≠ None})))"

    using updates_monotonic
    using <apply_w g g' (strat g') ≠ None> <weight g g' ≠ None> <{strat
g' | g'. weight g g' ≠ None} ⊆ energies ∧ finite {strat g' | g'. weight g g' ≠ None}>
leq by blast
    thus "winning_budget_len (the (application (the (weight g g')) (energy_sup
{strat g' | g'. weight g g' ≠ None}))) g'"
    using W upwards_closure_wb_len by blast
    qed
    qed
    qed

    have "(energy_sup {strat g' | g'. weight g g' ≠ None}) ∈ energies"
proof-
    have "{strat g' | g'. weight g g' ≠ None} ⊆ {strat g' | g'. g' ∈ positions}"
by auto
    hence fin: "finite {strat g' | g'. weight g g' ≠ None}" using finite_positions
    using rev_finite_subset by fastforce
    have "{strat g' | g'. weight g g' ≠ None} ⊆ energies"
    using A2
    by (smt (verit) inv_well_defined mem_Collect_eq subsetI winning_budget_len.cases)

    thus ?thesis using bounded_join_semilattice fin by auto
    qed
    thus "(energy_sup {strat g' | g'. weight g g' ≠ None}) ∈ energies ∧ g ∉
attacker ∧
    (∀g'. weight g g' ≠ None →
    application (the (weight g g')) (energy_sup {strat g' | g'. weight g g'
≠ None}) ≠ None ∧
    winning_budget_len (the (application (the (weight g g')) (energy_sup {strat
g' | g'. weight g g' ≠ None}))) g'"
    using A1 A by auto
    qed
    qed

    have min_winning_budget_is_inv_d: "∧e g. g ∉ attacker ⇒ minimal_winning_budget
e g ⇒ ∃strat. (∀g'. weight g g' ≠ None → strat g' ∈ {inv_upd (the (weight
g g')) e | e. winning_budget_len e g'})
    ∧ e = (energy_sup {strat g' | g'. weight g g' ≠ None})"
proof-
    fix e g
    assume A1: "g ∉ attacker" and A2: "minimal_winning_budget e g"
    show "∃strat. (∀g'. weight g g' ≠ None → strat g' ∈ {inv_upd (the (weight
g g')) e | e. winning_budget_len e g'})"

```

```

       $\wedge e = (\text{energy\_sup } \{\text{strat } g' \mid g'. \text{weight } g \ g' \neq \text{None}\})$ "
proof-
  from A2 have "e  $\in$  energies" using winning_budget_len.simps energy_Min_def
  by (metis (no_types, lifting) mem_Collect_eq)
  from A1 A2 have W: " $(\forall g'. \text{weight } g \ g' \neq \text{None} \longrightarrow$ 
    application (the (weight g g')) e  $\neq$  None  $\wedge$ 
    winning_budget_len (the (application (the (weight g g')) e)) g'"
using winning_budget_len.simps energy_Min_def
  by (metis (no_types, lifting) mem_Collect_eq)

  define strat where S: " $\forall g'. \text{strat } g' = \text{the } ((\text{inverse\_application (the (weight } g \ g')) (the (application (the (weight g g')) e))))$ "
  have A: " $(\forall g'. \text{weight } g \ g' \neq \text{None} \longrightarrow \text{strat } g' \in \{\text{the (inverse\_application (the (weight g g')) e) \mid e. \text{winning\_budget\_len } e \ g'\})}$ "
  proof
    fix g'
    show "weight g g'  $\neq$  None  $\longrightarrow$  strat g'  $\in$  {the (inverse_application (the (weight g g')) e)  $\mid$  e. winning_budget_len e g'"
    proof
      assume "weight g g'  $\neq$  None"
      hence "winning_budget_len (the (application (the (weight g g')) e))
g'" using W by auto
      thus "strat g'  $\in$  {the (inverse_application (the (weight g g')) e)  $\mid$  e.
winning_budget_len e g'}" using S by blast
    qed
  qed
  hence W: "winning_budget_len (energy_sup {strat g'  $\mid$  g'. weight g g'  $\neq$  None})
g" using sup_inv_in_winning_budget A1 by simp
  have " $\bigwedge g'. \text{weight } g \ g' \neq \text{None} \implies \text{order (strat g')} e$ "
  proof-
    fix g'
    assume "weight g g'  $\neq$  None"
    hence "application (the (weight g g')) e  $\neq$  None" using W
    using A1 A2 winning_budget_len.cases energy_Min_def
    by (metis (mono_tags, lifting) mem_Collect_eq)
    from <weight g g'  $\neq$  None> have "strat g' = the ((inverse_application
(the (weight g g')) (the (application (the (weight g g')) e))))" using S by auto
    thus "order (strat g') e" using inv_upd_decreasing <application (the
(weight g g')) e  $\neq$  None>
    using <e  $\in$  energies> <weight g g'  $\neq$  None> by presburger
  qed

  have BJS1: "finite {strat g'  $\mid$  g'. weight g g'  $\neq$  None}  $\wedge$  {strat g'  $\mid$  g'.
weight g g'  $\neq$  None}  $\subseteq$  energies"
  proof
    have "{strat g'  $\mid$  g'. weight g g'  $\neq$  None}  $\subseteq$  {strat g'  $\mid$  g'. g'  $\in$  positions}"
    by auto
    thus "finite {strat g'  $\mid$  g'. weight g g'  $\neq$  None}"
    using finite_positions
    using rev_finite_subset by fastforce
    show "{strat g'  $\mid$  g'. weight g g'  $\neq$  None}  $\subseteq$  energies"
  proof
    fix x
    assume "x  $\in$  {strat g'  $\mid$  g'. weight g g'  $\neq$  None}"
    from this obtain g' where "x = strat g'" and "weight g g'  $\neq$  None" by
auto

```

```

      hence "x ∈ {the (inverse_application (the (weight g g')) e) | e. winning_budget_len
e g'}}" using A
      by simp
      thus "x ∈ energies"
      using <weight g g' ≠ None> inv_well_defined nonpos_eq_pos winning_bugget_len_is_
by auto
      qed
      qed
      hence "(∀s. s ∈ {strat g' | g'. weight g g' ≠ None} → s e ≤ e) → energy_sup
{strat g' | g'. weight g g' ≠ None} e ≤ e"
      using bounded_join_semilattice
      by (meson <e ∈ energies>)
      hence "order (energy_sup {strat g' | g'. weight g g' ≠ None}) e"
      using <∧g'. weight g g' ≠ None ⇒ order (strat g') e>
      by blast
      hence "e = energy_sup {strat g' | g'. weight g g' ≠ None}" using W A1 A2
energy_Min_def
      by force
      thus ?thesis using A by blast
      qed
      qed

      have min_winning_budget_d_iff_energy_Min: "∧e g. g ∉ attacker ⇒ e ∈ energies
⇒ ((e ∈ energy_Min {e''. ∃ strat. (∀ g'. weight g g' ≠ None → strat g' ∈ {inv_upd
(the (weight g g')) e | e. winning_budget_len e g'}})
      ∧ e'' = (energy_sup {strat g' | g'. weight g g' ≠ None}))
      ⇔ minimal_winning_budget e g)"
proof-
  fix e g
  show "g ∉ attacker ⇒
e ∈ energies ⇒
(e ∈ energy_Min
{e''.
  ∃ strat.
    (∀ g'. weight g g' ≠ None →
      strat g'
        ∈ {inv_upd (the (weight g g')) e | e. winning_budget_len
e g'}}) ∧
      e'' = energy_sup {strat g' | g'. weight g g' ≠ None})) =
      minimal_winning_budget e g"
proof-
  assume A1: "g ∉ attacker" and A2: "e ∈ energies"
  show "(e ∈ energy_Min
{e''.
  ∃ strat.
    (∀ g'. weight g g' ≠ None →
      strat g'
        ∈ {inv_upd (the (weight g g')) e | e. winning_budget_len
e g'}}) ∧
      e'' = energy_sup {strat g' | g'. weight g g' ≠ None})) =
      minimal_winning_budget e g"
proof
  assume assumption: "e ∈ energy_Min {e''. ∃ strat. (∀ g'. weight g g' ≠ None
→ strat g' ∈ {the (inverse_application (the (weight g g')) e) | e. winning_budget_len
e g'})"

```

```

       $\wedge e'' = (\text{energy\_sup } \{\text{strat } g' \mid g'. \text{ weight } g \ g' \neq \text{None}\})$ "
    show "minimal_winning_budget e g"
      unfolding energy_Min_def
    proof
      show "e  $\in$  {e. winning_budget_len e g}  $\wedge$  ( $\forall e' \in$  {e. winning_budget_len
e g}. e  $\neq$  e'  $\longrightarrow$   $\neg$  e' e  $\leq$  e)"
    proof
      show "e  $\in$  {e. winning_budget_len e g}"
    proof
      from A1 A2 assumption have " $\exists$  strat. ( $\forall g'. \text{ weight } g \ g' \neq \text{None} \longrightarrow$ 
strat g'  $\in$  {the (inverse_application (the (weight g g'))) e}  $\mid$  e. winning_budget_len
e g'})"
       $\wedge e = (\text{energy\_sup } \{\text{strat } g' \mid g'. \text{ weight } g \ g' \neq \text{None}\})$ "
      using energy_Min_def by simp
      thus "winning_budget_len e g" using sup_inv_in_winning_budget A1
A2 by blast
    qed
    hence W: "winning_budget_len e g" by simp
    hence "e  $\in$  energies" using winning_budget_len.simps by blast
    hence "e  $\in$  {e''. order e'' e  $\wedge$  winning_budget_len e'' g  $\wedge$  e''  $\in$  energies}"
using W energy_order ordering_def <g  $\notin$  attacker>
      using energy_wqo reflp_onD wqo_on_imp_reflpl_on by fastforce
    hence "{e''. order e'' e  $\wedge$  winning_budget_len e'' g  $\wedge$  e''  $\in$  energies}
 $\neq$  {}" by auto
    hence "energy_Min {e''. order e'' e  $\wedge$  winning_budget_len e'' g  $\wedge$  e''
 $\in$  energies}  $\neq$  {}" using energy_Min_not_empty
      by (metis (no_types, lifting) mem_Collect_eq subsetI)
    hence " $\exists e''. e'' \in \text{energy\_Min } \{e''. \text{ order } e'' e \wedge \text{winning\_budget\_len}$ 
e'' g  $\wedge$  e''  $\in$  energies}" by auto
    from this obtain e'' where "e''  $\in$  energy_Min {e''. order e'' e  $\wedge$  winning_budget_len
e'' g  $\wedge$  e''  $\in$  energies}" by auto
    hence X: "order e'' e  $\wedge$  winning_budget_len e'' g  $\wedge$  e''  $\in$  energies
 $\wedge$  ( $\forall e'. e' \in$  {e''. order e'' e  $\wedge$  winning_budget_len e'' g
 $\wedge$  e''  $\in$  energies}  $\longrightarrow$  e''  $\neq$  e'  $\longrightarrow$   $\neg$  order e' e'')" using energy_Min_def
      by simp
    have "( $\forall e' \neq e''. \text{ order } e' e'' \longrightarrow \neg \text{winning\_budget\_len } e' g$ )"
    proof
      fix e'
      show "e'  $\neq$  e''  $\longrightarrow$  order e' e''  $\longrightarrow$   $\neg$  winning_budget_len e' g"
    proof
      assume "e'  $\neq$  e'"
      show "order e' e''  $\longrightarrow$   $\neg$  winning_budget_len e' g"
    proof
      assume "order e' e'"
      from <order e' e''> have "order e' e" using X energy_order
ordering_def
      by (metis (no_types, lifting) partial_preordering.trans)
    show " $\neg$  winning_budget_len e' g"
    proof (cases "e'  $\in$  energies")
      case True
      show ?thesis
    proof
      assume "winning_budget_len e' g"
      hence "e'  $\in$  {e''. order e'' e  $\wedge$  winning_budget_len e'' g  $\wedge$ 
e''  $\in$  energies}" using <e'  $\in$  energies> <order e' e> by auto

```



```

      hence "¬ order e' e'" using X <e' ≠ e''> by simp
      thus "False" using <order e' e''> by simp
    qed
  next
    case False
    then show ?thesis
      by (simp add: nonpos_eq_pos winning_budget_len_is_wb)
    qed
  qed
  qed
  qed
  hence "order e'' e ∧ winning_budget_len e'' g ∧ (∀e' ≠ e''. order
e' e'' → ¬ winning_budget_len e' g)" using X
    by meson
  hence E: "order e'' e ∧ minimal_winning_budget e'' g" using energy_Min_def
    by (smt (verit) mem_Collect_eq)
  hence "∃ strat. (∀ g'. weight g g' ≠ None → strat g' ∈ {the (inverse_application
(the (weight g g')) e) | e. winning_budget_len e g'})
    ∧ e'' = (energy_sup {strat g' | g'. weight g g' ≠ None})"
    using min_winning_budget_is_inv_d
    by (simp add: X A1)
  hence "e=e'" using assumption X energy_Min_def by auto
  thus "(∀ e' ∈ {e. winning_budget_len e g}. e ≠ e' → ¬ e' e ≤ e)" using
E
    using <∀ e'. e' ≠ e'' → e' e ≤ e'' → ¬ winning_budget_len e'
g> by fastforce
  qed
  qed
  next
    assume assumption: "minimal_winning_budget e g"
    show "e ∈ energy_Min {e''. ∃ strat. (∀ g'. weight g g' ≠ None → strat
g' ∈ {the (inverse_application (the (weight g g')) e) | e. winning_budget_len e
g'})
      ∧ e'' = (energy_sup {strat g' | g'. weight g g' ≠ None})}"
      unfolding energy_Min_def
    proof
      from assumption have "e ∈ energies" using winning_budget_len.simps energy_Min_def
        using A2 by blast
      show "e ∈ {e''."
    ∃ strat.
      (∀ g'. weight g g' ≠ None →
        strat g' ∈ {the (inverse_application (the (weight g g')) e) | e.
winning_budget_len e g'}) ∧
        e'' = energy_sup {strat g' | g'. weight g g' ≠ None} ∧
      (∀ e' ∈ {e''.
    ∃ strat.
      (∀ g'. weight g g' ≠ None →
        strat g' ∈ {the (inverse_application (the (weight g g')) e)
| e. winning_budget_len e g'}) ∧
        e'' = energy_sup {strat g' | g'. weight g g' ≠ None}.
      e ≠ e' → ¬ order e' e)"
      proof
        from A1 A2 assumption have "∃ strat. (∀ g'. weight g g' ≠ None →
strat g' ∈ {the (inverse_application (the (weight g g')) e) | e. winning_budget_len
e g'})
          ∧ e = (energy_sup {strat g' | g'. weight g g' ≠ None})" using

```

```

min_winning_budget_is_inv_d by simp
  thus "e ∈ {e'}. ∃ strat. (∀ g'. weight g g' ≠ None → strat g' ∈
    {the (inverse_application (the (weight g g'))) e | e. winning_budget_len e g'})
    ∧ e' = (energy_sup {strat g' | g'. weight g g' ≠ None})" by
auto
  show "∀ e' ∈ {e'}.
    ∃ strat.
      (∀ g'. weight g g' ≠ None →
        strat g' ∈ {the (inverse_application (the (weight g g'))) e | e.
winning_budget_len e g'}) ∧
        e' = energy_sup {strat g' | g'. weight g g' ≠ None}).
    e ≠ e' → ¬ order e' e"
  proof
    fix e'
    assume "e' ∈ {e'}.
    ∃ strat.
      (∀ g'. weight g g' ≠ None →
        strat g' ∈ {the (inverse_application (the (weight g g')))
e) | e. winning_budget_len e g'}) ∧
        e' = energy_sup {strat g' | g'. weight g g' ≠ None}"
    hence "∃ strat.
      (∀ g'. weight g g' ≠ None →
        strat g' ∈ {the (inverse_application (the (weight g g')))
e) | e. winning_budget_len e g'}) ∧
        e' = energy_sup {strat g' | g'. weight g g' ≠ None}" by auto
    from this obtain strat where S: "(∀ g'. weight g g' ≠ None →
      strat g' ∈ {the (inverse_application (the (weight g g')))
e) | e. winning_budget_len e g'}) ∧
      e' = energy_sup {strat g' | g'. weight g g' ≠ None}" by auto
    have "finite {strat g' | g'. weight g g' ≠ None} ∧ {strat g' | g'.
weight g g' ≠ None} ⊆ energies"
    proof
      have "{strat g' | g'. weight g g' ≠ None} ⊆ {strat g' | g'. g'
∈ positions}" by auto
      thus "finite {strat g' | g'. weight g g' ≠ None}" using finite_positions
        using rev_finite_subset by fastforce
      show "{strat g' | g'. weight g g' ≠ None} ⊆ energies"
    proof
      fix x
      assume "x ∈ {strat g' | g'. weight g g' ≠ None}"
      thus "x ∈ energies" using S
        by (smt (verit, ccfv_threshold) inv_well_defined mem_Collect_eq
nonpos_eq_pos winning_bugget_len_is_wb)
    qed
    qed
    hence "e' ∈ energies" using bounded_join_semilattice S
      by (meson empty_iff empty_subsetI finite.emptyI upward_closed_energies)

    show "e ≠ e' → ¬ order e' e"
    proof
      assume "e ≠ e'"
      have "(∀ g'. weight g g' ≠ None →
        application (the (weight g g'))) e' ≠ None ∧
        winning_budget_len (the (application (the (weight g g'))) e')) g'"
    proof
      fix g'

```

```

      show "weight g g' ≠ None →
        application (the (weight g g')) e' ≠ None ∧ winning_budget_len (the
(application (the (weight g g')) e')) g' "
      proof
        assume "weight g g' ≠ None"
        hence "strat g' ∈ {the (inverse_application (the (weight g
g')) e) | e. winning_budget_len e g'}" using S by auto
        hence "∃ e''. strat g' = the (inverse_application (the (weight
g g')) e'') ∧ winning_budget_len e'' g'" by auto
        from this obtain e'' where E: "strat g' = the (inverse_application
(the (weight g g')) e'') ∧ winning_budget_len e'' g'" by auto
        hence "e'' ∈ energies" using winning_budget_len.simps by blast
        show "application (the (weight g g')) e' ≠ None ∧ winning_budget_len
(the (application (the (weight g g')) e')) g' "
        proof
          have "{strat g' | g'. weight g g' ≠ None} ⊆ energies ∧ finite
{strat g' | g'. weight g g' ≠ None}"
          proof
            show "{strat g' | g'. weight g g' ≠ None} ⊆ energies"
          using S
            using <finite {strat g' | g'. weight g g' ≠ None} ∧
{strat g' | g'. weight g g' ≠ None} ⊆ energies> by blast
            have "{strat g' | g'. weight g g' ≠ None} ⊆ {strat g'
| g'. g' ∈ positions}" by auto
            thus "finite {strat g' | g'. weight g g' ≠ None}"
              using finite_positions
              using rev_finite_subset by fastforce
            qed
            hence "order (strat g') e'" using S bounded_join_semilattice
<weight g g' ≠ None>
              by (metis (mono_tags, lifting) mem_Collect_eq)
            have "application (the (weight g g')) (strat g') ≠ None"
          using E inv_well_defined inv_well_defined <e'' ∈ energies>
            by (metis <weight g g' ≠ None> )
            thus "application (the (weight g g')) e' ≠ None" using domain_upw_clos
<order (strat g') e'>
              using <weight g g' ≠ None> by blast
            have "order e'' (the (application (the (weight g g')) (strat
g')))" using E upd_inv_increasing
              using <e'' ∈ energies> <weight g g' ≠ None> by metis
            hence W: "winning_budget_len (the (application (the (weight
g g')) (strat g')) g'" using upwards_closure_wb_len
              using E by blast
            from <order (strat g') e'> have "order (the (application
(the (weight g g')) (strat g')) (the (application (the (weight g g')) e')))"
              using updates_monotonic <application (the (weight g g'))
(strat g') ≠ None>
              by (metis E <e'' ∈ energies> <weight g g' ≠ None> inv_well_defined)
            thus "winning_budget_len (the (application (the (weight
g g')) e')) g' " using upwards_closure_wb_len W
              by blast
            qed
          qed
        qed
        hence "winning_budget_len e' g" using winning_budget_len.intros(1)
A1 <e' ∈ energies>

```

```

      by blast
      thus "¬ order e' e" using assumption <e ≠ e'> energy_Min_def
by auto

      qed
    qed
  qed
  qed
  qed
  qed
  qed

  have min_winning_budget_is_minimal_inv_d: "∀e g. g ≠ attacker ⇒ minimal_winning_budget
e g ⇒ ∃ strat. (∀g'. weight g g' ≠ None ⇒ strat g' ∈ {the (inverse_application
(the (weight g g')) e) | e. minimal_winning_budget e g'})
  ∧ e = (energy_sup {strat g' | g'. weight g g' ≠ None})"

  proof-
    fix e g
    assume A1: "g ≠ attacker" and A2: "minimal_winning_budget e g"
    show "∃ strat. (∀g'. weight g g' ≠ None ⇒ strat g' ∈ {the (inverse_application
(the (weight g g')) e) | e. minimal_winning_budget e g'})
  ∧ e = (energy_sup {strat g' | g'. weight g g' ≠ None})"

    proof-
      from A1 A2 have "winning_budget_len e g" using energy_Min_def by simp
      from A1 A2 have "∀e' ≠ e. order e' e ⇒ ¬ winning_budget_len e' g" using
energy_Min_def
      using mem_Collect_eq by auto

      hence "e ∈ energy_Min {e'}. ∃ strat. (∀g'. weight g g' ≠ None ⇒ strat g'
∈ {the (inverse_application (the (weight g g')) e) | e. winning_budget_len e g'})
  ∧ e' = (energy_sup {strat g' | g'. weight g g' ≠ None})"
      using <winning_budget_len e g> A1 A2 min_winning_budget_d_iff_energy_Min
      by (meson winning_budget_len.cases)
      hence "∃ strat. (∀g'. weight g g' ≠ None ⇒ strat g' ∈ {the (inverse_application
(the (weight g g')) e) | e. winning_budget_len e g'})
  ∧ e = (energy_sup {strat g' | g'. weight g g' ≠ None})" using
energy_Min_def by auto

      from this obtain strat where Strat: "(∀g'. weight g g' ≠ None ⇒ strat
g' ∈ {the (inverse_application (the (weight g g')) e) | e. winning_budget_len e
g'})
  ∧ e = (energy_sup {strat g' | g'. weight g g' ≠ None})" by auto
      define strat_e where "strat_e ≡ λg'. (SOME e. strat g' = the (inverse_application
(the (weight g g')) e) ∧ winning_budget_len e g'"

      have Strat_E: "∀g'. weight g g' ≠ None ⇒ strat g' = the (inverse_application
(the (weight g g')) (strat_e g')) ∧ winning_budget_len (strat_e g') g'"
      proof-
        fix g'
        have Strat_E: "strat_e g' = (SOME e. strat g' = the (inverse_application
(the (weight g g')) e) ∧ winning_budget_len e g'" using strat_e_def by simp
        assume "weight g g' ≠ None"
        hence "strat g' ∈ {the (inverse_application (the (weight g g')) e) | e.
winning_budget_len e g'}" using Strat by simp
        hence "∃ e. strat g' = the (inverse_application (the (weight g g')) e)
  ∧ winning_budget_len e g'" by auto
        thus "strat g' = the (inverse_application (the (weight g g')) (strat_e

```

```

g'))  $\wedge$  winning_budget_len (strat_e g') g'"
    using Strat_E by (smt (verit, del_insts) some_eq_ex)
qed

    have exists: " $\wedge g'. \text{weight } g \ g' \neq \text{None} \implies \exists e. e \in \text{energy\_Min } \{e. \text{winning\_budget\_len } e \ g' \wedge \text{order } e \ (\text{strat\_e } g')\}$ "
    proof-
      fix g'
      assume "weight g g'  $\neq$  None"
      hence notempty: "strat_e g'  $\in$  {e. winning_budget_len e g'  $\wedge$  order e (strat_e g')}" using Strat_E energy_order ordering_def
      using partial_preordering.refl by fastforce
      have " $\forall e \in \{e. \text{winning\_budget\_len } e \ g' \wedge \text{order } e \ (\text{strat\_e } g')\}. e \in \text{energies}$ "
      using winning_budget_len.cases by auto
      hence "{ }  $\neq$  energy_Min {e. winning_budget_len e g'  $\wedge$  order e (strat_e g')}"
      using energy_Min_not_empty notempty
      by (metis (no_types, lifting) empty_iff subsetI)
      thus " $\exists e. e \in \text{energy\_Min } \{e. \text{winning\_budget\_len } e \ g' \wedge \text{order } e \ (\text{strat\_e } g')\}$ " by auto
    qed

    define strat' where "strat'  $\equiv$   $\lambda g'. \text{the } (\text{inverse\_application } (\text{the } (\text{weight } g \ g')) (\text{SOME } e. e \in \text{energy\_Min } \{e. \text{winning\_budget\_len } e \ g' \wedge \text{order } e \ (\text{strat\_e } g')\}))$ "

    have " $(\forall g'. \text{weight } g \ g' \neq \text{None} \longrightarrow \text{strat'} \ g' \in \{\text{the } (\text{inverse\_application } (\text{the } (\text{weight } g \ g')) \ e) \mid e. \text{minimal\_winning\_budget } e \ g'\})$ "
       $\wedge e = (\text{energy\_sup } \{\text{strat'} \ g' \mid g'. \text{weight } g \ g' \neq \text{None}\})$ "
    proof
      show win: " $\forall g'. \text{weight } g \ g' \neq \text{None} \longrightarrow \text{strat'} \ g' \in \{\text{the } (\text{inverse\_application } (\text{the } (\text{weight } g \ g')) \ e) \mid e. \text{minimal\_winning\_budget } e \ g'\}$ "
      proof
        fix g'
        show "weight g g'  $\neq$  None  $\longrightarrow$  strat' g'  $\in$  {the (inverse_application (the (weight g g')) e) | e. minimal_winning_budget e g'}"
        proof
          assume "weight g g'  $\neq$  None"
          hence "strat' g' = the (inverse_application (the (weight g g')) (SOME e. e  $\in$  energy_Min {e. winning_budget_len e g'  $\wedge$  order e (strat_e g')}))"
          using strat'_def by auto
          hence " $\exists e. e \in \text{energy\_Min } \{e. \text{winning\_budget\_len } e \ g' \wedge \text{order } e \ (\text{strat\_e } g')\} \wedge \text{strat'} \ g' = \text{the } (\text{inverse\_application } (\text{the } (\text{weight } g \ g')) \ e)$ "
          using exists <weight g g'  $\neq$  None> some_eq_ex
          by (metis (mono_tags))
          from this obtain e where E: "e  $\in$  energy_Min {e. winning_budget_len e g'  $\wedge$  order e (strat_e g')}  $\wedge$  strat' g' = the (inverse_application (the (weight g g')) e)" by auto
          have "minimal_winning_budget e g'"
          unfolding energy_Min_def
          proof
            show "e  $\in$  a_win g'  $\wedge$  ( $\forall e' \in \text{a\_win } g'. e \neq e' \longrightarrow \neg e' \ e \leq e$ )"
            proof
              show "e  $\in$  a_win g'"
              using E energy_Min_def
              by simp
            qed
          qed
        qed
      qed
    qed

```

```

show "( $\forall e' \in a\_win\ g'.\ e \neq e' \longrightarrow \neg e' \leq e$ )"
proof
  fix e'
  show " $e' \in a\_win\ g' \implies e \neq e' \longrightarrow \neg e' \leq e$ "
  proof
    assume " $e' \in a\_win\ g'$ " and " $e \neq e'$ "
    hence "winning_budget_len e' g'" by simp
    show " $\neg e' \leq e$ "
    proof
      assume " $e' \leq e$ "
      have "order e (strat_e g'" using E energy_Min_def by auto
      hence "order e' (strat_e g'" using  $\langle e' \leq e \rangle$  energy_order
ordering_def
      by (metis (no_types, lifting) partial_preordering_def)

      hence " $e' \in \{e.\ winning\_budget\_len\ e\ g' \wedge order\ e\ (strat\_e\ g')\}$ " using  $\langle winning\_budget\_len\ e' \ g' \rangle$  by auto
      thus "False" using E  $\langle e \neq e' \rangle \langle e' \leq e \rangle$  energy_Min_def
      by fastforce
    qed
  qed
qed
qed
qed
qed
qed
thus "strat' g'  $\in \{the\ (inverse\_application\ (the\ (weight\ g\ g'))\ e)\ | e.\ minimal\_winning\_budget\ e\ g'\}$ " using E
by blast
qed
qed

have "( $\bigwedge g'.\ weight\ g\ g' \neq None \implies$ 
  strat' g'  $\in \{the\ (inverse\_application\ (the\ (weight\ g\ g'))\ e)\ | e.\ winning\_budget\_len$ 
e g'})"
  using win energy_Min_def
  by (smt (verit, del_insts) mem_Collect_eq)
  hence win: "winning_budget_len (energy_sup {strat' g' | g'. weight g g'
 $\neq None$ }) g"
    using sup_inv_in_winning_budget A1 A2 by simp

  have "order (energy_sup {strat' g' | g'. weight g g'  $\neq None$ }) (energy_sup
{strat g' | g'. weight g g'  $\neq None$ })"
  proof (cases " $\{g'.\ weight\ g\ g' \neq None\} = \{\}$ ")
    case True
    then show ?thesis using bounded_join_semilattice
    by auto
  next
    case False
    show ?thesis
    proof (rule energy_sup_leq_energy_sup)
      show " $\{strat' g' | g'.\ weight\ g\ g' \neq None\} \neq \{\}$ " using False by simp

      have A: " $\bigwedge a.\ a \in \{strat' g' | g'.\ weight\ g\ g' \neq None\} \implies \exists b \in \{strat$ 
g' | g'. weight g g'  $\neq None\}.\ order\ a\ b \wedge a \in energies$ "
      proof-
        fix a
        assume "a  $\in \{strat' g' | g'.\ weight\ g\ g' \neq None\}$ "

```

```

    hence "∃g'. a = strat' g' ∧ weight g g' ≠ None" by auto
    from this obtain g' where "a = strat' g' ∧ weight g g' ≠ None"
by auto

    have "(strat' g') = (the (inverse_application (the (weight g g'))
    (SOME e. e ∈ energy_Min {e. winning_budget_len e g' ∧ order e
(strat_e g'))})))" using strat'_def by auto
    hence "∃e. e ∈ energy_Min {e. winning_budget_len e g' ∧ order e
(strat_e g')} ∧ strat' g' = the (inverse_application (the (weight g g')) e)"
    using exists <a = strat' g' ∧ weight g g' ≠ None> some_eq_ex
    by (metis (mono_tags))
    from this obtain e where E: "e ∈ energy_Min {e. winning_budget_len
e g' ∧ order e (strat_e g')} ∧ strat' g' = the (inverse_application (the (weight
g g')) e)" by auto
    hence "order e (strat_e g'" using energy_Min_def by auto

    hence "a ∈ energies " using <a = strat' g' ∧ weight g g' ≠ None>
energy_Min_def
    by (metis (no_types, lifting) E inv_well_defined mem_Collect_eq
nonpos_eq_pos winning_buget_len_is_wb)

    have leq: "order (the (inverse_application (the (weight g g')) e))
(the (inverse_application (the (weight g g')) (strat_e g')))"
    proof(rule inverse_monotonic)
    show "order e (strat_e g'" using <order e (strat_e g'))>.
    show "weight g g' ≠ None" using <a = strat' g' ∧ weight g g'
≠ None> by simp
    from E have "e ∈ {e. winning_budget_len e g' ∧ order e (strat_e
g'))}" using energy_Min_def
    by auto
    hence "winning_budget_len e g'"
    by simp
    thus "e ∈ energies"
    using winning_budget_len.simps
    by blast
    thus "inverse_application (the (weight g g')) e ≠ None"
    using inv_well_defined <weight g g' ≠ None>
    by simp
    qed
    have "the (inverse_application (the (weight g g')) (strat_e g'))
= strat g'" using Strat_E <a = strat' g' ∧ weight g g' ≠ None> by auto
    hence "order (strat' g') (strat g'" using leq E by simp
    hence "∃b∈{strat g' |g'. weight g g' ≠ None}. order a b" using
<a = strat' g' ∧ weight g g' ≠ None> by auto
    thus "∃b∈{strat g' |g'. weight g g' ≠ None}. order a b ∧ a ∈ energies"
using <a ∈ energies> by simp
    qed
    thus "∧a. a ∈ {strat' g' |g'. weight g g' ≠ None} ⇒ ∃b∈{strat
g' |g'. weight g g' ≠ None}. order a b" by simp
    show "∧a. a ∈ {strat' g' |g'. weight g g' ≠ None} ⇒ a ∈ energies
" using A by simp

    have "{strat' g' |g'. weight g g' ≠ None} ⊆ {strat' g' |g'. g' ∈
positions}" by auto
    thus "finite {strat' g' |g'. weight g g' ≠ None}" using finite_positions
rev_finite_subset by fastforce

```

```

      have "{strat g' | g'. weight g g' ≠ None} ⊆ {strat g' | g'. g' ∈ positions}"
    by auto
      thus "finite {strat g' | g'. weight g g' ≠ None}" using finite_positions
    rev_finite_subset by fastforce
      show "{strat g' | g'. weight g g' ≠ None} ⊆ energies"
      by (smt (verit) Strat_E inv_well_defined mem_Collect_eq subsetI
    winning_budget_len.simps)
      qed
      qed
      thus "e = energy_sup {strat' g' | g'. weight g g' ≠ None}" using <g ∉
    attacker> Strat win
      by (metis (no_types, lifting) <∀e'. e' ≠ e → order e' e → ¬ winning_budget_len
    e' g>)
      qed
      thus ?thesis by blast
      qed
      qed

    show "minimal_winning_budget e g =
      (e ∈ energy_Min
        {e''.
          ∃ strat.
            (∀ g'. weight g g' ≠ None →
              strat g'
                ∈ {inv_upd (the (weight g g')) e | e. minimal_winning_budget
    e g'}) ∧
              e'' = energy_sup {strat g' | g'. weight g g' ≠ None}})"

    proof
      assume "minimal_winning_budget e g"
      hence exist: "∃ strat. (∀ g'. weight g g' ≠ None → strat g' ∈ {the (inverse_application
    (the (weight g g')) e) | e. minimal_winning_budget e g'})
        ∧ e = (energy_sup {strat g' | g'. weight g g' ≠ None})"
      using min_winning_budget_is_minimal_inv_d <g ∉ attacker> by simp
      have "∧ e''. e'' e < e ⇒ ¬ e'' ∈ {e''. ∃ strat. (∀ g'. weight g g' ≠ None
    → strat g' ∈ {the (inverse_application (the (weight g g')) e) | e. minimal_winning_budget
    e g'})
        ∧ e'' = (energy_sup {strat g' | g'. weight g g' ≠ None}})"

      proof-
        fix e''
        show "e'' e < e ⇒ ¬ e'' ∈ {e''. ∃ strat. (∀ g'. weight g g' ≠ None →
    strat g' ∈ {the (inverse_application (the (weight g g')) e) | e. minimal_winning_budget
    e g'})
        ∧ e'' = (energy_sup {strat g' | g'. weight g g' ≠ None}})"

      proof-
        assume "e'' e < e"
        show "¬ e'' ∈ {e''. ∃ strat. (∀ g'. weight g g' ≠ None → strat g' ∈
    {the (inverse_application (the (weight g g')) e) | e. minimal_winning_budget e g'})
        ∧ e'' = (energy_sup {strat g' | g'. weight g g' ≠ None}})"

      proof
        assume "e'' ∈ {e''. ∃ strat. (∀ g'. weight g g' ≠ None → strat g' ∈
    {the (inverse_application (the (weight g g')) e) | e. minimal_winning_budget e g'})
        ∧ e'' = (energy_sup {strat g' | g'. weight g g' ≠ None}})"
        hence "∃ strat. (∀ g'. weight g g' ≠ None → strat g' ∈ {the (inverse_application
    (the (weight g g')) e) | e. minimal_winning_budget e g'})
        ∧ e'' = (energy_sup {strat g' | g'. weight g g' ≠ None})" by auto
        from this obtain strat where E': "(∀ g'. weight g g' ≠ None → strat

```



```

g' ∈ {the (inverse_application (the (weight g g')) e) | e. minimal_winning_budget
e g'})

      ∧ e'' = (energy_sup {strat g' | g'. weight g g' ≠ None})" by auto
hence "∧g'. weight g g' ≠ None ⇒
strat g' ∈ {inv_upd (the (weight g g')) e | e. winning_budget_len e g'}"
using energy_Min_def
by (smt (verit, del_insts) mem_Collect_eq)
hence "winning_budget_len (energy_sup {strat g' | g'. weight g g' ≠ None})
g" using sup_inv_in_winning_budget <g ∉ attacker> by simp
hence "winning_budget_len e'' g" using E'' by simp
thus "False" using <e'' e< e> <minimal_winning_budget e g> energy_Min_def
by auto

qed
qed
qed
thus "e ∈ energy_Min {e''. ∃ strat. (∀ g'. weight g g' ≠ None ⇒ strat g' ∈
{the (inverse_application (the (weight g g')) e) | e. minimal_winning_budget
e g'})
      ∧ e'' = (energy_sup {strat g' | g'. weight g g' ≠ None})}"
using exist energy_Min_def by (smt (verit) mem_Collect_eq)
next
assume A: "(e ∈ energy_Min {e''. ∃ strat. (∀ g'. weight g g' ≠ None ⇒ strat
g' ∈ {the (inverse_application (the (weight g g')) e) | e. minimal_winning_budget
e g'})
      ∧ e'' = (energy_sup {strat g' | g'. weight g g' ≠ None})}"
hence emin: "e ∈ energy_Min {e''. ∃ strat. (∀ g'. weight g g' ≠ None ⇒ strat
g' ∈ {the (inverse_application (the (weight g g')) e) | e. minimal_winning_budget
e g'})
      ∧ e'' = (energy_sup {strat g' | g'. weight g g' ≠ None})}" using
A by simp
hence "∃ strat. (∀ g'. weight g g' ≠ None ⇒ strat g' ∈ {the (inverse_application
(the (weight g g')) e) | e. minimal_winning_budget e g'})
      ∧ e = (energy_sup {strat g' | g'. weight g g' ≠ None})" using
energy_Min_def by auto
hence "∃ strat.
      (∀ g'. weight g g' ≠ None ⇒
        strat g' ∈ {inv_upd (the (weight g g')) e | e. winning_budget_len
e g'}) ∧
        e = energy_sup {strat g' | g'. weight g g' ≠ None}" using energy_Min_def
by (smt (verit, ccfv_threshold) mem_Collect_eq)
hence element_of: "e ∈ {e''.
      ∃ strat.
        (∀ g'. weight g g' ≠ None ⇒
          strat g' ∈ {inv_upd (the (weight g g')) e | e. winning_budget_len
e g'}) ∧
          e'' = energy_sup {strat g' | g'. weight g g' ≠ None}" by auto
hence "e ∈ energies"
using <g ∉ attacker> sup_inv_in_winning_budget winning_budget_len.simps
by blast

have "∧e'. e' e< e ⇒ e' ∉ {e''.
      ∃ strat.
        (∀ g'. weight g g' ≠ None ⇒
          strat g' ∈ {inv_upd (the (weight g g')) e | e. winning_budget_len
e g'}) ∧
          e'' = energy_sup {strat g' | g'. weight g g' ≠ None}"

```

```

proof
  fix e'
  assume "e' e< e"
  assume A: "e' ∈ {e'}. ∃ strat.
    (∀ g'. weight g g' ≠ None →
      strat g' ∈ {inv_upd (the (weight g g')) e | e. winning_budget_len
e g'}) ∧
      e' = energy_sup {strat g' | g'. weight g g' ≠ None}"
  hence "∃ strat.
    (∀ g'. weight g g' ≠ None →
      strat g' ∈ {inv_upd (the (weight g g')) e | e. winning_budget_len
e g'}) ∧
      e' = energy_sup {strat g' | g'. weight g g' ≠ None}" by auto
  from this obtain strat where Strat: "(∀ g'. weight g g' ≠ None →
    strat g' ∈ {inv_upd (the (weight g g')) e | e. winning_budget_len
e g'}) ∧
    e' = energy_sup {strat g' | g'. weight g g' ≠ None}" by auto
  hence "e' ∈ energies"
  proof-
    have "{strat g' | g'. weight g g' ≠ None} ⊆ {strat g' | g'. g' ∈ positions}"
  by auto
    hence fin: "finite {strat g' | g'. weight g g' ≠ None}"
    using finite_positions
    using rev_finite_subset by fastforce
    have "{strat g' | g'. weight g g' ≠ None} ⊆ energies" using Strat
    by (smt (verit, best) inv_well_defined mem_Collect_eq nonpos_eq_pos
subsetI winning_budget_len_is_wb)
    thus ?thesis using bounded_join_semilattice fin Strat
    by auto
  qed

  define the_e where "the_e ≡ λg'. (SOME x. strat g' = inv_upd (the (weight
g g')) x ∧ winning_budget_len x g'))"

  define strat' where "strat' ≡ λg'. (SOME x. x ∈ {inv_upd (the (weight g
g')) x |
                                                                    x. (minimal_winning_budget
x g' ∧ x e≤ the_e g'))}"

  have some_not_empty: "λg'. weight g g' ≠ None ⇒ {inv_upd (the (weight
g g')) x | x. (minimal_winning_budget x g' ∧ x e≤ the_e g')} ≠ {}"
  proof-
    fix g'
    assume "weight g g' ≠ None"
    hence "strat g' ∈ {inv_upd (the (weight g g')) e | e. winning_budget_len
e g'}" using Strat by auto
    hence "∃ e. strat g' = inv_upd (the (weight g g')) e ∧ winning_budget_len
e g'" by auto
    hence "strat g' = inv_upd (the (weight g g')) (the_e g') ∧ winning_budget_len
(the_e g') g'" using the_e_def some_eq_ex
    by (metis (mono_tags, lifting))
    hence "the_e g' ∈ {x. winning_budget_len x g'}" by auto
    hence "∃ x. (minimal_winning_budget x g' ∧ x e≤ the_e g'" using energy_Min_contains
<the_e g' ∈ {x. winning_budget_len x g'}>
    by (metis mem_Collect_eq nonpos_eq_pos subsetI winning_budget_len_is_wb)
    hence "{x. (minimal_winning_budget x g' ∧ x e≤ the_e g')} ≠ {}" by auto

```

```

      thus "{inv_upd (the (weight g g')) x | x. (minimal_winning_budget x g' ∧
x e ≤ the_e g'))} ≠ {}"
      by auto
    qed

    hence len: "∧a. a ∈ {strat' g' | g'. weight g g' ≠ None} ⇒ a ∈ energies"

  proof-
    fix a
    assume "a ∈ {strat' g' | g'. weight g g' ≠ None}"
    hence "∃ g'. a = strat' g' ∧ weight g g' ≠ None" by auto
    from this obtain g' where "a = strat' g' ∧ weight g g' ≠ None" by auto
    hence some_not_empty: "{inv_upd (the (weight g g')) x | x. (minimal_winning_budget
x g' ∧ x e ≤ the_e g'))} ≠ {}"
      using some_not_empty by blast

    have "strat' g' = (SOME x. x ∈ {inv_upd (the (weight g g')) x |
x. (minimal_winning_budget
x g' ∧ x e ≤ the_e g'))}"
      using strat'_def by auto
    hence "strat' g' ∈ {inv_upd (the (weight g g')) x | x. (minimal_winning_budget
x g' ∧ x e ≤ the_e g'))}"
      using some_not_empty some_in_eq
      by (smt (verit, ccfv_SIG) Eps_cong)
    hence "∃ x. strat' g' = inv_upd (the (weight g g')) x ∧ minimal_winning_budget
x g' ∧ x e ≤ the_e g'"
      by simp
    from this obtain x where X: "strat' g' = inv_upd (the (weight g g')) x
∧ minimal_winning_budget x g' ∧ x e ≤ the_e g'" by auto
    hence "winning_budget_len x g'" using energy_Min_def by simp
    hence "x ∈ energies" using winning_budget_len.simps
      by blast
    have "a = inv_upd (the (weight g g')) x" using X <a = strat' g' ∧ weight
g g' ≠ None> by simp
    thus "a ∈ energies"
      using <a = strat' g' ∧ weight g g' ≠ None> <x ∈ energies> inv_well_defined
by blast
  qed

  show "False"
  proof(cases "deadend g")
    case True

    from emin have "∃ strat.
      (∀ g'. weight g g' ≠ None →
        strat g' ∈ {inv_upd (the (weight g g')) e | e. minimal_winning_budget
e g'}) ∧
      e = energy_sup {strat g' | g'. weight g g' ≠ None}" using energy_Min_def
by auto
    from this obtain strat where "(∀ g'. weight g g' ≠ None →
      strat g' ∈ {inv_upd (the (weight g g')) e | e. minimal_winning_budget
e g'}) ∧
      e = energy_sup {strat g' | g'. weight g g' ≠ None}" by auto
    hence "e = energy_sup {}" using True by simp

    have "energy_sup {} ∈ energies ∧ (∀ s. s ∈ {} → order s (energy_sup

```

```

{}}))  $\wedge$  ( $e' \in \text{energies} \wedge (\forall s. s \in \{\} \longrightarrow \text{order } s \ e') \longrightarrow \text{order } (\text{energy\_sup } \{\}) \ e')$ "
  using bounded_join_semilattice by blast
  hence "e  $\leq$  e'" using <e = energy_sup {}> <e'  $\in$  energies> by auto

  from <e' e< e> have "e'  $\leq$  e  $\wedge$  e'  $\neq$  e" using energy_order ordering_def
ordering.strict_iff_order
  by simp
  hence "e'  $\leq$  e" by simp
  hence "e' = e" using <e  $\leq$  e'> using energy_order ordering_def ordering.antisym
  by fastforce
  thus ?thesis using <e'  $\leq$  e  $\wedge$  e'  $\neq$  e> by auto
next
  case False
  hence notempty: "{strat' g' | g'. weight g g'  $\neq$  None}  $\neq$  {}" by auto

  have fin: "finite {strat' g' | g'. weight g g'  $\neq$  None}  $\wedge$  finite {strat
g' | g'. weight g g'  $\neq$  None}"
  proof
    have "{strat' g' | g'. weight g g'  $\neq$  None}  $\subseteq$  {strat' g' | g'. g'  $\in$  positions}"
  by auto
    thus "finite {strat' g' | g'. weight g g'  $\neq$  None}" using finite_positions
    using finite_image_set rev_finite_subset by fastforce
    have "{strat g' | g'. weight g g'  $\neq$  None}  $\subseteq$  {strat g' | g'. g'  $\in$  positions}"
  by auto
    thus "finite {strat g' | g'. weight g g'  $\neq$  None}" using finite_positions
    using finite_image_set rev_finite_subset by fastforce
  qed

  have " $\wedge$  g'. weight g g'  $\neq$  None  $\implies$  strat' g'  $\leq$  strat g'"
  proof-
    fix g'
    assume "weight g g'  $\neq$  None"
    hence some_not_empty: "{inv_upd (the (weight g g')) x | x. (minimal_winning_budget
x g'  $\wedge$  x  $\leq$  the_e g'))}  $\neq$  {}"
    using some_not_empty by auto
    have "strat' g' = (SOME x. x  $\in$  {inv_upd (the (weight g g')) x |
x. (minimal_winning_budget
x g'  $\wedge$  x  $\leq$  the_e g'))}"
    using strat'_def by auto
    hence "strat' g'  $\in$  {inv_upd (the (weight g g')) x | x. (minimal_winning_budget
x g'  $\wedge$  x  $\leq$  the_e g'))}"
    using some_not_empty some_in_eq
    by (smt (verit, ccfv_SIG) Eps_cong)
    hence " $\exists$  x. strat' g' = inv_upd (the (weight g g')) x  $\wedge$  minimal_winning_budget
x g'  $\wedge$  x  $\leq$  the_e g'"
    by simp
    from this obtain x where X: "strat' g' = inv_upd (the (weight g g'))
x  $\wedge$  minimal_winning_budget x g'  $\wedge$  x  $\leq$  the_e g'" by auto
    hence "x  $\in$  energies" using winning_budget_len.simps energy_Min_def
    by (metis (mono_tags, lifting) mem_Collect_eq)
    hence "strat' g'  $\leq$  inv_upd (the (weight g g')) (the_e g'" using inverse_monoton
X
    by (metis <weight g g'  $\neq$  None> inv_well_defined)

    have "strat g'  $\in$  {inv_upd (the (weight g g')) e | e. winning_budget_len
e g'}" using Strat <weight g g'  $\neq$  None> by auto

```

```

      hence "∃e. strat g' = inv_upd (the (weight g g')) e ∧ winning_budget_len
e g'" by auto
      hence "strat g' = inv_upd (the (weight g g')) (the_e g') ∧ winning_budget_len
(the_e g') g'" using the_e_def some_eq_ex
      by (metis (mono_tags, lifting))
      thus "strat' g' e ≤ strat g'" using <strat' g' e ≤ inv_upd (the (weight
g g')) (the_e g'))> by auto
    qed

    hence leq: "(∧a. a ∈ {strat' g' | g'. weight g g' ≠ None} ⇒ ∃b∈{strat
g' | g'. weight g g' ≠ None}. a e ≤ b)" by auto
    have in_energy: "{strat g' | g'. weight g g' ≠ None} ⊆ energies ∧ {strat'
g' | g'. weight g g' ≠ None} ⊆ energies"
    proof
      show "{strat g' | g'. weight g g' ≠ None} ⊆ energies"
      using Strat
      by (smt (verit, ccfv_threshold) inv_well_defined mem_Collect_eq nonpos_eq_pos
subsetI winning_buget_len_is_wb)
      show "{strat' g' | g'. weight g g' ≠ None} ⊆ energies"
      unfolding strat'_def
      using len strat'_def by blast
    qed

    hence "energy_sup {strat' g' | g'. weight g g' ≠ None} e ≤ e'"
      using notempty len Strat energy_sup_leq_energy_sup fin leq
      by presburger
    hence le: "energy_sup {strat' g' | g'. weight g g' ≠ None} e < e" using
<e' e < e> in_energy
      by (smt (verit) <e ∈ energies> <e' ∈ energies> energy_order energy_wqo
fin galois_energy_game.bounded_join_semilattice galois_energy_game_axioms ordering.antisym
transp_onD wqo_on_imp_transp_on)

    have "energy_sup {strat' g' | g'. weight g g' ≠ None} ∈ {e''. ∃strat.
(∀g'. weight g g' ≠ None → strat g' ∈ {the (inverse_application (the (weight
g g')) e) | e. minimal_winning_budget e g'})}
      ∧ e'' = (energy_sup {strat g' | g'. weight g g' ≠ None})}"
    proof-
      have "(∀g'. weight g g' ≠ None → strat' g' ∈ {the (inverse_application
(the (weight g g')) e) | e. minimal_winning_budget e g'})"
      proof
        fix g'
        show "weight g g' ≠ None →
          strat' g' ∈ {inv_upd (the (weight g g')) e | e. minimal_winning_budget
e g'}"
      proof
        assume "weight g g' ≠ None"
        hence some_not_empty: "{inv_upd (the (weight g g')) x | x. minimal_winning_budg
e g' ∧ x e ≤ the_e g'} ≠ {}"
        using some_not_empty by auto
        have "strat' g' = (SOME x. x ∈ {inv_upd (the (weight g g')) x |
          x. (minimal_winning_budget
x g' ∧ x e ≤ the_e g')})"
        using strat'_def by auto
        hence "strat' g' ∈ {inv_upd (the (weight g g')) x | x. (minimal_winning_budget
x g' ∧ x e ≤ the_e g')}"
        using some_not_empty some_in_eq

```

```

      by (smt (verit, ccfv_SIG) Eps_cong)
      thus "strat' g' ∈ {inv_upd (the (weight g g')) e | e. minimal_winning_budget
e g'}"

      by auto
      qed
      qed
      hence "∃ strat. (∀ g'. weight g g' ≠ None → strat g' ∈ {the (inverse_application
(the (weight g g')) e) | e. minimal_winning_budget e g'})
      ∧ energy_sup {strat' g' | g'. weight g g' ≠ None} = (energy_sup
{strat g' | g'. weight g g' ≠ None})"
      by blast
      then show ?thesis
      by simp
      qed

      then show ?thesis
      using energy_Min_def emin le
      by (smt (verit) mem_Collect_eq)
      qed
      qed

      hence "e ∈ energy_Min
      {e'".
      ∃ strat.
      (∀ g'. weight g g' ≠ None →
      strat g' ∈ {inv_upd (the (weight g g')) e | e. winning_budget_len
e g'}) ∧
      e' = energy_sup {strat g' | g'. weight g g' ≠ None}" using element_of
energy_Min_def
      by (smt (verit) mem_Collect_eq)
      thus "minimal_winning_budget e g"
      using min_winning_budget_d_iff_energy_Min <g ∉ attacker> <e ∈ energies>
by blast
      qed
      qed

      have "∧ g e. e ∈ a_win_min g ⇒ e ∈ energies"
      using winning_budget_len.simps energy_Min_def
      by (metis (no_types, lifting) mem_Collect_eq)
      hence D: "∧ g e. e ∈ a_win_min g = (e ∈ a_win_min g ∧ e ∈ energies)" by auto
      fix g
      show "iteration a_win_min g = a_win_min g"
      proof (cases "g ∈ attacker")
      case True
      have "a_win_min g = {e. minimal_winning_budget e g}" by simp
      hence "a_win_min g = energy_Min {e. ∃ g' e'.
      weight g g' ≠ None ∧
      minimal_winning_budget e' g' ∧ e = inv_upd (the (weight g g'))
e'}"
      using minimal_winning_budget_attacker True by simp
      also have "... = energy_Min {inv_upd (the (weight g g')) e' | g' e'.
      weight g g' ≠ None ∧
      minimal_winning_budget e' g' }"
      by meson
      also have "... = energy_Min {inv_upd (the (weight g g')) e' | e' g'.

```

```

weight g g' ≠ None ∧ e' ∈ a_win_min g'}"
by (metis (no_types, lifting) mem_Collect_eq)
also have "... = energy_Min {inv_upd (the (weight g g')) e' | e' g'. e' ∈ energies
Λ
weight g g' ≠ None ∧ e' ∈ a_win_min g'}"
using D by meson
also have "... = iteration a_win_min g" using iteration_def True by simp
finally show ?thesis by simp
next
case False
have "a_win_min g = {e. minimal_winning_budget e g}" by simp
hence minwin: "a_win_min g = energy_Min {e''. ∃ strat. (∀ g'. weight g g' ≠ None
→ strat g' ∈ {the (inverse_application (the (weight g g')) e) | e. minimal_winning_budget
e g'})}"
Λ e'' = (energy_sup {strat g' | g'. weight g g' ≠ None})}"
using minimal_winning_budget_defender False by simp
hence "a_win_min g = energy_Min {energy_sup {strat g' | g'. weight g g' ≠ None}
| strat. (∀ g'. weight g g' ≠ None → strat g' ∈ {the (inverse_application (the
(weight g g')) e) | e. minimal_winning_budget e g'})}"
by (smt (z3) Collect_cong)
have iteration: "energy_Min {energy_sup {inv_upd (the (weight g g')) (e_index
g') | g'. weight g g' ≠ None} |
e_index. ∀ g'. weight g g' ≠ None → ((e_index g') ∈ energies ∧ e_index
g' ∈ a_win_min g')} = iteration a_win_min g"
using iteration_def False by simp

have "{e''. ∃ strat. (∀ g'. weight g g' ≠ None → strat g' ∈ {the (inverse_application
(the (weight g g')) e) | e. minimal_winning_budget e g'})"
Λ e'' = (energy_sup {strat g' | g'. weight g g' ≠ None})}
={energy_sup {inv_upd (the (weight g g')) (e_index g') | g'. weight g g'
≠ None} |
e_index. ∀ g'. weight g g' ≠ None → ((e_index g') ∈ energies ∧ e_index
g' ∈ a_win_min g')}}"
proof
show "{e''. ∃ strat. (∀ g'. weight g g' ≠ None → strat g' ∈ {the (inverse_application
(the (weight g g')) e) | e. minimal_winning_budget e g'})"
Λ e'' = (energy_sup {strat g' | g'. weight g g' ≠ None})}
⊆ {energy_sup {inv_upd (the (weight g g')) (e_index g') | g'. weight
g g' ≠ None} |
e_index. ∀ g'. weight g g' ≠ None → ((e_index g') ∈ energies ∧ e_index
g' ∈ a_win_min g')}}"
proof
fix e
assume "e ∈ {e''. ∃ strat. (∀ g'. weight g g' ≠ None → strat g' ∈ {the
(inverse_application (the (weight g g')) e) | e. minimal_winning_budget e g'})"
Λ e'' = (energy_sup {strat g' | g'. weight g g' ≠ None})}"
hence "∃ strat. (∀ g'. weight g g' ≠ None → strat g' ∈ {the (inverse_application
(the (weight g g')) e) | e. minimal_winning_budget e g'})"
Λ e = (energy_sup {strat g' | g'. weight g g' ≠ None})"
by auto
from this obtain strat where S: "(∀ g'. weight g g' ≠ None → strat g'
∈ {the (inverse_application (the (weight g g')) e) | e. minimal_winning_budget e
g'})"
Λ e = (energy_sup {strat g' | g'. weight g g' ≠ None})"
by auto
define e_index where "e_index ≡ λ g'. (SOME e''. e'' ∈ a_win_min g' ∧ strat

```

```

g' = the (inverse_application (the (weight g g')) e'))"
  hence index: " $\bigwedge g'. \text{weight } g \ g' \neq \text{None} \implies (\text{e\_index } g') \in \text{a\_win\_min } g' \wedge$ "
strat g' = the (inverse_application (the (weight g g')) (e_index g'))"
  proof-
    fix g'
    have I: "e_index g' = (SOME e''. e''  $\in$  a_win_min g'  $\wedge$  strat g' = the (inverse_applicatio
(the (weight g g')) e'))"
      using e_index_def by simp
      assume "weight g g'  $\neq$  None"
      hence "strat g'  $\in$  {the (inverse_application (the (weight g g')) e) | e.
minimal_winning_budget e g'}"
      using S by simp
      hence "strat g'  $\in$  {the (inverse_application (the (weight g g')) e) | e.
e  $\in$  a_win_min g'}" by simp
      hence " $\exists e''. e'' \in \text{a\_win\_min } g' \wedge \text{strat } g' = \text{the (inverse\_application}$ 
(the (weight g g')) e'))" by auto
      thus "(e_index g')  $\in$  a_win_min g'  $\wedge$  strat g' = the (inverse_application
(the (weight g g')) (e_index g'))"
      unfolding e_index_def using some_eq_ex
      by (smt (verit, del_insts))
    qed

    show "e  $\in$  {energy_sup {inv_upd (the (weight g g')) (e_index g') | g'. weight
g g'  $\neq$  None} |
      e_index.  $\forall g'. \text{weight } g \ g' \neq \text{None} \implies ((\text{e\_index } g') \in \text{energies} \wedge \text{e\_index}$ 
g'  $\in$  a_win_min g'))}"
      proof
        show " $\exists \text{e\_index. } e = \text{energy\_sup \{inv\_upd (the (weight g g')) (e\_index g')\}}$ 
|g'. weight g g'  $\neq$  None}  $\wedge$ 
      ( $\forall g'. \text{weight } g \ g' \neq \text{None} \implies ((\text{e\_index } g') \in \text{energies} \wedge \text{e\_index } g' \in \text{a\_win\_min}$ 
g')))"
        proof
          show "e = energy_sup {inv_upd (the (weight g g')) (e_index g') | g'.
weight g g'  $\neq$  None}  $\wedge$ 
      ( $\forall g'. \text{weight } g \ g' \neq \text{None} \implies ((\text{e\_index } g') \in \text{energies} \wedge \text{e\_index } g' \in \text{a\_win\_min}$ 
g')))"
          proof
            show "e = energy_sup {inv_upd (the (weight g g')) (e_index g') | g'.
weight g g'  $\neq$  None}"
            using index S
            by (smt (verit) Collect_cong)
            have " $\forall g'. \text{weight } g \ g' \neq \text{None} \implies \text{e\_index } g' \in \text{a\_win\_min } g'$ "
            using index by simp
            thus " $\forall g'. \text{weight } g \ g' \neq \text{None} \implies ((\text{e\_index } g') \in \text{energies} \wedge \text{e\_index}$ 
g'  $\in$  a_win_min g'))"
            using D by meson
          qed
        qed
      qed
    qed
  show "{energy_sup {inv_upd (the (weight g g')) (e_index g') | g'. weight g
g'  $\neq$  None} |
    e_index.  $\forall g'. \text{weight } g \ g' \neq \text{None} \implies ((\text{e\_index } g') \in \text{energies} \wedge \text{e\_index}$ 
g'  $\in$  a_win_min g'))}
     $\subseteq$  {e''.  $\exists \text{strat. } (\forall g'. \text{weight } g \ g' \neq \text{None} \implies \text{strat } g' \in \{\text{the (inverse\_application}$ 
(the (weight g g')) e) | e. minimal_winning_budget e g'})}

```



```

       $\wedge e'' = (\text{energy\_sup } \{\text{strat } g' \mid g'. \text{weight } g \ g' \neq \text{None}\})$ "
    proof
      fix e
      assume "e  $\in$  {energy_sup {inv_upd (the (weight g g')) (e_index g'))  $\mid$  g'.
weight g g'  $\neq$  None}  $\mid$ 
      e_index.  $\forall g'. \text{weight } g \ g' \neq \text{None} \longrightarrow ((\text{e\_index } g') \in \text{energies} \wedge \text{e\_index }
g' \in \text{a\_win\_min } g')}$ "
      from this obtain e_index where I: "e = energy_sup {inv_upd (the (weight
g g')) (e_index g')  $\mid$  g'. weight g g'  $\neq$  None}  $\wedge$  ( $\forall g'. \text{weight } g \ g' \neq \text{None} \longrightarrow \text{e\_index }
g' \in \text{a\_win\_min } g')$ "
      by blast
      define strat where "strat  $\equiv$   $\lambda g'. \text{inv\_upd (the (weight } g \ g')) (e\_index } g')$ "

      show "e  $\in$  {e'',  $\exists$  strat. ( $\forall g'. \text{weight } g \ g' \neq \text{None} \longrightarrow \text{strat } g' \in$  {the (inverse_applicat
(the (weight g g')) e)  $\mid$  e. minimal_winning_budget e g'})}
       $\wedge e'' = (\text{energy\_sup } \{\text{strat } g' \mid g'. \text{weight } g \ g' \neq \text{None}\})$ "
    proof
      show " $\exists$  strat.
      ( $\forall g'. \text{weight } g \ g' \neq \text{None} \longrightarrow$ 
      strat g'  $\in$  {inv_upd (the (weight g g')) e  $\mid$  e. minimal_winning_budget
e g'})  $\wedge$ 
      e = energy_sup {strat g'  $\mid$  g'. weight g g'  $\neq$  None}"
    proof
      show " $(\forall g'. \text{weight } g \ g' \neq \text{None} \longrightarrow$ 
      strat g'  $\in$  {inv_upd (the (weight g g')) e  $\mid$  e. minimal_winning_budget
e g'})  $\wedge$ 
      e = energy_sup {strat g'  $\mid$  g'. weight g g'  $\neq$  None}"
    proof
      show " $\forall g'. \text{weight } g \ g' \neq \text{None} \longrightarrow$ 
      strat g'  $\in$  {inv_upd (the (weight g g')) e  $\mid$  e. minimal_winning_budget e
g'})"
      using I strat_def by blast
      show "e = energy_sup {strat g'  $\mid$  g'. weight g g'  $\neq$  None}" using I strat_def
      by blast
    qed
  qed
  qed
  qed
  qed
  qed

  thus ?thesis using minwin iteration by simp
qed
qed

```

With this we can conclude that iteration maps subsets of winning budgets to subsets of winning budgets.

```

lemma iteration_stays_winning:
  assumes "F  $\in$  possible_pareto" and "F  $\preceq$  a_win_min"
  shows "iteration F  $\preceq$  a_win_min"
proof-
  have "iteration F  $\preceq$  iteration a_win_min"
  using assms iteration_monotonic a_win_min_in_pareto by blast
  thus ?thesis
  using a_win_min_is_fp by simp
qed

```

We now prepare the proof that `a_win_min` is the *least* fixed point of iteration by introducing `S`.

```

inductive S:: "'energy  $\Rightarrow$  'position  $\Rightarrow$  bool" where
  "S e g" if "g  $\notin$  attacker  $\wedge$  ( $\exists$  index. e = (energy_sup
    {inv_upd (the (weight g g')) (index g') | g'. weight g g'  $\neq$  None})
     $\wedge$  ( $\forall$  g'. weight g g'  $\neq$  None  $\longrightarrow$  S (index g') g'))" |
  "S e g" if "g  $\in$  attacker  $\wedge$  ( $\exists$  g'. (weight g g'  $\neq$  None
     $\wedge$  ( $\exists$  e'. S e' g'  $\wedge$  e = inv_upd (the (weight g g')) e')))"

lemma length_S:
  shows " $\bigwedge$  e g. S e g  $\implies$  e  $\in$  energies"
proof-
  fix e g
  assume "S e g"
  thus "e  $\in$  energies"
proof(rule S.induct)
  show " $\bigwedge$  g e. g  $\notin$  attacker  $\wedge$ 
    ( $\exists$  index.
      e =
        energy_sup
        {inv_upd (the (weight g g')) (index g') | g'. weight g g'  $\neq$  None}
     $\wedge$ 
      ( $\forall$  g'. weight g g'  $\neq$  None  $\longrightarrow$  S (index g') g'  $\wedge$  (index g')  $\in$  energies))
   $\implies$ 
    e  $\in$  energies"
proof-
  fix e g
  assume "g  $\notin$  attacker  $\wedge$ 
    ( $\exists$  index.
      e =
        energy_sup
        {inv_upd (the (weight g g')) (index g') | g'. weight g g'  $\neq$  None}
     $\wedge$ 
      ( $\forall$  g'. weight g g'  $\neq$  None  $\longrightarrow$  S (index g') g'  $\wedge$  (index g')  $\in$  energies))"
  from this obtain index where E: "e =
    energy_sup
    {inv_upd (the (weight g g')) (index g') | g'. weight g g'  $\neq$  None}"
  and "( $\forall$  g'. weight g g'  $\neq$  None  $\longrightarrow$  S (index g') g'  $\wedge$  (index g')  $\in$  energies)" by
  auto
  hence in_energy: "{inv_upd (the (weight g g')) (index g') | g'. weight g g'
     $\neq$  None}  $\subseteq$  energies"
  using inv_well_defined by blast
  have "{inv_upd (the (weight g g')) (index g') | g'. weight g g'  $\neq$  None}  $\subseteq$ 
    {inv_upd (the (weight g g')) (index g') | g'. g'  $\in$  positions}" by auto
  hence "finite {inv_upd (the (weight g g')) (index g') | g'. weight g g'  $\neq$  None}"
  using finite_positions rev_finite_subset by fastforce
  thus "e  $\in$  energies" using E in_energy bounded_join_semilattice by meson
qed

show " $\bigwedge$  g e. g  $\in$  attacker  $\wedge$ 
  ( $\exists$  g'. weight g g'  $\neq$  None  $\wedge$ 
    ( $\exists$  e'. (S e' g'  $\wedge$  e'  $\in$  energies)  $\wedge$ 
      e = inv_upd (the (weight g g')) e'))  $\implies$ 
    e  $\in$  energies"
proof-

```

```

fix e g
assume "g ∈ attacker ∧
  (∃ g'. weight g g' ≠ None ∧
    (∃ e'. (S e' g' ∧ e' ∈ energies) ∧
      e = inv_upd (the (weight g g')) e'))"
from this obtain g' e' where "weight g g' ≠ None" and "(S e' g' ∧ e' ∈ energies)
^
  e = inv_upd (the (weight g g')) e'" by auto
thus "e ∈ energies"
  using inv_well_defined by blast
qed
qed
qed

lemma a_win_min_is_minS:
  shows "energy_Min {e. S e g} = a_win_min g"
proof-
  have "{e. ∃ e'. S e' g ∧ e' e ≤ e} = a_win g"
  proof
    show "{e. ∃ e'. S e' g ∧ e' e ≤ e} ⊆ a_win g"
    proof
      fix e
      assume "e ∈ {e. ∃ e'. S e' g ∧ e' e ≤ e}"
      from this obtain e' where "S e' g ∧ e' e ≤ e" by auto
      have "e' ∈ a_win g"
      proof(rule S.induct)
        show "S e' g" using <S e' g ∧ e' e ≤ e> by simp
        show "∧ g e. g ∉ attacker ∧
          (∃ index.
            e =
              energy_sup
                {inv_upd (the (weight g g')) (index g') | g'. weight g g' ≠ None}
          (∀ g'. weight g g' ≠ None → S (index g') g' ∧ index g' ∈ a_win
            g')) ⇒
            e ∈ a_win g"
        proof
          fix e g
          assume A: "g ∉ attacker ∧
            (∃ index.
              e =
                energy_sup
                  {inv_upd (the (weight g g')) (index g') | g'. weight g g' ≠ None}
            (∀ g'. weight g g' ≠ None → S (index g') g' ∧ index g' ∈ a_win
              g'))"
          from this obtain index where E: "e =
            energy_sup
              {inv_upd (the (weight g g')) (index g') | g'. weight g g' ≠ None}
            (∀ g'. weight g g' ≠ None → S (index g') g' ∧ index g' ∈ a_win
              g'))" by auto
          show "winning_budget_len e g"
          proof(rule winning_budget_len.intros(1))
            show "e ∈ energies ∧
              g ∉ attacker ∧

```

```

    (∀g'. weight g g' ≠ None →
      apply_w g g' e ≠ None ∧ winning_budget_len (upd (the (weight g g')) e)
    g')"

  proof
    have "{inv_upd (the (weight g g')) (index g') | g'. weight g g' ≠ None}
    ⊆ {inv_upd (the (weight g g')) (index g') | g'. g' ∈ positions }" by auto
    hence fin: "finite {inv_upd (the (weight g g')) (index g') | g'. weight
    g g' ≠ None}"

    using finite_positions rev_finite_subset by fastforce
    have "{inv_upd (the (weight g g')) (index g') | g'. weight g g' ≠ None}
    ⊆ energies" using E
    using inv_well_defined length_S by blast
    thus "e ∈ energies" using E fin bounded_join_semilattice by meson

    show "g ∉ attacker ∧
    (∀g'. weight g g' ≠ None →
      apply_w g g' e ≠ None ∧ winning_budget_len (upd (the (weight g g')) e)
    g')"

    proof
      show "g ∉ attacker"
      using A by simp
      show "∀g'. weight g g' ≠ None →
      apply_w g g' e ≠ None ∧ winning_budget_len (upd (the (weight g g')) e)
      g'"

      proof
        fix g'
        show "weight g g' ≠ None →
        apply_w g g' e ≠ None ∧ winning_budget_len (upd (the (weight g g')) e)
        g'"

        proof
          assume "weight g g' ≠ None"
          hence "S (index g') g' ∧ index g' ∈ a_win g'" using E
          by simp
          show "apply_w g g' e ≠ None ∧ winning_budget_len (upd (the
          (weight g g')) e) g'"

          proof
            from E have E: "e = energy_sup {inv_upd (the (weight g g'))
            (index g') | g'. weight g g' ≠ None}" by simp

            have "∧s'. energy_sup {inv_upd (the (weight g g')) (index
            g') | g'. weight g g' ≠ None} ∈ energies ∧ (∀s. s ∈ {inv_upd (the (weight g g'))
            (index g') | g'. weight g g' ≠ None} → s e ≤ energy_sup {inv_upd (the (weight g
            g')) (index g') | g'. weight g g' ≠ None}) ∧ (s' ∈ energies ∧ (∀s. s ∈ {inv_upd
            (the (weight g g')) (index g') | g'. weight g g' ≠ None} → s e ≤ s') → energy_sup
            {inv_upd (the (weight g g')) (index g') | g'. weight g g' ≠ None} e ≤ s'))"

            proof(rule bounded_join_semilattice)
              show "∧s'. {inv_upd (the (weight g g')) (index g') | g'.
              weight g g' ≠ None} ⊆ energies"

              proof-
                fix s'
                show "{inv_upd (the (weight g g')) (index g') | g'. weight
                g g' ≠ None} ⊆ energies"

                using <{inv_upd (the (weight g g')) (index g') | g'.
                weight g g' ≠ None} ⊆ energies> by auto
              qed
            qed

```

```

      show "\s'. finite {inv_upd (the (weight g g')) (index g')}
|g'. weight g g' ≠ None}"
      proof-
        fix s'
        have "{inv_upd (the (weight g g')) (index g') |g'. weight
g g' ≠ None} ⊆ {inv_upd (the (weight g g')) (index g') |g'. g' ∈ positions}" by
auto
        thus "finite {inv_upd (the (weight g g')) (index g') |g'.
weight g g' ≠ None}" using finite_positions
        using rev_finite_subset by fastforce
      qed
      qed
      hence "(∀s. s ∈ {inv_upd (the (weight g g')) (index g') |g'.
weight g g' ≠ None} → s e≤ energy_sup {inv_upd (the (weight g g')) (index g')
|g'. weight g g' ≠ None})" by auto

      hence leq: "inv_upd (the (weight g g')) (index g') e≤ e"
      unfolding E
      using <weight g g' ≠ None> by blast

      show "apply_w g g' e ≠ None"
      using <weight g g' ≠ None> proof(rule domain_upw_closed)
        show "apply_w g g' (inv_upd (the (weight g g')) (index g'))
≠ None"
        using inv_well_defined <weight g g' ≠ None> <S (index
g') g' ∧ index g' ∈ a_win g'> winning_budget_len.simps
        by (metis inv_well_defined mem_Collect_eq)
        show "inv_upd (the (weight g g')) (index g') e≤ e" using
leq by simp
      qed

      have A1: "index g' e≤ upd (the (weight g g')) (inv_upd (the
(weight g g')) (index g'))"
      using upd_inv_increasing <S (index g') g' ∧ index g' ∈
a_win g'> winning_budget_len.simps
      using <weight g g' ≠ None> by blast
      have A2: "upd (the (weight g g')) (inv_upd (the (weight g
g')) (index g')) e≤
      upd (the (weight g g')) e" using leq updates_monotonic <weight g g' ≠ None>
      using <S (index g') g' ∧ index g' ∈ a_win g'> inv_well_defined
length_S by blast

      hence "index g' e≤ upd (the (weight g g')) e" using A1 energy_order
ordering_def
      by (metis (mono_tags, lifting) partial_preordering.trans)

      thus "winning_budget_len (upd (the (weight g g')) e) g'"
      using upwards_closure_wb_len <S (index g') g' ∧ index g'
∈ a_win g'> by blast
    qed
  qed
qed
qed
qed
qed

```

qed

```

show "∧g e. g ∈ attacker ∧
  (∃g'. weight g g' ≠ None ∧
    (∃e'. (S e' g' ∧ e' ∈ a_win g') ∧ e = inv_upd (the (weight g g'))
e')) ⇒
  e ∈ a_win g "
proof
  fix e g
  assume A: "g ∈ attacker ∧
    (∃g'. weight g g' ≠ None ∧
      (∃e'. (S e' g' ∧ e' ∈ a_win g') ∧ e = inv_upd (the (weight g g'))
e'))"
    from this obtain g' e' where "weight g g' ≠ None" and "(S e' g' ∧ e'
    ∈ a_win g') ∧ e = inv_upd (the (weight g g')) e'" by auto
    hence "e' e ≤ upd (the (weight g g')) e"
    using updates_monotonic inv_well_defined inv_well_defined
    by (metis length_S upd_inv_increasing)
    show "winning_budget_len e g"
    proof(rule winning_budget_len.intros(2))
      show "e ∈ energies ∧
        g ∈ attacker ∧
        (∃g'. weight g g' ≠ None ∧
          apply_w g g' e ≠ None ∧ winning_budget_len (upd (the (weight g g')) e)
g')"
        proof
          have "e' ∈ energies" using <(S e' g' ∧ e' ∈ a_win g') ∧ e = inv_upd
          (the (weight g g')) e'> winning_budget_len.simps
          by blast
          show "e ∈ energies"
          using <(S e' g' ∧ e' ∈ a_win g') ∧ e = inv_upd (the (weight g g'))
e'> <e' ∈ energies> <weight g g' ≠ None>
          using inv_well_defined by blast
          show "g ∈ attacker ∧
            (∃g'. weight g g' ≠ None ∧
              apply_w g g' e ≠ None ∧ winning_budget_len (upd (the (weight g g')) e)
g')"
            proof
              show "g ∈ attacker" using A by simp
              show "∃g'. weight g g' ≠ None ∧
                apply_w g g' e ≠ None ∧ winning_budget_len (upd (the (weight g g')) e)
g' "
                proof
                  show "weight g g' ≠ None ∧
                    apply_w g g' e ≠ None ∧ winning_budget_len (upd (the (weight g g')) e)
g'"
                    proof
                      show "weight g g' ≠ None"
                      using <weight g g' ≠ None> .
                      show "apply_w g g' e ≠ None ∧ winning_budget_len (upd (the
                      (weight g g')) e) g'"
                      proof
                        show "apply_w g g' e ≠ None"
                        using <weight g g' ≠ None> <(S e' g' ∧ e' ∈ a_win g')
                        ∧ e = inv_upd (the (weight g g')) e'>

```

```

      <e' e ≤ upd (the (weight g g')) e> updates_monotonic inv_well_defined
inv_well_defined
  by (metis mem_Collect_eq winning_budget_len.cases)
  show "winning_budget_len (upd (the (weight g g')) e) g'"
  using <e' e ≤ upd (the (weight g g')) e> upwards_closure_wb_len
<(S e' g' ∧ e' ∈ a_win g') ∧ e = inv_upd (the (weight g g')) e'> by blast
  qed
  qed
  qed
  qed
  qed
  qed
  qed
  thus "e ∈ a_win g" using <S e' g ∧ e' e ≤ e> upwards_closure_wb_len
  by blast
  qed
next
  show "a_win g ⊆ {e. ∃e'. S e' g ∧ e' e ≤ e}"
  proof

    define P where "P ≡ λ(g,e). (e ∈ {e. ∃e'. S e' g ∧ e' e ≤ e})"

    fix e
    assume "e ∈ a_win g"
    from this obtain s where S: "attacker_winning_strategy s e g"
    using nonpos_eq_pos
    by (metis winning_budgnet_len_is_wb mem_Collect_eq winning_budget.elims(2))

    have "reachable_positions_len s g e ⊆ reachable_positions s g e" by auto
    hence "wfp_on (strategy_order s) (reachable_positions_len s g e)"
    using strategy_order_well_founded S
    using Restricted_Predicates.wfp_on_subset by blast
    hence "inductive_on (strategy_order s) (reachable_positions_len s g e)"
    by (simp add: wfp_on_iff_inductive_on)

    hence "P (g,e)"
    proof(rule inductive_on_induct)
      show "(g,e) ∈ reachable_positions_len s g e"
      unfolding reachable_positions_def proof-
      have "lfinite LNil ∧
        llast (LCons g LNil) = g ∧
        valid_play (LCons g LNil) ∧ play_consistent_attacker s (LCons g LNil)
e ∧
        Some e = energy_level e (LCons g LNil) (the_enat (llength LNil))"
      using valid_play.simps play_consistent_attacker.simps energy_level.simps
      by (metis lfinite_code(1) llast_singleton llength_LNil neq_LNil_conv
the_enat_0)
      thus "(g, e)
∈ {(g', e').
  (g', e')
∈ {(g', e') | g' e'.
  ∃p. lfinite p ∧
    llast (LCons g p) = g' ∧
    valid_play (LCons g p) ∧
    play_consistent_attacker s (LCons g p) e ∧ Some e' = energy_level

```

```

e (LCons g p) (the_enat (llength p)))} ∧
  e' ∈ energies}"
  using <e ∈ a_win g> nonpos_eq_pos winning_bugget_len_is_wb
  by auto
qed

show "∧y. y ∈ reachable_positions_len s g e ⇒
  (∧x. x ∈ reachable_positions_len s g e ⇒ strategy_order s x y ⇒
P x) ⇒ P y"
proof-
  fix y
  assume "y ∈ reachable_positions_len s g e"
  hence "∃e' g'. y = (g', e'" using reachable_positions_def by auto
  from this obtain e' g' where "y = (g', e'" by auto

  hence y_len: "(∃p. lfinite p ∧ llast (LCons g p) = g'
    ∧ valid_play (LCons g p)
    ∧ play_consistent_attacker s
(LCons g p) e
    ∧ (Some e' = energy_level e
(LCons g p) (the_enat (llength p))))
  using <y ∈ reachable_positions_len s g e> unfolding reachable_positions_def
  by auto
  from this obtain p where P: "(lfinite p ∧ llast (LCons g p) = g'
    ∧ valid_play (LCons g p)
    ∧ play_consistent_attacker s
(LCons g p) e)
    ∧ (Some e' = energy_level e
(LCons g p) (the_enat (llength p))))" by auto

  show "(∧x. x ∈ reachable_positions_len s g e ⇒ strategy_order s x y
⇒ P x) ⇒ P y"
  proof-
    assume ind: "(∧x. x ∈ reachable_positions_len s g e ⇒ strategy_order
s x y ⇒ P x)"
    thus "P y"
    proof(cases "g' ∈ attacker")
      case True
      then show ?thesis
      proof(cases "deadend g'")
        case True
        hence "attacker_stuck (LCons g p)" using <g' ∈ attacker> P
          by (meson defender_wins_play_def attacker_winning_strategy.elims(2))

        hence "defender_wins_play e (LCons g p)" using defender_wins_play_def
by simp

        have "¬defender_wins_play e (LCons g p)" using P S by simp
        then show ?thesis using <defender_wins_play e (LCons g p)> by simp
      next
        case False
        hence "(s e' g') ≠ None ∧ (weight g' (the (s e' g'))) ≠ None" using
S attacker_winning_strategy.simps
          by (simp add: True attacker_strategy_def)

        define x where "x = (the (s e' g'), the (apply_w g' (the (s e' g'))))

```



```

e'))"

    define p' where "p' = (lappend p (LCons (the (s e' g')) LNil))"
    hence "lfinite p'" using P by simp
    have "llast (LCons g p') = the (s e' g'" using p'_def <lfinite
p'>

        by (simp add: llast_LCons)

    have "the_enat (llength p') > 0" using P
    by (metis LNil_eq_lappend_iff <lfinite p'> bot_nat_0.not_eq_extremum
    enat_0_iff(2) lfinite_conv_llength_enat llength_eq_0 llist.collapse(1) llist.distinct(1)
    p'_def the_enat.simps)
    hence "∃ i. Suc i = the_enat (llength p'"
    using less_iff_Suc_add by auto
    from this obtain i where "Suc i = the_enat (llength p'" by auto
    hence "i = the_enat (llength p)" using p'_def P
    by (metis Suc_leI <lfinite p'> length_append_singleton length_list_of_conv_t
    less_Suc_eq_le less_irrefl_nat lfinite_LConsI lfinite_LNil list_of_LCons list_of_LNil
    list_of_lappend not_less_less_Suc_eq)
    hence "Some e' = (energy_level e (LCons g p) i)" using P by simp

    have A: "lfinite (LCons g p) ∧ i < the_enat (llength (LCons g p))
    ∧ energy_level e (LCons g p) (the_enat (llength (LCons g p)) - 1) ≠ None"
    proof
    show "lfinite (LCons g p)" using P by simp
    show "i < the_enat (llength (LCons g p)) ∧ energy_level e (LCons
    g p) (the_enat (llength (LCons g p)) - 1) ≠ None"
    proof
    show "i < the_enat (llength (LCons g p))" using <i = the_enat
    (llength p)> P
    by (metis <lfinite (LCons g p)> length_Cons length_list_of_conv_the_enat
    lessI list_of_LCons)
    show "energy_level e (LCons g p) (the_enat (llength (LCons g
    p)) - 1) ≠ None" using P <i = the_enat (llength p)>
    using S_defender_wins_play_def by auto
    qed
    qed

    hence "Some e' = (energy_level e (LCons g p') i)" using p'_def energy_level_app
P <Some e' = (energy_level e (LCons g p) i)>
    by (metis lappend_code(2))
    hence "energy_level e (LCons g p') i ≠ None"
    by (metis option.distinct(1))

    have "enat (Suc i) = llength p'" using <Suc i = the_enat (llength
p')>

    by (metis <lfinite p'> lfinite_conv_llength_enat the_enat.simps)
    also have "... < eSuc (llength p'"
    by (metis calculation illess_Suc_eq order_refl)
    also have "... = llength (LCons g p'" using <lfinite p'> by simp
    finally have "enat (Suc i) < llength (LCons g p')".

    have "(lnth (LCons g p) i) = g'" using <i = the_enat (llength p)>
P

    by (metis lfinite_conv_llength_enat llast_conv_lnth llength_LCons
    the_enat.simps)
    hence "(lnth (LCons g p') i) = g'" using p'_def

```

```

    by (metis P <i = the_enat (llength p)> enat_ord_simps(2) energy_level.elims
lessI lfinite_llength_enat lnth_0 lnth_Suc_LCons lnth_lappend1 the_enat.simps)

    have "energy_level e (LCons g p') (the_enat (llength p')) = energy_level
e (LCons g p') (Suc i)"
    using <Suc i = the_enat (llength p')> by simp
    also have "... = apply_w (lnth (LCons g p') i) (lnth (LCons g p'))
(Suc i)) (the (energy_level e (LCons g p') i))"
    using energy_level.simps <enat (Suc i) < llength (LCons g p')>
<energy_level e (LCons g p') i ≠ None>
    by (meson leD)
    also have "... = apply_w (lnth (LCons g p') i) (lnth (LCons g p'))
(Suc i)) e'" using <Some e' = (energy_level e (LCons g p') i)>
    by (metis option.sel)
    also have "... = apply_w (lnth (LCons g p') i) (the (s e' g'))
e'" using p'_def <enat (Suc i) = llength p'>
    by (metis <eSuc (llength p') = llength (LCons g p')> <llast (LCons
g p') = the (s e' g')> llast_conv_lnth)
    also have "... = apply_w g' (the (s e' g')) e'" using <(lnth (LCons
g p') i) = g'> by simp
    finally have "energy_level e (LCons g p') (the_enat (llength p'))
= apply_w g' (the (s e' g')) e'" .

    have P': "lfinite p' ∧
llast (LCons g p') = (the (s e' g')) ∧
valid_play (LCons g p') ∧ play_consistent_attacker s (LCons g p') e
^
Some (the (apply_w g' (the (s e' g')) e')) = energy_level e (LCons g
p') (the_enat (llength p'))"
    proof
      show "lfinite p'" using p'_def P by simp
      show "llast (LCons g p') = the (s e' g') ∧
valid_play (LCons g p') ∧
play_consistent_attacker s (LCons g p') e ∧
Some (the (apply_w g' (the (s e' g')) e')) = energy_level e (LCons g p') (the_enat
(llength p'))"
    proof
      show "llast (LCons g p') = the (s e' g')" using p'_def <lfinite
p'>
      by (simp add: llast_LCons)
      show "valid_play (LCons g p') ∧
play_consistent_attacker s (LCons g p') e ∧
Some (the (apply_w g' (the (s e' g')) e')) = energy_level e (LCons g p') (the_enat
(llength p'))"
    proof
      show "valid_play (LCons g p')" using p'_def P
      using <s e' g' ≠ None ∧ weight g' (the (s e' g')) ≠ None>
valid_play.intros(2) valid_play_append by auto
      show "play_consistent_attacker s (LCons g p') e ∧
Some (the (apply_w g' (the (s e' g')) e')) = energy_level e (LCons g p') (the_enat
(llength p'))"
    proof
      have "(LCons g p') = lappend (LCons g p) (LCons (the (s
e' g')) LNil)" using p'_def
      by simp
      have "play_consistent_attacker s (lappend (LCons g p) (LCons

```

```

(the (s e' g')) LNil)) e"
  proof (rule play_consistent_attacker_append_one)
    show "play_consistent_attacker s (LCons g p) e"
      using P by auto
    show "lfinite (LCons g p)" using P by auto
    show "energy_level e (LCons g p) (the_enat (llength (LCons
g p)) - 1) ≠ None" using P
      using A by auto
    show "valid_play (lappend (LCons g p) (LCons (the (s e'
g')) LNil)))"
      using <valid_play (LCons g p')> <(LCons g p') = lappend
(LCons g p) (LCons (the (s e' g')) LNil)> by simp
    show "llast (LCons g p) ∈ attacker →
      Some (the (s e' g')) =
      s (the (energy_level e (LCons g p) (the_enat (llength (LCons g p)) - 1))) (llast
(LCons g p))"
      proof
        assume "llast (LCons g p) ∈ attacker"
        show "Some (the (s e' g')) =
          s (the (energy_level e (LCons g p) (the_enat (llength (LCons g p)) - 1))) (llast
(LCons g p))"
          using <llast (LCons g p) ∈ attacker> P
          by (metis One_nat_def <s e' g' ≠ None ∧ weight g'
(the (s e' g')) ≠ None> diff_Suc_1' eSuc_enat lfinite_llength_enat llength_LCons
option.collapse option.sel the_enat.simps)
        qed
      qed
    thus "play_consistent_attacker s (LCons g p') e" using <(LCons
g p') = lappend (LCons g p) (LCons (the (s e' g')) LNil)> by simp

    show "Some (the (apply_w g' (the (s e' g')) e')) = energy_level
e (LCons g p') (the_enat (llength p'))"
      by (metis <eSuc (llength p') = llength (LCons g p')> <enat
(Suc i) = llength p'> <energy_level e (LCons g p') (the_enat (llength p')) = apply_w
g' (the (s e' g')) e'> <play_consistent_attacker s (LCons g p') e> <valid_play
(LCons g p')> S defender_wins_play_def diff_Suc_1 eSuc_enat option.collapse attacker_winning_st
the_enat.simps)
    qed
  qed
  qed
  qed

  have x_len: "(upd (the (weight g' (the (s e' g')))) e') ∈ energies"
using y_len
  by (metis P' <energy_level e (LCons g p') (the_enat (llength p'))
= apply_w g' (the (s e' g')) e'> <s e' g' ≠ None ∧ weight g' (the (s e' g')) ≠
None> option.distinct(1) upd_well_defined)
  hence "x ∈ reachable_positions_len s g e" using P' reachable_positions_def
x_def by auto

  have "(apply_w g' (the (s e' g')) e') ≠ None" using P'
  by (metis <energy_level e (LCons g p') (the_enat (llength p'))
= apply_w g' (the (s e' g')) e'> option.distinct(1))

  have "Some (the (apply_w g' (the (s e' g')) e')) = apply_w g' (the
(s e' g')) e' ∧ (if g' ∈ attacker then Some (the (s e' g')) = s e' g' else weight

```

```

g' (the (s e' g')) ≠ None)"
  using <(s e' g') ≠ None ∧ (weight g' (the (s e' g'))))≠None> <(apply_w
g' (the (s e' g')) e') ≠ None> by simp
  hence "strategy_order s x y" unfolding strategy_order_def using
x_def <y = (g', e')>
  by blast
  hence "P x" using ind <x ∈ reachable_positions_len s g e> by simp

  hence "∃e''. S e'' (the (s e' g')) ∧ e'' e ≤ (upd (the (weight
g' (the (s e' g')))) e'" unfolding P_def x_def by simp
  from this obtain e'' where E: "S e'' (the (s e' g')) ∧ e'' e ≤ (upd
(the (weight g' (the (s e' g')))) e'" by auto
  hence "S (inv_upd (the (weight g' (the (s e' g')))) e'') g'" using
True S.intros(2)
  using <s e' g' ≠ None ∧ weight g' (the (s e' g')) ≠ None> by
blast

  have "(inv_upd (the (weight g' (the (s e' g')))) e'') e ≤ inv_upd
(the (weight g' (the (s e' g')))) (upd (the (weight g' (the (s e' g')))) e'"
  using E inverse_monotonic <s e' g' ≠ None ∧ weight g' (the (s
e' g')) ≠ None>
  using x_len
  using inv_well_defined length_S by blast
  hence "(inv_upd (the (weight g' (the (s e' g')))) e'') e ≤ e'" using
inv_upd_decreasing <s e' g' ≠ None ∧ weight g' (the (s e' g')) ≠ None>
  using <apply_w g' (the (s e' g')) e' ≠ None> energy_order ordering_def

  by (metis (mono_tags, lifting) E <apply_w g' (the (s e' g')) e'
≠ None> <y = (g', e')> <y ∈ reachable_positions_len s g e> case_prodD galois_energy_game.gal
galois_energy_game_decidable.length_S galois_energy_game_decidable_axioms galois_energy_game_ax
mem_Collect_eq)
  thus "P y" unfolding P_def <y = (g', e')>
  using <S (inv_upd (the (weight g' (the (s e' g')))) e'') g'> by
blast

qed
next
case False
  hence P: "g' ∉ attacker ∧
(∀g''. weight g' g'' ≠ None →
apply_w g' g'' e' ≠ None ∧ P (g'', (the (apply_w g' g'' e'))))"
  proof
    show "∀g''. weight g' g'' ≠ None →
apply_w g' g'' e' ≠ None ∧ P (g'', (the (apply_w g' g'' e')))"
    proof
      fix g''
      show "weight g' g'' ≠ None →
apply_w g' g'' e' ≠ None ∧ P (g'', (the (apply_w g' g'' e')))"
      proof
        assume "weight g' g'' ≠ None"
        show "apply_w g' g'' e' ≠ None ∧ P (g'', (the (apply_w g'
g'' e')))"
        proof
          show "apply_w g' g'' e' ≠ None"
          proof
            assume "apply_w g' g'' e' = None"

```

```

define p' where "p' ≡ (LCons g (lappend p (LCons g'' LNil)))"
hence "lfinite p'" using P by simp
have "∃ i. llength p = enat i" using P
  by (simp add: lfinite_llength_enat)
from this obtain i where "llength p = enat i" by auto
hence "llength (lappend p (LCons g'' LNil)) = enat (Suc
i)"

  by (simp add: <llength p = enat i> eSuc_enat iadd_Suc_right)
hence "llength p' = eSuc (enat (Suc i))" using p'_def
  by simp
hence "the_enat (llength p') = Suc (Suc i)"
  by (simp add: eSuc_enat)
hence "the_enat (llength p') - 1 = Suc i"
  by simp
hence "the_enat (llength p') - 1 = the_enat (llength (lappend
p (LCons g'' LNil)))"
  using <llength (lappend p (LCons g'' LNil)) = enat (Suc
i)>
  by simp

have "(lnth p' i) = g'" using p'_def <llength p = enat i>
P
  by (smt (verit) One_nat_def diff_Suc_1' enat_ord_simps(2)
energy_level.elims lessI llast_conv_lnth llength_LCons lnth_0 lnth_LCons' lnth_lappend
the_enat.simps)
have "(lnth p' (Suc i)) = g''" using p'_def <llength p =
enat i>
  by (metis <llength p' = eSuc (enat (Suc i))> lappend.disc(2)
llast_LCons llast_conv_lnth llast_lappend_LCons llength_eq_enat_lfiniteD llist.disc(1)
llist.disc(2))
have "p' = lappend (LCons g p) (LCons g'' LNil)" using p'_def
by simp
hence "the (energy_level e p' i) = the (energy_level e (lappend
(LCons g p) (LCons g'' LNil)) i)" by simp
also have "... = the (energy_level e (LCons g p) i)" using
<llength p = enat i> energy_level_append P
  by (metis diff_Suc_1 eSuc_enat lessI lfinite_LConsI llength_LCons
option.distinct(1) the_enat.simps)
also have "... = e'" using P
  by (metis <llength p = enat i> option.sel the_enat.simps)

finally have "the (energy_level e p' i) = e'" .
hence "apply_w (lnth p' i) (lnth p' (Suc i)) (the (energy_level
e p' i)) = None" using <apply_w g' g'' e'=None> <(lnth p' i) = g'> <(lnth p' (Suc
i)) = g''> by simp

have "energy_level e p' (the_enat (llength p') - 1) =
  energy_level e p' (the_enat (llength (lappend p (LCons
g'' LNil))))"
  using <the_enat (llength p') - 1 = the_enat (llength (lappend
p (LCons g'' LNil)))>
  by simp
also have "... = energy_level e p' (Suc i)" using <llength
(lappend p (LCons g'' LNil)) = enat (Suc i)> by simp
also have "... = (if energy_level e p' i = None ∨ llength
p' ≤ enat (Suc i) then None

```

```

else apply_w (lnth p' i) (lnth p' (Suc i))
(the (energy_level e p' i)))" using energy_level.simps by simp
  also have "... = None" using <apply_w (lnth p' i) (lnth
p' (Suc i)) (the (energy_level e p' i)) = None>
    by simp
  finally have "energy_level e p' (the_enat (llength p')) -
1) = None" .
  hence "defender_wins_play e p'" unfolding defender_wins_play_def
by simp

  have "valid_play p'"
    by (metis P <p' = lappend (LCons g p) (LCons g'' LNil)>
<weight g' g'' ≠ None> energy_game.valid_play.intros(2) energy_game.valid_play_append
lfinite_LConsI)

  have "play_consistent_attacker s (lappend (LCons g p) (LCons
g'' LNil)) e"
    proof(rule play_consistent_attacker_append_one)
      show "play_consistent_attacker s (LCons g p) e"
        using P by simp
      show "lfinite (LCons g p)" using P by simp
      show "energy_level e (LCons g p) (the_enat (llength (LCons
g p)) - 1) ≠ None"
        using P
        by (meson S defender_wins_play_def attacker_winning_strategy.elims(
show "valid_play (lappend (LCons g p) (LCons g'' LNil))"
        using <valid_play p'> <p' = lappend (LCons g p) (LCons
g'' LNil)> by simp
        show "llast (LCons g p) ∈ attacker →
Some g'' =
s (the (energy_level e (LCons g p) (the_enat (llength (LCons g p)) - 1))) (llast
(LCons g p))"
        using False P by simp
      qed
      hence "play_consistent_attacker s p' e"
        using <p' = lappend (LCons g p) (LCons g'' LNil)> by
simp
      hence "¬defender_wins_play e p'" using <valid_play p'>
p'_def S by simp
      thus "False" using <defender_wins_play e p'> by simp
    qed

  define x where "x = (g'', the (apply_w g' g'' e'))"
  have "P x"
  proof(rule ind)
    have X: "(∃p. lfinite p ∧
llast (LCons g p) = g'' ∧
valid_play (LCons g p) ∧ play_consistent_attacker s (LCons g p) e ∧
Some (the (apply_w g' g'' e')) = energy_level e (LCons g p) (the_enat
(llength p)))"
    proof
      define p' where "p' = lappend p (LCons g'' LNil)"
      show "lfinite p' ∧
llast (LCons g p') = g'' ∧
valid_play (LCons g p') ∧ play_consistent_attacker s (LCons g p') e ∧

```

```

    Some (the (apply_w g' g'' e')) = energy_level e (LCons g p') (the_enat (llength
p')))"

    proof
      show "lfinite p'" using P p'_def by simp
      show "llast (LCons g p') = g'" ∧
    valid_play (LCons g p') ∧
    play_consistent_attacker s (LCons g p') e ∧
    Some (the (apply_w g' g'' e')) = energy_level e (LCons g p') (the_enat (llength
p')))"

    proof
      show "llast (LCons g p') = g'" using p'_def
      by (metis <lfinite p'> lappend.disc_iff(2) lfinite_lappend
llast_LCons llast_lappend_LCons llast_singleton llist.discI(2))
      show "valid_play (LCons g p') ∧
    play_consistent_attacker s (LCons g p') e ∧
    Some (the (apply_w g' g'' e')) = energy_level e (LCons g p') (the_enat (llength
p')))"

    proof
      show "valid_play (LCons g p')" using p'_def P
      using <weight g' g'' ≠ None> lfinite_LCons valid_play.intros

valid_play_append by auto

      show "play_consistent_attacker s (LCons g p') e
    ∧
    Some (the (apply_w g' g'' e')) = energy_level e (LCons g p') (the_enat (llength
p'))"

    proof

      have "play_consistent_attacker s (lappend (LCons
g p) (LCons g'' LNil)) e"

      proof(rule play_consistent_attacker_append_one)
        show "play_consistent_attacker s (LCons g p)

e"

        using P by simp
        show "lfinite (LCons g p)" using P by simp
        show "energy_level e (LCons g p) (the_enat (llength
(LCons g p)) - 1) ≠ None"

        using P
        by (meson S defender_wins_play_def attacker_winning_strat
show "valid_play (lappend (LCons g p) (LCons
g'' LNil))"

        using <valid_play (LCons g p')> p'_def by

simp

        show "llast (LCons g p) ∈ attacker →
          Some g'' =
          s (the (energy_level e (LCons g p) (the_enat
(llength (LCons g p)) - 1))) (llast (LCons g p))"
          using False P by simp
        qed
        thus "play_consistent_attacker s (LCons g p')

e" using p'_def

        by (simp add: lappend_code(2))

        have "∃ i. Suc i = the_enat (llength p'" using

p'_def <lfinite p'>
        by (metis P length_append_singleton length_list_of_conv_the
lfinite_LConsI lfinite_LNil list_of_LCons list_of_LNil list_of_lappend)

```

```

from this obtain i where "Suc i = the_enat (llength
p')" by auto

hence "i = the_enat (llength p)" using p'_def
by (smt (verit) One_nat_def <lfinite p'> add.commute
add_Suc_shift add_right_cancel length_append length_list_of_conv_the_enat lfinite_LNil
lfinite_lappend list.size(3) list.size(4) list_of_LCons list_of_LNil list_of_lappend
plus_1_eq_Suc)

hence "Suc i = llength (LCons g p)"
using P eSuc_enat lfinite_llength_enat by fastforce
have "(LCons g p') = lappend (LCons g p) (LCons
g'' LNil)" using p'_def by simp

have A: "lfinite (LCons g p) ∧ i < the_enat (llength
(LCons g p)) ∧ energy_level e (LCons g p) (the_enat (llength (LCons g p)) - 1)
≠ None"

proof
show "lfinite (LCons g p)" using P by simp
show " i < the_enat (llength (LCons g p)) ∧
energy_level e (LCons g p) (the_enat (llength (LCons g p)) - 1) ≠ None "
proof
have "(llength p') = llength (LCons g p)"
using p'_def
by (metis P <lfinite p'> length_Cons length_append_singl
length_list_of lfinite_LConsI lfinite_LNil list_of_LCons list_of_LNil list_of_lappend)

thus "i < the_enat (llength (LCons g p))"
using <Suc i = the_enat (llength p')>
using lessI by force
show "energy_level e (LCons g p) (the_enat
(llength (LCons g p)) - 1) ≠ None" using P
by (meson S energy_game.defender_wins_play_def
energy_game.play_consistent_attacker.intros(2) attacker_winning_strategy.simps)
qed
qed
hence "energy_level e (LCons g p') i ≠ None"
using energy_level_append
by (smt (verit) Nat.lessE Suc_leI <LCons g p'
= lappend (LCons g p) (LCons g'' LNil)> diff_Suc_1 energy_level_nth)
have "enat (Suc i) < llength (LCons g p')"
using <Suc i = the_enat (llength p')>
by (metis Suc_ile_eq <lfinite p'> ldropsn_Suc_LCons
leI lfinite_conv_llength_enat lnull_ldropsn nless_le the_enat.simps)
hence el_premis: "energy_level e (LCons g p')
i ≠ None ∧ llength (LCons g p') > enat (Suc i)" using <energy_level e (LCons g
p') i ≠ None> by simp

have "(lnth (LCons g p') i) = lnth (LCons g p)
i"
unfolding <(LCons g p') = lappend (LCons g p)
(LCons g'' LNil)> using <i = the_enat (llength p)> lnth_lappend1
by (metis A enat_ord_simps(2) length_list_of
length_list_of_conv_the_enat)
have "lnth (LCons g p) i = llast (LCons g p)"
using <Suc i = llength (LCons g p)>
by (metis enat_ord_simps(2) lappend_LNil2 ldropsn_LNil
ldropsn_Suc_conv_ldropsn ldropsn_lappend lessI less_not_refl llast_ldropsn llast_singleton)
hence "(lnth (LCons g p') i) = g'" using P

```



```

      by (simp add: <lnth (LCons g p') i = lnth (LCons
g p) i>)

      have "(lnth (LCons g p') (Suc i)) = g'"
      using p'_def <Suc i = the_enat (llength p')>
      by (smt (verit) <enat (Suc i) < llength (LCons
g p')> <lfinite p'> <llast (LCons g p') = g'>> lappend_snocL1_conv_LCons2 ldropn_LNil
ldropn_Suc_LCons ldropn_Suc_conv_ldropn ldropn_lappend2 lfinite_llength_enat llast_ldropn
llast_singleton the_enat.simps wlog_linorder_le)

      have "energy_level e (LCons g p) i = energy_level
e (LCons g p') i"
      using energy_level_append A <(LCons g p') =
lappend (LCons g p) (LCons g'' LNil)>
      by presburger
      hence "Some e' = (energy_level e (LCons g p')
i)"
      using P <i = the_enat (llength p)>
      by argo

      have "energy_level e (LCons g p') (the_enat (llength
p')) = energy_level e (LCons g p') (Suc i)" using <Suc i = the_enat (llength p')>
      by simp
      also have "... = apply_w (lnth (LCons g p') i)
(lnth (LCons g p') (Suc i)) (the (energy_level e (LCons g p') i))"
      using energy_level.simps el_prem
      by (meson leD)
      also have "... = apply_w g' g'' (the (energy_level
e (LCons g p') i))"
      using <(lnth (LCons g p') i) = g'> <(lnth (LCons
g p') (Suc i)) = g''> by simp
      finally have "energy_level e (LCons g p') (the_enat
(llength p')) = (apply_w g' g'' e)"
      using <Some e' = (energy_level e (LCons g p')
i)>
      by (metis option.sel)
      thus "Some (the (apply_w g' g'' e')) = energy_level
e (LCons g p') (the_enat (llength p'))"
      using <apply_w g' g'' e' ≠ None> by auto
      qed
      qed
      qed
      qed
      qed

      have x_len: "(upd (the (weight g' g'')) e') ∈ energies"
using y_len
      using <apply_w g' g'' e' ≠ None> <weight g' g'' ≠ None>
upd_well_defined by blast

      thus "x ∈ reachable_positions_len s g e"
      using X x_def reachable_positions_def
      by (simp add: mem_Collect_eq)

      have "Some (the (apply_w g' g'' e')) = apply_w g' g'' e'
^
      (if g' ∈ attacker then Some g'' = s e' g' else weight g' g'' ≠ None)"

```

```

proof
  show "Some (the (apply_w g' g'' e')) = apply_w g' g''
e'"
  using <apply_w g' g'' e' ≠ None> by auto
  show "(if g' ∈ attacker then Some g'' = s e' g' else weight
g' g'' ≠ None)"
  using False
  by (simp add: <weight g' g'' ≠ None>)
qed
thus "strategy_order s x y" using strategy_order_def x_def
<y = (g', e')>
  by simp
qed

  thus "P (g'', (the (apply_w g' g'' e')))" using x_def by simp
qed
qed
qed
qed

  hence "∧g''. weight g' g'' ≠ None ⇒ ∃e0. S e0 g'' ∧ e0 e ≤ (the
(apply_w g' g'' e'))" using P_def
  by blast
  define index where "index = (λg''. SOME e0. S e0 g'' ∧ e0 e ≤ (the
(apply_w g' g'' e')))"
  hence I: "∧g''. weight g' g'' ≠ None ⇒ S (index g'') g'' ∧ (index
g'') e ≤ (the (apply_w g' g'' e'))"
  using <∧g''. weight g' g'' ≠ None ⇒ ∃e0. S e0 g'' ∧ e0 e ≤ (the
(apply_w g' g'' e'))> some_eq_ex
  by (smt (verit, del_insts))
  hence "∧g''. weight g' g'' ≠ None ⇒ inv_upd (the (weight g' g''))
(index g'') e ≤ inv_upd (the (weight g' g'')) (the (apply_w g' g'' e'))"
  using inverse_monotonic P
  by (meson inv_well_defined length_S)
  hence "∧g''. weight g' g'' ≠ None ⇒ inv_upd (the (weight g' g''))
(index g'') e ≤ e'"
  using inv_upd_decreasing P
  by (meson I galois length_S y_len)
  hence all: "∀s. s ∈ {inv_upd (the (weight g' g'')) (index g'') | g''.
weight g' g'' ≠ None} → s e ≤ e'"
  by auto

  have "∧s'. energy_sup {inv_upd (the (weight g' g'')) (index g'') |
g''. weight g' g'' ≠ None} ∈ energies ∧ (∀s. s ∈ {inv_upd (the (weight g' g''))
(index g'') | g''. weight g' g'' ≠ None} → s e ≤ energy_sup {inv_upd (the (weight
g' g'')) (index g'') | g''. weight g' g'' ≠ None}) ∧ (s' ∈ energies ∧ (∀s. s ∈
{inv_upd (the (weight g' g'')) (index g'') | g''. weight g' g'' ≠ None} → s e ≤
s') → energy_sup {inv_upd (the (weight g' g'')) (index g'') | g''. weight g' g''
≠ None} e ≤ s'))"
  proof(rule bounded_join_semilattice)
    show "∧s'. {inv_upd (the (weight g' g'')) (index g'') | g''. weight
g' g'' ≠ None} ⊆ energies"
    proof-
      fix s'
      show "{inv_upd (the (weight g' g'')) (index g'') | g''. weight
g' g'' ≠ None} ⊆ energies"

```

```

        using I inv_well_defined length_S by blast
      qed
      show "\s'. finite {inv_upd (the (weight g' g'')) (index g'') | g''.
weight g' g'' ≠ None}"
    proof-
      fix s'
      have "{inv_upd (the (weight g g')) (index g') | g'. weight g g'
≠ None} ⊆ {inv_upd (the (weight g g')) (index g') | g'. g' ∈ positions}" by auto
      thus "finite {inv_upd (the (weight g' g'')) (index g'') | g''.
weight g' g'' ≠ None}" using finite_positions
        using rev_finite_subset by fastforce
      qed
    qed

    hence leq: "energy_sup {inv_upd (the (weight g' g'')) (index g'') |
g''. weight g' g'' ≠ None} e ≤ e'"
      using all
      using y_len by blast
      have "S (energy_sup {inv_upd (the (weight g' g'')) (index g'') | g''.
weight g' g'' ≠ None}) g'"
        using False S.intros(1) I
        by blast
      thus "P y" using leq P_def
        using <y = (g', e')> by blast
    qed
  qed
  qed
  qed
  thus "e ∈ {e. ∃e'. S e' g ∧ e' e ≤ e}" using P_def by simp
  qed
  qed
  hence "energy_Min {e. ∃e'. S e' g ∧ e' e ≤ e} = a_win_min g" by simp

  have "energy_Min {e. ∃e'. S e' g ∧ e' e ≤ e} = energy_Min {e. S e g}"
  proof
    have "{e. S e g} ⊆ {e. ∃e'. S e' g ∧ e' e ≤ e}"
      using energy_order ordering.eq_iff by fastforce

    show "energy_Min {e. ∃e'. S e' g ∧ e' e ≤ e} ⊆ energy_Min {e. S e g}"
    proof
      fix x
      assume "x ∈ energy_Min {e. ∃e'. S e' g ∧ e' e ≤ e}"
      hence "∃e'. S e' g ∧ e' e ≤ x"
        using energy_Min_def by auto
      from this obtain e' where "S e' g ∧ e' e ≤ x" by auto
      hence "S e' g ∧ e' e ≤ e'" using energy_order ordering_def
        using ordering.eq_iff by fastforce
      hence "e' ∈ {e. ∃e'. S e' g ∧ e' e ≤ e} ∧ e' e ≤ x"
        using <S e' g ∧ e' e ≤ x> by auto
      hence "x = e'" using energy_Min_def
        using <x ∈ energy_Min {e. ∃e'. S e' g ∧ e' e ≤ e}> by auto
      hence "S x g"
        by (simp add: <S e' g ∧ e' e ≤ x>)
      show "x ∈ energy_Min {e. S e g}"
    proof(rule ccontr)

```

```

    assume "x ∉ energy_Min {e. S e g}"
    hence "∃x'. x' e< x ∧ x' ∈ {e. S e g}"
      using <S x g> energy_Min_def
      by auto
    from this obtain x' where "x' e< x" and "S x' g"
      by auto
    hence "S x' g ∧ x' e≤ x'" using energy_order ordering_def
      using ordering.eq_iff by fastforce
    hence "x' ∈ {e. ∃e'. S e' g ∧ e' e≤ e}" by auto
    thus "False"
      using <x ∈ energy_Min {e. ∃e'. S e' g ∧ e' e≤ e}> unfolding energy_Min_def
using <x' e< x>
  by auto
qed
qed
show "energy_Min {e. S e g} ⊆ energy_Min {e. ∃e'. S e' g ∧ e' e≤ e} "
proof
  fix x
  assume "x ∈ energy_Min {e. S e g}"
  hence "S x g" using energy_Min_def by auto
  hence "x ∈ {e. ∃e'. S e' g ∧ e' e≤ e}" using energy_Min_def energy_order
ordering_def
  using ordering.eq_iff by fastforce
  show "x ∈ energy_Min {e. ∃e'. S e' g ∧ e' e≤ e} "
  proof(rule ccontr)
    assume "x ∉ energy_Min {e. ∃e'. S e' g ∧ e' e≤ e}"
    from this obtain x' where "x' ∈ {e. ∃e'. S e' g ∧ e' e≤ e}" and "x' e< x"
      using energy_Min_def
      using <x ∈ {e. ∃e'. S e' g ∧ e' e≤ e}> by auto
    from this(1) obtain e' where "S e' g ∧ e' e≤ x'" by auto
    hence "e' e< x" using <x' e< x> energy_order ordering_def
      by (metis (no_types, lifting) ordering_axioms_def partial_preordering_def)

    thus "False"
      using <S e' g ∧ e' e≤ x'> <x ∈ energy_Min {e. S e g}> energy_Min_def
      by auto
  qed
qed
qed
qed

  thus " energy_Min {e. S e g} = a_win_min g" using <energy_Min {e. ∃e'. S e' g
  ∧ e' e≤ e} = a_win_min g> by simp
qed

```

We now conclude that the algorithm indeed returns the minimal attacker winning budgets.

```

lemma a_win_min_is_lfp_sup:
  shows "pareto_sup {(iteration ^^ i) (λg. { }) |. i} = a_win_min"
proof(rule antisymmetry)

  have in_pareto_leq: "∧n. (iteration ^^ n) (λg. { }) ∈ possible_pareto ∧ (iteration
  ^^ n) (λg. { }) ≤ a_win_min"
  proof-
    fix n
    show "(iteration ^^ n) (λg. { }) ∈ possible_pareto ∧ (iteration ^^ n) (λg. { })

```

```

  ⌊ a_win_min"
  proof(induct n)
    case 0
    show ?case
    proof
      show "(iteration ^^ 0) (λg. {g}) ∈ possible_pareto"
        using funpow_simps_right(1) possible_pareto_def by auto
      have "(λg. {g}) ≤ a_win_min"
        unfolding pareto_order_def by simp
      thus "(iteration ^^ 0) (λg. {g}) ≤ a_win_min" using funpow_simps_right(1)
    by simp
    qed
  next
    case (Suc n)
    have "(iteration ^^ (Suc n)) (λg. {g}) = iteration ((iteration ^^ n) (λg. {g}))"

      by simp
    then show ?case using Suc iteration_stays_winning iteration_pareto_functor
  by simp
  qed
  qed

  show "pareto_sup {(iteration ^^ n) (λg. {g}) |. n} ∈ possible_pareto"
  proof(rule pareto_sup_is_sup)
    show "{(iteration ^^ n) (λg. {g}) |. n} ⊆ possible_pareto"
      using in_pareto_leq by auto
  qed

  show "a_win_min ∈ possible_pareto"
    using a_win_min_in_pareto by simp

  show "pareto_sup {(iteration ^^ n) (λg. {g}) |. n} ≤ a_win_min"
    using pareto_sup_is_sup in_pareto_leq a_win_min_in_pareto image_iff rangeE
    by (smt (verit) subsetI)

  define Smin where "Smin = (λg. energy_Min {e. S e g})"

  have "Smin ≤ pareto_sup {(iteration ^^ n) (λg. {g}) |. n}"
    unfolding pareto_order_def proof
      fix g
      show "∀e. e ∈ Smin g →
        (∃e'. e' ∈ pareto_sup {(iteration ^^ n) (λg. {g}) |. n} g ∧ e' ≤ e)"
    proof
      fix e
      show "e ∈ Smin g →
        (∃e'. e' ∈ pareto_sup {(iteration ^^ n) (λg. {g}) |. n} g ∧ e' ≤ e)"
    proof
      assume "e ∈ Smin g"
      hence "S e g" using energy_Min_def Smin_def by simp
      thus "∃e'. e' ∈ pareto_sup {(iteration ^^ n) (λg. {g}) |. n} g ∧ e' ≤ e"
      proof(rule S.induct)
        show "∧g e. g ∉ attacker ∧
          (∃index.
            e =
              energy_sup
                {inv_upd (the (weight g g')) (index g') | g'. weight g g' ≠ None})"

```

```

 $\wedge$ 
      ( $\forall g'. \text{weight } g \ g' \neq \text{None} \longrightarrow$ 
         $S (\text{index } g') \ g' \wedge$ 
         $(\exists e'. e' \in \text{pareto\_sup } \{(\text{iteration } \sim n) (\lambda g. \{ \}) \mid . n\} \ g'$ 
 $\wedge$ 
           $e' \leq \text{index } g')) \implies$ 
 $\exists e'. e' \in \text{pareto\_sup } \{(\text{iteration } \sim n) (\lambda g. \{ \}) \mid . n\} \ g \wedge e' \leq e$ 
proof-
  fix e g
  assume A: "g  $\notin$  attacker  $\wedge$ 
  ( $\exists \text{index}.$ 
    e =
    energy_sup
    {inv_upd (the (weight g g')) (index g') | g'. weight g g'  $\neq$  None}
 $\wedge$ 
    ( $\forall g'. \text{weight } g \ g' \neq \text{None} \longrightarrow$ 
       $S (\text{index } g') \ g' \wedge$ 
       $(\exists e'. e' \in \text{pareto\_sup } \{(\text{iteration } \sim n) (\lambda g. \{ \}) \mid . n\} \ g'$ 
 $\wedge$ 
         $e' \leq \text{index } g'))$ "
  from this obtain index where "e =
    energy_sup
    {inv_upd (the (weight g g')) (index g') | g'. weight g g'  $\neq$  None}"
and
  " $\forall g'. \text{weight } g \ g' \neq \text{None} \longrightarrow$ 
     $S (\text{index } g') \ g' \wedge$ 
     $(\exists e'. e' \in \text{pareto\_sup } \{(\text{iteration } \sim n) (\lambda g. \{ \}) \mid . n\} \ g'$ 
 $\wedge$ 
       $e' \leq \text{index } g'))$ " by auto

  define index' where "index'  $\equiv \lambda g'. \text{SOME } e'. e' \in \text{pareto\_sup } \{(\text{iteration}$ 
 $\sim n) (\lambda g. \{ \}) \mid . n\} \ g' \wedge$ 
     $e' \leq \text{index } g''$ "

  have " $\bigwedge g'. \text{weight } g \ g' \neq \text{None} \implies \exists e'. e' \in \text{pareto\_sup } \{(\text{iteration}$ 
 $\sim n) (\lambda g. \{ \}) \mid . n\} \ g' \wedge$ 
     $e' \leq \text{index } g''$ " using < $\forall g'. \text{weight } g \ g' \neq \text{None} \longrightarrow$ 
     $S (\text{index } g') \ g' \wedge$ 
     $(\exists e'. e' \in \text{pareto\_sup } \{(\text{iteration } \sim n) (\lambda g. \{ \}) \mid . n\} \ g'$ 
 $\wedge$ 
       $e' \leq \text{index } g'))$ > by simp
  hence " $\bigwedge g'. \text{weight } g \ g' \neq \text{None} \implies \text{index}' \ g' \in \text{pareto\_sup } \{(\text{iteration}$ 
 $\sim n) (\lambda g. \{ \}) \mid . n\} \ g' \wedge$ 
     $\text{index}' \ g' \leq \text{index } g''$ " unfolding index'_def using some_eq_ex
    by (metis (mono_tags, lifting))
  hence F: " $\bigwedge g'. \text{weight } g \ g' \neq \text{None} \implies \exists F. F \in \{(\text{iteration } \sim n) (\lambda g.$ 
 $\{ \}) \mid . n\} \wedge \text{index}' \ g' \in F \ g''$ "
    unfolding pareto_sup_def using energy_Min_def by simp
  have index'_len: " $\bigwedge g'. \text{weight } g \ g' \neq \text{None} \implies (\text{index}' \ g') \in \text{energies}$ "

proof-
  fix g'
  assume "weight g g'  $\neq$  None"
  hence " $\exists F. F \in \{(\text{iteration } \sim n) (\lambda g. \{ \}) \mid . n\} \wedge \text{index}' \ g' \in F \ g''$ "
using F by auto
  from this obtain F where F: "F  $\in \{(\text{iteration } \sim n) (\lambda g. \{ \}) \mid . n\}$ "

```

```

 $\wedge \text{index}' g' \in F g''$ 
  by auto
  hence "F  $\in$  possible_pareto"
  using in_pareto_leq by auto
  thus "(index' g')  $\in$  energies"
  unfolding possible_pareto_def using F
  using subset_iff by blast
qed

define index_F where "index_F = ( $\lambda g'$ . (SOME F. (F  $\in$  {(iteration  $\wedge\wedge$  n)
( $\lambda g$ . { }) |. n}  $\wedge$  index' g'  $\in$  F g'})))"
  have IF: " $\bigwedge g'$ . weight g g'  $\neq$  None  $\implies$  index_F g'  $\in$  {(iteration  $\wedge\wedge$  n)
( $\lambda g$ . { }) |. n}  $\wedge$  index' g'  $\in$  index_F g' g'"
  unfolding index_F_def using some_eq_ex < $\bigwedge g'$ . weight g g'  $\neq$  None  $\implies$ 
 $\exists F$ . F  $\in$  {(iteration  $\wedge\wedge$  n) ( $\lambda g$ . { }) |. n}  $\wedge$  index' g'  $\in$  F g'>
  by (metis (mono_tags, lifting))

  have " $\exists F$ . (F  $\in$  {(iteration  $\wedge\wedge$  n) ( $\lambda g$ . { }) |. n}  $\wedge$  ( $\forall g'$ . weight g g'
 $\neq$  None  $\longrightarrow$  index_F g'  $\preceq$  F)))"
  proof-
    define P' where "P' = {index_F g' | g'. weight g g'  $\neq$  None}"
    have " $\exists F'$ . F'  $\in$  {(iteration  $\wedge\wedge$  n) ( $\lambda g$ . { }) |. n}  $\wedge$  ( $\forall F$ . F  $\in$  P'  $\longrightarrow$ 
F  $\preceq$  F')"
      proof(rule finite_directed_set_upper_bound)
        show " $\bigwedge F$  F'."
        F  $\in$  {(iteration  $\wedge\wedge$  n) ( $\lambda g$ . { }) |. n}  $\implies$ 
        F'  $\in$  {(iteration  $\wedge\wedge$  n) ( $\lambda g$ . { }) |. n}  $\implies$ 
         $\exists F''$ . F''  $\in$  {(iteration  $\wedge\wedge$  n) ( $\lambda g$ . { }) |. n}  $\wedge$  F  $\preceq$  F''  $\wedge$  F'  $\preceq$  F''
        proof-
          fix F F'
          assume "F  $\in$  {(iteration  $\wedge\wedge$  n) ( $\lambda g$ . { }) |. n}" and "F'  $\in$  {(iteration
 $\wedge\wedge$  n) ( $\lambda g$ . { }) |. n}"
          from this obtain n m where "F = (iteration  $\wedge\wedge$  n) ( $\lambda g$ . { })" and
          "F' = (iteration  $\wedge\wedge$  m) ( $\lambda g$ . { })" by auto
          show " $\exists F''$ . F''  $\in$  {(iteration  $\wedge\wedge$  n) ( $\lambda g$ . { }) |. n}  $\wedge$  F  $\preceq$  F''
 $\wedge$  F'  $\preceq$  F'"
            proof
              show "((iteration  $\wedge\wedge$  (max n m)) ( $\lambda g$ . { }))  $\in$  {(iteration  $\wedge\wedge$  n)
( $\lambda g$ . { }) |. n}  $\wedge$  F  $\preceq$  ((iteration  $\wedge\wedge$  (max n m)) ( $\lambda g$ . { }))  $\wedge$  F'  $\preceq$  ((iteration  $\wedge\wedge$ 
(max n m)) ( $\lambda g$ . { }))"
              proof-
                have " $\bigwedge i j$ .  $i \leq j \implies ((\text{iteration } \wedge\wedge i) (\lambda g. \{ \})) \preceq ((\text{iteration } \wedge\wedge j) (\lambda g. \{ \}))$ "
                proof-
                  fix i j
                  show " $i \leq j \implies ((\text{iteration } \wedge\wedge i) (\lambda g. \{ \})) \preceq ((\text{iteration } \wedge\wedge j) (\lambda g. \{ \}))$ "
                  proof-
                    assume " $i \leq j$ "
                    thus "(iteration  $\wedge\wedge i$ ) ( $\lambda g$ . { })  $\preceq$  (iteration  $\wedge\wedge j$ ) ( $\lambda g$ .
{ })"
                    proof(induct "j-i" arbitrary: i j)
                      case 0
                      hence "i = j" by simp
                      then show ?case
                      by (simp add: in_pareto_leq reflexivity)

```

```

      next
      case (Suc x)
      show ?case
      proof(rule transitivity)
        show A: "(iteration ^^ i) (λg. { }) ∈ possible_pareto"
using in_pareto_leq by simp
        show B: "(iteration ^^ (Suc i)) (λg. { }) ∈ possible_pareto"
using in_pareto_leq by blast
        show C: "(iteration ^^ j) (λg. { }) ∈ possible_pareto"
using in_pareto_leq by simp

        have D: "(iteration ^^ (Suc i)) (λg. { }) = iteration
((iteration ^^ i) (λg. { })))" using funpow.simps by simp

        have "((iteration ^^ i) (λg. { })) ≤ iteration ((iteration
^^ i) (λg. { })))"

        proof(induct i)
          case 0
          then show ?case using pareto_minimal_element in_pareto_leq
            by simp
        next
          case (Suc i)
          then show ?case using in_pareto_leq iteration_monotonic
funpow.simps(2)

            by (smt (verit, del_insts) comp_eq_dest_lhs)
          qed
          thus "(iteration ^^ i) (λg. { }) ≤ (iteration ^^ (Suc
i)) (λg. { })"

            unfolding D by simp

          have "x = j - (Suc i)" using Suc by simp
          have "(Suc i) ≤ j"
            using diff_diff_left Suc by simp
          show "(iteration ^^ (Suc i)) (λg. { }) ≤ (iteration
^^ j) (λg. { })"

            using Suc <x = j - (Suc i)> <(Suc i) ≤ j> by blast
          qed
        qed
      qed
    thus ?thesis
      using <F = (iteration ^^ n) (λg. { })> <F' = (iteration
^^ m)(λg. { })> <F' ∈ {(iteration ^^ n) (λg. { }) |. n}> max.cobounded2 by auto
    qed
  qed
qed

show "{(iteration ^^ n) (λg. { }) |. n} ≠ {}"
  by auto
show "P' ⊆ {(iteration ^^ n) (λg. { }) |. n}" using P'_def IF
  by blast
have "finite {g'. weight g g' ≠ None}" using finite_positions
  by (smt (verit) Collect_cong finite_Collect_conjI)
thus "finite P'" unfolding P'_def using nonpos_eq_pos
  by auto
show "{(iteration ^^ n) (λg. { }) |. n} ⊆ possible_pareto" using

```



```

in_pareto_leq by auto
qed
from this obtain F' where "F' ∈ {(iteration ^^ n) (λg. { }) |. n} ∧
(∀F. F ∈ P' → F ≤ F')" by auto
hence "F' ∈ {(iteration ^^ n) (λg. { }) |. n} ∧ (∀g'. weight g g'
≠ None → index_F g' ≤ F')"
using P'_def
by auto
thus ?thesis by auto
qed
from this obtain F' where F': "F' ∈ {(iteration ^^ n) (λg. { }) |. n}
∧ (∀g'. weight g g' ≠ None → index_F g' ≤ F'" by auto

have IE: "∧g'. weight g g' ≠ None ⇒ ∃e'. e' ∈ F' g' ∧ e' e ≤ index'
g'"
proof-
fix g'
assume "weight g g' ≠ None"
hence "index_F g' ≤ F'" using F' by simp
thus "∃e'. e' ∈ F' g' ∧ e' e ≤ index' g'" unfolding pareto_order_def
using IF <weight g g' ≠ None>
by simp
qed

define e_index where "e_index = (λg'. SOME e'. e' ∈ F' g' ∧ e' e ≤
index' g')"
hence "∧g'. weight g g' ≠ None ⇒ e_index g' ∈ F' g' ∧ e_index g'
e ≤ index' g'"
using IE some_eq_ex
by (metis (no_types, lifting))

have sup_leq1: "energy_sup {inv_upd (the (weight g g')) (e_index g')) |
g'. weight g g' ≠ None} e ≤ energy_sup {inv_upd (the (weight g g')) (index' g')) |
g'. weight g g' ≠ None}"
proof(cases "{g'. weight g g' ≠ None} = { }")
case True
then show ?thesis
by (simp add: bounded_join_semilattice)
next
case False
hence "{inv_upd (the (weight g g')) (e_index g')) | g'. weight g g'
≠ None} ≠ { }" by simp
then show ?thesis
proof(rule energy_sup_leq_energy_sup)
show "∧a. a ∈ {inv_upd (the (weight g g')) (e_index g')) | g'. weight
g g' ≠ None} ⇒
∃b∈{inv_upd (the (weight g g')) (index' g')) | g'. weight g g' ≠ None}.
a e ≤ b"
proof-
fix a
assume "a ∈ {inv_upd (the (weight g g')) (e_index g')) | g'. weight
g g' ≠ None}"
from this obtain g' where "weight g g' ≠ None" and "a=inv_upd
(the (weight g g')) (e_index g')" by auto
have "a e ≤ inv_upd (the (weight g g')) (index' g')"
unfolding <a=inv_upd (the (weight g g')) (e_index g')>

```

```

      using <weight g g' ≠ None>
    proof(rule inverse_monotonic)
      show "e_index g' e ≤ index' g'" using <∧g'. weight g g' ≠ None
⇒ e_index g' ∈ F' g' ∧ e_index g' e ≤ index' g'> <weight g g' ≠ None> by auto
      hence "(e_index g') ∈ energies" using index'_len <weight g
g' ≠ None> energy_order ordering_def
      by (smt (z3) F' <∧g'. weight g g' ≠ None ⇒ e_index g' ∈
F' g' ∧ e_index g' e ≤ index' g'> full_SetCompr_eq in_pareto_leq mem_Collect_eq
possible_pareto_def subset_iff)
      thus "inverse_application (the (weight g g')) (e_index g') ≠
None"

      using inv_well_defined <weight g g' ≠ None>
      by auto
      show "(e_index g') ∈ energies"
      using <(e_index g') ∈ energies> by auto
    qed
    thus "∃b∈{inv_upd (the (weight g g')) (index' g') | g'. weight
g g' ≠ None}. a e ≤ b"
      using <weight g g' ≠ None>
      by blast
    qed
    have "∧g'. weight g g' ≠ None ⇒ (e_index g') ∈ energies"
      using index'_len energy_order ordering_def
      by (smt (z3) F' <∧g'. weight g g' ≠ None ⇒ e_index g' ∈ F'
g' ∧ e_index g' e ≤ index' g'> full_SetCompr_eq in_pareto_leq mem_Collect_eq possible_pareto_def
subset_iff)
      thus "∧a. a ∈ {inv_upd (the (weight g g')) (e_index g') | g'. weight
g g' ≠ None} ⇒
a ∈ energies"
      using inv_well_defined by blast

      have "{inv_upd (the (weight g g')) (e_index g') | g'. weight g g'
≠ None} ⊆ {inv_upd (the (weight g g')) (e_index g') | g'. g' ∈ positions}" by auto

      thus "finite {inv_upd (the (weight g g')) (e_index g') | g'. weight
g g' ≠ None}"

      using finite_positions finite_image_set rev_finite_subset by fastforce
      have "{inv_upd (the (weight g g')) (index' g') | g'. weight g g'
≠ None} ⊆ {inv_upd (the (weight g g')) (index' g') | g'. g' ∈ positions}" by auto

      thus "finite {inv_upd (the (weight g g')) (index' g') | g'. weight
g g' ≠ None}"

      using finite_positions finite_image_set rev_finite_subset by fastforce
      show "{inv_upd (the (weight g g')) (index' g') | g'. weight g g'
≠ None} ⊆ energies"
    proof-
      have "∧g'. weight g g' ≠ None ⇒ index' g' ∈ energies"
      by (simp add: index'_len)
      thus ?thesis
      using inv_well_defined by blast
    qed
  qed
qed

```

have sup\_leq2: "energy\_sup {inv\_upd (the (weight g g')) (index' g') | g'.  
 weight g g' ≠ None} e ≤ energy\_sup {inv\_upd (the (weight g g')) (index' g') | g'.

```

weight g g' ≠ None}"
  proof(cases "{g'. weight g g' ≠ None} = {}")
    case True
      then show ?thesis
        using sup_leq1 by force
    next
      case False
        hence "{inv_upd (the (weight g g')) (index' g')) |g'. weight g g' ≠
None} ≠ {}" by simp
        then show ?thesis
          proof(rule energy_sup_leq_energy_sup)
            show "∧a. a ∈ {inv_upd (the (weight g g')) (index' g')) |g'. weight
g g' ≠ None} ⇒
              ∃b∈{inv_upd (the (weight g g')) (index g')) |g'. weight g g' ≠ None}. a
e ≤ b"
              proof-
                fix a
                assume "a ∈ {inv_upd (the (weight g g')) (index' g')) |g'. weight
g g' ≠ None}"
                from this obtain g' where "weight g g' ≠ None" and "a=inv_upd
(the (weight g g')) (index' g'))" by auto
                hence "a e ≤ inv_upd (the (weight g g')) (index g'))"
                  using inverse_monotonic <∧g'. weight g g' ≠ None ⇒ e_index
g' ∈ F' g' ∧ e_index g' e ≤ index' g'> F' possible_pareto_def
                  using <∧g'. weight g g' ≠ None ⇒ index' g' ∈ pareto_sup
{(iteration ^^ n) (λg. { }) |. n} g' ∧ index' g' e ≤ index g'> energy_order
                  by (meson inv_well_defined index'_len)
                thus "∃b∈{inv_upd (the (weight g g')) (index g')) |g'. weight
g g' ≠ None}. a e ≤ b"
                  using <weight g g' ≠ None>
                  by blast
              qed
              show "∧a. a ∈ {inv_upd (the (weight g g')) (index' g')) |g'. weight
g g' ≠ None} ⇒
                a ∈ energies"
                using index'_len inv_well_defined by blast

                have "{inv_upd (the (weight g g')) (e_index g')) |g'. weight g g'
≠ None} ⊆ {inv_upd (the (weight g g')) (e_index g')) |g'. g' ∈ positions}" by auto

                thus "finite {inv_upd (the (weight g g')) (index g')) |g'. weight
g g' ≠ None}"
                  using finite_positions finite_image_set rev_finite_subset by fastforce
                  have "{inv_upd (the (weight g g')) (index' g')) |g'. weight g g'
≠ None} ⊆ {inv_upd (the (weight g g')) (index' g')) |g'. g' ∈ positions}" by auto

                  thus "finite {inv_upd (the (weight g g')) (index' g')) |g'. weight
g g' ≠ None}"
                    using finite_positions finite_image_set rev_finite_subset by fastforce
                    show "{inv_upd (the (weight g g')) (index g')) |g'. weight g g'
≠ None} ⊆ energies"
                    using inv_well_defined
                    by (smt (verit, best) <∧g'. weight g g' ≠ None ⇒ index' g'
∈ pareto_sup {(iteration ^^ n) (λg. { }) |. n} g' ∧ index' g' e ≤ index g'> galois_energy_game.
galois_energy_game_axioms index'_len mem_Collect_eq subsetI)
                    qed

```

```

qed

have "⋀g'. weight g g' ≠ None ⇒ (e_index g') ∈ energies"
proof-
  fix g'
  assume "weight g g' ≠ None"
  hence "e_index g' ∈ F' g' ∧ e_index g' ≤ index' g'" using <⋀g'.
weight g g' ≠ None ⇒ e_index g' ∈ F' g' ∧ e_index g' ≤ index' g'>
    by simp
  thus "(e_index g') ∈ energies" using F' possible_pareto_def
    using in_pareto_leq by blast
qed
hence es_in: "energy_sup {inv_upd (the (weight g g')) (e_index g')} | g'.
weight g g' ≠ None} ∈ {energy_sup
  {inv_upd (the (weight g g')) (e_index g')} | g'. weight g
g' ≠ None} |
  e_index.
  ∀g'. weight g g' ≠ None →
    (e_index g') ∈ energies ∧ e_index g' ∈ F' g'}"
using <⋀g'. weight g g' ≠ None ⇒ e_index g' ∈ F' g' ∧ e_index
g' ≤ index' g'>
  by blast
have "{energy_sup {inv_upd (the (weight g g')) (e_index g')} | g'. weight
g g' ≠ None} | e_index. ∀g'. weight g g' ≠ None → e_index g' ∈ energies ∧ e_index
g' ∈ F' g'} ⊆ energies"
proof
  fix x
  assume "x ∈ {energy_sup {inv_upd (the (weight g g')) (e_index g')}
| g'. weight g g' ≠ None} | e_index. ∀g'. weight g g' ≠ None → e_index g' ∈ energies
∧ e_index g' ∈ F' g'}"
  from this obtain e_index where "x = energy_sup {inv_upd (the (weight
g g')) (e_index g')} | g'. weight g g' ≠ None}" and "∀g'. weight g g' ≠ None →
e_index g' ∈ energies ∧ e_index g' ∈ F' g'"
  by auto
  have "{inv_upd (the (weight g g')) (e_index g')} | g'. weight g g' ≠
None} ⊆ {inv_upd (the (weight g g')) (e_index g')} | g'. g' ∈ positions}"
  by auto
  hence fin: "finite {inv_upd (the (weight g g')) (e_index g')} | g'.
weight g g' ≠ None}"
  using finite_positions
  by (simp add: Collect_mem_eq finite_image_set rev_finite_subset)
  have "{inv_upd (the (weight g g')) (e_index g')} | g'. weight g g' ≠
None} ⊆ energies"
  using inv_well_defined
  using <∀g'. weight g g' ≠ None → e_index g' ∈ energies ∧ e_index
g' ∈ F' g'> by blast
  thus "x ∈ energies" unfolding <x = energy_sup {inv_upd (the (weight
g g')) (e_index g')} | g'. weight g g' ≠ None}> using bounded_join_semilattice fin
  by simp
qed
hence "∃em. em ∈ energy_Min
  {energy_sup
    {inv_upd (the (weight g g')) (e_index g')} | g'. weight g
g' ≠ None} |
    e_index.

```

```

       $\forall g'. \text{weight } g \ g' \neq \text{None} \longrightarrow$ 
       $(e\_index \ g') \in \text{energies} \wedge e\_index \ g' \in F' \ g'$ 
       $\wedge \text{em } e \leq \text{energy\_sup } \{\text{inv\_upd } (\text{the } (\text{weight } g \ g')) \ (e\_index \ g')) \mid$ 
 $g'. \text{weight } g \ g' \neq \text{None}\}$ 
      using energy_Min_contains_smaller es_in
      by meson
      hence " $\exists \text{em}. \text{em} \in \text{iteration } F' \ g \wedge \text{em } e \leq \text{energy\_sup } \{\text{inv\_upd } (\text{the } (\text{weight } g \ g')) \ (e\_index \ g')) \mid g'. \text{weight } g \ g' \neq \text{None}\}$ "
      unfolding iteration_def using A
      by simp
      from this obtain em where EM: " $\text{em} \in \text{iteration } F' \ g \wedge \text{em } e \leq \text{energy\_sup } \{\text{inv\_upd } (\text{the } (\text{weight } g \ g')) \ (e\_index \ g')) \mid g'. \text{weight } g \ g' \neq \text{None}\}$ "
      by auto
      from F' have F': " $\text{iteration } F' \in \{(\text{iteration } \wedge n) \ (\lambda g. \{ \}) \mid n\}$ " using
      funpow.simps image_iff rangeE
      by (smt (z3) UNIV_I comp_eq_dest_lhs)
      hence EM0: " $\text{em} \in \{e. \exists F. F \in \{(\text{iteration } \wedge n) \ (\lambda g. \{ \}) \mid n\} \wedge e$ 
 $\in F \ g\}$ "
      using EM by auto
      have " $\{e. \exists F. F \in \{(\text{iteration } \wedge n) \ (\lambda g. \{ \}) \mid n\} \wedge e \in F \ g\} \subseteq \text{energies}$ "
      using possible_pareto_def
      using in_pareto_leq by fastforce
      hence " $\exists \text{em}'. \text{em}' \in \text{pareto\_sup } \{(\text{iteration } \wedge n) \ (\lambda g. \{ \}) \mid n\} \ g \wedge$ 
 $\text{em}' \ e \leq \text{em}$ "
      unfolding pareto_sup_def using F' energy_Min_contains_smaller EM0
      by meson
      from this obtain em' where EM': " $\text{em}' \in \text{pareto\_sup } \{(\text{iteration } \wedge n) \ (\lambda g. \{ \}) \mid n\} \ g \wedge \text{em}' \ e \leq \text{em}$ " by auto
      hence " $\text{em}' \ e \leq \text{em}$ " by simp
      hence " $\text{em}' \ e \leq \text{energy\_sup } \{\text{inv\_upd } (\text{the } (\text{weight } g \ g')) \ (e\_index \ g')) \mid$ 
 $g'. \text{weight } g \ g' \neq \text{None}\}$ " using EM energy_order ordering_def
      by (metis (no_types, lifting) partial_preordering_def)
      hence " $\text{em}' \ e \leq \text{energy\_sup } \{\text{inv\_upd } (\text{the } (\text{weight } g \ g')) \ (\text{index}' \ g') \mid g'. \text{weight } g \ g' \neq \text{None}\}$ " using sup_leq1 energy_order ordering_def
      by (metis (no_types, lifting) partial_preordering_def)
      hence " $\text{em}' \ e \leq \text{energy\_sup } \{\text{inv\_upd } (\text{the } (\text{weight } g \ g')) \ (\text{index } g') \mid g'. \text{weight } g \ g' \neq \text{None}\}$ " using sup_leq2 energy_order ordering_def
      by (metis (no_types, lifting) partial_preordering_def)
      hence " $\text{em}' \ e \leq e$ " using <e =
      energy_sup
      {inv_upd (the (weight g g')) (index g') | g'. weight g g' ≠ None}>
      energy_order ordering_def
      by (metis (no_types, lifting) partial_preordering_def)
      thus " $\exists e'. e' \in \text{pareto\_sup } \{(\text{iteration } \wedge n) \ (\lambda g. \{ \}) \mid n\} \ g \wedge e'$ 
 $e \leq e$ "
      using EM' by auto
      qed
      show " $\bigwedge g \ e. g \in \text{attacker} \wedge$ 
      ( $\exists g'. \text{weight } g \ g' \neq \text{None} \wedge$ 
      ( $\exists e'. (\text{S } e' \ g' \wedge$ 
      ( $\exists e'a. e'a \in \text{pareto\_sup } \{(\text{iteration } \wedge n) \ (\lambda g. \{ \}) \mid n\}$ 
 $g' \wedge e'a \ e \leq e')$ )  $\wedge$ 
       $e = \text{inv\_upd } (\text{the } (\text{weight } g \ g')) \ e') \implies$ 
       $\exists e'. e' \in \text{pareto\_sup } \{(\text{iteration } \wedge n) \ (\lambda g. \{ \}) \mid n\} \ g \wedge e' \ e \leq e$ "

```

```

proof-
  fix g e
  assume A: "g ∈ attacker ∧
    (∃ g'. weight g g' ≠ None ∧
      (∃ e'. (S e' g' ∧
        (∃ e'a. e'a ∈ pareto_sup {(iteration ^^ n) (λg. { }) |. n}
          g' ∧ e'a e ≤ e')) ∧
          e = inv_upd (the (weight g g')) e'))"
  from this obtain g' e' e'' where "weight g g' ≠ None" and "S e' g'"
  and "e = inv_upd (the (weight g g')) e'" and
    "e'' ∈ pareto_sup {(iteration ^^ n) (λg. { }) |. n} g' ∧ e''
e ≤ e'" by auto

  have "e'' ∈ energies" using <e'' ∈ pareto_sup {(iteration ^^ n) (λg.
{ }) |. n} g' ∧ e'' e ≤ e'> in_pareto_leq possible_pareto_def
  using <pareto_sup {(iteration ^^ n) (λg. { }) |. n} ∈ possible_pareto>
by blast
  have "inv_upd (the (weight g g')) e'' e ≤ inv_upd (the (weight g g'))
e'"
  using <weight g g' ≠ None>
proof(rule inverse_monotonic)
  show "e'' e ≤ e'" using <e'' ∈ pareto_sup {(iteration ^^ n) (λg. { })
|. n} g' ∧ e'' e ≤ e'> by auto
  have "e' ∈ energies" using length_S <weight g g' ≠ None> <S e' g'>
by auto
  show "inverse_application (the (weight g g')) e'' ≠ None"
  using inv_well_defined <weight g g' ≠ None> <e'' ∈ energies>
  by blast
  show "e'' ∈ energies"
  by (simp add: <e'' ∈ energies>)
qed
  have "e'' ∈ energy_Min {e. ∃ F. F ∈ {(iteration ^^ n) (λg. { }) |. n}
  ∧ e ∈ F g'}"
  using <e'' ∈ pareto_sup {(iteration ^^ n) (λg. { }) |. n} g' ∧ e''
e ≤ e'> unfolding pareto_sup_def by simp
  hence "∃ n. e'' ∈ (iteration ^^ n) (λg. { }) g'"
  using energy_Min_def
  by auto
  from this obtain n where "e'' ∈ (iteration ^^ n) (λg. { }) g'" by auto

  hence e''in: "inv_upd (the (weight g g')) e'' ∈ {inv_upd (the (weight
g g')) e' | e' g'.
  e' ∈ energies ∧ weight g g' ≠ None ∧ e' ∈ (iteration ^^ n) (λg. { })
g'}"
  using <weight g g' ≠ None> length_S <S e' g'> <e'' ∈ pareto_sup
{(iteration ^^ n) (λg. { }) |. n} g' ∧ e'' e ≤ e'> <e'' ∈ energies> by blast

  define Fn where "Fn = (iteration ^^ n) (λg. { })"
  have "{inv_upd (the (weight g g')) e' | e' g'. e' ∈ energies ∧ weight
g g' ≠ None ∧ e' ∈ Fn g'} ⊆ energies"
  using inv_well_defined by auto
  hence "∃ e'''. e''' ∈ iteration Fn g ∧ e''' e ≤ inv_upd (the (weight
g g')) e'"
  unfolding iteration_def using Fn_def energy_Min_contains_smaller A
e''in
  by meson

```

```

      from this obtain e''' where E''': "e''' ∈ iteration ((iteration ^^ n)
(λg. {})) g ∧ e''' ≤ inv_upd (the (weight g g')) e'"
      using Fn_def by auto
      hence "e''' ∈ ((iteration ^^ (Suc n)) (λg. {})) g" by simp
      hence E''': "e''' ∈ {e. ∃F. F ∈ {(iteration ^^ n) (λg. {}) |. n}
      ∧ e ∈ F g}" by blast
      have "{e. ∃F. F ∈ {(iteration ^^ n) (λg. {}) |. n} ∧ e ∈ F g} ⊆ energies"
      using possible_pareto_def in_pareto_leq by blast
      hence "∃em. em ∈ pareto_sup {(iteration ^^ n) (λg. {}) |. n} g ∧ em
e ≤ e'"
      unfolding pareto_sup_def using energy_Min_contains_smaller E'''
      by meson
      from this obtain em where EM: "em ∈ pareto_sup {(iteration ^^ n) (λg.
      {} |. n} g ∧ em ≤ e'" by auto

      show "∃e'. e' ∈ pareto_sup {(iteration ^^ n) (λg. {}) |. n} g ∧ e'
e ≤ e"

      proof
        show "em ∈ pareto_sup {(iteration ^^ n) (λg. {}) |. n} g ∧ em ≤
e"

        proof
          show "em ∈ pareto_sup {(iteration ^^ n) (λg. {}) |. n} g" using
EM by simp
          have "inv_upd (the (weight g g')) e' ≤ e"
          using <e = inv_upd (the (weight g g')) e'> <inv_upd (the (weight
g g')) e'> e ≤ inv_upd (the (weight g g')) e'> by simp
          hence "e' ≤ e" using E''' energy_order ordering_def
          by (metis (mono_tags, lifting) partial_preordering_def)
          thus "em ≤ e" using EM energy_order ordering_def
          by (metis (mono_tags, lifting) partial_preordering_def)
        qed
      qed
    qed
  qed
  qed
  qed
  qed
  qed
  thus "a_win_min ≤ pareto_sup {(iteration ^^ n) (λg. {}) |. n}"
  using a_win_min_is_minS Smin_def by simp
qed

```

We can argue that the algorithm always terminates by showing that only finitely many iterations are needed before a fixed point (the minimal attacker winning budgets) is reached.

```

lemma finite_iterations:
  shows "∃i. a_win_min = (iteration ^^ i) (λg. {})"
proof
  have in_pareto_leq: "∧n. (iteration ^^ n) (λg. {}) ∈ possible_pareto ∧ (iteration
  ^^ n) (λg. {}) ≤ a_win_min"
  proof-
    fix n
    show "(iteration ^^ n) (λg. {}) ∈ possible_pareto ∧ (iteration ^^ n) (λg. {})
    ≤ a_win_min"
    proof(induct n)
      case 0

```

```

show ?case
proof
  show "(iteration ^^ 0) (λg. { }) ∈ possible_pareto"
  using funpow.simps possible_pareto_def by auto
  have "(λg. { }) ≤ a_win_min"
  unfolding pareto_order_def by simp
  thus "(iteration ^^ 0) (λg. { }) ≤ a_win_min" using funpow.simps by simp
qed
next
case (Suc n)
have "(iteration ^^ (Suc n)) (λg. { }) = iteration ((iteration ^^ n) (λg. { })))"

  using funpow.simps by simp
then show ?case using Suc iteration_stays_winning iteration_pareto_functor
by simp
qed
qed

have A: "∧g n m e. n ≤ m ⇒ e ∈ a_win_min g ⇒ e ∈ (iteration ^^ n) (λg. { })
g ⇒ e ∈ (iteration ^^ m) (λg. { }) g"
proof-
  fix g n m e
  assume "n ≤ m" and "e ∈ a_win_min g" and "e ∈ (iteration ^^ n) (λg. { }) g"
  thus "e ∈ (iteration ^^ m) (λg. { }) g"
  proof(induct "m-n" arbitrary: n m)
    case 0
    then show ?case by simp
  next
    case (Suc x)
    hence "Suc n ≤ m"
    by linarith
    have "x = m - (Suc n)" using Suc by simp
    have "e ∈ (iteration ^^ (Suc n)) (λg. { }) g"
    proof-
      have "(iteration ^^ n) (λg. { }) ≤ (iteration ^^ (Suc n)) (λg. { })"
      proof(induct n)
        case 0
        then show ?case
        by (simp add: pareto_minimal_element)
      next
        case (Suc n)
        have "(iteration ^^ (Suc (Suc n))) (λg. { }) = iteration ((iteration ^^
(Suc n)) (λg. { })))"
        using funpow.simps by simp
        then show ?case using Suc iteration_monotonic in_pareto_leq funpow.simps(2)
        by (smt (verit) comp_apply)
      qed
      hence "∃e'. e' ∈ (iteration ^^ (Suc n)) (λg. { }) g ∧ e' e ≤ e"
      unfolding pareto_order_def using Suc by simp
      from this obtain e' where "e' ∈ (iteration ^^ (Suc n)) (λg. { }) g ∧ e'
e ≤ e" by auto
      hence "(∃e''. e'' ∈ a_win_min g ∧ e'' e ≤ e'" using in_pareto_leq unfolding
pareto_order_def
      by blast
      from this obtain e'' where "e'' ∈ a_win_min g ∧ e'' e ≤ e'" by auto
      hence "e'' = e" using Suc energy_Min_def <e' ∈ (iteration ^^ (Suc n)) (λg.

```



```

{ }) g ∧ e' e ≤ e >
  by (smt (verit, ccfv_SIG) mem_Collect_eq upwards_closure_wb_len)
  hence "e = e'" using <e' ∈ (iteration ^^ (Suc n)) (λg. { }) g ∧ e' e ≤ e >
  <e'' ∈ a_win_min g ∧ e'' e ≤ e' >
    by (meson energy_order ordering.antisym)
    thus ?thesis using <e' ∈ (iteration ^^ (Suc n)) (λg. { }) g ∧ e' e ≤ e > by
simp
  qed
  then show ?case using Suc <x = m - (Suc n) > <Suc n ≤ m > by auto
  qed
  qed
  hence A1: "∧g n m. n ≤ m ⇒ a_win_min g = (iteration ^^ n) (λg. { }) g ⇒ a_win_min
g = (iteration ^^ m) (λg. { }) g"
  proof-
    fix g n m
    assume "n ≤ m" and "a_win_min g = (iteration ^^ n) (λg. { }) g"
    show "a_win_min g = (iteration ^^ m) (λg. { }) g"
    proof
      show "a_win_min g ⊆ (iteration ^^ m) (λg. { }) g"
      proof
        fix e
        assume "e ∈ a_win_min g"
        hence "e ∈ (iteration ^^ n) (λg. { }) g" using <a_win_min g = (iteration
^^ n) (λg. { }) g > by simp
        thus "e ∈ (iteration ^^ m) (λg. { }) g" using A <n ≤ m > <e ∈ a_win_min g >
by auto
        qed
        show "(iteration ^^ m) (λg. { }) g ⊆ a_win_min g"
        proof
          fix e
          assume "e ∈ (iteration ^^ m) (λg. { }) g"
          hence "∃e'. e' ∈ a_win_min g ∧ e' e ≤ e" using in_pareto_leq unfolding pareto_order_def
by auto
          from this obtain e' where "e' ∈ a_win_min g ∧ e' e ≤ e" by auto
          hence "e' ∈ (iteration ^^ n) (λg. { }) g" using <a_win_min g = (iteration
^^ n) (λg. { }) g > by simp
          hence "e' ∈ (iteration ^^ m) (λg. { }) g" using A <n ≤ m > <e' ∈ a_win_min
g ∧ e' e ≤ e > by simp
          hence "e = e'" using in_pareto_leq unfolding possible_pareto_def
          using <e ∈ (iteration ^^ m) (λg. { }) g > <e' ∈ a_win_min g ∧ e' e ≤ e >
by blast
          thus "e ∈ a_win_min g" using <e' ∈ a_win_min g ∧ e' e ≤ e > by simp
        qed
      qed
    qed
  have "∧g e. e ∈ a_win_min g ⇒ ∃n. e ∈ (iteration ^^ n) (λg. { }) g"
  proof-
    fix g e
    assume "e ∈ a_win_min g"
    hence "e ∈ (pareto_sup {(iteration ^^ n) (λg. { }) |. n}) g" using a_win_min_is_lfp_sup
finite_positions nonpos_eq_pos by simp
    thus "∃n. e ∈ (iteration ^^ n) (λg. { }) g" unfolding pareto_sup_def energy_Min_def
by auto
  qed
  define n_e where "n_e = (λ g e. SOME n. e ∈ (iteration ^^ n) (λg. { }) g)"

```

```

hence "⋀g e. n_e g e = (SOME n. e ∈ (iteration ^^ n) (λg. { }) g)"
  by simp
hence n_e: "⋀g e. e ∈ a_win_min g ⇒ e ∈ (iteration ^^ (n_e g e)) (λg. { }) g"
  using some_eq_ex <⋀g e. e ∈ a_win_min g ⇒ ∃n. e ∈ (iteration ^^ n) (λg.
{ }) g>
  by metis

have fin_e: "⋀g. finite {n_e g e | e. e ∈ a_win_min g}"
  using minimal_winning_budget_finite by fastforce
define m_g where "m_g = (λg. Max {n_e g e | e. e ∈ a_win_min g})"
hence n_e_leq: "⋀g e. e ∈ a_win_min g ⇒ n_e g e ≤ m_g g" using A fin_e
  using Collect_mem_eq Max.coboundedI by fastforce
have MG: "⋀g. a_win_min g = (iteration ^^ (m_g g)) (λg. { }) g"
proof
  fix g
  show "a_win_min g ⊆ (iteration ^^ (m_g g)) (λg. { }) g"
  proof
    fix e
    assume "e ∈ a_win_min g"
    hence "e ∈ (iteration ^^ (n_e g e)) (λg. { }) g"
      using n_e by simp
    thus "e ∈ (iteration ^^ (m_g g)) (λg. { }) g"
      using A <e ∈ a_win_min g> n_e_leq
      by blast
  qed
  show "(iteration ^^ (m_g g)) (λg. { }) g ⊆ a_win_min g"
  proof
    fix e
    assume "e ∈ (iteration ^^ (m_g g)) (λg. { }) g"
    hence "∃e'. e' ∈ a_win_min g ∧ e' e ≤ e"
      using in_pareto_leq unfolding pareto_order_def
      by simp
    from this obtain e' where "e' ∈ a_win_min g ∧ e' e ≤ e" by auto
    hence "e' ∈ (iteration ^^ (m_g g)) (λg. { }) g" using <a_win_min g ⊆ (iteration
^^ (m_g g)) (λg. { }) g> by auto
    hence "e = e'" using <e' ∈ a_win_min g ∧ e' e ≤ e> in_pareto_leq unfolding
possible_pareto_def
      using <e ∈ (iteration ^^ (m_g g)) (λg. { }) g> by blast
    thus "e ∈ a_win_min g" using <e' ∈ a_win_min g ∧ e' e ≤ e> by auto
  qed
qed

have fin_m: "finite {m_g g | g. g ∈ positions}"
proof-
  have "finite {p. p ∈ positions}"
    using finite_positions by fastforce
  then show ?thesis
    using finite_image_set by blast
qed
hence "⋀g. m_g g ≤ Max {m_g g | g. g ∈ positions}"
  using Max_ge by blast
have "⋀g. a_win_min g = (iteration ^^ (Max {m_g g | g. g ∈ positions})) (λg. { })
g"
proof-
  fix g
  have G: "a_win_min g = (iteration ^^ (m_g g)) (λg. { }) g" using MG by simp

```

```

    from fin_m have "\/g. m_g g ≤ Max {m_g g | g. g ∈ positions}"
    using Max_ge by blast
    thus "a_win_min g = (iteration ^^ (Max {m_g g | g. g ∈ positions})) (λg. {})"
g"
    using A1 G by simp
qed

hence "a_win_min ≤ (iteration ^^ (Max {m_g g | g. g ∈ positions})) (λg. {})"
    using pareto_order_def
    using in_pareto_leq by auto
thus "a_win_min = (iteration ^^ (Max {m_g g | g. g ∈ positions})) (λg. {})"
    using in_pareto_leq <\/g. a_win_min g = (iteration ^^ (Max {m_g g | g. g ∈ positions}))
(λg. {}) g> by auto
qed

```

### 4.3 Applying Kleene's Fixed Point Theorem

We now establish compatablity with Complete\_Non\_Orders.thy.

```

sublocale attractive possible_pareto pareto_order
    unfolding attractive_def using pareto_partial_order(2,3)
    by (smt (verit) attractive_axioms_def semiattractiveI transp_on_def)

```

```

abbreviation pareto_order_dual (infix "≥" 80) where
    "pareto_order_dual ≡ (λx y. y ≤ x)"

```

We now conclude, that Kleene's fixed point theorem is applicable.

```

lemma kleene_lfp_iteration:
    shows "extreme_bound possible_pareto (≤) {(iteration ^^ i) (λg. {}) |. i} =
        extreme {s ∈ possible_pareto. sympartp (≤) (iteration s) s} (≥)"
proof(rule kleene_qfp_is_dual_extreme)
    show "omega_chain-complete possible_pareto (≤)"
    unfolding omega_chain_def complete_def
    proof
        fix P
        show "P ⊆ possible_pareto ⟶
            (∃f. monotone (≤) (≤) f ∧ range f = P) ⟶ (∃s. extreme_bound possible_pareto
(≤) P s)"
        proof
            assume "P ⊆ possible_pareto"
            show "(∃f. monotone (≤) (≤) f ∧ range f = P) ⟶ (∃s. extreme_bound possible_pareto
(≤) P s) "
            proof
                assume "∃f. monotone (≤) (≤) f ∧ range f = P"
                show "∃s. extreme_bound possible_pareto (≤) P s"
                proof
                    show "extreme_bound possible_pareto (≤) P (pareto_sup P)"
                    unfolding extreme_bound_def extreme_def using pareto_sup_is_sup
                    using <P ⊆ possible_pareto> by fastforce
                qed
            qed
        qed
    qed
    show "omega_chain-continuous possible_pareto (≤) possible_pareto (≤) iteration"
    using finite_positions iteration_scott_continuous scott_continuous_imp_omega_continuous
by simp

```

```

show "(λg. {}) ∈ possible_pareto"
  unfolding possible_pareto_def
  by simp
show "∀x ∈ possible_pareto. x ≥ (λg. {})"
  using pareto_minimal_element
  by simp
qed

```

We now apply Kleene's fixed point theorem, showing that minimal attacker winning budgets are the least fixed point.

```

lemma a_win_min_is_lfp:
  shows "extreme {s ∈ possible_pareto. (iteration s) = s} (≥) a_win_min"
proof-
  have in_pareto_leq: "∧n. (iteration ^^ n) (λg. {}) ∈ possible_pareto ∧ (iteration
^^ n) (λg. {}) ≤ a_win_min"
  proof-
    fix n
    show "(iteration ^^ n) (λg. {}) ∈ possible_pareto ∧ (iteration ^^ n) (λg. {})
≤ a_win_min"
    proof(induct n)
      case 0
      show ?case
      proof
        show "(iteration ^^ 0) (λg. {}) ∈ possible_pareto"
          using funpow.simps possible_pareto_def by auto
        have "(λg. {}) ≤ a_win_min"
          unfolding pareto_order_def by simp
        thus "(iteration ^^ 0) (λg. {}) ≤ a_win_min" using funpow.simps by simp
      qed
    next
      case (Suc n)
      have "(iteration ^^ (Suc n)) (λg. {}) = iteration ((iteration ^^ n) (λg. {}))"

        using funpow.simps by simp
      then show ?case using Suc iteration_stays_winning iteration_pareto_functor
    by simp
  qed
qed

have "extreme_bound possible_pareto (≤) {(iteration ^^ n) (λg. {}) |. n} a_win_min"
proof
  show "∧b. bound {(iteration ^^ n) (λg. {}) |. n} (≤) b ⇒ b ∈ possible_pareto
⇒ b ≥ a_win_min"
  proof-
    fix b
    assume "bound {(iteration ^^ n) (λg. {}) |. n} (≤) b" and "b ∈ possible_pareto"
    hence "∧n. (iteration ^^ n) (λg. {}) ≤ b"
      by blast
    hence "pareto_sup {(iteration ^^ n) (λg. {}) |. n} ≤ b"
      using pareto_sup_is_sup in_pareto_leq <b ∈ possible_pareto>
      using nonpos_eq_pos finite_iterations a_win_min_is_lfp_sup by auto
    thus "b ≥ a_win_min"
      using nonpos_eq_pos a_win_min_is_lfp_sup
      by simp
  qed
  show "∧x. x ∈ {(iteration ^^ n) (λg. {}) |. n} ⇒ a_win_min ≥ x"

```

```

proof-
  fix F
  assume "F ∈ {(iteration ^^ n) (λg. { }) |. n}"
  thus "a_win_min ≽ F"
    using pareto_sup_is_sup in_pareto_leq by force
qed

  show "a_win_min ∈ possible_pareto"
    by (simp add: a_win_min_in_pareto)
qed

  thus "extreme {s ∈ possible_pareto. (iteration s) = s} (≽) a_win_min"
    using kleene_lfp_iteration nonpos_eq_pos
    by (smt (verit, best) Collect_cong antisymmetry iteration_pareto_functor reflexivity
sympartp_def)
qed

end
end

```

## 5 Vectors of (extended) Naturals as Energies

```
theory Energy_Order
  imports Main List_Lemmas "HOL-Library.Extended_Nat" Well_Quasi_Orders.Well_Quasi_Orders
begin
```

We consider vectors with entries in the extended naturals as energies and fix a dimension later. In this theory we introduce the component-wise order on energies (represented as lists of enats) as well as a minimum and supremum.

```
type_synonym energy = "enat list"
```

```
definition energy_leq:: "energy  $\Rightarrow$  energy  $\Rightarrow$  bool" (infix "e $\leq$ " 80) where
  "energy_leq e e' = ((length e = length e')
     $\wedge$  ( $\forall i < \text{length } e. (e ! i) \leq (e' ! i)))$ "
```

```
abbreviation energy_l:: "energy  $\Rightarrow$  energy  $\Rightarrow$  bool" (infix "e<" 80) where
  "energy_l e e'  $\equiv$  e e $\leq$  e'  $\wedge$  e  $\neq$  e'"
```

We now establish that energy\_leq is a partial order.

```
interpretation energy_leq: ordering "energy_leq" "energy_l"
proof
  fix e e' e''
  show "e e $\leq$  e" using energy_leq_def by simp
  show "e e $\leq$  e'  $\implies$  e' e $\leq$  e''  $\implies$  e e $\leq$  e'' using energy_leq_def by fastforce
  show "e e< e' = e e< e'" by simp
  show "e e $\leq$  e'  $\implies$  e' e $\leq$  e  $\implies$  e = e'" using energy_leq_def
    by (metis (no_types, lifting) nth_equalityI order_antisym_conv)
qed
```

We now show that it is well-founded when considering a fixed dimension  $n$ . For the proof we define the subsequence of a given sequence of energies such that the last entry is increasing but never equals  $\infty$ .

```
fun subsequence_index:: "(nat  $\Rightarrow$  energy)  $\Rightarrow$  nat  $\Rightarrow$  nat" where
  "subsequence_index f 0 = (SOME x. (last (f x)  $\neq$   $\infty$ ))" |
  "subsequence_index f (Suc n) = (SOME x. (last (f x)  $\neq$   $\infty$ 
     $\wedge$  (subsequence_index f n) < x
     $\wedge$  (last (f (subsequence_index f n))  $\leq$  last (f x))))"
```

```
lemma energy_leq_wqo:
```

```
  shows "wqo_on energy_leq {e::energy. length e = n}"
proof
  show "transp_on {e. length e = n} (e $\leq$ )"
    by (metis energy_leq.trans transp_onI)
  show "almost_full_on (e $\leq$ ) {e::energy. length e = n}"
  proof(induct n)
    case 0
    then show ?case
      by (smt (verit, del_insts) almost_full_onI energy_leq.refl good_def length_0_conv
        mem_Collect_eq zero_less_Suc)
    next
    case (Suc n)
    hence allF: " $\forall f \in \text{SEQ } \{e::\text{energy}. \text{length } e = n\}. (\exists i j. i < j \wedge (f i) e \leq (f j))$ "
      unfolding almost_full_on_def good_def by simp
```

```

    have "{e::energy. length e = Suc n} = {e@[x]|e x::enat. e ∈ {e::energy. length
e = n}}"
    using length_Suc_conv_rev by auto
    show ?case
    proof
      fix f
      show "∀i. f i ∈ {e::energy. length e = Suc n} ⇒ good (e≤) f"
      proof-
        assume "∀i. f i ∈ {e::energy. length e = Suc n}"
        show "good (e≤) f" unfolding good_def proof-
          show "∃i j. i < j ∧ f i e≤ f j"
          proof(cases "finite {i::nat. (f i) ! n = ∞}")
            case True
            define upbound where "upbound = Sup {(f i) ! n | i::nat. (f i) ! n ≠
∞}"
            then show ?thesis
            proof(cases "upbound = ∞")
              case True
              have exist: "∧i. (f i) ! n ≠ ∞ ⇒ ∃j. i < j ∧ (f j) ! n ≠ ∞ ∧
(f i) ! n ≤ (f j) ! n"
              proof-
                fix i
                assume "(f i) ! n ≠ ∞"
                have "¬(∃j. i < j ∧ (f j) ! n ≠ ∞ ∧ (f i) ! n ≤ (f j) ! n) ⇒
False"
                proof-
                  assume "¬(∃j. i < j ∧ (f j) ! n ≠ ∞ ∧ (f i) ! n ≤ (f j) ! n)"
                  hence A: "∧j. i < j ⇒ (f j) ! n = ∞ ∨ (f i) ! n > (f j) !"
                  n" by auto

                define max_value where "max_value = Max {(f k) ! n | k. k ≤ i ∧
(f k) ! n ≠ ∞}"
                have "∧k. (f k) ! n ≠ ∞ ⇒ (f k) ! n ≤ max_value"
                proof-
                  fix k
                  assume not_inf: "(f k) ! n ≠ ∞"
                  show "(f k) ! n ≤ max_value"
                  proof(cases "k ≤ i")
                    case True
                    hence "(f k) ! n ∈ {(f k) ! n | k. k ≤ i ∧ (f k) ! n ≠ ∞}"
                    using not_inf by auto
                    then show ?thesis using Max_ge <(f k) ! n ∈ {(f k) ! n | k.
k ≤ i ∧ (f k) ! n ≠ ∞}> max_value_def by auto
                    next
                    case False
                    hence "(f k) ! n < (f i) ! n" using A not_inf
                    by (meson leI)
                    have "(f i) ! n ∈ {(f k) ! n | k. k ≤ i ∧ (f k) ! n ≠ ∞}"
                    using <(f i) ! n ≠ ∞> by auto
                    hence "(f i) ! n ≤ max_value" using Max_ge max_value_def by
auto
                    then show ?thesis using <(f k) ! n < (f i) ! n> by auto
                  qed
                qed
                hence "upbound = max_value" using upbound_def

```

```

    by (smc (verit) Sup_least True antisym enat_ord_code(3) mem_Collect_eq)

    have " (f i) ! n ∈ {(f k) ! n | k. k ≤ i ∧ (f k) ! n ≠ ∞}" using
<(f i) ! n ≠ ∞> by auto
    hence notempty: "{(f k) ! n | k. k ≤ i ∧ (f k) ! n ≠ ∞} ≠ {}"
by auto
    have "finite {(f k) ! n | k. k ≤ i ∧ (f k) ! n ≠ ∞}" by simp
    hence "max_value ∈ {(f k) ! n | k. k ≤ i ∧ (f k) ! n ≠ ∞}" unfolding
max_value_def using Max_in notempty by blast
    hence "max_value ≠ ∞" using max_value_def by auto
    hence "upbound ≠ ∞" using <upbound = max_value> by simp
    thus "False" using True by simp
qed
    thus "∃j. i < j ∧ (f j) ! n ≠ ∞ ∧ (f i) ! n ≤ (f j) ! n"
    by blast
qed

define f' where "f' ≡ λi. butlast (f (subsequence_index f i))"

have "f' ∈ SEQ {e::energy. length e = n}"
proof
    show "∀i. f' i ∈ {e. length e = n}"
    proof
        fix i
        have "(f (subsequence_index f i)) ∈ {e. length e = Suc n}" using
<∀i. f i ∈ {e::energy. length e = Suc n}>
        by simp
        thus "f' i ∈ {e. length e = n}"
        using f'_def by auto
    qed
qed
    hence "(∃j. i < j ∧ (f' i) e ≤ (f' j))"
    using allF by simp
    from this obtain i j where ij: "i < j ∧ (f' i) e ≤ (f' j)" by auto
    hence le: "butlast (f (subsequence_index f i)) e ≤ butlast (f (subsequence_index
f j))" using f'_def by simp

    have last: "∧x. last (f x) = (f x) ! n" using last_len
    using <∀i. f i ∈ {e. length e = Suc n}> by auto
    have "{x. (last (f x) ≠ ∞)} ≠ {}"
proof
    assume "{x. last (f x) ≠ ∞} = {}"
    hence "∧x. last (f x) = ∞" by auto
    hence "∧x. (f x) ! n = ∞" using <∧x. last (f x) = (f x) ! n> by
auto
    thus "False" using <finite {i::nat. (f i) ! n = ∞}> by simp
qed
    hence subsequence_index_0: "(last (f (subsequence_index f 0)) ≠ ∞)"

    using subsequence_index.simps(1)
    by (metis (mono_tags, lifting) Collect_empty_eq some_eq_imp)

    have subsequence_index_Suc: "∧m. (last (f (subsequence_index f (Suc
m))) ≠ ∞ ∧ (subsequence_index f m) < (subsequence_index f (Suc m)) ∧ (last (f
(subsequence_index f m)) ≤ last (f (subsequence_index f (Suc m)))))"
proof-

```



```

      fix m
      have some: "subsequence_index f (Suc m) = (SOME x. last (f x) ≠
∞ ∧ subsequence_index f m < x ∧ last (f (subsequence_index f m)) ≤ last (f x))"
using subsequence_index.simps(2) by auto
      show "(last (f (subsequence_index f (Suc m))) ≠ ∞ ∧ (subsequence_index
f m) < (subsequence_index f (Suc m)) ∧ (last (f (subsequence_index f m)) ≤ last
(f (subsequence_index f (Suc m)))))"
      proof(induct m)
      case 0
      have "{x. last (f x) ≠ ∞ ∧ subsequence_index f 0 < x ∧ last
(f (subsequence_index f 0)) ≤ last (f x)} ≠ {}"
      unfolding last using subsequence_index_0 exist
      by (simp add: last)
      then show ?case using some some_eq_ex
      by (smt (z3) empty_Collect_eq subsequence_index.simps(2))
    next
      case (Suc m)
      hence "{x. last (f x) ≠ ∞ ∧ subsequence_index f (Suc m) < x
∧ last (f (subsequence_index f (Suc m))) ≤ last (f x)} ≠ {}"
      unfolding last using exist by simp
      then show ?case using some some_eq_ex
      by (smt (z3) empty_Collect_eq subsequence_index.simps(2))
    qed
  qed
  hence "∧i j. i < j ⇒ subsequence_index f i < subsequence_index
f j"
    by (simp add: lift_Suc_mono_less)
  hence "subsequence_index f i < subsequence_index f j" using <i < j
∧ (f' i) e≤ (f' j)> by simp

  have "∧i j. i < j ⇒ last (f (subsequence_index f i)) ≤ last (f
(subsequence_index f j))"
  proof-
    fix i j
    show "i < j ⇒ last (f (subsequence_index f i)) ≤ last (f (subsequence_index
f j))"

    proof-
      assume "i < j"
      show "last (f (subsequence_index f i)) ≤ last (f (subsequence_index
f j))" using <i < j>
      proof(induct "j-i" arbitrary: i j)
      case 0
      then show ?case by simp
    next
      case (Suc x)
      then show ?case
      proof(cases "x = 0")
      case True
      hence "j = Suc i" using Suc
      by (simp add: Nat.lessE Suc_pred diff_diff_cancel)
      then show ?thesis using subsequence_index_Suc by simp
    next
      case False
      hence "∃x'. x = Suc x'"
      by (simp add: not0_implies_Suc)
      then show ?thesis using Suc subsequence_index_Suc

```

```

      by (smt (verit, ccfv_SIG) Suc_leD diff_Suc_Suc diff_diff_cancel
diff_le_self dual_order.strict_trans2 not_less_eq_eq verit_comp_simplify1(3) zero_less_diff)
      qed
    qed
  qed
  qed
  hence "(f (subsequence_index f i))!n ≤ (f (subsequence_index f j))!n"
using <i < j ∧ (f' i) e ≤ (f' j)> last by simp

  have "(f (subsequence_index f i)) e ≤ (f (subsequence_index f j))"
unfolding energy_leq_def
proof
  show "length (f (subsequence_index f i)) = length (f (subsequence_index
f j))" using <∀i. f i ∈ {e::energy. length e = Suc n}> by simp
  show "∀ia < length (f (subsequence_index f i)). f (subsequence_index
f i) ! ia ≤ f (subsequence_index f j) ! ia "
  proof
    fix ia
    show "ia < length (f (subsequence_index f i)) → f (subsequence_index
f i) ! ia ≤ f (subsequence_index f j) ! ia"
  proof
    assume "ia < length (f (subsequence_index f i))"
    hence "ia < Suc n" using <∀i. f i ∈ {e::energy. length e =
Suc n}> by simp
    show "f (subsequence_index f i) ! ia ≤ f (subsequence_index
f j) ! ia "
    proof(cases "ia < n")
      case True
      hence "f (subsequence_index f i) ! ia = (butlast (f (subsequence_index
f i))) ! ia" using nth_butlast <ia < length (f (subsequence_index f i))> <∀i. f
i ∈ {e::energy. length e = Suc n}>
      by (metis (mono_tags, lifting) SEQ_iff <f' ∈ SEQ {e. length
e = n}> f'_def mem_Collect_eq)
      also have "... ≤ (butlast (f (subsequence_index f j))) ! ia"
using le unfolding energy_leq_def using True <f' ∈ SEQ {e. length e = n}> f'_def
by simp
      also have "... = f (subsequence_index f j) ! ia" using True
nth_butlast <ia < length (f (subsequence_index f i))> <∀i. f i ∈ {e::energy. length
e = Suc n}>
      by (metis (mono_tags, lifting) SEQ_iff <f' ∈ SEQ {e. length
e = n}> f'_def mem_Collect_eq)
      finally show ?thesis .
    next
      case False
      hence "ia = n" using <ia < Suc n> by simp
      then show ?thesis using <(f (subsequence_index f i))!n ≤
(f (subsequence_index f j))!n> by simp
    qed
  qed
  qed
  then show ?thesis using <subsequence_index f i < subsequence_index
f j> by auto
next
case False
hence "∃upbound_nat. upbound = enat upbound_nat" by simp

```

```

from this obtain upbound_nat where "upbound = enat upbound_nat" by
auto

have "¬(∃x. infinite {i::nat. (f i) ! n = x}) ⇒ False "
proof-
  assume "¬(∃x. infinite {i::nat. (f i) ! n = x})"
  hence allfinite: "∧x. x ≤ upbound ⇒ finite {i::nat. (f i) ! n
= x}" by auto

  have "∧k. k ≠ ∞ ⇒ finite {n::enat. n ≤ k}"
  by (metis finite_enat_bounded mem_Collect_eq not_enat_eq)
  hence "finite ({x. x ≤ upbound} ∪ {∞}) " using False by simp
  hence "finite {{i::nat. (f i) ! n = x} | x. x ≤ upbound ∨ x = ∞}"
by simp

  hence union_finite: "finite (∪ {{i::nat. (f i) ! n = x} | x. x ≤
upbound ∨ x = ∞})" using finite_Union allfinite True by auto

  have "{i::nat. True} = (∪ {{i::nat. (f i) ! n = x} | x. x ≤ upbound
∨ x = ∞})"
  proof
    show "{i. True} ⊆ ∪ {{i. f i ! n = x} | x. x ≤ upbound ∨ x =
∞}"

    proof
      fix x
      show "x ∈ {i. True} ⇒ x ∈ ∪ {{i. f i ! n = x} | x. x ≤ upbound
∨ x = ∞}"

      proof-
        assume "x ∈ {i. True}"
        hence "x ∈ {i. f i ! n = f x ! n}" by simp
        show "x ∈ ∪ {{i. f i ! n = x} | x. x ≤ upbound ∨ x = ∞}"

        proof(cases "f x ! n = ∞")
          case True
          thus "x ∈ ∪ {{i. f i ! n = x} | x. x ≤ upbound ∨ x = ∞}"
using <x ∈ {i. f i ! n = f x ! n}>
          by auto
          next
          case False
          hence "f x ! n ≤ upbound" using upbound_def
          by (metis (mono_tags, lifting) Sup_upper mem_Collect_eq)

          thus "x ∈ ∪ {{i. f i ! n = x} | x. x ≤ upbound ∨ x = ∞}"
using <x ∈ {i. f i ! n = f x ! n}>
          by auto
          qed
          qed
          qed
          show "∪ {{i. f i ! n = x} | x. x ≤ upbound ∨ x = ∞} ⊆ {i. True}"
by simp

        qed
        thus "False" using union_finite by simp
        qed
        hence "∃x. infinite {i::nat. (f i) ! n = x}" by auto
        from this obtain x where inf_x: "infinite {i::nat. (f i) ! n = x}"
by auto

```

```

    define f' where "f' ≡ λi. butlast (f (enumerate {i::nat. (f i) !
n = x} i))"
    have "∀i. f' i ∈ {e. length e = n}"
    proof
      fix i
      have "f (enumerate {i::nat. (f i) ! n = x} i) ∈ {e. length e = Suc
n}" using <∀i. f i ∈ {e::energy. length e = Suc n}> by simp
      hence "length (f (enumerate {i::nat. (f i) ! n = x} i)) = Suc n"
    by simp
      hence "length (butlast (f (enumerate {i::nat. (f i) ! n = x} i)))
= n" using length_butlast
      by simp
      hence "butlast (f (enumerate {i::nat. (f i) ! n = x} i)) ∈ {e. length
e = n}" by simp
      thus "f' i ∈ {e. length e = n}" using f'_def by simp
    qed
    hence "f' ∈ SEQ {e::energy. length e = n}"
    unfolding SEQ_def by simp
    hence "(∃i j. i < j ∧ (f' i) e ≤ (f' j))"
    using allF by simp
    from this obtain i j where ij: "i < j ∧ (f' i) e ≤ (f' j)" by auto
    hence le: "(enumerate {i::nat. (f i) ! n = x} i) < (enumerate {i::nat.
(f i) ! n = x} j)"
      using enumerate_mono inf_x by simp
      have "(f (enumerate {i::nat. (f i) ! n = x} i)) e ≤ (f (enumerate {i::nat.
(f i) ! n = x} j))"
      unfolding energy_leq_def
    proof
      show "length (f (wellorder_class.enumerate {i. f i ! n = x} i))
=
      length (f (wellorder_class.enumerate {i. f i ! n = x} j))"

      using <∀i. f i ∈ {e::energy. length e = Suc n}> by simp
      show "∀ia < length (f (wellorder_class.enumerate {i. f i ! n = x}
i)).
      f (wellorder_class.enumerate {i. f i ! n = x} i) ! ia
      ≤ f (wellorder_class.enumerate {i. f i ! n = x} j) ! ia"

    proof
      fix ia
      show "ia < length (f (wellorder_class.enumerate {i. f i ! n =
x} i)) →
      f (wellorder_class.enumerate {i. f i ! n = x} i) ! ia
      ≤ f (wellorder_class.enumerate {i. f i ! n = x} j) ! ia"
    proof
      assume "ia < length (f (wellorder_class.enumerate {i. f i !
n = x} i))"
      hence "ia < Suc n" using <∀i. f i ∈ {e::energy. length e =
Suc n}> by simp
      show "f (wellorder_class.enumerate {i. f i ! n = x} i) ! ia
      ≤ f (wellorder_class.enumerate {i. f i ! n = x} j) ! ia"

    proof(cases "ia < n")
      case True
      hence "f (wellorder_class.enumerate {i. f i ! n = x} i) !"

```

```

ia = (f' i) ! ia" using f'_def
      by (smt (verit) SEQ_iff <f' ∈ SEQ {e. length e = n}> mem_Collect_eq
nth_butlast)
      also have "... ≤ (f' j) ! ia" using ij energy_leq_def True
<f' ∈ SEQ {e. length e = n}>
      by simp
      also have "... = f (wellorder_class.enumerate {i. f i ! n
= x} j) ! ia" using f'_def True
      by (smt (verit) SEQ_iff <f' ∈ SEQ {e. length e = n}> mem_Collect_eq
nth_butlast)
      finally show ?thesis .
next
case False
hence "ia = n" using <ia < Suc n> by simp
hence "f (wellorder_class.enumerate {i. f i ! n = x} i) !
ia = x"
      using enumerate_in_set <infinite {i::nat. (f i) ! n = x}>
      by auto
      hence "f (wellorder_class.enumerate {i. f i ! n = x} i) !
ia = f (wellorder_class.enumerate {i. f i ! n = x} j) ! ia"
      using enumerate_in_set <infinite {i::nat. (f i) ! n = x}>
<ia = n>
      by force
      then show ?thesis by simp
qed
qed
qed
qed
then show ?thesis using le by auto
qed
next
case False
define f' where "f' ≡ λi. butlast (f (enumerate {i::nat. (f i) ! n
= ∞} i))"
have "∀i. f' i ∈ {e. length e = n}"
proof
fix i
have "f (enumerate {i::nat. (f i) ! n = ∞} i) ∈ {e. length e = Suc
n}" using <∀i. f i ∈ {e::energy. length e = Suc n}> by simp
hence "length (f (enumerate {i::nat. (f i) ! n = ∞} i)) = Suc n"
by simp
hence "length (butlast (f (enumerate {i::nat. (f i) ! n = ∞} i)))
= n" using length_butlast
by simp
hence "butlast (f (enumerate {i::nat. (f i) ! n = ∞} i)) ∈ {e. length
e = n}" by simp
thus "f' i ∈ {e. length e = n}" using f'_def by simp
qed
hence "f' ∈ SEQ {e::energy. length e = n}"
unfolding SEQ_def by simp
hence "(∃i j. i < j ∧ (f' i) e ≤ (f' j))"
using allF by simp
from this obtain i j where ij: "i < j ∧ (f' i) e ≤ (f' j)" by auto
hence le: "(enumerate {i::nat. (f i) ! n = ∞} i) < (enumerate {i::nat.
(f i) ! n = ∞} j)"
using enumerate_mono False by simp

```

```

      have "(f (enumerate {i::nat. (f i) ! n = ∞} i)) e ≤ (f (enumerate {i::nat.
(f i) ! n = ∞} j))"
      unfolding energy_leq_def
    proof
      show "length (f (wellorder_class.enumerate {i. f i ! n = ∞} i))
=
      length (f (wellorder_class.enumerate {i. f i ! n = ∞} j))"

      using <∀i. f i ∈ {e::energy. length e = Suc n}> by simp
    show "∀ia<length (f (wellorder_class.enumerate {i. f i ! n = ∞} i)).
      f (wellorder_class.enumerate {i. f i ! n = ∞} i) ! ia
      ≤ f (wellorder_class.enumerate {i. f i ! n = ∞} j) ! ia "

    proof
      fix ia
      show "ia < length (f (wellorder_class.enumerate {i. f i ! n = ∞}
i)) →
      f (wellorder_class.enumerate {i. f i ! n = ∞} i) ! ia
      ≤ f (wellorder_class.enumerate {i. f i ! n = ∞} j) ! ia"

    proof
      assume "ia < length (f (wellorder_class.enumerate {i. f i ! n
= ∞} i))"
      hence "ia < Suc n" using <∀i. f i ∈ {e::energy. length e = Suc
n}> by simp
      show "f (wellorder_class.enumerate {i. f i ! n = ∞} i) ! ia
      ≤ f (wellorder_class.enumerate {i. f i ! n = ∞} j) ! ia"

      proof(cases "ia < n")
        case True
        hence "f (wellorder_class.enumerate {i. f i ! n = ∞} i) !
ia = (f' i) ! ia" using f'_def
        by (smt (verit) SEQ_iff <f' ∈ SEQ {e. length e = n}> mem_Collect_eq
nth_butlast)
        also have "... ≤ (f' j) ! ia" using ij energy_leq_def True <f'
∈ SEQ {e. length e = n}>
        by simp
        also have "... = f (wellorder_class.enumerate {i. f i ! n =
∞} j) ! ia" using f'_def True
        by (smt (verit) SEQ_iff <f' ∈ SEQ {e. length e = n}> mem_Collect_eq
nth_butlast)
        finally show ?thesis .
      next
        case False
        hence "ia = n" using <ia < Suc n> by simp
        hence "f (wellorder_class.enumerate {i. f i ! n = ∞} i) ! ia
= ∞"
        using enumerate_in_set <infinite {i::nat. (f i) ! n = ∞}>
        by auto
        hence "f (wellorder_class.enumerate {i. f i ! n = ∞} i) ! ia
= f (wellorder_class.enumerate {i. f i ! n = ∞} j) ! ia"
        using enumerate_in_set <infinite {i::nat. (f i) ! n = ∞}>
        <ia = n>
        by force
        then show ?thesis by simp
      qed

```

```

      qed
    qed
  qed
  thus "∃ i j. i < j ∧ (f i) e ≤ (f j)" using le by auto
  qed
  qed
  qed
  qed
  qed
  qed

```

## Minimum

```

definition energy_Min:: "energy set ⇒ energy set" where
  "energy_Min A = {e ∈ A . ∀ e' ∈ A. e ≠ e' ⟶ ¬ (e' e ≤ e)}"

```

We now observe that the minimum of a non-empty set is not empty. Further, each element  $a \in A$  has a lower bound in  $\text{energy\_Min } A$ .

```

lemma energy_Min_not_empty:
  assumes "A ≠ {}" and "∧ e. e ∈ A ⟹ length e = n"
  shows "energy_Min A ≠ {}"
using assms proof(induction n arbitrary: A)
  case 0
  hence "{[]} = A" using assms by auto
  hence "[] ∈ energy_Min A" using energy_Min_def by auto
  then show ?case by auto
next
  case (Suc n)
  have "{butlast a | a. a ∈ A} ≠ {}" using Suc(2) by simp
  have "∧ a. a ∈ {butlast a | a. a ∈ A} ⟹ length a = n" using Suc(3) by auto
  hence "energy_Min {butlast a | a. a ∈ A} ≠ {}" using <{butlast a | a. a ∈ A} ≠ {}>
  Suc(1)
  by meson
  hence "∃ x. x ∈ energy_Min {butlast a | a. a ∈ A}" by auto
  from this obtain x where "x ∈ energy_Min {butlast a | a. a ∈ A}" by auto
  hence "x ∈ {butlast a | a. a ∈ A}" using energy_Min_def by auto
  hence "∃ a. a ∈ A ∧ x = butlast a" by auto
  from this obtain a where "a ∈ A ∧ x = butlast a" by auto

  have "last a ∈ {x. (butlast a)@[x] ∈ A}"
  by (metis Suc.prem1(2) Zero_neq_Suc <a ∈ A ∧ x = butlast a> append_butlast_last_id
list.size(3) mem_Collect_eq)
  hence "{x. (butlast a)@[x] ∈ A} ≠ {}" by auto
  have "∃ B. finite B ∧ B ⊆ {x. (butlast a)@[x] ∈ A} ∧ Inf {x. (butlast a)@[x] ∈
A} = Min B"
  proof(cases "Inf {x. butlast a @ [x] ∈ A} = ∞")
    case True
    hence "∞ ∈ {x. (butlast a)@[x] ∈ A}" using <{x. (butlast a)@[x] ∈ A} ≠ {}>
    by (metis <last a ∈ {x. butlast a @ [x] ∈ A}> wellorder_InfI)
    hence "finite {∞} ∧ {∞} ⊆ {x. (butlast a)@[x] ∈ A} ∧ Inf {x. (butlast a)@[x]
∈ A} = Min {∞}"
    by (simp add: True)
    then show ?thesis by blast
  next
    case False

```

```

    hence "∃m. (enat m) ∈ {x. butlast a @ [x] ∈ A}"
    by (metis Inf_top_conv(2) Succ_def <a ∈ A ∧ x = butlast a> not_infinity_eq
top_enat_def)
    from this obtain m where "(enat m) ∈ {x. butlast a @ [x] ∈ A}" by auto
    hence finite: "finite {x. (butlast a)@[x] ∈ A ∧ x ≤ enat m}"
    by (metis (no_types, lifting) finite_enat_bounded mem_Collect_eq)
    have subset: "{x. (butlast a)@[x] ∈ A ∧ x ≤ enat m} ⊆ {x. (butlast a)@[x] ∈
A}" by (simp add: Collect_mono)
    have "Inf {x. (butlast a)@[x] ∈ A} = Inf {x. (butlast a)@[x] ∈ A ∧ x ≤ enat
m}" using <(enat m) ∈ {x. butlast a @ [x] ∈ A}>
    by (smt (verit) Inf_lower mem_Collect_eq nle_le wellorder_InfI)
    hence "Inf {x. (butlast a)@[x] ∈ A} = Min {x. (butlast a)@[x] ∈ A ∧ x ≤ enat
m}" using <(enat m) ∈ {x. butlast a @ [x] ∈ A}>
    using finite
    by (smt (verit, best) False Inf_enat_def Min_Inf)
    hence "finite {x. (butlast a)@[x] ∈ A ∧ x ≤ enat m} ∧ {x. (butlast a)@[x] ∈
A ∧ x ≤ enat m} ⊆ {x. (butlast a)@[x] ∈ A} ∧ Inf {x. (butlast a)@[x] ∈ A} = Min
{x. (butlast a)@[x] ∈ A ∧ x ≤ enat m}"
    using finite subset by simp
    then show ?thesis by blast
qed
from this obtain B where B: "finite B ∧ B ⊆ {x. (butlast a)@[x] ∈ A} ∧ Inf {x.
(butlast a)@[x] ∈ A} = Min B" by auto
hence "((butlast a)@[Min B]) ∈ A"
by (metis <last a ∈ {x. butlast a @ [x] ∈ A}> mem_Collect_eq wellorder_InfI)

have "∀b ∈ A. ((butlast a)@[Min B]) ≠ b ⟶ ¬ (energy_leq b ((butlast a)@[Min B]))"

proof
  fix b
  assume "b ∈ A"
  have "energy_leq b (butlast a @ [Min B]) ⟹ butlast a @ [Min B] = b"
  proof-
    assume "energy_leq b (butlast a @ [Min B])"
    have "energy_leq (butlast b) (butlast a)"
    unfolding energy_leq_def proof
      show "length (butlast b) = length (butlast a)"
      using <∧a. a ∈ {butlast a | a. a ∈ A} ⟹ length a = n> <a ∈ A ∧ x =
butlast a> <b ∈ A> mem_Collect_eq by blast
      show "∀i < length (butlast b). butlast b ! i ≤ butlast a ! i"
      proof
        fix i
        show "i < length (butlast b) ⟹ butlast b ! i ≤ butlast a ! i"
        proof
          assume "i < length (butlast b)"
          hence "i < length b"
          by (simp add: Suc.prems(2) <b ∈ A>)
          hence B: "b ! i ≤ (butlast a @ [Min B]) ! i" using <energy_leq b (butlast
a @ [Min B])> energy_leq_def by auto

          have "butlast b ! i = b ! i" using <i < length (butlast b)> nth_butlast
by auto
          have "butlast a ! i = (butlast a @ [Min B]) ! i"
          by (metis <i < length (butlast b)> <length (butlast b) = length (butlast
a)> butlast_snoc nth_butlast)

```



```

      thus "butlast b ! i ≤ butlast a ! i " using B <butlast b ! i = b! i>
by auto
  qed
  qed
  qed
  hence "butlast b = butlast a" using <x ∈ energy_Min {butlast a | a. a ∈ A}>
<a ∈ A ∧ x = butlast a> energy_Min_def <b ∈ A> by auto
  hence "(butlast a)@[last b] ∈ A" using Suc(3)
  by (metis <b ∈ A> append_butlast_last_id list.size(3) nat.discI)
  hence "Min B ≤ last b"
  by (metis (no_types, lifting) B Inf_lower mem_Collect_eq)

  have "last b ≤ Min B" using <energy_leq b (butlast a @ [Min B])> energy_leq_def
  by (metis (no_types, lifting) <butlast b = butlast a> append_butlast_last_id
butlast.simps(1) dual_order.refl impossible_Cons length_Cons length_append_singleton
lessI nth_append_length)
  hence "last b = Min B" using <Min B ≤ last b> by simp
  thus "butlast a @ [Min B] = b" using <butlast b = butlast a> Suc(3)
  by (metis Zero_not_Suc <b ∈ A> append_butlast_last_id list.size(3))
  qed
  thus "butlast a @ [Min B] ≠ b → ¬ energy_leq b (butlast a @ [Min B])"
  by auto
  qed
  hence "((butlast a)@[Min B]) ∈ energy_Min A" using energy_Min_def <((butlast
a)@[Min B]) ∈ A>
  by simp
  thus ?case by auto
  qed

lemma energy_Min_contains_smaller:
  assumes "a ∈ A"
  shows "∃ b ∈ energy_Min A. b ≤ a"
proof-
  define set where "set ≡ {e. e ∈ A ∧ e ≤ a}"
  hence "a ∈ set"
  by (simp add: assms(1) energy_leq.refl)
  hence "set ≠ {}" by auto
  have "∧ s. s ∈ set ⇒ length s = length a" using energy_leq_def set_def
  by simp
  hence "energy_Min set ≠ {}" using <set ≠ {}> energy_Min_not_empty by simp
  hence "∃ b. b ∈ energy_Min set" by auto
  from this obtain b where "b ∈ energy_Min set" by auto
  hence "∧ b'. b' ∈ A ⇒ b' ≠ b ⇒ ¬ (b' ≤ b)"
  proof-
    fix b'
    assume "b' ∈ A"
    assume "b' ≠ b"
    show "¬ (b' ≤ b)"
  proof
    assume "(b' ≤ b)"
    hence "b' ≤ a" using <b ∈ energy_Min set> energy_Min_def
    by (simp add: energy_leq.trans local.set_def)
    hence "b' ∈ set" using <b' ∈ A> set_def by simp
    thus "False" using <b ∈ energy_Min set> energy_Min_def <b' ≤ b> <b' ≠
b> by auto
  qed
  qed

```

```

qed
hence "b ∈ energy_Min A" using energy_Min_def
  using <b ∈ energy_Min set> local.set_def by auto
thus ?thesis using <b ∈ energy_Min set> energy_Min_def set_def by auto
qed

```

We now establish how the minimum relates to subsets.

```

lemma energy_Min_subset:
  assumes "A ⊆ B"
  shows "A ∩ (energy_Min B) ⊆ energy_Min A" and
    "energy_Min B ⊆ A ⇒ energy_Min B = energy_Min A"
proof-
  show "A ∩ energy_Min B ⊆ energy_Min A"
  proof
    fix e
    assume "e ∈ A ∩ energy_Min B"
    hence "∃ a ∈ energy_Min A. a ≤ e" using assms energy_Min_contains_smaller by
blast
    from this obtain a where "a ∈ energy_Min A" and "a ≤ e" by auto
    hence "a = e" using <e ∈ A ∩ energy_Min B> unfolding energy_Min_def
      using assms by auto
    thus "e ∈ energy_Min A" using <a ∈ energy_Min A> by simp
  qed
  assume "energy_Min B ⊆ A"
  hence "energy_Min B ⊆ energy_Min A" using <A ∩ energy_Min B ⊆ energy_Min A> by
auto
  have "energy_Min A ⊆ energy_Min B"
  proof
    fix a
    assume "a ∈ energy_Min A"
    hence "a ∈ B" unfolding energy_Min_def using assms by blast
    hence "∃ b ∈ energy_Min B. b ≤ a" using assms energy_Min_contains_smaller by
blast
    from this obtain b where "b ∈ energy_Min B" and "b ≤ a" by auto
    hence "a = b" using <energy_Min B ⊆ A> energy_Min_def
      using <a ∈ energy_Min A> by auto
    thus "a ∈ energy_Min B"
      using <b ∈ energy_Min B> by simp
  qed
  thus "energy_Min B = energy_Min A" using <energy_Min B ⊆ energy_Min A> by simp
qed

```

We now show that by well-foundedness the minimum is a finite set. For the proof we first generalise enumerate.

```

fun enumerate_arbitrary :: "'a set ⇒ nat ⇒ 'a" where
  "enumerate_arbitrary A 0 = (SOME a. a ∈ A)" |
  "enumerate_arbitrary A (Suc n)
    = enumerate_arbitrary (A - {enumerate_arbitrary A 0}) n"

lemma enumerate_arbitrary_in:
  shows "infinite A ⇒ enumerate_arbitrary A i ∈ A"
proof(induct i arbitrary: A)
  case 0
  then show ?case using enumerate_arbitrary.simps finite.simps some_in_eq by auto
next

```

```

    case (Suc i)
    hence "infinite (A - {enumerate_arbitrary A 0})" using infinite_remove by simp
    hence "Energy_Order.enumerate_arbitrary (A - {enumerate_arbitrary A 0}) i ∈ (A
- {enumerate_arbitrary A 0})" using Suc.hyps by blast
    hence "enumerate_arbitrary A (Suc i) ∈ (A - {enumerate_arbitrary A 0})" using
enumerate_arbitrary.simps by simp
    then show ?case by auto
qed

lemma enumerate_arbitrary_neq:
  shows "infinite A  $\implies$  i < j
 $\implies$  enumerate_arbitrary A i  $\neq$  enumerate_arbitrary A j"
proof(induct i arbitrary: j A)
  case 0
  then show ?case using enumerate_arbitrary.simps
  by (metis Diff_empty Diff_iff enumerate_arbitrary_in finite_Diff_insert gr0_implies_Suc
insert_iff)
next
  case (Suc i)
  hence " $\exists j'. j = \text{Suc } j'$ "
  by (simp add: not0_implies_Suc)
  from this obtain j' where "j = Suc j'" by auto
  hence "i < j'" using Suc by simp
  from Suc have "infinite (A - {enumerate_arbitrary A 0})" using infinite_remove
by simp
  hence "enumerate_arbitrary (A - {enumerate_arbitrary A 0}) i  $\neq$  enumerate_arbitrary
(A - {enumerate_arbitrary A 0}) j'" using Suc <i < j'>
  by force
  then show ?case using enumerate_arbitrary.simps
  by (simp add: <j = Suc j'>)
qed

lemma energy_Min_finite:
  assumes " $\bigwedge e. e \in A \implies \text{length } e = n$ "
  shows "finite (energy_Min A)"
proof-
  have "wqo_on energy_leq (energy_Min A)" using energy_leq_wqo assms
  by (smt (verit, del_insts) Collect_mono_iff energy_Min_def wqo_on_subset)
  hence wqoMin: " $(\forall f \in \text{SEQ } (\text{energy\_Min } A). (\exists i j. i < j \wedge \text{energy\_leq } (f \ i) (f \ j)))$ "
unfolding wqo_on_def almost_full_on_def good_def by simp
  have " $\neg \text{finite } (\text{energy\_Min } A) \implies \text{False}$ "
  proof-
    assume " $\neg \text{finite } (\text{energy\_Min } A)$ "
    hence "infinite (energy_Min A)"
    by simp

    define f where "f  $\equiv$  enumerate_arbitrary (energy_Min A)"
    have fneq: " $\bigwedge i j. f \ i \in \text{energy\_Min } A \wedge (j \neq i \longrightarrow f \ j \neq f \ i)$ "
    proof
      fix i j
      show "f i  $\in$  energy_Min A" unfolding f_def using enumerate_arbitrary_in <infinite
(energy_Min A)> by auto
      show "j  $\neq$  i  $\longrightarrow$  f j  $\neq$  f i" proof
        assume "j  $\neq$  i"
        show "f j  $\neq$  f i" proof(cases "j < i")
          case True

```

```

      then show ?thesis unfolding f_def using enumerate_arbitrary_neq <infinite
    (energy_Min A) > by auto
  next
    case False
    hence "i < j" using <j ≠ i> by auto
    then show ?thesis unfolding f_def using enumerate_arbitrary_neq <infinite
  (energy_Min A) >
    by metis
  qed
qed
qed
hence "∃i j. i < j ∧ energy_leq (f i) (f j)" using wqoMin SEQ_def by simp
thus "False" using energy_Min_def fneq by force
qed
thus ?thesis by auto
qed

```

## Supremum

**definition** energy\_sup :: "nat ⇒ energy set ⇒ energy" **where**  
 "energy\_sup n A = map (λi. Sup {(e!i) | e. e ∈ A}) [0.. $n$ ]"

We now show that we indeed defined a supremum, i.e. a least upper bound, when considering a fixed dimension  $n$ .

```

lemma energy_sup_is_sup:
  shows energy_sup_in: "∧a. a ∈ A ⇒ length a = n ⇒ a e≤ (energy_sup n A)" and
    energy_sup_leq: "∧s. (∧a. a ∈ A ⇒ a e≤ s) ⇒ length s = n
      ⇒ (energy_sup n A) e≤ s"

proof-
  fix a
  assume A1: "a ∈ A" and A2: "length a = n"
  show "a e≤ (energy_sup n A)"
    unfolding energy_leq_def energy_sup_def
  proof
    show "length a = length (map (λi. Sup {(v!i) | v. v ∈ A}) [0.. $n$ ])" using A2
    by simp
    show "∀i < length a. a ! i ≤ map (λi. Sup {(v!i) | v. v ∈ A}) [0.. $n$ ] ! i"
    proof
      fix i
      show "i < length a → a ! i ≤ map (λi. Sup {(v!i) | v. v ∈ A}) [0.. $n$ ] ! i"
    "
      proof
        assume "i < length a"
        thus "a ! i ≤ map (λi. Sup {(v!i) | v. v ∈ A}) [0.. $n$ ] ! i" using A1 A2
        by (smt (verit, del_insts) Sup_upper diff_add_inverse length_upt mem_Collect_eq
          minus_nat.diff_0 nth_map nth_upt)
      qed
    qed
  qed
next
  fix x
  assume A1: "∧a. a ∈ A ⇒ a e≤ x" and A2: "length x = n"
  show "(energy_sup n A) e≤ x"
    unfolding energy_leq_def
  proof
    show L: "length (energy_sup n A) = length x" using A2 energy_sup_def by simp
  
```

```

show "∀i<length (energy_sup n A). energy_sup n A ! i ≤ x ! i "
proof
  fix i
  show "i < length (energy_sup n A) → energy_sup n A ! i ≤ x ! i "
  proof
    assume "i < length (energy_sup n A)"
    hence "i < length [0.. $n$ ]" using L A2 by simp
    from A1 have "∧a. a∈{v ! i | v. v ∈ A} ⇒ a ≤ x ! i"
    proof-
      fix a
      assume "a∈{v ! i | v. v ∈ A} "
      hence "∃v∈A. a = v ! i" by auto
      from this obtain v where "v∈ A" and "a=v ! i" by auto
      thus " a ≤ x ! i" using A1 energy_leq_def L <i < length (energy_sup n
A)> by simp
    qed
  qed

  have "(energy_sup n A) ! i = (map (λi. Sup {(v!i)|v. v ∈ A}) [0.. $n$ ] ! i)
" using energy_sup_def by auto
  also have "... = (λi. Sup {(v!i)|v. v ∈ A}) ([0.. $n$ ] ! i)" using nth_map
<i < length [0.. $n$ ]>
  by auto
  also have "... = Sup {v ! i | v. v ∈ A}"
  using <i < length [0.. $n$ ]> by auto
  also have "... ≤ (x ! i) " using <∧a. a∈{v ! i | v. v ∈ A} ⇒ a ≤ x ! i>
  by (meson Sup_least)
  finally show "energy_sup n A ! i ≤ x ! i " .
qed
qed
qed
qed

```

We now observe a version of monotonicity. Afterwards we show that the supremum of the empty set is the zero-vector.

```

lemma energy_sup_leq_energy_sup:
  assumes "A ≠ {}" and "∧a. a ∈ A ⇒ ∃b ∈ B. energy_leq a b" and
    "∧a. a ∈ A ⇒ length a = n"
  shows "energy_leq (energy_sup n A) (energy_sup n B)"
proof-
  have len: "length (energy_sup n B) = n" using energy_sup_def by simp
  have "∧a. a ∈ A ⇒ energy_leq a (energy_sup n B)"
  proof-
    fix a
    assume "a ∈ A"
    hence "∃b ∈ B. energy_leq a b" using assms by simp
    from this obtain b where "b ∈ B" and "energy_leq a b" by auto
    hence "energy_leq b (energy_sup n B)" using energy_sup_in energy_leq_def
      by (simp add: <a ∈ A> assms(3))
    thus "energy_leq a (energy_sup n B)" using <energy_leq a b> energy_leq.trans
  by blast
  qed
  thus ?thesis using len energy_sup_leq by blast
qed

```

```

lemma empty_Sup_is_zero:
  assumes "i < n"

```

```

    shows "(energy_sup n {}) ! i = 0"
  proof-
    have "(energy_sup n {}) ! i = (map (λi. Sup {(v!i)|v. v ∈ {}}) [0..<n]) ! i"
      using energy_sup_def by auto
    also have "... = (λi. Sup {(v!i)|v. v ∈ {}}) ([0..<n] ! i)" using nth_map assms
  by simp
  finally show "(energy_sup n {}) ! i = 0"
    by (simp add: bot_enat_def)
qed

end

```

## 6 Bisping's Updates

```
theory Update
  imports Energy_Order
begin
```

In this theory we define a superset of Bisping's updates and their application. Further, we introduce Bisping's "inversion" of updates and relate the two.

### 6.1 Bisping's Updates

Bisping allows three ways of updating a component of an energy: `zero` does not change the respective entry, `minus_one` subtracts one and `min_set A` for some set  $A$  replaces the entry by the minimum of entries whose index is contained in  $A$ . We further add `plus_one` to add one and omit the assumption that the a minimum has to consider the component it replaces. Updates are vectors where each entry contains the information, how the update changes the respective component of energies. We now introduce a datatype such that updates can be represented as lists of `update_components`.

```
datatype update_component = zero | minus_one | min_set "nat set" | plus_one
type_synonym update = "update_component list"
```

```
abbreviation "valid_update u  $\equiv$  ( $\forall i$  D. u ! i = min_set D
 $\longrightarrow$  D  $\neq$  {}  $\wedge$  D  $\subseteq$  {x. x < length u})"
```

Now the application of updates `apply_update` will be defined.

```
fun apply_component::"nat  $\Rightarrow$  update_component  $\Rightarrow$  energy  $\Rightarrow$  enat option" where
  "apply_component i zero e = Some (e ! i)" |
  "apply_component i minus_one e = (if ((e ! i) > 0) then Some ((e ! i) - 1)
    else None)" |
  "apply_component i (min_set A) e = Some (min_list (nth_s e A))"|
  "apply_component i plus_one e = Some ((e ! i)+1)"
```

```
fun apply_update:: "update  $\Rightarrow$  energy  $\Rightarrow$  energy option" where
  "apply_update u e = (if (length u = length e)
    then (those (map ( $\lambda i$ . apply_component i (u ! i) e) [0..

```

```
abbreviation "upd u e  $\equiv$  the (apply_update u e)"
```

We now observe some properties of updates and their application. In particular, the application of an update preserves the dimension and the domain of an update is upward closed.

```
lemma len_appl:
  assumes "apply_update u e  $\neq$  None"
  shows "length (upd u e) = length e"
proof -
  from assms have "apply_update u e = those (map ( $\lambda n$ . apply_component n (u ! n)
e) [0..

```

```
lemma apply_to_comp_n:
  assumes "apply_update u e  $\neq$  None" and "i < length e"
```

```

    shows "(upd u e) ! i = the (apply_component i (u ! i) e)"
proof-
  have "(the (apply_update u e)) ! i = (the (those (map (λn. apply_component n (u ! n) e) [0..

```

Now we show that all valid updates are monotonic. The proof follows directly from the definition of `apply_update` and `valid_update`.

```

lemma updates_monotonic:
  assumes "apply_update u e ≠ None" and "e ≤ e'" and "valid_update u"
  shows "(upd u e) ≤ (upd u e')"
  unfolding energy_leq_def proof
  have "apply_update u e' ≠ None" using assms upd_domain_upward_closed by auto

```



```

    thus "length (the (apply_update u e)) = length (the (apply_update u e'))" using
  assms len_appl
    by (metis energy_leq_def)
    show "∀n<length (the (apply_update u e)). the (apply_update u e) ! n ≤ the (apply_update
  u e') ! n "
    proof
      fix n
      show "n < length (the (apply_update u e)) → the (apply_update u e) ! n ≤ the
  (apply_update u e') ! n"
    proof
      assume "n < length (the (apply_update u e))"
      hence "n < length e" using len_appl assms(1)
      by simp
      hence "e ! n ≤ e' ! n" using assms energy_leq_def
      by simp
      consider (zero) "(u ! n) = zero" | (minus_one) "(u ! n) = minus_one" | (min_set)
  "(∃A. (u ! n) = min_set A)" | (plus_one) "(u ! n) = plus_one"
      using update_component.exhaust by auto
      thus "the (apply_update u e) ! n ≤ the (apply_update u e') ! n"
    proof (cases)
      case zero
      then show ?thesis using apply_update.simps apply_component.simps assms <e
  ! n ≤ e' ! n> <apply_update u e' ≠ None>
      by (metis <n < length (the (apply_update u e))> apply_to_comp_n len_appl
  option.sel)
      next
      case minus_one
      hence "the (apply_update u e) ! n = the (apply_component n (u ! n) e)" using
  apply_to_comp_n assms(1)
      by (simp add: <n < length e>)

      from assms(1) have A: "(map (λn. apply_component n (u ! n) e) [0..

```

```

    from this obtain A where "u ! n = min_set A" by auto
    hence "A ⊆ {x. x < length e}" using assms(3) by (metis apply_update.elims
assms(1))
    hence "∀a ∈ A. e!a ≤ e'!a" using assms(2) energy_leq_def
    by blast
    have "∀a ∈ A. (Min (set (nth e A))) ≤ e! a" proof
    fix a
    assume "a ∈ A"
    hence "e!a ∈ set (nth e A)" using set_nth nth_def
    using <A ⊆ {x. x < length e}> in_mono by fastforce
    thus "Min (set (nth e A)) ≤ e! a" using Min_le by simp
qed
    hence "∀a ∈ A. (Min (set (nth e A))) ≤ e'! a" using <∀a ∈ A. e!a ≤ e'!a>
    using dual_order.trans by blast
    hence "∀x ∈ (set (nth e' A)). (Min (set (nth e A))) ≤ x" using set_nth
    by (smt (verit) mem_Collect_eq)

    from assms(2) have "A ≠ {}"
    using <u ! n = min_set A> assms(3) by auto
    have "A ⊆ {x. x < length e}" using <A ⊆ {x. x < length e}> assms
    using energy_leq_def by auto
    hence "set (nth e' A) ≠ {}" using <A ≠ {}> set_nth
    by (smt (verit, best) Collect_empty_eq Collect_mem_eq Collect_mono_iff)

    hence "(nth e' A) ≠ []" by simp
    from <A ≠ {}> have "set (nth e A) ≠ {}" using set_nth <A ⊆ {x. x < length
e}> Collect_empty_eq <n < length e> <u ! n = min_set A>
    by (smt (verit, best) <set (nth e' A) ≠ {}> assms(2) energy_leq_def)
    hence "(nth e A) ≠ []" by simp
    hence "(min_list (nth e A)) = Min (set (nth e A))" using min_list_Min
by auto
    also have "... ≤ Min (set (nth e' A))"
    using <∀x ∈ (set (nth e' A)). (Min (set (nth e A))) ≤ x>
    by (simp add: <nth e' A ≠ []>)
    finally have "(min_list (nth e A)) ≤ min_list (nth e' A)" using min_list_Min
<(nth e' A) ≠ []> by metis
    then show ?thesis using apply_to_comp_n assms(1) <apply_update u e' ≠ None>
apply_component.simps(3) <u ! n = min_set A>
    by (metis <length (the (apply_update u e)) = length (the (apply_update
u e'))> <n < length e> len_appl option.sel)
  next
    case plus_one
    have "upd u e ! n = the (apply_component n (u ! n) e)" using apply_to_comp_n
<n < length e> assms(1) by auto
    also have "... = (e!n) + 1" using apply_component.elims plus_one
    by simp
    also have "... ≤ (e'!n) + 1"
    using <e ! n ≤ e' ! n> by auto
    also have "... = upd u e' ! n" using apply_to_comp_n <n < length e> assms
apply_component.elims plus_one
    by (metis <apply_update u e' ≠ None> apply_component.simps(4) energy_leq_def
option.sel)
    finally show ?thesis by simp
  qed
qed
qed
qed

```

qed

## 6.2 Bisping's Inversion

The “inverse” of an update  $u$  is a function mapping energies  $e$  to  $\min\{e' \mid e \leq u(e')\}$  w.r.t the component-wise order. We start by giving a calculation and later show that we indeed calculate such minima. For an energy  $e = (e_0, \dots, e_{n-1})$  we calculate this component-wise such that the  $i$ -th component is the maximum of  $e_i$  (plus or minus one if applicable) and each entry  $e_j$  where  $i \in u_j \subseteq \{0, \dots, n-1\}$ . Note that this generalises the inversion proposed by Bisping [1].

```
fun apply_inv_component::"nat  $\Rightarrow$  update  $\Rightarrow$  energy  $\Rightarrow$  enat" where
  "apply_inv_component i u e = Max (set (map ( $\lambda$ (j,up).
    (case up of zero  $\Rightarrow$  (if i=j then (e ! i) else 0) |
      minus_one  $\Rightarrow$  (if i=j then (e ! i)+1 else 0) |
      min_set A  $\Rightarrow$  (if i $\in$ A then (e ! j) else 0) |
      plus_one  $\Rightarrow$  (if i=j then (e ! i)-1 else 0)))
    (List.enumerate 0 u)))"
```

```
fun apply_inv_update:: "update  $\Rightarrow$  energy  $\Rightarrow$  energy option" where
  "apply_inv_update u e = (if (length u = length e)
    then Some (map ( $\lambda$ i. apply_inv_component i u e) [0.. $\text{length } e$ ])
    else None)"
```

```
abbreviation "inv_upd u e  $\equiv$  the (apply_inv_update u e)"
```

We now observe the following properties, if an update  $u$  and an energy  $e$  have the same dimension:

- `apply_inv_update` preserves dimension.
- The domain of `apply_inv_update u` is  $\{e \mid |e| = |u|\}$ .
- `apply_inv_update u e` is in the domain of the update  $u$ .

The first two proofs follow directly from the definition of `apply_inv_update`, while the proof of `inv_not_none_then` is done by a case analysis of the possible `update_components`.

```
lemma len_inv_appl:
  assumes "length u = length e"
  shows "length (inv_upd u e) = length e"
  using assms apply_inv_update.simps length_map option.sel by auto
```

```
lemma inv_not_none:
  assumes "length u = length e"
  shows "apply_inv_update u e  $\neq$  None"
  using assms apply_inv_update.simps by simp
```

```
lemma inv_not_none_then:
  assumes "apply_inv_update u e  $\neq$  None"
  shows "(apply_update u (inv_upd u e))  $\neq$  None"
proof -
  have len: "length u = length (the (apply_inv_update u e))" using assms apply_inv_update.simps
  len_those
  by auto
  have " $\forall n < \text{length } u. (\text{apply\_component } n (u ! n) (\text{the } (\text{apply\_inv\_update } u e))) \neq \text{None}$ "
```

```

proof
  fix n
  show "n < length u → apply_component n (u ! n) (the (apply_inv_update u e))
≠ None "
proof
  assume "n < length u"
  consider (zero) "(u ! n) = zero" | (minus_one) "(u ! n) = minus_one" | (min_set)
"(∃ A. (u ! n) = min_set A)" | (plus_one) "(u ! n) = plus_one"
  using update_component.exhaust by auto
  then show "apply_component n (u ! n) (the (apply_inv_update u e)) ≠ None"

proof(cases)
  case zero
  then show ?thesis by simp
next
  case minus_one
  have nth: "(the (apply_inv_update u e)) ! n = apply_inv_component n u e"
using apply_inv_update.simps
  by (metis (no_types, lifting) <n < length u> add_0 assms len length_map
nth_map nth_upt option.sel)

  have n_minus_one: "List.enumerate 0 u ! n = (n, minus_one) " using minus_one
  by (simp add: <n < length u> nth_enumerate_eq)
  have "(λ(m, up). (case up of
    zero ⇒ (if n=m then (nth e n) else 0) |
    minus_one ⇒ (if n=m then (nth e n)+1 else 0) |
    min_set A ⇒ (if n∈A then (nth e m) else 0))) (n, minus_one) = (e
! n) +1"
  by simp
  hence "(e ! n) +1 ∈ set (map (λ(m, up). (case up of
    zero ⇒ (if n=m then (nth e n) else 0) |
    minus_one ⇒ (if n=m then (nth e n)+1 else 0) |
    min_set A ⇒ (if n∈A then (nth e m) else 0) |
    plus_one ⇒ (if n=m then (nth e n)-1 else 0))) (List.enumerate 0 u)))"
using n_minus_one
  by (metis (no_types, lifting) <n < length u> case_prod_conv length_enumerate
length_map nth_map nth_mem update_component.simps(15))
  hence "(nth e n)+1 ≤ apply_inv_component n u e" using minus_one nth apply_inv_component
Max_ge
  by simp
  hence "(nth (the (apply_inv_update u e)) n > 0)" using nth by fastforce
  then show ?thesis by (simp add: minus_one)
next
  case min_set
  then show ?thesis by auto
next
  case plus_one
  then show ?thesis by simp
qed
qed
qed
hence "∀ n < length (the (apply_inv_update u e)). apply_component n (u ! n) (the
(apply_inv_update u e)) ≠ None"
  using len by presburger
  hence "those (map (λ n. apply_component n (u ! n) (the (apply_inv_update u e)))
[0..<length (the (apply_inv_update u e))]) ≠ None"

```

```

    using those_map_not_None
  by (smt (verit) add_less_cancel_left gen_length_def length_code length_map map_nth
nth_upt)
  thus ?thesis using apply_update.simps len by presburger
qed

```

Now we show that `apply_inv_update u` is monotonic for all updates `u`. The proof follows directly from the definition of `apply_inv_update` and a case analysis of the possible update components.

```

lemma inverse_monotonic:
  assumes "e ≤ e'" and "length u = length e'"
  shows "(inv_upd u e) ≤ (inv_upd u e')"
  unfolding energy_leq_def proof
    show "length (the (apply_inv_update u e)) = length (the (apply_inv_update u e'))"
  using assms len_inv_appl energy_leq_def
    by simp
    show "∀i < length (the (apply_inv_update u e)). the (apply_inv_update u e) ! i ≤
the (apply_inv_update u e') ! i "
  proof
    fix i
    show "i < length (the (apply_inv_update u e)) → the (apply_inv_update u e)
! i ≤ the (apply_inv_update u e') ! i "
  proof
    assume "i < length (the (apply_inv_update u e))"
    have "the (apply_inv_update u e) ! i = (map (λi. apply_inv_component i u e)
[0..<length e]) ! i"
      using apply_inv_update.simps assms
      using energy_leq_def by auto
    also have "... = (λi. apply_inv_component i u e) ([0..<length e] ! i)" using
nth_map
      by (metis (full_types) <i < length (the (apply_inv_update u e))> add_less_mono
assms(1) assms(2) energy_leq_def diff_add_inverse gen_length_def len_inv_appl length_code
less_add_same_cancel2 not_less_less_Suc_eq nth_map_upt nth_upt plus_1_eq_Suc)
    also have "... = apply_inv_component i u e"
      using <i < length (the (apply_inv_update u e))> assms(1) assms(2) energy_leq_def
by auto
    finally have E: "the (apply_inv_update u e) ! i =
      Max (set (map (λ(m,up). (case up of
        zero ⇒ (if i=m then (nth e i) else 0) |
        minus_one ⇒ (if i=m then (e ! i)+1 else 0) |
        min_set A ⇒ (if i∈A then (e ! m) else 0) |
        plus_one ⇒ (if i=m then (nth e i)-1 else 0))) (List.enumerate 0
u)))" using apply_inv_component.simps
      by presburger

    have "the (apply_inv_update u e') ! i = (map (λi. apply_inv_component i u
e') [0..<length e']) ! i"
      using apply_inv_update.simps assms
      using energy_leq_def by auto
    also have "... = (λi. apply_inv_component i u e') ([0..<length e'] ! i)"
  using nth_map
    by (metis (full_types) <i < length (the (apply_inv_update u e))> add_less_mono
assms(1) assms(2) energy_leq_def diff_add_inverse gen_length_def len_inv_appl length_code
less_add_same_cancel2 not_less_less_Suc_eq nth_map_upt nth_upt plus_1_eq_Suc)
    also have "... = apply_inv_component i u e'"

```

```

    using <i < length (the (apply_inv_update u e))> assms(1) assms(2) energy_leq_def
  by auto
  finally have E': "the (apply_inv_update u e') ! i =
    Max (set (map (λ(m,up). (case up of
      zero ⇒ (if i=m then (nth e' i) else 0) |
      minus_one ⇒ (if i=m then (e' ! i)+1 else 0) |
      min_set A ⇒ (if i∈A then (e' ! m) else 0) |
      plus_one ⇒ (if i=m then (nth e' i)-1 else 0))) (List.enumerate
0 u)))" using apply_inv_component.simps
    by presburger

  have fin': "finite (set (map (λ(m,up). (case up of
    zero ⇒ (if i=m then (nth e' i) else 0) |
    minus_one ⇒ (if i=m then (e' ! i)+1 else 0) |
    min_set A ⇒ (if i∈A then (e' ! m) else 0) |
    plus_one ⇒ (if i=m then (nth e' i)-1 else 0))) (List.enumerate 0
u)))" by simp
  have fin: "finite (set (map (λ(m, up).
    case up of zero ⇒ (if i=m then (nth e i) else 0) | minus_one
⇒ if i = m then e ! i + 1 else 0
    | min_set A ⇒ if i ∈ A then e ! m else 0 |
    plus_one ⇒ (if i=m then (nth e i)-1 else 0))
(List.enumerate 0 u)))" by simp

  have "∧x. x ∈ (set (map (λ(m,up). (case up of
    zero ⇒ (if i=m then (nth e i) else 0) |
    minus_one ⇒ (if i=m then (e ! i)+1 else 0) |
    min_set A ⇒ (if i∈A then (e ! m) else 0) |
    plus_one ⇒ (if i=m then (nth e i)-1 else 0))) (List.enumerate 0
u))) ⇒ (∃y. x ≤ y ∧ y ∈ (set (map (λ(m,up). (case up of
    zero ⇒ (if i=m then (nth e' i) else 0) |
    minus_one ⇒ (if i=m then (e' ! i)+1 else 0) |
    min_set A ⇒ (if i∈A then (e' ! m) else 0) |
    plus_one ⇒ (if i=m then (nth e' i)-1 else 0))) (List.enumerate
0 u))))"
  proof-
    fix x
    assume "x ∈ set (map (λ(m, up).
      case up of zero ⇒ (if i=m then (nth e i) else 0) | minus_one
⇒ if i = m then e ! i + 1 else 0
      | min_set A ⇒ if i ∈ A then e ! m else 0 |
      plus_one ⇒ (if i=m then (nth e i)-1 else 0))
(List.enumerate 0 u)))"
    hence "∃j < length u. x = (map (λ(m, up).
      case up of zero ⇒ (if i=m then (nth e i) else 0) | minus_one
⇒ if i = m then e ! i + 1 else 0
      | min_set A ⇒ if i ∈ A then e ! m else 0 |
      plus_one ⇒ (if i=m then (nth e i)-1 else 0))
(List.enumerate 0 u)) ! j" using in_set_conv_nth
    by (metis (no_types, lifting) length_enumerate length_map)
    from this obtain j where "j < length u" and X: "x = (map (λ(m, up).
      case up of zero ⇒ (if i=m then (nth e i) else 0) | minus_one
⇒ if i = m then e ! i + 1 else 0
      | min_set A ⇒ if i ∈ A then e ! m else 0 |
      plus_one ⇒ (if i=m then (nth e i)-1 else 0))
(List.enumerate 0 u)) ! j" by auto

```

```

hence "(List.enumerate 0 u) ! j = (j, (u ! j))"
by (simp add: nth_enumerate_eq)
hence X: "x=(case (u ! j) of zero  $\Rightarrow$  (if i=j then (nth e i) else 0) | minus_one
 $\Rightarrow$  if i = j then e ! i + 1 else 0
| min_set A  $\Rightarrow$  if i  $\in$  A then e ! j else 0 |
plus_one  $\Rightarrow$  (if i=j then (nth e i)-1 else 0)))" using X
by (simp add: <j < length u>)

consider (zero) "(u ! j) = zero" | (minus_one) "(u ! j) = minus_one" | (min_set)
" $\exists$  A. (u ! j) = min_set A" | (plus_one) "(u ! j) = plus_one"
by (meson update_component.exhaust)

thus "( $\exists$  y. x  $\leq$  y  $\wedge$  y  $\in$  (set (map ( $\lambda$ (m,up). (case up of
zero  $\Rightarrow$  (if i=m then (nth e' i) else 0) |
minus_one  $\Rightarrow$  (if i=m then (e' ! i)+1 else 0) |
min_set A  $\Rightarrow$  (if i $\in$ A then (e' ! m) else 0) |
plus_one  $\Rightarrow$  (if i=m then (nth e' i)-1 else 0)))) (List.enumerate
0 u))))"
proof(cases)
case zero
hence "x= (if i=j then (nth e i) else 0)" using X by simp
also have "...  $\leq$  (if i=j then (nth e' i) else 0)" using assms
using <i < length (the (apply_inv_update u e))> energy_leq_def
by force
also have "... = ( $\lambda$ (m, up).
case up of zero  $\Rightarrow$  (if i=m then (nth e' i) else 0) |
minus_one  $\Rightarrow$  if i = m then e' ! i + 1 else 0
| min_set A  $\Rightarrow$  if i  $\in$  A then e' ! m else 0 |
plus_one  $\Rightarrow$  (if i=m then (nth e' i)-1 else 0))(j, (u ! j)))"
by (simp add: zero)
finally have "x  $\leq$  (map ( $\lambda$ (m, up).
case up of zero  $\Rightarrow$  (if i=m then (nth e' i) else 0) |
minus_one  $\Rightarrow$  if i = m then e' ! i + 1 else 0
| min_set A  $\Rightarrow$  if i  $\in$  A then e' ! m else 0 |
plus_one  $\Rightarrow$  (if i=m then (nth e' i)-1 else 0))
(List.enumerate 0 u))!j"
by (simp add: <List.enumerate 0 u ! j = (j, u ! j)> <j < length u>)
then show ?thesis
using <j < length u> by auto
next
case minus_one
hence X: "x = (if i=j then (e ! i)+1 else 0)" using X by simp
then show ?thesis proof(cases "i=j")
case True
hence "x = (e ! i) +1" using X by simp
also have "...  $\leq$  (e' ! i) +1" using assms
using True <j < length u> energy_leq_def by auto
also have "... = ( $\lambda$ (m, up).
case up of zero  $\Rightarrow$  (if i=m then (nth e' i) else 0) |
minus_one  $\Rightarrow$  if i = m then e' ! i + 1 else 0
| min_set A  $\Rightarrow$  if i  $\in$  A then e' ! m else 0 |
plus_one  $\Rightarrow$  (if i=m then (nth e' i)-1 else 0))(j, (u ! j)))"
by (simp add: minus_one True)
finally have "x  $\leq$  (map ( $\lambda$ (m, up).
case up of zero  $\Rightarrow$  (if i=m then (nth e' i) else 0) |
minus_one  $\Rightarrow$  if i = m then e' ! i + 1 else 0

```

```

      | min_set A  $\Rightarrow$  if  $i \in A$  then  $e' ! m$  else 0 |
    plus_one  $\Rightarrow$  (if  $i=m$  then (nth  $e' i$ )-1 else 0))
      (List.enumerate 0 u))!j"
  by (simp add: <List.enumerate 0 u ! j = (j, u ! j)> <j < length u>)
then show ?thesis
  using <j < length u> by auto
next
  case False
  hence "x = 0" using X by simp
  also have "...  $\leq$  0"
    by simp
  also have "... = ( $\lambda(m, up)$ .
    case up of zero  $\Rightarrow$  (if  $i=m$  then (nth  $e' i$ ) else 0) |
minus_one  $\Rightarrow$  if  $i = m$  then  $e' ! i + 1$  else 0
    | min_set A  $\Rightarrow$  if  $i \in A$  then  $e' ! m$  else 0 |
    plus_one  $\Rightarrow$  (if  $i=m$  then (nth  $e' i$ )-1 else 0))(j, (u ! j))"
  by (simp add: minus_one False)
  finally have "x  $\leq$  (map ( $\lambda(m, up)$ .
    case up of zero  $\Rightarrow$  (if  $i=m$  then (nth  $e' i$ ) else 0) |
minus_one  $\Rightarrow$  if  $i = m$  then  $e' ! i + 1$  else 0
    | min_set A  $\Rightarrow$  if  $i \in A$  then  $e' ! m$  else 0 | plus_one
 $\Rightarrow$  (if  $i=m$  then (nth  $e' i$ )-1 else 0))
    (List.enumerate 0 u))!j"
  by (simp add: <List.enumerate 0 u ! j = (j, u ! j)> <j < length u>)
then show ?thesis
  using <j < length u> by auto
qed
next
  case min_set
  from this obtain A where A: "u ! j = min_set A" by auto
  hence X: "x = (if  $i \in A$  then  $e' ! j$  else 0)" using X by auto
  then show ?thesis proof(cases "i  $\in$  A")
    case True
    hence "x= $e' ! j$ " using X by simp
    also have "...  $\leq e' ! j$ " using assms
      using <j < length u> energy_leq_def by auto
    also have "... = ( $\lambda(m, up)$ .
      case up of zero  $\Rightarrow$  (if  $i=m$  then (nth  $e' i$ ) else 0) |
minus_one  $\Rightarrow$  if  $i = m$  then  $e' ! i + 1$  else 0
      | min_set A  $\Rightarrow$  if  $i \in A$  then  $e' ! m$  else 0 | plus_one
 $\Rightarrow$  (if  $i=m$  then (nth  $e' i$ )-1 else 0))(j, (u ! j))"
    by (simp add: A True)
    finally have "x  $\leq$  (map ( $\lambda(m, up)$ .
      case up of zero  $\Rightarrow$  (if  $i=m$  then (nth  $e' i$ ) else 0) |
minus_one  $\Rightarrow$  if  $i = m$  then  $e' ! i + 1$  else 0
      | min_set A  $\Rightarrow$  if  $i \in A$  then  $e' ! m$  else 0 | plus_one
 $\Rightarrow$  (if  $i=m$  then (nth  $e' i$ )-1 else 0))
      (List.enumerate 0 u))!j"
    by (simp add: <List.enumerate 0 u ! j = (j, u ! j)> <j < length u>)
  then show ?thesis
    using <j < length u> by auto
  next
    case False
    hence "x=0" using X by simp
    also have "... = ( $\lambda(m, up)$ .
      case up of zero  $\Rightarrow$  (if  $i=m$  then (nth  $e' i$ ) else 0) |

```



```

minus_one ⇒ if i = m then e' ! i + 1 else 0
              | min_set A ⇒ if i ∈ A then e' ! m else 0 | plus_one
⇒ (if i=m then (nth e' i)-1 else 0))(j, (u ! j))"
  by (simp add: A False)
  finally have "x ≤ (map (λ(m, up)).
    case up of zero ⇒ (if i=m then (nth e' i) else 0) |
minus_one ⇒ if i = m then e' ! i + 1 else 0
              | min_set A ⇒ if i ∈ A then e' ! m else 0 | plus_one
⇒ (if i=m then (nth e' i)-1 else 0))
    (List.enumerate 0 u))!j"
  by (simp add: <List.enumerate 0 u ! j = (j, u ! j)> <j < length u>)
  then show ?thesis
  using <j < length u> by auto
qed
next
case plus_one
then show ?thesis proof(cases "i=j")
  case True
  hence "x=e!i -1" using X plus_one by simp
  have "x ≤ e' ! i -1"
  proof(cases "e!i =0")
    case True
    then show ?thesis
    by (simp add: <x = e ! i - 1>)
  next
  case False
  then show ?thesis
  proof(cases "e!i = ∞")
    case True
    then show ?thesis using assms
    using <i < length (inv_upd u e)> energy_leq_def by fastforce
  next
  case False
  from this obtain b where "e!i = enat (Suc b)" using <e ! i ≠ 0>
  by (metis list_decode.cases not_enat_eq zero_enat_def)
  then show ?thesis
  proof(cases "e'!i = ∞")
    case True
    then show ?thesis
    by simp
  next
  case False
  from this obtain c where "e'!i = enat (Suc c)" using <e!i = enat
(Suc b)> assms
  by (metis (no_types, lifting) Nat.lessE Suc_ile_eq <i < length
(inv_upd u e)> enat.exhaust enat_ord_simps(2) energy_leq_def len_inv_appl)
  hence "b ≤ c" using assms
  using <e ! i = enat (Suc b)> <i < length (inv_upd u e)> energy_leq_def
by auto
  then show ?thesis using <e!i = enat (Suc b)> <e'!i = enat (Suc
c)>
  by (simp add: <x = e ! i - 1> one_enat_def)
qed
qed
qed
show ?thesis using plus_one True

```

```

      by (smt (verit) <List.enumerate 0 u ! j = (j, u ! j)> <j < length
u> <x ≤ e' ! i - 1> case_prod_conv length_enumerate length_map nth_map_enumerate
nth_mem update_component.simps(17))
    next
      case False
      hence "x = 0" using X
      using plus_one by auto
      also have "... ≤ 0" by simp
      also have "... = (λ(m, up).
        case up of zero ⇒ (if i=m then (nth e' i) else 0) |
minus_one ⇒ if i = m then e' ! i + 1 else 0
        | min_set A ⇒ if i ∈ A then e' ! m else 0 |
        plus_one ⇒ (if i=m then (nth e' i)-1 else 0))(j, (u ! j))"
      by (simp add: plus_one False)
      finally have "x ≤ (map (λ(m, up).
        case up of zero ⇒ (if i=m then (nth e' i) else 0) |
minus_one ⇒ if i = m then e' ! i + 1 else 0
        | min_set A ⇒ if i ∈ A then e' ! m else 0 | plus_one
⇒ (if i=m then (nth e' i)-1 else 0))
        (List.enumerate 0 u))!j"
      by (simp add: <List.enumerate 0 u ! j = (j, u ! j)> <j < length u>)
      then show ?thesis
      using <j < length u> by auto
    qed
  qed
qed

  hence "∀x ∈ (set (map (λ(m, up).
    case up of zero ⇒ (if i=m then (nth e i) else 0) | minus_one
⇒ if i = m then e ! i + 1 else 0
    | min_set A ⇒ if i ∈ A then e ! m else 0 | plus_one ⇒
(if i=m then (nth e i)-1 else 0))
    (List.enumerate 0 u)))
  x ≤ Max (set (map (λ(m, up). (case up of
    zero ⇒ (if i=m then (nth e' i) else 0) |
    minus_one ⇒ (if i=m then (e' ! i)+1 else 0) |
    min_set A ⇒ (if i ∈ A then (e' ! m) else 0) | plus_one ⇒ (if i=m
then (nth e' i)-1 else 0))) (List.enumerate 0 u)))"
  using fin'
  by (meson Max.coboundedI dual_order.trans)
  hence "Max (set (map (λ(m, up).
    case up of zero ⇒ (if i=m then (nth e i) else 0) | minus_one
⇒ if i = m then e ! i + 1 else 0
    | min_set A ⇒ if i ∈ A then e ! m else 0 | plus_one ⇒
(if i=m then (nth e i)-1 else 0))
    (List.enumerate 0 u)))
  ≤ Max (set (map (λ(m, up). (case up of
    zero ⇒ (if i=m then (nth e' i) else 0) |
    minus_one ⇒ (if i=m then (e' ! i)+1 else 0) |
    min_set A ⇒ (if i ∈ A then (e' ! m) else 0) | plus_one ⇒ (if i=m
then (nth e' i)-1 else 0))) (List.enumerate 0 u)))"
  using fin assms Max_less_iff
  by (metis (no_types, lifting) Max_in <i < length (the (apply_inv_update
u e))> <length (the (apply_inv_update u e)) = length (the (apply_inv_update u e'))>
ex_in_conv len_inv_appl length_enumerate length_map nth_mem)

```

```

      thus "the (apply_inv_update u e) ! i ≤ the (apply_inv_update u e') ! i " using
E E'
      by presburger
    qed
  qed
qed

```

### 6.3 Relating Updates and “Inverse” Updates

Since the minimum is not an injective function, for many updates there does not exist an inverse. The following 2-dimensional examples show, that the function `apply_inv_update` does not map an update to its inverse.

```

lemma not_right_inverse_example:
  shows "apply_update [minus_one, (min_set {0,1})] [1,2] = Some [0,1]"
        "apply_inv_update [minus_one, (min_set {0,1})] [0,1] = Some [1,1]"
  by (auto simp add: nth_def)

lemma not_right_inverse:
  shows "∃u. ∃e. apply_inv_update u (upd u e) ≠ Some e"
  using not_right_inverse_example by force

lemma not_left_inverse_example:
  shows "apply_inv_update [zero, (min_set {0,1})] [0,1] = Some [1,1]"
        "apply_update [zero, (min_set {0,1})] [1,1] = Some [1,1]"
  by (auto simp add: nth_def)

lemma not_left_inverse:
  shows "∃u. ∃e. apply_update u (inv_upd u e) ≠ Some e"
  by (metis option.sel apply_update.simps length_0_conv not_Cons_self2 option.distinct(1))

```

We now show that the given calculation `apply_inv_update` indeed calculates  $e \mapsto \min\{e' \mid e \leq u(e')\}$  for all valid updates  $u$ . For this we first name this set `possible_inv u e`. Then we show that `inv_upd u e` is an element of that set before showing that it is minimal. Considering one component at a time, the proofs follow by a case analysis of the possible update components from the definition of `apply_inv_update`

```

abbreviation "possible_inv u e ≡ {e'. apply_update u e' ≠ None
                                     ∧ (e e≤ (upd u e'))}"

lemma leq_up_inv:
  assumes "length u = length e" and "valid_update u"
  shows "e e≤ (upd u (inv_upd u e))"
  unfolding energy_leq_def proof
    from assms have notNone: "apply_update u (the (apply_inv_update u e)) ≠ None"
  using inv_not_none_then inv_not_none by blast
    thus len1: "length e = length (the (apply_update u (the (apply_inv_update u e))))"
  using assms len_appl len_inv_appl
    by presburger

  show "∀n<length e. e ! n ≤ the (apply_update u (the (apply_inv_update u e)))
! n "
  proof
    fix n
    show "n < length e ⟶ e ! n ≤ the (apply_update u (the (apply_inv_update u
e)))) ! n "
    proof

```

```

    assume "n < length e"

    have notNone_n: "(map (λn. apply_component n (u ! n) (the (apply_inv_update
u e))) [0..<length (the (apply_inv_update u e))]) ! n ≠ None" using notNone apply_update.simps
    by (smt (verit) <n < length e> assms(1) length_map map_nth nth_map option.distinct(1)
those_all_Some)

    have "the (apply_update u (the (apply_inv_update u e))) ! n = the (those (map
(λn. apply_component n (u ! n) (the (apply_inv_update u e))) [0..<length (the (apply_inv_update
u e))])) ! n"
    using apply_update.simps assms(1) len1 notNone by presburger
    also have "... = the ((map (λn. apply_component n (u ! n) (the (apply_inv_update
u e))) [0..<length (the (apply_inv_update u e))]) ! n)" using the_those_n notNone
    by (smt (verit) <n < length e> apply_update.elims calculation assms(1)
length_map map_nth nth_map)
    also have "... = the ((λn. apply_component n (u ! n) (the (apply_inv_update
u e))) ([0..<length (the (apply_inv_update u e))]) ! n))" using nth_map
    using <n < length e> assms len_inv_appl minus_nat.diff_0 nth_upt by auto
    also have "... = the (apply_component n (u ! n) (the (apply_inv_update u
e)))" using <n < length e> assms len_inv_appl
    by (simp add: plus_nat.add_0)
    finally have unfolded_apply_update: "the (apply_update u (the (apply_inv_update
u e))) ! n = the (apply_component n (u ! n) (the (apply_inv_update u e)))" .

    have "(the (apply_inv_update u e)) ! n = (the (Some (map (λn. apply_inv_component
n u e) [0..<length e])) ! n)" using apply_inv_update.simps assms(1) by auto
    also have "... = (map (λn. apply_inv_component n u e) [0..<length e]) ! n"
    by auto
    also have "... = apply_inv_component n u e" using nth_map map_nth
    by (smt (verit) Suc_diff_Suc <n < length e> add_diff_inverse_nat diff_add_0
length_map less_diff_conv less_one nat_1 nat_one_as_int nth_upt plus_1_eq_Suc)
    finally have unfolded_apply_inv: "(the (apply_inv_update u e)) ! n = apply_inv_component
n u e".

    consider (zero) "u ! n = zero" |(minus_one) "u ! n = minus_one" |(min_set)
"∃A. min_set A = u ! n" |(plus_one) "u ! n = plus_one"
    by (metis update_component.exhaust)
    thus "e ! n ≤ the (apply_update u (the (apply_inv_update u e))) ! n"
    proof (cases)
    case zero
    hence "(List.enumerate 0 u) ! n = (n, zero)"
    by (simp add: <n < length e> assms(1) nth_enumerate_eq)
    hence nth_in_set: "e ! n ∈ set (map (λ(m,up). (case up of
zero ⇒ (if n=m then (nth e n) else 0) |
minus_one ⇒ (if n=m then (nth e n)+1 else 0) |
min_set A ⇒ (if n∈A then (nth e m) else 0) |
plus_one ⇒ (if n=m then (nth e n)-1 else 0))) (List.enumerate 0
u))" using nth_map
    by (smt (verit) <n < length e> assms(1) length_enumerate length_map nth_mem
old.prod.case update_component.simps(14))

    from zero have "the (apply_update u (the (apply_inv_update u e))) ! n =
the (apply_component n zero (the (apply_inv_update u e)))" using unfolded_apply_update
    by auto
    also have "... = ((the (apply_inv_update u e)) ! n)" using apply_component.simps(1)
    by simp

```

```

    also have "... = apply_inv_component n u e" using unfolded_apply_inv by
auto
    also have "... = Max (set (map (λ(m,up). (case up of
      zero ⇒ (if n=m then (nth e n) else 0) |
      minus_one ⇒ (if n=m then (nth e n)+1 else 0) |
      min_set A ⇒ (if n∈A then (nth e m) else 0) |
      plus_one ⇒ (if n=m then (nth e n)-1 else 0))) (List.enumerate 0
u)))" using apply_inv_component.simps by auto
    also have "... ≥ e ! n " using nth_in_set by simp
    finally show ?thesis .
next
  case minus_one

  hence A: "(λ(m,up). (case up of
    zero ⇒ (if n=m then (nth e n) else 0) |
    minus_one ⇒ (if n=m then (nth e n)+1 else 0) |
    min_set A ⇒ (if n∈A then (nth e m) else 0)|
    plus_one ⇒ (if n=m then (nth e n)-1 else 0))) (n,(u!n)) = (e!n)
+1"

    by simp
  have "(List.enumerate 0 u)!n = (n,(u!n))"
    using <n < length e> assms(1) nth_enumerate_eq
    by (metis add_0)
  hence "(e!n) +1 ∈ (set (map (λ(m,up). (case up of
    zero ⇒ (if n=m then (nth e n) else 0) |
    minus_one ⇒ (if n=m then (nth e n)+1 else 0) |
    min_set A ⇒ (if n∈A then (nth e m) else 0)|
    plus_one ⇒ (if n=m then (nth e n)-1 else 0))) (List.enumerate 0
u)))" using A nth_map_enumerate
    by (metis (no_types, lifting) <n < length e> assms(1) length_enumerate
length_map nth_mem)
  hence leq: "(e!n) +1 ≤ Max (set (map (λ(m,up). (case up of
    zero ⇒ (if n=m then (nth e n) else 0) |
    minus_one ⇒ (if n=m then (nth e n)+1 else 0) |
    min_set A ⇒ (if n∈A then (nth e m) else 0)|
    plus_one ⇒ (if n=m then (nth e n)-1 else 0))) (List.enumerate 0
u)))" using Max_ge by simp

  have notNone_comp: "apply_component n minus_one (the (apply_inv_update u
e)) ≠ None" using notNone
    by (smt (z3) <n < length e> add_0 len1 len_appl length_map length_upt
map_nth minus_one notNone_n nth_map_upt)

  from minus_one have "the (apply_update u (the (apply_inv_update u e))) !
n = the (apply_component n minus_one (the (apply_inv_update u e)))" using unfolded_apply_update
by auto
  also have "... = ((the (apply_inv_update u e)) ! n) -1" using apply_component.simps(2)
notNone_comp
    using calculation option.sel by auto
  also have "... = apply_inv_component n u e -1" using unfolded_apply_inv
by auto
  also have "... = Max (set (map (λ(m,up). (case up of
    zero ⇒ (if n=m then (nth e n) else 0) |
    minus_one ⇒ (if n=m then (nth e n)+1 else 0) |
    min_set A ⇒ (if n∈A then (nth e m) else 0)|
    plus_one ⇒ (if n=m then (nth e n)-1 else 0))) (List.enumerate 0

```

```

u))) -1" using apply_inv_component.simps by auto
  also have "... ≥ e ! n" using leq
    by (smt (verit) add.assoc add_diff_assoc_enat le_iff_add)
  finally show ?thesis .
next
  case min_set
  from this obtain A where "min_set A = u ! n" by auto

  have "upd u (inv_upd u e) ! n = the (apply_component n (min_set A) (inv_upd
u e))"
    using <min_set A = u ! n> unfolded_apply_update by auto
  also have "... = (min_list (nth (inv_upd u e) A))"
    using apply_component.elims
    by simp

  have leq: "∧j. j ∈ A ⇒ e!n ≤ (inv_upd u e)!j"
  proof-
    fix j
    assume "j ∈ A"
    hence "j < length e" using assms
      by (metis <min_set A = u ! n> in_mono mem_Collect_eq)
    hence "j < length [0..<length e]"
      by simp
    have "(inv_upd u e)!j = (map (λi. apply_inv_component i u e) [0..<length
e])!j"
      using apply_inv_update.simps assms
      by simp
    hence "(inv_upd u e)!j = apply_inv_component j u e"
      using nth_map <j < length [0..<length e]>
      by (metis <j < length e> nth_upt plus_nat.add_0)
    hence "(inv_upd u e)!j = Max (set (map (λ(m,up). (case up of
      zero ⇒ (if j=m then (nth e j) else 0) |
      minus_one ⇒ (if j=m then (nth e j)+1 else 0) |
      min_set A ⇒ (if j∈A then (nth e m) else 0)|
      plus_one ⇒ (if j=m then (nth e j)-1 else 0))) (List.enumerate 0
u)))"
      by auto

    have "(List.enumerate 0 u)! n = (n, u ! n)"
      by (simp add: <n < length e> assms(1) nth_enumerate_eq)

    have fin: "finite (set (map (λ(m,up). (case up of
      zero ⇒ (if j=m then (nth e j) else 0) |
      minus_one ⇒ (if j=m then (nth e j)+1 else 0) |
      min_set A ⇒ (if j∈A then (nth e m) else 0)|
      plus_one ⇒ (if j=m then (nth e j)-1 else 0))) (List.enumerate 0
u)))" by auto
    have "e!n = (case (min_set A) of
      zero ⇒ (if j=n then (nth e j) else 0) |
      minus_one ⇒ (if j=n then (nth e j)+1 else 0) |
      min_set A ⇒ (if j∈A then (nth e n) else 0)|
      plus_one ⇒ (if j=n then (nth e j)-1 else 0))"
      by (simp add: <j ∈ A>)
    hence "e!n = (λ(m,up). (case up of
      zero ⇒ (if j=m then (nth e j) else 0) |
      minus_one ⇒ (if j=m then (nth e j)+1 else 0) |

```

```

      min_set A  $\Rightarrow$  (if  $j \in A$  then (nth e m) else 0) |
      plus_one  $\Rightarrow$  (if  $j=m$  then (nth e j)-1 else 0))) (n, u ! n)"
    using <min_set A = u ! n> by simp
  hence "e!n  $\in$  (set (map ( $\lambda(m,up)$ ). (case up of
    zero  $\Rightarrow$  (if  $j=m$  then (nth e j) else 0) |
    minus_one  $\Rightarrow$  (if  $j=m$  then (nth e j)+1 else 0) |
    min_set A  $\Rightarrow$  (if  $j \in A$  then (nth e m) else 0) |
    plus_one  $\Rightarrow$  (if  $j=m$  then (nth e j)-1 else 0))) (List.enumerate 0
u)))"

    using <(List.enumerate 0 u)! n = (n, u ! n)> nth_map_enumerate
    by (metis (no_types, lifting) <n < length e> assms(1) in_set_conv_nth
length_enumerate length_map)

    thus "e!n  $\leq$  (inv_upd u e)!j"
      using fin Max_le_iff
      using <inv_upd u e ! j = Max (set (map ( $\lambda(k, y)$ ). case y of zero  $\Rightarrow$ (if
j=k then (nth e j) else 0) | minus_one  $\Rightarrow$  if  $j = k$  then e ! j + 1 else 0 | min_set
A  $\Rightarrow$  if  $j \in A$  then e ! k else 0 | plus_one  $\Rightarrow$  if  $j = k$  then e ! j - 1 else 0) (List.enumerate
0 u)))> by fastforce
    qed

    have "A  $\neq \{\}$   $\wedge$  A  $\subseteq \{x. x < \text{length } u\}$ " using assms
      by (simp add: <min_set A = u ! n>)
    hence "A  $\neq \{\}$   $\wedge$  A  $\subseteq \{x. x < \text{length } (\text{inv\_upd } u \text{ e})\}$ " using assms
      by auto

    have "set (nth (inv_upd u e) A) = {(inv_upd u e) ! i | i. i < length (inv_upd
u e)  $\wedge$  i  $\in$  A}"
      using set_nth by metis
    hence not_empty: "(set (nth (inv_upd u e) A))  $\neq \{\}$ "
      using <A  $\neq \{\}$   $\wedge$  A  $\subseteq \{x. x < \text{length } (\text{inv\_upd } u \text{ e})\}$ >
      by (smt (z3) Collect_empty_eq equals0I in_mono mem_Collect_eq)
    hence "(nth (inv_upd u e) A)  $\neq []$ "
      by blast
    hence min_eq_Min: "min_list (nth (inv_upd u e) A) = Min (set (nth (inv_upd
u e) A))"
      using min_list_Min by blast

    have "finite (set (nth (inv_upd u e) A))" using assms <min_set A = u !
n>
      by simp
    hence "(e!n  $\leq$  Min (set (nth (inv_upd u e) A))) = ( $\forall a \in$  (set (nth (inv_upd
u e) A)). e!n  $\leq$  a)"
      using not_empty Min_ge_iff by auto

    have "e!n  $\leq$  Min (set (nth (inv_upd u e) A))"
      unfolding <(e!n  $\leq$  Min (set (nth (inv_upd u e) A))) = ( $\forall a \in$  (set (nth
(inv_upd u e) A)). e!n  $\leq$  a)>
      proof
        fix x
        assume "x  $\in$  set (nth (inv_upd u e) A)"
        hence "x  $\in$  {(inv_upd u e) ! i | i. i < length (inv_upd u e)  $\wedge$  i  $\in$  A}"
          using set_nth
          by metis
        hence " $\exists j. j \in A \wedge x = (\text{inv\_upd } u \text{ e})!j$ "
          by blast

```

```

      thus "e ! n ≤ x " using leq
      by auto
    qed

    hence "e!n ≤ (min_list (nth (inv_upd u e) A))"
      using min_eq_Min
      by metis
    thus ?thesis
      using calculation by auto
  next
    case plus_one
    hence A: "(λ(m,up). (case up of
      zero ⇒ (if n=m then (nth e n) else 0) |
      minus_one ⇒ (if n=m then (nth e n)+1 else 0) |
      min_set A ⇒ (if n∈A then (nth e m) else 0)|
      plus_one ⇒ (if n=m then (nth e n)-1 else 0))) (n,(u!n)) = (e!n)
    -1"
      by simp
    have "(List.enumerate 0 u)!n = (n,(u!n))"
      using <n < length e> assms(1) nth_enumerate_eq
      by (metis add_0)
    hence "(e!n) -1 ∈ (set (map (λ(m,up). (case up of
      zero ⇒ (if n=m then (nth e n) else 0) |
      minus_one ⇒ (if n=m then (nth e n)+1 else 0) |
      min_set A ⇒ (if n∈A then (nth e m) else 0)|
      plus_one ⇒ (if n=m then (nth e n)-1 else 0))) (List.enumerate 0
    u)))" using plus_one nth_map_enumerate A
      by (metis (no_types, lifting) <n < length e> assms(1) length_enumerate
    length_map nth_mem)
    hence leq: "(e!n) -1 ≤ Max (set (map (λ(m,up). (case up of
      zero ⇒ (if n=m then (nth e n) else 0) |
      minus_one ⇒ (if n=m then (nth e n)+1 else 0) |
      min_set A ⇒ (if n∈A then (nth e m) else 0)|
      plus_one ⇒ (if n=m then (nth e n)-1 else 0))) (List.enumerate 0
    u)))" using Max_ge by simp

    have "e ! n ≤ ((e!n)-1)+1"
      by (metis dual_order.trans eSuc_minus_1 eSuc_plus_1 le_iff_add linorder_le_cases
    plus_1_eSuc(1))
    also have "... ≤ ( Max (set (map (λ(m,up). (case up of
      zero ⇒ (if n=m then (nth e n) else 0) |
      minus_one ⇒ (if n=m then (nth e n)+1 else 0) |
      min_set A ⇒ (if n∈A then (nth e m) else 0)|
      plus_one ⇒ (if n=m then (nth e n)-1 else 0))) (List.enumerate 0
    u)))) +1" using leq
      by simp
    also have "... = (inv_upd u e) ! n +1"
      using apply_inv_component.simps unfolded_apply_inv by presburger
    also have "... = upd u (inv_upd u e) ! n"
      using unfolded_apply_update plus_one by auto
    finally show ?thesis .
  qed
qed
qed
qed

```



```

lemma apply_inv_is_min:
  assumes "length u = length e" and "valid_update u"
  shows "energy_Min (possible_inv u e) = {inv_upd u e}"
proof
  have apply_inv_leq_possible_inv: " $\forall e' \in (\text{possible\_inv } u \ e). (\text{inv\_upd } u \ e) \leq e'$ "
  proof
    fix e'
    assume "e'  $\in$  possible_inv u e"
    hence "energy_leq e (the (apply_update u e'))" by auto
    hence B: " $\forall n < \text{length } e. e!n \leq (\text{the } (\text{apply\_update } u \ e')) ! n$ " unfolding energy_leq_def
  by auto

    from <e'  $\in$  possible_inv u e> have "apply_update u e'  $\neq$  None" by simp
    have geq_0: " $\bigwedge i. i < \text{length } u \implies u!i = \text{minus\_one} \implies e'!i > 0$ "
  proof-
    fix i
    assume "i < length u" and "u!i = minus_one"
    have "e'!i = 0  $\implies$  False"
  proof-
    assume "e'!i = 0"
    hence "apply_component i minus_one e' = None"
      by simp
    hence "apply_component i (u!i) e' = None"
      using <u!i = minus_one> by simp

    from <apply_update u e'  $\neq$  None> have "those (map ( $\lambda i. \text{apply\_component } i$ 
      (u ! i) e')) [0.. $\text{length } e'$ ])  $\neq$  None" unfolding apply_update.simps
      by meson
    hence "(map ( $\lambda i. \text{apply\_component } i$  (u ! i) e')) [0.. $\text{length } e'$ ]) ! i  $\neq$  None"
  using those_all_Some
    by (metis <apply_update u e'  $\neq$  None> <i < length u> apply_update.simps
      length_map map_nth)
    hence " $(\lambda i. \text{apply\_component } i$  (u ! i) e') ([0.. $\text{length } e'$ ] ! i)  $\neq$  None"
  using nth_map
    by (metis <apply_update u e'  $\neq$  None> <i < length u> apply_update.simps
      length_map map_nth)
    hence "apply_component i (u ! i) e'  $\neq$  None"
    by (metis <apply_update u e'  $\neq$  None> <i < length u> apply_update.elims
      nth_upt plus_nat.add_0)
    thus "False"
    using <apply_component i (u!i) e' = None> by simp
  qed

  then show "e'!i > 0"
  by auto
qed

  show "energy_leq (the (apply_inv_update u e)) e'" unfolding energy_leq_def
  proof
    show "length (the (apply_inv_update u e)) = length e'" using assms
      by (metis (mono_tags, lifting) <e'  $\in$  possible_inv u e> energy_leq_def len_appl
        len_inv_appl mem_Collect_eq)
    show " $\forall n < \text{length } (\text{the } (\text{apply\_inv\_update } u \ e)). \text{the } (\text{apply\_inv\_update } u \ e) !$ 
      n  $\leq e' ! n$ "

```

```

proof
  fix n
  show "n < length (the (apply_inv_update u e)) → the (apply_inv_update
u e) ! n ≤ e' ! n"
  proof
    assume "n < length (the (apply_inv_update u e))"
    have "the (apply_inv_update u e) ! n = (map (λn. apply_inv_component n
u e) [0..

```

```

qed

show "the (apply_inv_update u e) ! n ≤ e' ! n"
  unfolding inv_max Max_iff
proof
  fix a
  assume "a ∈ (set (map (λ(m, up). case up of zero ⇒ if n = m then e !
n else 0 | minus_one ⇒ if n = m then e ! n + 1 else 0 | min_set A ⇒ if n ∈ A then
e ! m else 0 | plus_one ⇒ if n = m then e ! n - 1 else 0) (List.enumerate 0 u)))"
  hence "∃ i. i < length (List.enumerate 0 u) ∧ a = (λ(m, up). case up
of zero ⇒ if n = m then e ! n else 0 | minus_one ⇒ if n = m then e ! n + 1 else
0 | min_set A ⇒ if n ∈ A then e ! m else 0 | plus_one ⇒ if n = m then e ! n -
1 else 0) ((List.enumerate 0 u) ! i) "
    using set_map
    by (metis (no_types, lifting) in_set_conv_nth length_map nth_map)
  from this obtain m where A: "a = (λ(m, up). case up of zero ⇒ if n
= m then e ! n else 0 | minus_one ⇒ if n = m then e ! n + 1 else 0 | min_set A
⇒ if n ∈ A then e ! m else 0 | plus_one ⇒ if n = m then e ! n - 1 else 0) (m,
(u!m))"
    and "m < length u"
    using nth_enumerate_eq by fastforce

  consider (zero) "u ! m = zero" | (minus_one) "u ! m = minus_one" | (min)
"∃ A. u ! m = min_set A" | (plus_one) "u ! m = plus_one"
    using update_component.exhaust by auto
  then show "a ≤ e' ! n" proof(cases)
    case zero
    hence A: "a = (if n = m then e ! n else 0)" using A by simp
    then show ?thesis
    proof(cases "n=m")
      case True
      hence "a = e!n" using zero A by simp
      also have "... ≤ the (apply_component n (u ! n) e'))" using upd_v
by simp
      also have "... = the (apply_component n zero e'))" using zero True
by simp
      also have "... = e'!n"
        by simp
      finally show ?thesis by simp
    next
      case False
      then show ?thesis using zero A by simp
    qed
  next
    case minus_one
    hence A: "a = (if n = m then e ! n + 1 else 0)" using A by simp
    then show ?thesis
    proof(cases "n=m")
      case True
      hence "a = e!n + 1" using minus_one A by simp
      also have "... ≤ (the (apply_component n (u ! n) e')) + 1" using
upd_v by simp
      also have "... = (the (apply_component n minus_one e')) + 1" using
minus_one True by simp
      also have "... = (the (if ((e' ! n) > 0) then Some ((e' ! n) - 1)

```

```

else None)) +1" using apply_component.simps
  by auto
also have "... = (e'!n -1) +1" using geq_0
  using True <n < length (inv_upd u e)> assms(1) minus_one by fastforce

also have "... = e'!n"
proof(cases "e'!n = ∞")
  case True
  then show ?thesis
  by simp
next
  case False
  hence "∃b. e' ! n = enat (Suc b)" using geq_0 True <n < length
(inv_upd u e)> assms(1) minus_one
  by (metis len_inv_appl not0_implies_Suc not_enat_eq not_iless0
zero_enat_def)
  from this obtain b where "e' ! n = enat (Suc b)" by auto
  then show ?thesis
  by (metis eSuc_enat eSuc_minus_1 eSuc_plus_1)
qed

finally show ?thesis .
next
  case False
  then show ?thesis using minus_one A by simp
qed
next
  case min
  from this obtain A where "u !m = min_set A" by auto
  hence A: "a = (if n ∈ A then e' ! m else 0)" using A by simp
  then show ?thesis
  proof(cases "n ∈ A")
    case True
    hence "a = e'!m" using A min by simp

    have "(set (nth e' A)) ≠ {}" using set_nth True assms
      by (smt (verit) Collect_empty_eq <length (inv_upd u e) = length
e'> <n < length (inv_upd u e)>)
    hence "(nth e' A) ≠ []"
      by auto

    from B have "e' ! m ≤ (the (apply_update u e')) ! m"
      using <m < length u> assms(1) len_inv_appl by auto
    hence upd_v: "e' ! m ≤ the (apply_component m (u ! m) e'" using
apply_to_comp_n <m < length u>
      by (metis <apply_update u e' ≠ None> <length (inv_upd u e) =
length e'> assms(1) len_inv_appl)
    hence "e' ! m ≤ the (apply_component m (min_set A) e'" using <u
!m = min_set A> by simp
    also have "... = the (Some (min_list (nth e' A)))"
      by simp
    also have "... = (min_list (nth e' A))"
      by simp
    also have "... = Min (set (nth e' A))" using min_list_Min <(nth
e' A) ≠ []>

```

```

      by auto
    also have "... ≤ e'!n" using True Min_le
      using <length (inv_upd u e) = length e'> <n < length (inv_upd
u e)> set_nth by fastforce
    finally show ?thesis using <a = e!m>
      by simp
  next
    case False
    then show ?thesis using <u !m = min_set A> A by simp
  qed
next
  case plus_one
  hence A: "a = (if n = m then e ! n - 1 else 0)" using A by simp
  then show ?thesis
  proof(cases "n=m")
    case True

    hence "a = (e!n) - 1" using plus_one A by simp
    also have "... ≤ (the (apply_component n (u ! n) e')) - 1"
    proof(cases "(the (apply_component n (u ! n) e')) = 0")
      case True
      hence "e!n = 0" using upd_v
        by simp
      then show ?thesis using True
        by auto
    next
      case False
      then show ?thesis
      proof(cases "e!n = ∞")
        case True
        then show ?thesis using upd_v
          by simp
      next
        case False
        then show ?thesis
        proof(cases "e!n = 0")
          case True
          then show ?thesis
            by simp
        next
          case False
          hence "∃ a. e!n = enat (Suc a)" using < e ! n ≠ ∞ >
            by (metis enat.exhaust old.nat.exhaust zero_enat_def)
          then show ?thesis
          proof(cases "(the (apply_component n (u ! n) e')) = ∞")
            case True
            then show ?thesis
              by simp
          next
            case False
            hence "∃ b. (the (apply_component n (u ! n) e')) = enat (Suc
b)" using < (the (apply_component n (u ! n) e')) ≠ 0 >
              by (metis enat.exhaust old.nat.exhaust zero_enat_def)
            then show ?thesis using <∃ a. e!n = enat (Suc a)> upd_v
              by (metis Suc_le_eq diff_Suc_1 enat_ord_simps(1) idiff_enat_enat
less_Suc_eq_le one_enat_def)

```

```

      qed
    qed
  qed
  qed
  also have "... = (the (apply_component n plus_one e')) -1" using
plus_one True by simp
  also have "... = the (Some ((e'!n)+1)) -1" using apply_component.simps
  by auto
  also have "... = (e'!n +1) -1"
  using True <n < length (inv_upd u e)> assms(1) plus_one by fastforce

  also have "... = e'!n"
  proof(cases "e'!n = ∞")
    case True
    then show ?thesis
    by simp
  next
    case False
    then show ?thesis
    by (simp add: add.commute)
  qed

  finally show ?thesis .
next
  case False
  then show ?thesis using plus_one A by simp
qed
qed
qed
qed
qed
qed
qed
qed

```

```

  have apply_inv_is_possible_inv: " $\bigwedge u v. \text{length } u = \text{length } v \implies \text{valid\_update } u \implies \text{inv\_upd } u v \in \text{possible\_inv } u v$ "
  using leq_up_inv inv_not_none_then inv_not_none by blast

  show "energy_Min (possible_inv u e)  $\subseteq$  {the (apply_inv_update u e)}"
  using apply_inv_leq_possible_inv apply_inv_is_possible_inv energy_Min_def assms
  by (smt (verit, ccfv_SIG) Collect_cong insert_iff mem_Collect_eq subsetI)
  show "{the (apply_inv_update u e)}  $\subseteq$  energy_Min (possible_inv u e)"
  using apply_inv_leq_possible_inv apply_inv_is_possible_inv energy_Min_def
  by (smt (verit, ccfv_SIG) <energy_Min (possible_inv u e)  $\subseteq$  {the (apply_inv_update u e)}> assms(1) assms(2) energy_leq.strict_trans1 insert_absorb mem_Collect_eq subset_iff subset_singletonD)
  qed

```

We now show that `apply_inv_update u` is decreasing.

```

lemma inv_up_leq:
  assumes "apply_update u e  $\neq$  None" and "valid_update u"
  shows "(inv_upd u (upd u e)) e  $\leq$  e"
  unfolding energy_leq_def proof
  from assms(1) have "length e = length u"
  by (metis apply_update.simps)

```

```

hence "length (the (apply_update u e)) = length u" using len_appl assms(1)
  by presburger
hence "(apply_inv_update u (the (apply_update u e))) ≠ None"
  using inv_not_none by presburger
thus "length (the (apply_inv_update u (the (apply_update u e)))) = length e" using
len_inv_appl <length (the (apply_update u e)) = length u> <length e = length u>
  by presburger
show "∀n<length (the (apply_inv_update u (the (apply_update u e)))) .
  the (apply_inv_update u (the (apply_update u e))) ! n ≤ e ! n "
proof
  fix n
  show "n < length (the (apply_inv_update u (the (apply_update u e)))) →
    the (apply_inv_update u (the (apply_update u e))) ! n ≤ e ! n"
  proof
    assume "n < length (the (apply_inv_update u (the (apply_update u e))))"
    hence "n < length e"
      using <length (the (apply_inv_update u (the (apply_update u e)))) = length
e> by auto
    show "the (apply_inv_update u (the (apply_update u e))) ! n ≤ e ! n"
    proof-
      have "the (apply_inv_update u (the (apply_update u e))) ! n = (map (λn. apply_inv_compon
n u (the (apply_update u e))) [0..

```

```

      case up of zero  $\Rightarrow$  (if n=m then (nth (the (apply_update
u e)) n) else 0)
      | minus_one  $\Rightarrow$  if n = m then the (apply_update u e) !
n + 1 else 0
      | min_set A  $\Rightarrow$  if n  $\in$  A then the (apply_update u e) !
m else 0 |
      plus_one  $\Rightarrow$  (if n=m then (nth (the (apply_update u e)) n)-1 else
0))
      (List.enumerate 0 u)) ! m" using in_set_conv_nth
      by (metis (no_types, lifting) length_map)
      from this obtain m where "m < length (List.enumerate 0 u)" and "x =
(map ( $\lambda$ (m, up).
      case up of zero  $\Rightarrow$  (if n=m then (nth (the (apply_update
u e)) n) else 0)
      | minus_one  $\Rightarrow$  if n = m then the (apply_update u e) !
n + 1 else 0
      | min_set A  $\Rightarrow$  if n  $\in$  A then the (apply_update u e) !
m else 0 |
      plus_one  $\Rightarrow$  (if n=m then (nth (the (apply_update u e)) n)-1 else
0))
      (List.enumerate 0 u)) ! m" by auto
      hence "x = ( $\lambda$ (m, up).
      case up of zero  $\Rightarrow$  (if n=m then (nth (the (apply_update
u e)) n) else 0)
      | minus_one  $\Rightarrow$  if n = m then the (apply_update u e) !
n + 1 else 0
      | min_set A  $\Rightarrow$  if n  $\in$  A then the (apply_update u e) !
m else 0 |
      plus_one  $\Rightarrow$  (if n=m then (nth (the (apply_update u e)) n)-1 else
0))
      ((List.enumerate 0 u) ! m)" using nth_map <m < length (List.enumerate
0 u)>
      by simp
      hence X: "x= ( $\lambda$ (m, up).
      case up of zero  $\Rightarrow$  (if n=m then (nth (the (apply_update
u e)) n) else 0)
      | minus_one  $\Rightarrow$  if n = m then the (apply_update u e) !
n + 1 else 0
      | min_set A  $\Rightarrow$  if n  $\in$  A then the (apply_update u e) !
m else 0 |
      plus_one  $\Rightarrow$  (if n=m then (nth (the (apply_update u e)) n)-1 else
0))
      (m, (u ! m))"
      by (metis (no_types, lifting) <m < length (List.enumerate 0 u)> add_cancel_left_
length_enumerate nth_enumerate_eq)

      consider (zero) "u ! m = zero" | (minus_one) "u ! m = minus_one" | (min)
"  $\exists$  A. u ! m = min_set A" | (plus_one) "u ! m = plus_one"
      using update_component.exhaust by auto
      thus "x  $\leq$  e ! n" proof(cases)
      case zero
      hence "x = (if n=m then (nth (the (apply_update u e)) n) else 0)"
using X by simp
      then show ?thesis proof(cases "n=m")

```



```

    case True
    hence "x= upd u e ! n"
      by (simp add: <x = (if n = m then upd u e ! n else 0)>)
    also have "... = the (apply_component n (u!n) e)"
      using <n < length e> apply_to_comp_n assms(1) by auto
    also have "... = the (apply_component n zero e)" using zero True
by simp

    also have "... = e!n"
      by simp
    finally show ?thesis by auto
next
case False
hence "x= 0"
  by (simp add: <x = (if n = m then upd u e ! n else 0)>)
then show ?thesis by simp
qed
next
case minus_one
then show ?thesis proof(cases "m=n")
  case True
  hence "u ! n = minus_one" using minus_one by simp
  have "(apply_component n (u ! n) e) ≠ None" using assms(1) those_all_Some
apply_update.simps apply_component.simps <n < length e>
  by (smt (verit) add_cancel_right_left length_map map_nth nth_map
nth_upt)

  hence "e ! n > 0" using <u ! n = minus_one> by auto
  hence "((e!n) -1 )+1 = e!n" proof(cases "e ! n = ∞")
    case True
    then show ?thesis by simp
  next
  case False
  hence "∃b. e ! n = enat b" by simp
  from this obtain b where "e ! n = enat b" by auto
  hence "∃b'. b = Suc b'" using <e ! n > 0>
  by (simp add: not0_implies_Suc zero_enat_def)
  from this obtain b' where "b = Suc b'" by auto
  hence "e ! n = enat (Suc b')" using <e ! n = enat b> by simp
  hence "(e!n)-1 = enat b'"
  by (metis eSuc_enat eSuc_minus_1)
  hence "((e!n) -1 )+1 = enat (Suc b')"
  using eSuc_enat plus_1_eSuc(2) by auto
  then show ?thesis using <e ! n = enat (Suc b')> by simp
qed

  from True have "x = (the (apply_update u e) ! n) +1" using X minus_one
by simp

  also have "... = (the (apply_component n (u ! n) e)) +1" using
apply_to_comp_n assms
  using <length (the (apply_inv_update u (the (apply_update u
e)))) = length e> <n < length (the (apply_inv_update u (the (apply_update u e))))>
by presburger

  also have "... = ((e !n) -1 ) +1" using <u ! n = minus_one> assms
those_all_Some apply_update.simps apply_component.simps
  using <0 < e ! n> by auto
  finally have "x = e ! n" using <((e!n) -1 )+1 = e!n> by simp
  then show ?thesis by simp

```

```

      next
      case False
      hence "x = 0" using X minus_one by simp
      then show ?thesis
      by simp
    qed
  next
  case min
  from this obtain A where "u ! m = min_set A" by auto
  hence "A ≠ {} ∧ A ⊆ {x. x < length e}" using assms
  by (simp add: <length e = length u>)
  then show ?thesis proof(cases "n ∈ A")
    case True
    hence "x = the (apply_update u e) ! m" using X <u ! m = min_set
A> by simp
      also have "... = (the (apply_component n (u ! m) e))" using apply_to_comp_n
      by (metis <length e = length u> <m < length (List.enumerate
0 u)> <u ! m = min_set A> apply_component.simps(3) assms(1) length_enumerate)
      also have "... = (min_list (nth e A))" using <u ! m = min_set
A> apply_component.simps by simp
      also have "... = Min (set (nth e A))" using <A ≠ {} ∧ A ⊆ {x.
x < length e}> min_list_Min
      by (smt (z3) True <n < length e> less_zeroE list.size(3) mem_Collect_eq
set_conv_nth set_nth)
      also have "... ≤ e ! n" using True Min_le <A ≠ {} ∧ A ⊆ {x. x <
length e}>
      using List.finite_set <n < length e> set_nth by fastforce
      finally show ?thesis .
    next
    case False
    hence "x = 0" using X <u ! m = min_set A> by simp
    then show ?thesis by simp
  qed
  next
  case plus_one
  hence X: "x = (if n = m then (nth (the (apply_update u e)) n) - 1 else
0)" using X
      by simp
  then show ?thesis
  proof(cases "n = m")
    case True
    hence X: "x = (nth (the (apply_update u e)) n) - 1" using X by simp
      have "nth (the (apply_update u e)) n = the (apply_component n
(u ! n) e)" using apply_update.simps
      using <n < length e> apply_to_comp_n assms(1) by auto
      also have "... = the (apply_component n plus_one e)" using plus_one
True by simp
      also have "... = (e ! n + 1)" unfolding apply_component.simps
      by simp
      finally have "x = (e ! n + 1) - 1" using X
      by simp
      then show ?thesis
      by (simp add: add.commute)
    next
    case False

```

```

      hence "x = 0" using X plus_one by simp
      then show ?thesis by simp
    qed
  qed
qed

  hence leq: "∀x ∈ (set (map (λ(m,up). (case up of
    zero ⇒ (if n=m then (nth (the (apply_update u e)) n) else 0) |
    minus_one ⇒ (if n=m then (nth (the (apply_update u e)) n)+1 else
0) |
    min_set A ⇒ (if n∈A then (nth (the (apply_update u e)) m) else
0) |
    plus_one ⇒ (if n=m then (nth (the (apply_update u e)) n)-1 else
0))) (List.enumerate 0 u))). x ≤ e ! n" by blast

  have "apply_inv_component n u (the (apply_update u e)) = Max (set (map
(λ(m,up). (case up of
    zero ⇒ (if n=m then (nth (the (apply_update u e)) n) else 0) |
    minus_one ⇒ (if n=m then (nth (the (apply_update u e)) n)+1 else
0) |
    min_set A ⇒ (if n∈A then (nth (the (apply_update u e)) m) else
0) |
    plus_one ⇒ (if n=m then (nth (the (apply_update u e)) n)-1 else
0))) (List.enumerate 0 u)))" using apply_inv_component.simps
  by blast
  also have "... ≤ e ! n" using leq Max_le_iff
  by (smt (verit) List.finite_set <length e = length u > <n < length e >
empty_iff length_enumerate length_map nth_mem)
  finally show ?thesis .
qed
thus ?thesis using A by presburger
qed
qed
qed
qed

```

We now conclude that for any valid update the functions  $e \mapsto \min\{e' \mid e \leq u(e')\}$  and  $u$  form a Galois connection between the domain of  $u$  and the set of energies of the same length as  $u$  w.r.t to the component-wise order.

```

lemma galois_connection:
  assumes "apply_update u e' ≠ None" and "length e = length e'" and
    "valid_update u"
  shows "(inv_upd u e) e ≤ e' = e ≤ (upd u e')"
proof
  show "energy_leq (the (apply_inv_update u e)) e' ⇒ energy_leq e (upd u e')"

  proof-
    assume A: "energy_leq (the (apply_inv_update u e)) e'"
    from assms(1) have "length u = length e" using apply_update.simps assms(2) by
metis
    hence leq: "energy_leq e (the (apply_update u (the (apply_inv_update u e))))"
using leq_up_inv assms(3) inv_not_none
  by presburger
    have "(apply_update u (the (apply_inv_update u e))) ≠ None" using <length u
= length e>

```

```

    using inv_not_none inv_not_none_then by blast
  hence "energy_leq (the (apply_update u (the (apply_inv_update u e)))) (the (apply_update
u e'))" using A updates_monotonic
    using <length u = length e> assms(3) inv_not_none len_inv_appl by presburger

  thus "energy_leq e (the (apply_update u e'))" using leq
    using energy_leq.trans by blast
qed
show "energy_leq e (the (apply_update u e'))  $\implies$  energy_leq (the (apply_inv_update
u e)) e' "
proof-
  assume A: "energy_leq e (the (apply_update u e'))"
  have "apply_inv_update u e  $\neq$  None" using assms
    by (metis apply_update.simps inv_not_none)
  have "length u = length e" using assms
    by (metis apply_update.elims)
  from A have "e'  $\in$  possible_inv u e"
    using assms(1) mem_Collect_eq by auto
  thus "energy_leq (the (apply_inv_update u e)) e'" using apply_inv_is_min assms
<length u = length e> energy_Min_def
    by (smt (verit) A Collect_cong energy_leq.strict_trans1 inv_up_leq inverse_monotonic
len_appl)
qed
qed
end

```

## 7 Galois Energy Games over Naturals

```
theory Natural_Galois_Energy_Game
  imports Energy_Game Energy_Order Decidability Update
begin
```

We now define Galois energy games over vectors of naturals with the component-wise order. We formalise this in this theory as an `energy_game` with a fixed dimension. In particular, we assume all updates to have an upward-closed domain (as `domain_upw_closed`) and be length-preserving (as `upd_preserves_length`). We assume the latter for the inversion of updates too (as `inv_preserves_length`) and assume that the inversion of an update is a total mapping from energies to the domain of the update (as `domain_inv`). (This corresponds to section 4.2 in the preprint [6].)

```
locale natural_galois_energy_game = energy_game attacker weight application
  for attacker :: "'position set" and
    weight :: "'position ⇒ 'position ⇒ 'label option" and
    application :: "'label ⇒ energy ⇒ energy option" and
    inverse_application :: "'label ⇒ energy ⇒ energy option"
+
  fixes dimension :: "nat"
  assumes
    domain_upw_closed: " $\bigwedge p p' e e'. \text{weight } p p' \neq \text{None} \implies e \leq e' \implies \text{application}$   

    (the (weight p p')) e  $\neq$  None  $\implies \text{application}$  (the (weight p p')) e'  $\neq$  None"  

    and updgalois: " $\bigwedge p p' e. \text{weight } p p' \neq \text{None} \implies \text{application}$  (the (weight p  

    p')) e  $\neq$  None  $\implies \text{length}$  (the (application (the (weight p p')) e)) =  $\text{length } e$ "  

    and inv_preserves_length: " $\bigwedge p p' e. \text{weight } p p' \neq \text{None} \implies \text{length } e = \text{dimension}$   

 $\implies \text{length}$  (the (inverse_application (the (weight p p')) e)) =  $\text{length } e$ "  

    and domain_inv: " $\bigwedge p p' e. \text{weight } p p' \neq \text{None} \implies \text{length } e = \text{dimension} \implies$  (inverse_applica  

    (the (weight p p')) e)  $\neq$  None  $\wedge$  application (the (weight p p')) (the (inverse_application  

    (the (weight p p')) e))  $\neq$  None"  

    and galois: " $\bigwedge p p' e e'. \text{weight } p p' \neq \text{None} \implies \text{application}$  (the (weight p  

    p')) e'  $\neq$  None  $\implies \text{length } e = \text{dimension} \implies \text{length } e' = \text{dimension} \implies$  (the (inverse_application  

    (the (weight p p')) e))  $e \leq e' = e \leq$  (the (application (the (weight p p')) e')))"

sublocale natural_galois_energy_game  $\subseteq$  galois_energy_game attacker weight application
  inverse_application "{e::energy. length e = dimension}" energy_leq " $\lambda s. \text{energy\_sup}$   

  dimension s"
proof
  show "wqo_on energy_leq {e::energy. length e = dimension}"
    using Energy_Order.energy_leq_wqo .

  show " $\bigwedge \text{set } s'. \text{set} \subseteq \{e::\text{energy. length } e = \text{dimension}\} \implies \text{finite set} \implies \text{energy\_sup}$   

  dimension set  $\in \{e::\text{energy. length } e = \text{dimension}\} \wedge (\forall s. s \in \text{set} \longrightarrow \text{energy\_leq}$   

  s (energy_sup dimension set))  $\wedge (s' \in \{e::\text{energy. length } e = \text{dimension}\} \wedge (\forall s.$   

  s  $\in \text{set} \longrightarrow \text{energy\_leq } s s') \longrightarrow \text{energy\_leq}$  (energy_sup dimension set) s')"
  proof-
    fix set
    show " $\bigwedge s'. \text{set} \subseteq \{e::\text{energy. length } e = \text{dimension}\} \implies \text{finite set} \implies \text{energy\_sup}$   

    dimension set  $\in \{e::\text{energy. length } e = \text{dimension}\} \wedge (\forall s. s \in \text{set} \longrightarrow \text{energy\_leq}$   

    s (energy_sup dimension set))  $\wedge (s' \in \{e::\text{energy. length } e = \text{dimension}\} \wedge (\forall s.$   

    s  $\in \text{set} \longrightarrow \text{energy\_leq } s s') \longrightarrow \text{energy\_leq}$  (energy_sup dimension set) s')"
    proof
      fix s'
      assume "set  $\subseteq \{e. \text{length } e = \text{dimension}\}"$  and "finite set"
      show "energy_sup dimension set  $\in \{e. \text{length } e = \text{dimension}\}"$ 
```

```

    unfolding energy_sup_def
    by simp
    show "( $\forall s. s \in \text{set} \longrightarrow \text{energy\_leq } s (\text{energy\_sup dimension set})) \wedge (s' \in \{e::\text{energy}.\text{length } e = \text{dimension}\} \wedge (\forall s. s \in \text{set} \longrightarrow \text{energy\_leq } s s') \longrightarrow \text{energy\_leq } (\text{energy\_sup dimension set}) s')$ "
    proof
      show "( $\forall s. s \in \text{set} \longrightarrow s \leq \text{energy\_sup dimension set})$ "
        using energy_sup_is_sup(1) <math>\langle \text{set} \subseteq \{e.\text{length } e = \text{dimension}\} \rangle</math>
        by auto
      show " $s' \in \{e.\text{length } e = \text{dimension}\} \wedge (\forall s. s \in \text{set} \longrightarrow s \leq s') \longrightarrow \text{energy\_sup dimension set } e \leq s'$ "
        proof
          assume " $s' \in \{e.\text{length } e = \text{dimension}\} \wedge (\forall s. s \in \text{set} \longrightarrow s \leq s')$ "
          show " $\text{energy\_sup dimension set } e \leq s'$ "
            proof(rule energy_sup_is_sup(2))
              show " $\bigwedge a. a \in \text{set} \implies a \leq s'$ "
                by (simp add: <math>\langle s' \in \{e.\text{length } e = \text{dimension}\} \wedge (\forall s. s \in \text{set} \longrightarrow s \leq s') \rangle</math>
                s <math>\leq s'>>)</math>
              show " $\text{length } s' = \text{dimension}$ "
                using <math>\langle s' \in \{e.\text{length } e = \text{dimension}\} \wedge (\forall s. s \in \text{set} \longrightarrow s \leq s') \rangle</math>
                by auto
            qed
          qed
        qed
      qed
    qed

    show " $\bigwedge e e'. e \in \{e::\text{energy}.\text{length } e = \text{dimension}\} \implies e \leq e' \implies e' \in \{e::\text{energy}.\text{length } e = \text{dimension}\}$ "
      unfolding Energy_Order.energy_leq_def by simp

    show " $\bigwedge p p' e. \text{weight } p p' \neq \text{None} \implies \text{application } (\text{the } (\text{weight } p p')) e \neq \text{None} \implies e \in \{e::\text{energy}.\text{length } e = \text{dimension}\} \implies (\text{the } (\text{application } (\text{the } (\text{weight } p p')) e)) \in \{e::\text{energy}.\text{length } e = \text{dimension}\}$ "
      using inv_preserves_length
      by (simp add: updgalois)

    show " $\bigwedge p p' e. \text{weight } p p' \neq \text{None} \implies e \in \{e::\text{energy}.\text{length } e = \text{dimension}\} \implies (\text{inverse\_application } (\text{the } (\text{weight } p p')) e) \neq \text{None} \wedge (\text{the } (\text{inverse\_application } (\text{the } (\text{weight } p p')) e)) \in \{e::\text{energy}.\text{length } e = \text{dimension}\} \wedge \text{application } (\text{the } (\text{weight } p p')) (\text{the } (\text{inverse\_application } (\text{the } (\text{weight } p p')) e)) \neq \text{None}$ "
      using inv_preserves_length domain_inv by simp

    show " $\bigwedge p p' e e'. \text{weight } p p' \neq \text{None} \implies \text{energy\_leq } e e' \implies \text{application } (\text{the } (\text{weight } p p')) e \neq \text{None} \implies \text{application } (\text{the } (\text{weight } p p')) e' \neq \text{None}$ "
      using local.domain_upw_closed .

    show " $\bigwedge p p' e e'. \text{weight } p p' \neq \text{None} \implies \text{application } (\text{the } (\text{weight } p p')) e' \neq \text{None} \implies e \in \{e::\text{energy}.\text{length } e = \text{dimension}\} \implies e' \in \{e::\text{energy}.\text{length } e = \text{dimension}\} \implies \text{energy\_leq } (\text{the } (\text{inverse\_application } (\text{the } (\text{weight } p p')) e)) e' = \text{energy\_leq } e (\text{the } (\text{application } (\text{the } (\text{weight } p p')) e'))$ "
      using galois by simp
    qed

    locale natural_galois_energy_game_decidable = natural_galois_energy_game attacker
    weight application inverse_application dimension

```

```

for attacker :: "'position set" and
  weight :: "'position ⇒ 'position ⇒ 'label option" and
  application :: "'label ⇒ energy ⇒ energy option" and
  inverse_application :: "'label ⇒ energy ⇒ energy option" and
  dimension :: "nat"
+
assumes nonpos_eq_pos: "nonpos_winning_budget = winning_budget" and
        finite_positions: "finite positions"

sublocale natural_galois_energy_game_decidable ⊆ galois_energy_game_decidable attacker
weight application inverse_application "{e::energy. length e = dimension}" energy_leq
"λs. energy_sup dimension s"
proof
  show "nonpos_winning_budget = winning_budget" and "finite positions" using nonpos_eq_pos
finite_positions by auto
qed

```

Bisping's only considers declining energy games over vectors of naturals. We generalise this by considering all valid updates. We formalise this in this theory as an `energy_game` with a fixed dimension and show that such games are Galois energy games.

```

locale bispings_energy_game = energy_game attacker weight apply_update
  for attacker :: "'position set" and
    weight :: "'position ⇒ 'position ⇒ update option"
+
  fixes dimension :: "nat"
  assumes
    valid_updates: "∀p. ∀p'. ((weight p p' ≠ None )
      → ((length (the (weight p p')) = dimension)
      ∧ valid_update (the (weight p p')))))"

sublocale bispings_energy_game ⊆ natural_galois_energy_game attacker weight apply_update
apply_inv_update dimension
proof
  show "∧p p' e e'. weight p p' ≠ None ⇒ e e ≤ e' ⇒ apply_w p p' e ≠ None ⇒
  apply_w p p' e' ≠ None"
    using upd_domain_upward_closed
    by blast
  show "∧p p' e. weight p p' ≠ None ⇒ apply_w p p' e ≠ None ⇒ length (upd
  (the (weight p p')) e) = length e"
    using len_appl
    by simp
  show "∧p p' e. weight p p' ≠ None ⇒ length e = dimension ⇒ length (inv_upd
  (the (weight p p')) e) = length e"
    using len_inv_appl valid_updates
    by blast
  show "∧p p' e.
    weight p p' ≠ None ⇒
    length e = dimension ⇒
    apply_inv_update (the (weight p p')) e ≠ None ∧ apply_w p p' (inv_upd (the
  (weight p p')) e) ≠ None"
    using inv_not_none inv_not_none_then
    using valid_updates by presburger
  show "∧p p' e e'.
    weight p p' ≠ None ⇒
    apply_w p p' e' ≠ None ⇒
    length e = dimension ⇒

```

```

      length e' = dimension  $\implies$  inv_upd (the (weight p p')) e  $\leq$  e' = e  $\leq$  upd
(the (weight p p')) e'"
    using galois_connection
    by (metis valid_updates)
qed

locale bispings_energy_game_decidable = bispings_energy_game attacker weight dimension
  for attacker :: "'position set" and
    weight :: "'position  $\Rightarrow$  'position  $\Rightarrow$  update option" and
    dimension :: "nat"
+
assumes nonpos_eq_pos: "nonpos_winning_budget = winning_budget" and
    finite_positions: "finite positions"

sublocale bispings_energy_game_decidable  $\subseteq$  natural_galois_energy_game_decidable
attacker weight apply_update apply_inv_update dimension
proof
  show "nonpos_winning_budget = winning_budget" using nonpos_eq_pos.
  show "finite positions" using finite_positions .
qed

end

```



## 8 References

### References

- [1] B. Bisping. Process equivalence problems as energy games. In *Computer Aided Verification (CAV)*, volume 13964 of *Lecture Notes in Computer Science*, pages 85–106. Springer Nature Switzerland, 2023.
- [2] T. Brihaye and A. Goeminne. Multi-weighted reachability games. In *Reachability Problems*, volume 14235 of *Lecture Notes in Computer Science*, pages 85–97. Springer Nature Switzerland, 2023.
- [3] M. Ern , J. Koslowski, A. Melton, and G. E. Strecker. A primer on galois connections. *Annals of the New York Academy of Sciences*, 704(1):103–125, 1993.
- [4] M. K nnemann, F. Mazowiecki, L. Sch tze, H. Sinclair-Banks, and K. W grzycki. Coverability in vass revisited: Improving rackoffs bound to obtain conditional optimality. 261:131:1–131:20, 2023.
- [5] C. Lemke. A formal proof of decidability of multi-weighted declining energy games. Master’s thesis, Technische Universit t Berlin, 2024.
- [6] C. Lemke and B. Bisping. Galois energy games: To solve all kinds of quantitative reachability problems, 2025.

## A Appendix

### A.1 List Lemmas

```
theory List_Lemmas
  imports Main
begin
```

In this theory some simple equalities about lists are established.

```
lemma len_those:
  assumes "those l ≠ None"
  shows "length (the (those l)) = length l"
using assms proof(induct l)
  case Nil
  then show ?case by simp
next
  case (Cons a l)
  hence "∃x. a = Some x" using those.simps
  using option.collapse by fastforce
  then obtain x where "a=Some x" by auto
  hence AL: "those (a#l) = map_option (Cons x) (those l)" using those.simps by auto
  hence "those l ≠ None" using assms Cons.prem1 by auto
  hence "length (the (those l)) = length l" using Cons by simp
  then show ?case using AL <those l ≠ None> by (simp add: option.map_sel)
qed

lemma the_those_n:
  assumes "those (l:: 'a option list) ≠ None" and "(n::nat) < length l"
  shows "(the (those l)) ! n = the (l ! n)"
  using assms proof(induct l arbitrary: n)
  case Nil
  then show ?case by simp
next
  case (Cons a l)
  from assms(1) have l_notNone: "those l ≠ None" using those.simps(2)
  by (metis (no_types, lifting) Cons.prem1 option.collapse option.map_disc_iff
option.simps(4) option.simps(5))
  from assms(1) have "∃b. a=Some b" using those.simps
  using Cons.prem1 not_None_eq by fastforce
  from this obtain b where "a=Some b" by auto
  hence those_al: "those (a#l) = (Some (b# (the (those l))))" using those.simps
  l_notNone by simp
  then show ?case proof(cases "n=0")
    case True
    have "the (those (a # l)) ! n = the (Some (b# (the (those l)))) ! n" using those_al
    nth_def by simp
    also have "... = b" using True by simp
    also have "... = the ((a # l) ! n)" using <a=Some b> True by simp
    finally show ?thesis .
  next
  case False
  hence "∃m. n = Suc m" using old.nat.exhaust by auto
  from this obtain m where "n = Suc m" by auto
  hence "m < length l" using assms(2) Cons.prem2 by auto
  hence "the (those l) ! m = the (l ! m)" using Cons l_notNone by simp
  hence A: "the (those l) ! m = the ((a#l) ! n)" using <n = Suc m> by auto
```

```

    have "the (those l) ! m = the (those (a # l)) ! n" using <n = Suc m> those.simps(2)
those_al nth_def
  by simp
  then show ?thesis using A by simp
qed
qed

lemma those_all_Some:
  assumes "those l ≠ None" and "n < length l"
  shows "(l ! n) ≠ None"
  using assms proof (induct l arbitrary: n)
  case Nil
  then show ?case by simp
next
  case (Cons a l)
  from assms(1) have l_notNone: "those l ≠ None" using those.simps(2)
  by (metis (no_types, lifting) Cons.prem1 option.collapse option.map_disc_iff
option.simps(4) option.simps(5))
  from assms(1) have "∃ b. a = Some b" using those.simps
  using Cons.prem1 not_None_eq by fastforce
  from this obtain b where "a = Some b" by auto
  then show ?case proof (cases "n = 0")
  case True
  then show ?thesis using <a = Some b> by fastforce
  next
  case False
  hence "∃ m. n = Suc m" using old.nat.exhaust by auto
  from this obtain m where "n = Suc m" by auto
  hence "m < length l" using assms(2) Cons.prem2 by auto
  hence "l ! m ≠ None" using Cons.l_notNone by simp
  then show ?thesis using <n = Suc m> by simp
qed
qed

lemma nth_map_enumerate:
  shows "n < length xs ⟹ (map f (List.enumerate 0 xs))!n = f((List.enumerate 0
xs)!n)"
proof (induct xs arbitrary: n)
  case Nil
  then show ?case by simp
next
  case (Cons a xs)
  then show ?case using less_Suc_eq_0_disj
  by (metis length_enumerate nth_map)
qed

lemma those_map_not_None:
  assumes "∀ n < length xs. f (xs ! n) ≠ None"
  shows "those (map f xs) ≠ None"
using assms proof (induct xs)
  case Nil
  then show ?case by simp
next
  case (Cons a xs)
  hence "f ((a # xs) ! 0) ≠ None" using Cons(2) by auto
  hence "∃ b. f a = Some b" by auto

```

```

from this obtain b where "f a = Some b" by auto
have "those (map f xs) ≠ None" using Cons(1) assms those.simps
  by (smt (verit) Cons.prem Ex_less_Suc length_Cons less_trans_Suc nth_Cons_Suc)

then show ?case using those.simps <f a = Some b>
  by (simp add: option.simps(5))
qed

lemma last_len:
  assumes "length xs = Suc n"
  shows "last xs = xs ! n"
  using assms proof(induct xs arbitrary: n)
  case Nil
  then show ?case by simp
next
  case (Cons a xs)
  show ?case proof(cases "xs = Nil")
  case True
  then show ?thesis
    using Cons.prem by auto
next
  case False
  hence "∃m. n = Suc m" using Cons
    using not0_implies_Suc by auto
  from this obtain m where "n = Suc m" by auto
  hence "length xs = Suc m" using Cons by simp
  have "last (a#xs) = last xs"
    using False by simp
  also have "... = xs ! m" using Cons <length xs = Suc m> by simp
  also have "... = (a#xs) ! (Suc m)" by simp
  finally show ?thesis using <n = Suc m> by simp
qed
qed

end

```