Galois Energy Games

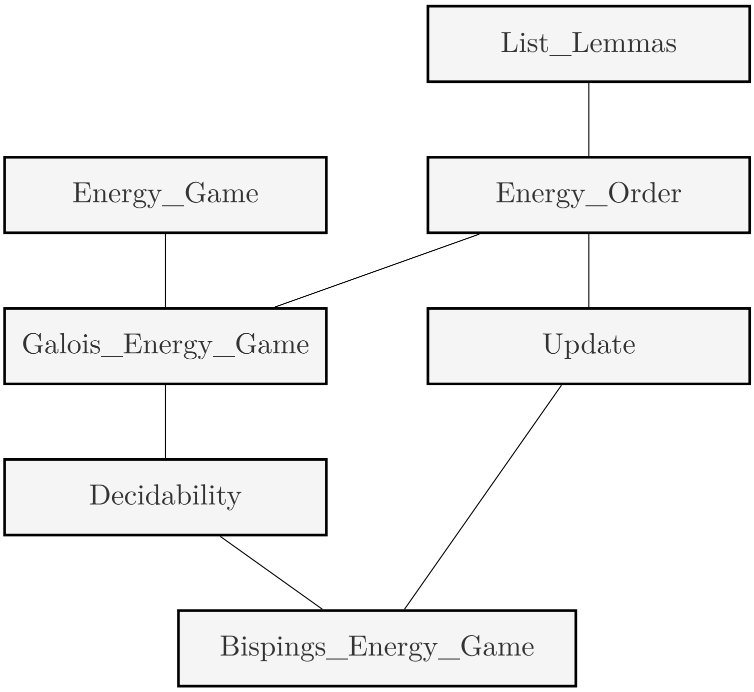
Caroline Lemke

March 17, 2025

Contents

1 Introduction

We provide a formal proof of decidability of Galois energy games over vectors of naturals with the component-wise order. Afterwards we consider an instantiation of *Bisping's declining energy games*. Bisping, Nestmann, and Jansen [3, 2] generalised Stirling's bisimulation game [5] to find Hennessy-Milner logic (HML) formulae distinguishing processes. Those formalae are elements of some HML-sublanguage from van Glabbeeks linear-time-branching-time spectrum[6] and thus their existence is a statement about behavioural equivalences. The HML-sublanguages from the linear-time-branching-time spectrum can be characterised by depth properties, which can be represented by six-dimensional vectors of extended natural numbers. Understanding these vectors as energies Bisping [1] developed a multi-weighted energy game deciding all common notions of (strong) behavioural equivalences at once, the *spectroscopy game*.
This game is part of a class of energy games Bisping [1] calls *declining energy games*. Bisping provides an algorithm, which he claims decides this class of energy games if the set of positions is finite. We substantiate this claim by providing a proof in Isabelle/HOL using a simplyfied and generalised version of that algorithm. To do so we first formalise energy games with reachability winning conditions in Energy_Game.thy. Building upon this, we 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 Bispings_Energy_Game.thy. In particular, we do not assume the games to be declining. An overview of all our theories is given by the following figure, where the theories above are imported by the ones below.



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.
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.
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.
Galois energy games over such energies with a fixed dimension are formalised in Galois_energy_game.thy.
In Decidability.thy we formalise one iteration of a simplyfied 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.
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.
Finally, we formalise energy 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) in Bispings_Energy_Game.thy.

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.

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

abbreviation "positions ≡ {g. g ∈ attacker ∨ g ∉ attacker}"
abbreviation "apply_w g g' ≡ application (the (weight g g'))"
```

Plays

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

```
coinductive valid_play :: "'position llist ⇒ bool" where
  "valid_play LNil" |
  "valid_play (LCons v LNil)" |
  "⟦weight v (lhd Ps) ≠ None; valid_play Ps; ¬lnull Ps⟧
    ⟹ 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' ≠ 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
          "⋀v Ps. Q (LCons v Ps) ⟹ Ps≠LNil ⟹ Q Ps ∧ weight v (lhd Ps) ≠ 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 "(∃v. p = LCons v LNil)")
      case True
      then show ?thesis by simp
    next
      case False
      hence "∃v Ps. p = LCons v Ps ∧ ¬ lnull Ps" using ‹¬p = LNil›
        by (metis llist.collapse(1) not_lnull_conv)
      from this obtain v Ps where "p = LCons v Ps ∧ ¬ lnull Ps" by blast
      hence "Q Ps ∧ weight v (lhd Ps) ≠ None" using valid_play
        using llist.disc(1) by blast
      then show ?thesis using valid_play.simps valid_play
        using ‹p = LCons v Ps ∧ ¬ 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)) ≠ None"
proof-
  have "∃prefix p'. p = lappend prefix p' ∧ llength prefix = Suc i ∧ weight (llast
prefix) (lhd p') ≠ None ∧ valid_play p'"
    using assms proof(induct i)
    case 0
    hence "∃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) ≠ None" using 0 valid_play.simps ‹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 ∧ llength (LCons v LNil) = Suc 0 ∧ weight
(llast (LCons v LNil)) (lhd Ps) ≠ None" using ‹p = LCons v Ps›
      using ‹p = lappend (LCons v LNil) Ps› ‹llength (LCons v LNil) = Suc 0›
      by simp
    hence "p = lappend (LCons v LNil) Ps ∧ llength (LCons v LNil) = Suc 0 ∧ weight
(llast (LCons v LNil)) (lhd Ps) ≠ None ∧ valid_play Ps" using valid_play.simps
0
      by (metis (no_types, lifting) ‹p = LCons v Ps› llist.distinct(1) ltl_simps(2))

    then show ?case by blast
  next
    case (Suc l)
    hence "∃prefix p'. p = lappend prefix p' ∧ llength prefix = enat (Suc l) ∧
weight (llast prefix) (lhd p') ≠ None ∧ 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 l) ∧ 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 l)) ∧ 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.prems(2) enat_ord_simps(2) lappend_LNil2 lappend_snocL1_conv_LCons2
lessI llist.exhaust_sel order.asym)
      show "llength (lappend prefix (LCons (lhd p') LNil)) = enat (Suc (Suc l))
∧
    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.disc(2) ltl_simps(2) zero_neq_one)
        thus "llength (lappend prefix (LCons (lhd p') LNil)) = enat (Suc (Suc l))"
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.prems(2) ‹llength (lappend prefix (LCons (lhd p') LNil))
= 1 + llength prefix› ‹llength (lappend prefix (LCons (lhd p') LNil)) = enat (Suc
(Suc l))› ‹p = lappend (lappend prefix (LCons (lhd p') LNil)) (ltl p')› add.commute
enat_add_mono eq_LConsD lappend_LNil2 less_numeral_extra(4) llast_lappend_LCons
llast_singleton llength_eq_enat_lfiniteD ltl_simps(1))
          show "valid_play (ltl p')" using P valid_play.simps
            by (metis (full_types) energy_game.valid_play.intros(1) ltl_simps(1)
ltl_simps(2))
        qed
      qed
    qed
    then show ?case by blast
  qed
  thus ?thesis
    by (smt (z3) assms(2) cancel_comm_monoid_add_class.diff_cancel eSuc_enat enat_ord_simps(2)
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)) ≠ None)) ∧
```

```
       weight v (lhd Ps) ≠ None"
  proof-
    fix v Ps p
    show "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)) ≠ None)) ∧
       weight v (lhd Ps) ≠ None"
    proof-
      assume "LCons v Ps = p"
      show "∀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)) ≠ None)) ∧
       weight v (lhd Ps) ≠ None"
      proof-
        assume A: "∀i. enat (Suc i) < llength p ⟶ weight (lnth p i) (lnth p (Suc
i)) ≠ None"
        show "Ps ≠ LNil ⟹
       (∃p. Ps = p ∧ (∀i. enat (Suc i) < llength p ⟶ weight (lnth p i) (lnth
p (Suc i)) ≠ None)) ∧
       weight v (lhd Ps) ≠ None"
        proof-
          assume "Ps ≠ LNil"
          show "(∃p. Ps = p ∧ (∀i. enat (Suc i) < llength p ⟶ weight (lnth p
i) (lnth p (Suc i)) ≠ None)) ∧
       weight v (lhd Ps) ≠ None"
          proof
            show "∃p. Ps = p ∧ (∀i. enat (Suc i) < llength p ⟶ weight (lnth p
i) (lnth p (Suc i)) ≠ None)"
            proof
              have "(∀i. enat (Suc i) < llength Ps ⟶ weight (lnth Ps i) (lnth
Ps (Suc i)) ≠ None)"
              proof
                fix i
                show "enat (Suc i) < llength Ps ⟶ weight (lnth Ps i) (lnth Ps
(Suc i)) ≠ None "
                proof
                  assume "enat (Suc i) < llength Ps"
                  hence "enat (Suc (Suc i)) < llength (LCons v Ps)"
                    by (metis ldropn_Suc_LCons ldropn_eq_LNil linorder_not_le)
                  have "(lnth Ps i) = (lnth (LCons v Ps) (Suc i))" by simp
                  have "(lnth Ps (Suc i)) = (lnth (LCons v Ps) (Suc (Suc i)))" by
simp
                  thus "weight (lnth Ps i) (lnth Ps (Suc i)) ≠ None"
                    using A ‹(lnth Ps i) = (lnth (LCons v Ps) (Suc i))›
                    using ‹LCons v Ps = p› ‹enat (Suc (Suc i)) < llength (LCons
v Ps)› by auto
                qed
              qed
              thus "Ps = Ps ∧ (∀i. enat (Suc i) < llength Ps ⟶ weight (lnth Ps
i) (lnth Ps (Suc i)) ≠ 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 ∨ llength (LCons v Ps) ≤
(Suc (Suc n)) ∨ 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 ∨ 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 ≠ None" and "Suc i ≤ m"
  shows "apply_w (lnth p i) (lnth p (Suc i)) (the (energy_level e p i)) ≠ None
         ∧ energy_level e p i ≠ 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))) ≠ None ∧ (energy_level e p (Suc i)) ≠ 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) ≠ None"
  shows "energy_level e p i = energy_level e (lappend p p') i"
proof-
  have A: "⋀i. i < the_enat (llength p) ⟹ energy_level e p i ≠ 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 ≡ (∀g'. weight g g' = None)"
abbreviation "attacker_stuck p ≡ (llast p)∈ attacker ∧ deadend (llast p)"

definition defender_wins_play:: "'energy ⇒ 'position llist ⇒ bool" where
  "defender_wins_play e p ≡ lfinite p ⟶
       (energy_level e p (the_enat (llength p)-1) = None ∨ 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 ⇒ 'position ⇒ 'position option) ⇒ bool"
where
  "attacker_strategy s = (∀g e. (g ∈ attacker ∧ ¬ deadend g) ⟶
                          (s e g ≠ None ∧ weight g (the (s e g))  ≠ None))"
```

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

```
coinductive play_consistent_attacker::"('energy ⇒ 'position ⇒ 'position option)
⇒ 'position llist  ⇒ 'energy ⇒ bool" where
  "play_consistent_attacker _ LNil _" |
  "play_consistent_attacker _ (LCons v LNil) _" |
  "⟦play_consistent_attacker s Ps (the (apply_w v (lhd Ps) e)); ¬lnull Ps;
    v ∈ attacker ⟶ (s e v) = Some (lhd Ps)⟧
    ⟹ 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
          "⋀s v Ps e'. Q s (LCons v Ps) e' ∧ ¬lnull Ps ⟹
                        Q s Ps (the (apply_w v (lhd Ps) e')) ∧
                        (v ∈ attacker ⟶ 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 "∃ 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)) ∧ (v ∈ attacker ⟶ 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) ≠ None" and
          "valid_play (lappend p (LCons g LNil))" and "llast p ∈ attacker ⟶
           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 "∃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.prems(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.prems(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.prems(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 (smt (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_Cons 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 ⇒ 'position option) ⇒ bool"
where
  "attacker_nonpos_strategy s = (∀list ≠ []. ((last list) ∈ attacker
   ∧ ¬deadend (last list)) ⟶ s list ≠ None
                                  ∧ (weight (last list) (the (s list)))≠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 ⇒ 'position option)
⇒ ('position llist) ⇒ ('position list) ⇒ bool" where
  "play_consistent_attacker_nonpos s LNil _" |
  "play_consistent_attacker_nonpos s (LCons v LNil) []" |
  "(last (w#l))∉attacker
  ⟹ play_consistent_attacker_nonpos s (LCons v LNil) (w#l)" |
  "⟦(last (w#l))∈attacker; the (s (w#l)) = v ⟧
  ⟹ play_consistent_attacker_nonpos s (LCons v LNil) (w#l)" |
  "⟦play_consistent_attacker_nonpos s Ps (l@[v]); ¬lnull Ps; v∉attacker⟧
   ⟹ play_consistent_attacker_nonpos s (LCons v Ps) l" |
  "⟦play_consistent_attacker_nonpos s Ps (l@[v]); ¬lnull Ps; v∈attacker;
    lhd Ps = the (s (l@[v]))⟧
    ⟹ 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: "⋀s v l. Q s (LCons v LNil) l ⟹ (l = [] ∨ (last l) ∉ attacker

                 ∨ ((last l)∈attacker ∧ the (s l) = v))" and
          step: "⋀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])))"
  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 "∃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 consitent play while remaining consistent.

```
lemma consistent_nonpos_append_defender:
  assumes "play_consistent_attacker_nonpos s (LCons v Ps) l" and
          "llast (LCons v Ps) ∉ 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.prems(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 "¬lnull Ps"
    by (metis list.discI list_of_LNil llist.collapse(1))

  have "¬ lnull (lappend Ps (LCons g' LNil))" by simp

  have "x = list_of (ltl Ps)" using Cons.hyps(2)
    by (metis Cons.prems(3) lfinite_code(2) list.sel(3) tl_list_of)
  have "llast (LCons (lhd Ps) (ltl Ps)) ∉ attacker" using Cons.prems(2)
    by (simp add: ‹¬ lnull Ps› llast_LCons)
  have "lfinite (LCons (lhd Ps) (ltl Ps))" using Cons.prems(3) by simp
  have "play_consistent_attacker_nonpos s (LCons (lhd Ps) (ltl Ps)) (l @ [v])" using
Cons.prems(1) play_consistent_attacker_nonpos.simps
    by (smt (verit, best) ‹¬ 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)) ∉ attacker› ‹x = list_of (ltl Ps)›
    by (metis ‹¬ lnull Ps› lhd_LCons_ltl)
```

```
  have "play_consistent_attacker_nonpos s (LCons v (lappend Ps (LCons g' LNil)))
l"
  proof(cases "v ∈ attacker")
    case True
    have "lhd Ps = the (s (l @ [v]))" using True Cons.prems(1) 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 (l @ [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)) (l @ [v])› ‹lhd (lappend Ps (LCons g' LNil)) = the
(s (l @ [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)) (l @ [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) l"
          and "llast (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)) 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
(the (s (l@(list_of (LCons v Ps))))) LNil)) l
        = play_consistent_attacker_nonpos s (LCons v (LCons (the (s (l@[v]))) LNil))
l"
    by (metis lappend_code(1) lappend_code(2) lfinite_code(2) list_of_LCons llist_of.simps(1)
llist_of_list_of)
  have "play_consistent_attacker_nonpos s (LCons v (LCons (the (s (l@[v]))) LNil))
l" 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 (l @ list_of (LCons v Ps)))) LNil)) l
                    = play_consistent_attacker_nonpos s (LCons v (lappend Ps (LCons
(the (s (l @ [v]@list_of Ps))) LNil))) l"
    using Cons.prems(3) by auto
  have "x = list_of (ltl Ps)" using Cons.hyps(2)
    by (metis Cons.prems(3) lfinite_code(2) list.sel(3) tl_list_of)
  have "play_consistent_attacker_nonpos s (LCons (lhd Ps) (ltl Ps)) (l@[v])" using
Cons.prems(1) play_consistent_attacker_nonpos.simps
    by (smt (verit) Cons.hyps(2) eq_LConsD lhd_LCons list.discI list_of_LNil ltl_simps(2))
  have "llast (LCons (lhd Ps) (ltl Ps)) ∈ attacker" using Cons.prems(2)
    by (metis Cons.hyps(2) lhd_LCons_ltl list.distinct(1) list_of_LNil llast_LCons
llist.collapse(1))
  have "lfinite (LCons (lhd Ps) (ltl Ps))" using Cons.prems(3) 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_Cons_self2)
  hence "play_consistent_attacker_nonpos s (LCons v (lappend Ps (LCons (the (s ((l
@[v])@(list_of Ps)))) LNil))) l"
    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.sim

  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
(llength (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 (llength
(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 (llength (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 (llength (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 (llength (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 (llength (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.prems 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_llength_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.prems energy_level_nth by blast
```

```
                have "llength (LCons g p) > Suc len" using Suc.prems
                  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_de
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
(LCons g p') e)
                                               ∧ (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
(LCons g p') e)
                                               ∧ (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_strate
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
  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_lfiniteD
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
      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 "(⋀s v Ps e'.
        Q s (LCons v Ps) e' ∧ ¬ lnull Ps ⟹ (apply_w v (lhd Ps) e') ≠ None ∧
        Q s Ps (the (apply_w v (lhd Ps) e')) ∧ (v ∈ attacker ⟶ 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)) ∧ 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 ‹⋀i. lnth p i = fst (f i)› ‹¬ 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) ‹¬ 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' ∧ llast (LCons g p') = lhd p ∧ play_consistent_attacker
s p (the (energy_level e (LCons g p') (the_enat (llength p'))))
             ∧ play_consistent_attacker s (LCons g p') e ∧ valid_play (LCons g p')
∧ 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 ‹⋀i. lnth p i = fst (f i)› ‹¬ 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 ‹¬ 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
llist.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 llist.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
llist.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
              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_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 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.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
            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 ∨ 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(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
```

```
                                  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.
                                    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_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_singlet
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_prems: "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_prems
                                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'))"
                                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
  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]@(l @ [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 (l @ [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] @ l) ⟹
       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] @ l) ∧ Ps ≠ LNil ⟹
       play_consistent_attacker_nonpos (nonpos_strat_from_next g g' s) Ps ([g] @
l @ [v]) ∧ (v ∈ attacker ⟶ lhd Ps = the (s (l @ [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_de
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_l:"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_l 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_win
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
    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
(llength p) - 1) ≠ None ∧ ¬(llast p ∈ attacker ∧ deadend (llast p))" using defender_wins_play_
            by simp
          hence "lfinite (LCons g p)" by simp

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

          from  ‹lfinite (LCons g p)› have "the_enat (llength (LCons g p)) = Suc
(the_enat (llength p))"
            by (metis eSuc_enat lfinite_LCons lfinite_conv_llength_enat llength_LCons
the_enat.simps)
          hence E:"(the_enat (llength (LCons g p)) - 1) = Suc (the_enat (llength
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_llength_enat llength_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 (llength (LCons g p)) - 1)
= energy_level (the (apply_w g g' e)) p (the_enat (llength p) - 1)"
            using energy_level_cons A not False A1 E
            by (metis ‹the_enat (llength (LCons g p)) = Suc (the_enat (llength p))›
diff_Suc_1 enat_ord_simps(2) lessI lfinite_conv_llength_enat play_consistent_attacker_nonpos_co
the_enat.simps)
          hence "energy_level e (LCons g p) (the_enat (llength (LCons g p)) - 1)
≠ None" using not by auto
          then show ?thesis using defender_wins_play_def ‹lfinite (LCons g p)› ‹¬(llast
(LCons g p) ∈ attacker ∧ deadend (llast (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_win
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 ≠ [] ⟶
       last list ∈ attacker ∧ ¬ deadend (last list) ⟶
       (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 ∧
       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
      assume "list ≠ []"
      show "last list ∈ attacker ∧ ¬ deadend (last list) ⟶
     (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 ∧
    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
        assume "last list ∈ attacker ∧ ¬ deadend (last list)"
        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 ∧
    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_st
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_nonpo
        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
```

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

end
end
```

3 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 ⇒ energy ⇒ bool" (infix "e≤" 80) where
  "energy_leq e e' = ((length e = length e')
                    ∧ (∀i < length e. (e ! i) ≤ (e' ! i)))"

abbreviation energy_l:: "energy ⇒ energy ⇒ bool" (infix "e<" 80) where
  "energy_l e e' ≡  e e≤ e' ∧ e ≠ 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≤ e" using energy_leq_def by simp
  show "e e≤ e' ⟹ e' e≤ e'' ⟹ e e≤ e''" using energy_leq_def by fastforce
  show "e e< e' = e e< e'" by simp
  show "e e≤ e' ⟹ e' e≤ e ⟹ 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 ∞.

```
fun subsequence_index::"(nat ⇒ energy) ⇒ nat ⇒ nat" where
  "subsequence_index f 0 = (SOME x. (last (f x) ≠ ∞))" |
  "subsequence_index f (Suc n) = (SOME x. (last (f x) ≠ ∞
            ∧ (subsequence_index f n) < x
            ∧ (last (f (subsequence_index f n)) ≤ 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≤)"
    by (metis energy_leq.trans transp_onI)
  show "almost_full_on (e≤) {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: "∀f ∈ SEQ {e::energy. length e = n}. (∃i j. i < j ∧ (f i) e≤ (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 (smt (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 "(∃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: "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
                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 `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.prems(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 e≤ a"
proof-
  define set where "set ≡ {e. e ∈ A ∧ e 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' 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
        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' 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
```

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≤ e" using assms energy_Min_contains_smaller by
blast
    from this obtain a where "a ∈ energy_Min A" and " a e≤ 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 e≤ a" using assms energy_Min_contains_smaller by
blast
    from this obtain b where "b ∈ energy_Min B" and "b e≤ 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 ⟹ i < j
        ⟹ enumerate_arbitrary A i ≠ 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 "∃j'. j = 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 ≠ 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 "⋀e. e∈ A ⟹ 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: "(∀f ∈ SEQ (energy_Min A). (∃i j. i < j ∧ energy_leq (f i) (f j)))"
unfolding wqo_on_def almost_full_on_def good_def by simp
  have "¬ finite (energy_Min A) ⟹ False"
  proof-
    assume "¬ finite (energy_Min A)"
    hence "infinite (energy_Min A)"
      by simp

    define f where "f ≡ enumerate_arbitrary (energy_Min A)"
    have fneq: "⋀i j. f i ∈ energy_Min A ∧ (j ≠ i ⟶ f j ≠ f i)"
    proof
      fix i j
      show "f i ∈ energy_Min A" unfolding f_def using enumerate_arbitrary_in ‹infinite
(energy_Min A)› by auto
      show "j ≠ i ⟶ f j ≠ f i" proof
        assume "j ≠ i"
        show "f j ≠ 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

        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
```

4 Galois Energy Games

```
theory Galois_Energy_Game
  imports Energy_Game Energy_Order
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_legth`). 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`).

```
locale 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: "⋀p p' e e'. weight p p' ≠ None ⟹ e e≤ e'
                       ⟹ application (the (weight p p')) e ≠ None
                       ⟹ application (the (weight p p')) e' ≠ None"
    and upd_preserves_legth: "⋀p p' e. weight p p' ≠ None
                   ⟹ application (the (weight p p')) e ≠ None
                   ⟹ length (the (application (the (weight p p')) e))
                       = length e"
    and inv_preserves_length: "⋀p p' e. weight p p' ≠ None ⟹ length e = dimension

                       ⟹ length (the (inverse_application (the (weight p p')) e))

                           = length e"
    and domain_inv: "⋀p p' e. weight p p' ≠ None ⟹ length e = dimension
                     ⟹ (inverse_application (the (weight p p')) e) ≠ None
                         ∧ application (the (weight p p')) (the (inverse_application
(the (weight p p')) e)) ≠ None"
    and galois: "⋀p p' e e'. weight p p' ≠ None
                ⟹ application (the (weight p p')) e' ≠ None
                ⟹ length e = dimension ⟹ length e' = dimension
                ⟹ (the (inverse_application (the (weight p p')) e)) e≤ e'
                    = e e≤ (the (application (the (weight p p')) e'))"
begin

abbreviation "upd u e ≡ the (application u e)"
abbreviation "inv_upd u e ≡ the (inverse_application u e)"
```

We now show that the energy game being a Galois energy games implies that $u \circ u^{-1}$ is increasing, $u^{-1} \circ u$ is decreasing and u^{-1} and u are monotonic. Note that this actually is equivalent to u^{-1} and u forming a Galois connection as stated by Erné et al. [4].

```
lemma upd_inv_increasing:
  "⋀p p' e. weight p p' ≠ None ⟹ length e = dimension
  ⟹ e e≤ the (application (the (weight p p')) (the (inverse_application (the (weight
p p')) e)))"
proof-
  fix p p' e
```

```
  assume "weight p p' ≠ None"
  define u where "u= the (weight p p')"
  show "length e = dimension ⟹e e≤ the (application (the (weight p p')) (the (inverse_applica
(the (weight p p')) e)))"
  proof-
    assume "length e = dimension"
    have "inv_upd u e e≤ inv_upd u e"
      by (simp add: energy_leq.refl)

    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"
        using ‹length e = dimension› ‹weight p p' ≠ None› e'_def domain_inv u_def
by presburger
      show "length e = dimension" using ‹length e = dimension›.
      show "length e' = dimension" unfolding e'_def
        using ‹length e = dimension› ‹weight p p' ≠ None› inv_preserves_length
u_def by auto
    qed
    hence "e e≤ upd u (inv_upd u e)"
      using  ‹inv_upd u e e≤ inv_upd u e› e'_def by auto
    thus "e e≤ the (application (the (weight p p')) (the (inverse_application (the
(weight p p')) e)))"
      using u_def by auto
  qed
qed

lemma inv_upd_decreasing:
  "⋀p p' e. weight p p' ≠ None ⟹ length e = dimension
  ⟹ 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 "length e = dimension ⟹ 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 "length e = dimension" 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 ‹length e = dimension›  using ‹length e = dimension› .
      show ‹length e' = dimension› unfolding e'_def using ‹length e = dimension›
        by (simp add: ‹apply_w p p' e ≠ None› ‹weight p p' ≠ None› upd_preserves_legth
u_def)
    qed
    hence "inv_upd u (upd u e) e≤ e" using e'_def
      by (simp add: energy_leq.refl)
    thus "the (inverse_application (the (weight p p')) (the (application (the (weight
p p')) e))) e≤ e"
      using u_def by simp
```

```
    qed
  qed


  lemma updates_monotonic:
    "⋀p p' e e'. weight p p' ≠ None ⟹ length e = dimension ⟹ e 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 "length e = dimension" and "e 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≤ e'› domain_upw_closed
        using ‹weight p p' ≠ None› by blast
      show "length (upd (the (weight p p')) e) = dimension"
        using ‹length e = dimension› ‹weight p p' ≠ None› upd_preserves_legth
        using ‹apply_w p p' e ≠ None› by blast
      show "length e' = dimension"
        using ‹length e = dimension› ‹e e≤ e'›
        by (simp add: energy_leq_def)
    qed

    have "inv_upd u (upd u e) e≤ e"
      unfolding u_def using ‹weight p p' ≠ None› ‹length e = dimension› ‹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'›
      by (meson energy_leq.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

  lemma inverse_monotonic:
    "⋀p p' e e'. weight p p' ≠ None ⟹ length e = dimension ⟹ 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 "length e = dimension ⟹ 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 "length e = dimension" 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›
        by (metis ‹e e≤ e'› ‹length e = dimension› ‹weight p p' ≠ None› domain_inv
energy_leq_def u_def)
      show "length e = dimension" using ‹ length e = dimension›.
      hence "length e' = dimension"
        using ‹e e≤ e'› by (simp add: energy_leq_def)
      thus "length e'' = dimension"
        unfolding e''_def
        by (simp add: ‹weight p p' ≠ None› inv_preserves_length u_def)
    qed

    have "e' e≤ upd u e''"
      unfolding e''_def u_def using ‹weight p p' ≠ None› proof(rule upd_inv_increasing)
      from ‹length e = dimension› show "length e' = dimension"
        using ‹e e≤ e'› by (simp add: energy_leq_def)
    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_leq.trans by blast
    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
```

The set of energies is `{e::energy. length e = dimension}`. For this reason length checks are needed and we redefine attacker winning budgets.

```
inductive winning_budget_len::"energy ⇒ 'position ⇒ bool" where
 defender: "winning_budget_len e g" if "length e = dimension ∧ 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 "length e = dimension ∧ 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≤ 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 "energy_leq (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.prems by blast
        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 (smt (verit, del_insts) energy_leq_def)
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. energy_leq (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 "energy_leq (the (application (the (weight g g')) e)) (the (application
(the (weight g g')) e'))" using assms G updates_monotonic
        by (simp add: attacker.hyps attacker.prems)
      thus "winning_budget_len (the (application (the (weight g g')) e')) g' " using
G by blast
    qed
  qed
  thus ?case using winning_budget_len.intros(2) attacker by (smt (verit, del_insts)
energy_leq_def)
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 ≡ {(g',e') ∈ reachable_positions s
```

```
g e . length e' = dimension}"

lemma winning_bugget_len_is_wb:
  assumes "nonpos_winning_budget = winning_budget"
  shows "winning_budget_len e g = (winning_budget e g ∧ length e = dimension)"
proof
  assume "winning_budget_len e g"
  show "winning_budget e g ∧ length e = dimension"
  proof
    have "winning_budget_ind e g"
      using ‹winning_budget_len e g› proof(rule winning_budget_len.induct)
      show "⋀e g. length e = dimension ∧
           g ∉ attacker ∧
           (∀g'. weight g g' ≠ None ⟶
                 apply_w g g' e ≠ None ∧
                 winning_budget_len (upd (the (weight g g')) e) g' ∧
                 winning_budget_ind (upd (the (weight g g')) e) g') ⟹
           winning_budget_ind e g"
        using winning_budget_ind.simps
        by meson
      show "⋀e g. length e = dimension ∧
           g ∈ attacker ∧
           (∃g'. weight g g' ≠ None ∧
                 apply_w g g' e ≠ None ∧
                 winning_budget_len (upd (the (weight g g')) e) g' ∧
                 winning_budget_ind (upd (the (weight g g')) e) g') ⟹
           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 "length e = dimension" using ‹winning_budget_len e g› winning_budget_len.simps
by blast
  qed
next
  show "winning_budget e g ∧ length e = dimension ⟹ winning_budget_len e g"
  proof-
    assume A: "winning_budget e g ∧ length e = dimension"
    hence "winning_budget_ind e g" using assms inductive_winning_budget by fastforce
    show "winning_budget_len e g"
    proof-

      define wb where "wb ≡ λ(g,e). winning_budget_len e g"

      from A have "∃s. 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 ⊆ 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) ∈ 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))} ∧
        length e' = dimension}" using A

            by blast
        qed

        show "⋀y. y ∈ reachable_positions_len s g e ⟹
             (⋀x. x ∈ reachable_positions_len s g e ⟹ strategy_order s x y ⟹
wb x) ⟹ wb 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))))
                ∧ length e' = dimension"
            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
⟹ wb x) ⟹ wb y"
          proof-
            assume ind: "(⋀x. x ∈ reachable_positions_len s g e ⟹ strategy_order
s x y ⟹ wb x)"
            have "winning_budget_len 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 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 "¬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') ≠ 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_th
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 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: "length (upd (the (weight g' (the (s e' g')))) e') =
dimension" 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› upd_preserves_legth option.distinct(1))
          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 (metis ‹apply_w g' (the (s e' g')) e' ≠ None› ‹s e' g' ≠
None ∧ weight g' (the (s e' g')) ≠ None› upd_preserves_legth winning_budget_len.attacker
x_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
(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_sing
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_prems: "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 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_prems
                                  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: "length (upd (the (weight g' g'')) e') = dimension"
using y_len
                        using ‹apply_w g' g'' e' ≠ None› upd_preserves_legth
                        using ‹weight g' g'' ≠ None› 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 "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
```

5 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 simplifyed version of Bisping's Algorithm to calculate minimal attacker winning budgets. We further formalise the key argument for its termination.
For this we make two assumptions: First, we assume energy-positional determinacy and secondly, we assume the set of positions to be finite. Note that the first assumption is a fact lacking only a formalized proof.

```
locale galois_energy_game_decidable = 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"
begin
```

5.1 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)"
  using energy_Min_finite
  by (metis mem_Collect_eq winning_budget_len.cases)
```

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. length e = dimension}
                        ∧ (∀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. length e =
dimension} ∧
        (∀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. length e = dimension}
∧
         (∀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. length e =
dimension}"
        using winning_budget_len.simps
        by blast
      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
```

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
               ⟹ F ≼ F' ⟹ F' ≼ F ⟹ F = F'"
proof-
```

```
  fix F F' F''
  assume "F ∈ possible_pareto" and "F' ∈ possible_pareto" and "F'' ∈ possible_pareto"
  show "F ≼ F"
    unfolding pareto_order_def
    using energy_leq.refl by auto
  show "F ≼ F' ⟹ F' ≼ F'' ⟹ F ≼ F'' "
  proof-
    assume "F ≼ F'" and "F' ≼ F''"
    show " F ≼ F'' "
      unfolding pareto_order_def proof
      show "⋀g. ∀e. e ∈ F g ⟶ (∃e'. e' ∈ F'' g ∧ e' e≤ e)"
      proof
        fix g e
        show "e ∈ F g ⟶ (∃e'. e' ∈ F'' g ∧ e' e≤ e)"
        proof
          assume "e ∈ F g"
          hence "(∃e'. e' ∈ F' g ∧ e' e≤ e)" using ‹F ≼ F'› unfolding pareto_order_def
by simp
          from this obtain e' where "e' ∈ F' g ∧ e' e≤ e" by auto
          hence "(∃e''. e'' ∈ F'' g ∧ e'' e≤ e')" using ‹F' ≼ F''› unfolding pareto_order_def
by simp
          from this obtain e'' where "e'' ∈ F'' g ∧ e'' e≤ e'" by auto
          hence "e'' ∈ F'' g ∧ e'' e≤ e" using ‹e' ∈ F' g ∧ e' e≤ e› energy_leq.trans
by auto
          thus "∃e'. e' ∈ F'' g ∧ e' e≤ e" by auto
        qed
      qed
    qed
  qed
  show "F ≼ F' ⟹ F' ≼ F ⟹ F = F'"
  proof-
    assume "F ≼ F'" and "F' ≼ F"
    show "F = F'"
    proof
      fix g
      show "F g = F' g"
      proof
        show "F g ⊆ F' g"
        proof
          fix e
          assume "e ∈ F g"
          hence "∃e'. e' ∈ F' g ∧ e' e≤ e" using ‹F ≼ F'› unfolding pareto_order_def
by auto
          from this obtain e' where "e' ∈ F' g ∧ e' e≤ e" by auto
          hence "∃e''. e'' ∈ F g ∧ e'' e≤ e'" using ‹F' ≼ F› unfolding pareto_order_def
by auto
          from this obtain e'' where "e'' ∈ F g ∧ e'' e≤ e'" by auto
          hence "e'' = e ∧ e' = e" using possible_pareto_def ‹F ∈ possible_pareto›
            by (smt (verit) ‹e ∈ F g› ‹e' ∈ F' g ∧ e' e≤ e› energy_leq.strict_trans1
mem_Collect_eq)
          thus "e ∈ F' g" using ‹e' ∈ F' g ∧ e' e≤ e› by auto
        qed
        show "F' g ⊆ F g"
        proof
          fix e
          assume "e ∈ F' g"
```

```
          hence "∃e'. e' ∈ F g ∧ e' e≤ e" using ‹F' ≼ F› unfolding pareto_order_def
by auto
          from this obtain e' where "e' ∈ F g ∧ e' e≤ e" by auto
          hence "∃e''. e'' ∈ F' g ∧ e'' e≤ e'" using ‹F ≼ F'› unfolding pareto_order_def
by auto
          from this obtain e'' where "e'' ∈ F' g ∧ e'' e≤ e'" by auto
          hence "e'' = e ∧ e' = e" using possible_pareto_def ‹F' ∈ possible_pareto›
            by (smt (verit) ‹e ∈ F' g› ‹e' ∈ F g ∧ e' e≤ e› energy_leq.strict_trans1
mem_Collect_eq)
          thus "e ∈ F g" using ‹e' ∈ F g ∧ e' e≤ e› by auto
        qed
      qed
    qed
  qed
qed

lemma pareto_partial_order:
  shows "reflp_on possible_pareto (≼)" and
        "transp_on possible_pareto (≼)" and
        "antisymp_on possible_pareto (≼)"
proof-
  show "reflp_on possible_pareto (≼)"
    using reflexivity
    by (simp add: reflp_onI)
  show "transp_on possible_pareto (≼)"
    using transitivity
    using transp_onI by blast
  show "antisymp_on possible_pareto (≼)"
    using antisymmetry
    using antisymp_onI by auto
qed
```

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

```
definition pareto_sup:: "('position ⇒ energy set) set ⇒ ('position ⇒ energy set)"
where
  "pareto_sup P g = energy_Min {e. ∃F. F∈ P ∧ e ∈ F g}"

lemma pareto_sup_is_sup:
  assumes "P ⊆ possible_pareto"
  shows "pareto_sup P ∈ possible_pareto" and
        "⋀F. F ∈ P ⟹ F ≼ pareto_sup P" and
        "⋀Fs. Fs ∈ possible_pareto ⟹ (⋀F. F ∈ P ⟹ F ≼ Fs)
         ⟹ pareto_sup P ≼ Fs"
proof-
  show "pareto_sup P ∈ 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 "⋀F. F ∈ P ⟹ F ≼ 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≤ e)"
```

```
      proof
        fix g e
        show "e ∈ F g ⟶ (∃e'. e' ∈ pareto_sup P g ∧ e' e≤ e)"
        proof
          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≤ e"
            using energy_Min_contains_smaller
            by (smt (verit) Collect_mono_iff mem_Collect_eq)
          thus "∃e'. e' ∈ pareto_sup P g ∧ e' 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≤ e) "
      proof
        fix g e
        show "e ∈ pareto_sup P g ⟶ (∃e'. e' ∈ Fs g ∧ e' 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≤ e" using ‹(⋀F. F ∈ P ⟹ F ≼ Fs)› pareto_order_def
by auto
        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 (≼) ∧ P ≠ {} ⟶ (∃s. extreme_bound possible_pareto (≼) P
s)"
    proof
      assume "P ⊆ possible_pareto"
      show "directed P (≼) ∧ P ≠ {} ⟶ (∃s. extreme_bound possible_pareto (≼)
P s)"
      proof
```

```
        assume "directed P (≼) ∧ P ≠ {}"
        show "∃s. extreme_bound possible_pareto (≼) P s"
        proof
          show "extreme_bound possible_pareto (≼) P (pareto_sup P)"
            unfolding extreme_bound_def
          proof
            show "pareto_sup P ∈ {b ∈ possible_pareto. bound P (≼) b}"
              using pareto_sup_is_sup ‹P ⊆ possible_pareto› ‹directed P (≼) ∧
P ≠ {}›
              by blast
            show "⋀x. x ∈ {b ∈ possible_pareto. bound P (≼) b} ⟹ pareto_sup
P ≼ x"
            proof-
              fix x
              assume "x ∈ {b ∈ possible_pareto. bound P (≼) b}"
              thus "pareto_sup P ≼ x"
                using pareto_sup_is_sup ‹P ⊆ possible_pareto› ‹directed P (≼)
∧ P ≠ {}›
                by auto
            qed
          qed
        qed
      qed
    qed
  qed
qed

lemma pareto_minimal_element:
  shows "(λg. {}) ≼ F"
  unfolding pareto_order_def by simp
```

5.2 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 ⇒ energy set) ⇒ ('position ⇒ energy set)"
where
  "iteration F g ≡ (if g ∈ attacker
                    then energy_Min {inv_upd (the (weight g g')) e' | e' g'.
                        length e' = dimension ∧ weight g g' ≠ None ∧ e' ∈ F g'}
                    else energy_Min {energy_sup dimension
                         {inv_upd (the (weight g g')) (e_index g') | g'.
                         weight g g' ≠ None} | e_index. ∀g'. weight g g' ≠ None
                         ⟶ length (e_index g')= dimension ∧ e_index g' ∈ 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. length e = dimension} ∧
        (∀e e'. e ∈ iteration F g ∧ e' ∈ iteration F g ∧ e ≠ e' ⟶ incomparable
(e≤) e e')"
```

```
  proof
    fix g
    show "iteration F g ⊆ {e. length e = dimension} ∧
        (∀e e'. e ∈ iteration F g ∧ e' ∈ iteration F g ∧ e ≠ e' ⟶ incomparable
(e≤) e e')"
    proof
      show "iteration F g ⊆ {e. length e = dimension}"
      proof
        fix e
        assume "e ∈ iteration F g"
        show "e ∈ {e. length e = dimension}"
        proof
          show "length e = dimension"
          proof(cases "g ∈ attacker")
            case True
            hence "e ∈ energy_Min {inv_upd (the (weight g g')) e' | e' g'. length
e' = dimension ∧ weight g g' ≠ None ∧ e' ∈ F g'}"
              using ‹e ∈ iteration F g› iteration_def by auto
            then show ?thesis using assms energy_Min_def inv_preserves_length
              by force
          next
            case False
            hence "e ∈ energy_Min {energy_sup dimension {inv_upd (the (weight g
g')) (e_index g')| g'. weight g g' ≠ None}| e_index. (∀g'. weight g g' ≠ None ⟶
(length (e_index g')= dimension ∧ e_index g' ∈ F g'))}"
              using ‹e ∈ iteration F g› iteration_def by auto
            then show ?thesis unfolding energy_sup_def using energy_Min_def
              using Ex_list_of_length by force
          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≤ e)"
  proof
    fix e
    show "e ∈ iteration F g ⟶ (∃e'. e' ∈ iteration F' g ∧ e' e≤ e)"
    proof
      assume "e ∈ iteration F g"
      show "(∃e'. e' ∈ iteration F' g ∧ e' e≤ e)"
      proof(cases"g∈ attacker")
        case True
        hence "e ∈ energy_Min {inv_upd (the (weight g g')) e' | e' g'. length e'
= dimension ∧ 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' ∧ length
e' = dimension ∧ weight g g' ≠ None ∧ e' ∈ F g'"
          using energy_Min_def by auto
        hence "∃e''. e'' ∈ F' g' ∧ e'' e≤ e'" using pareto_order_def assms by simp
        from this obtain e'' where "e'' ∈ F' g' ∧ e'' e≤ e'" by auto
        hence uE: "inv_upd (the (weight g g')) e'' e≤  inv_upd (the (weight g g'))
e'"
          using E inverse_monotonic domain_inv
          using energy_leq_def by blast
        hence "inv_upd (the (weight g g')) e'' ∈ {inv_upd (the (weight g g')) e'
| e' g'. length e' = dimension ∧ weight g g' ≠ None ∧ e' ∈ F' g'}"
          using E iteration_def True ‹e'' ∈ F' g' ∧ e'' e≤ e'›
          using energy_leq_def by blast
        hence "∃e'''. e'''∈ energy_Min {inv_upd (the (weight g g')) e' | e' g'.
length e' = dimension ∧ weight g g' ≠ None ∧ e' ∈ F' g'} ∧ e''' e≤ inv_upd (the
(weight g g')) e''"
          using energy_Min_contains_smaller
          by meson
        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
          using energy_leq.trans by blast
        then show ?thesis using E''' by auto
      next
        case False
        hence "e∈ (energy_Min {energy_sup dimension {inv_upd (the (weight g g'))
(e_index g')| g'. weight g g' ≠ None}| e_index. (∀g'. weight g g' ≠ None ⟶ (length
(e_index g')= dimension ∧ e_index g' ∈ F g'))})"
          using iteration_def ‹e ∈ iteration F g› by simp
        from this obtain e_index where E: "e= energy_sup dimension {inv_upd (the
(weight g g')) (e_index g')| g'. weight g g' ≠ None}" and "(∀g'. weight g g' ≠
None ⟶ (length (e_index g')= dimension ∧ 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 ⟶ length (e_index g') = dimension ∧
e_index g' ∈ F g'›
          by (simp add: domain_inv energy_leq_def)
        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 e≤ b"
          by blast
```

```
        have len: "⋀a. a∈ {inv_upd (the (weight g g')) (e_index' g')| g'. weight
g g' ≠ None} ⟹ length a = dimension"
          using  E' E inv_preserves_length
          using ‹∀g'. weight g g' ≠ None ⟶ length (e_index g') = dimension ∧
e_index g' ∈ F g'› energy_leq_def by force
        hence leq: "energy_sup dimension {inv_upd (the (weight g g')) (e_index'
g')| g'. weight g g' ≠ None} e≤ energy_sup dimension {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 using empty_Sup_is_zero
            using energy_leq.refl by fastforce
        next
          case False
          hence "{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
            by meson
        qed

        have "⋀g'. weight g g' ≠ None ⟹ length (e_index' g') = dimension" using
E' ‹∀g'. weight g g' ≠ None ⟶ length (e_index g') = dimension ∧ e_index g' ∈
F g'›
          by (simp add: energy_leq_def)
        hence "energy_sup dimension {inv_upd (the (weight g g')) (e_index' g')|
g'. weight g g' ≠ None} ∈ {energy_sup dimension {inv_upd (the (weight g g')) (e_index
g')| g'. weight g g' ≠ None}| e_index. (∀g'. weight g g' ≠ None ⟶ (length (e_index
g')= dimension ∧ e_index g' ∈ F' g'))}"
          using E'
          by blast
        hence "∃e'. e' ∈ energy_Min {energy_sup dimension {inv_upd (the (weight
g g')) (e_index g')| g'. weight g g' ≠ None}| e_index. (∀g'. weight g g' ≠ None
⟶ (length (e_index g')= dimension ∧ e_index g' ∈ F' g'))}
            ∧ e' e≤ energy_sup dimension {inv_upd (the (weight g g')) (e_index'
g')| g'. weight g g' ≠ None}"
          using energy_Min_contains_smaller
          by meson
        hence "∃e'. e' ∈ iteration F' g ∧ e' e≤ energy_sup dimension {inv_upd (the
(weight g g')) (e_index' g')| g'. weight g g' ≠ None} "
          unfolding iteration_def using False by auto
        then show ?thesis using leq E
          using energy_leq.trans by blast
      qed
    qed
  qed
qed

lemma finite_directed_set_upper_bound:
  assumes "⋀F F'. F ∈ P ⟹ F' ∈ P ⟹ ∃F''. F'' ∈ P ∧ F ≼ F'' ∧ F' ≼ F''"
          and "P ≠ {}" and "P' ⊆ P" and "finite P'" and "P ⊆ possible_pareto"
  shows "∃F'. F' ∈ P ∧ (∀F. F ∈ P' ⟶ F ≼ F')"
  using assms proof(induct "card P'" arbitrary: P')
  case 0
```

```
  then show ?case
    by auto
next
  case (Suc x)
  hence "∃F. F ∈ P'"
    by auto
  from this obtain F where "F ∈ P'" by auto
  define P'' where "P'' = P' - {F}"
  hence "card P'' = x" using Suc card_Suc_Diff1 ‹F ∈ P'› by simp
  hence "∃F'. F' ∈ P ∧ (∀F. F ∈ P'' ⟶ F ≼ F')" using Suc
    using P''_def by blast
  from this obtain F' where "F' ∈ P ∧ (∀F. F ∈ P'' ⟶ F ≼ F')" by auto
  hence "∃F''. F'' ∈ P ∧ F ≼ F'' ∧ F' ≼ F''" using Suc
    by (metis (no_types, lifting) ‹F ∈ P'› in_mono)
  from this obtain F'' where "F'' ∈ P ∧ F ≼ F'' ∧ F' ≼ F''" by auto
  show ?case
  proof
    show "F'' ∈ P ∧ (∀F. F ∈ P' ⟶ F ≼ F'')"
    proof
      show "F'' ∈ P" using ‹F'' ∈ P ∧ F ≼ F'' ∧ F' ≼ F''› by simp
      show "∀F. F ∈ P' ⟶ F ≼ F''"
      proof
        fix F0
        show "F0 ∈ P' ⟶ F0 ≼ F''"
        proof
          assume "F0 ∈ P'"
          show "F0 ≼ F''"
          proof(cases "F0 = F")
            case True
            then show ?thesis using ‹F'' ∈ P ∧ F ≼ F'' ∧ F' ≼ F''› by simp
          next
            case False
            hence "F0 ∈ P''" using P''_def ‹F0 ∈ P'› by auto
            hence "F0 ≼ F'" using ‹F' ∈ P ∧ (∀F. F ∈ P'' ⟶ F ≼ F')› by simp
            then show ?thesis using ‹F'' ∈ P ∧ F ≼ F'' ∧ F' ≼ F''› transitivity
Suc
              by (smt (z3) ‹F' ∈ P ∧ (∀F. F ∈ P'' ⟶ F ≼ F')› ‹F0 ∈ P'› subsetD)

          qed
        qed
      qed
    qed
  qed
qed

lemma iteration_scott_continuous_vanilla:
  assumes "P ⊆ possible_pareto" and
          "⋀F F'. F ∈ P ⟹ F' ∈ P ⟹ ∃F''. F'' ∈ P ∧ F ≼ F'' ∧ F' ≼ 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≤ e)"
    proof
      fix e
      show "e ∈ iteration (pareto_sup P) g ⟶
                  (∃e'. e' ∈ pareto_sup {iteration F |F. F ∈ P} g ∧ e' e≤ e)"
      proof
        assume "e ∈ iteration (pareto_sup P) g"
        show "∃e'. e' ∈ pareto_sup {iteration F |F. F ∈ P} g ∧ e' e≤ e"
        proof(cases "g ∈ attacker")
          case True
          hence "e ∈ energy_Min {inv_upd (the (weight g g')) e' | e' g'. length
e' = dimension ∧ 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
"length e' = dimension ∧ 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'. length e' = dimension
∧ weight g g' ≠ None ∧ e' ∈ F g'}" using ‹e = inv_upd (the (weight g g')) e'›
            using ‹length e' = dimension ∧ weight g g' ≠ None ∧ e' ∈ pareto_sup
P g'› by blast

          hence "∃e''. e'' ∈ energy_Min {inv_upd (the (weight g g')) e' | e' g'.
length e' = dimension ∧ weight g g' ≠ None ∧ e' ∈ F g'} ∧ e'' e≤ e"
            using energy_Min_contains_smaller
            by meson
          hence "∃e''. e'' ∈ iteration F g ∧ e'' e≤ e" using True iteration_def
by simp
          from this obtain e'' where "e'' ∈ iteration F g ∧ e'' e≤ e" by auto
          hence "∃e'''. e''' ∈ pareto_sup {iteration F |F. F ∈ P} g ∧ e''' e≤
e''"
            unfolding pareto_sup_def energy_Min_contains_smaller
            by (metis (mono_tags, lifting) ‹F ∈ P ∧ e' ∈ F g'› energy_Min_contains_smaller
mem_Collect_eq)
          then show ?thesis
            using ‹e'' ∈ iteration F g ∧ e'' e≤ e› energy_leq.trans by blast
        next
          case False
          hence "e ∈ energy_Min {energy_sup dimension {inv_upd (the (weight g g'))
(e_index g')| g'. weight g g' ≠ None}| e_index. (∀g'. weight g g' ≠ None ⟶ (length
```

```
(e_index g')= dimension ∧ 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 dimension {inv_upd (the
(weight g g')) (e_index g')| g'. weight g g' ≠ None}" and "(∀g'. weight g g' ≠
None ⟶ (length (e_index g')= dimension ∧ 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 ≡ λg'. 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 dimension {inv_upd (the (weight g g')) (e_index' g')|
g'. weight g g' ≠ None} e≤ energy_sup dimension {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_leq_def using empty_Sup_is_zero ‹{inv_upd
(the (weight g g')) (e_index' g')| g'. weight g g' ≠ None} = {}›
              by (simp add: order_refl)
          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 (simp add: energy_leq.refl)
                have "length (e_index' g') = dimension"
                  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 energy_leq.refl by blast
                thus "∃b∈{inv_upd (the (weight g g')) (e_index g') |g'. weight
g g' ≠ None}. a e≤ b"
                  by (smt (verit, best) F ‹⋀g'. weight g g' ≠ None ⟹ e_index
g' ∈ pareto_sup P g'› ‹⋀g'. weight g g' ≠ None ⟹ e_index' g' ∈ F g' ∧ e_index'
g' e≤ e_index g'› ‹pareto_sup P ∈ possible_pareto› ‹weight g g' ≠ None› assms
energy_leq.strict_trans1 mem_Collect_eq pareto_order_def pareto_sup_is_sup(2) possible_pareto_d

              qed
              show "⋀a. a ∈ {inv_upd (the (weight g g')) (e_index' g') |g'. weight
g g' ≠ None} ⟹
                    length a = dimension"
              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 "length (e_index' g') = dimension" using ‹F ∈ possible_pareto›
possible_pareto_def
                     by blast
                   thus "length a = dimension" using ‹a=inv_upd (the (weight g g'))
(e_index' g')› inv_preserves_length ‹weight g g' ≠ None› by blast
               qed
             qed
           qed
           hence leq: "energy_sup dimension {inv_upd (the (weight g g')) (e_index'
g')| g'. weight g g' ≠ None} e≤ e"
             using ‹e= energy_sup dimension {inv_upd (the (weight g g')) (e_index
g')| g'. weight g g' ≠ None}› by simp

           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 ⟹ length (e_index' g') = dimension"
using ‹F ∈ possible_pareto› possible_pareto_def
             by blast
           hence "(energy_sup dimension {inv_upd (the (weight g g')) (e_index' g')|
g'. weight g g' ≠ None}) ∈ {energy_sup dimension
             {inv_upd (the (weight g g')) (e_index g') |g'. weight g g' ≠ None} |
            e_index.
            ∀g'. weight g g' ≠ None ⟶ length (e_index g') = dimension ∧ 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 dimension {inv_upd
(the (weight g g')) (e_index' g')| g'. weight g g' ≠ None})"
             unfolding iteration_def using energy_Min_contains_smaller False
             by meson
           from this obtain e' where E': "e' ∈ iteration F g ∧ e' e≤ (energy_sup
dimension {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≤ e'"
             unfolding pareto_sup_def using energy_Min_contains_smaller by meson
           from this obtain a where "a ∈ pareto_sup {iteration F |F. F ∈ P} g ∧
a e≤ e'" by auto
           hence "a e≤ e" using E' leq
             using energy_leq.trans by blast
           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 "length e' = dimension" using winning_budget_len.simps
          by blast
        show "length (the (inverse_application (the (weight g g')) e')) = dimension
∧ 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 "length (the (inverse_application (the (weight g g')) e')) = dimension"
using ‹length e' = dimension› A2
            by (simp add: inv_preserves_length)
          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 domain_inv
                    by (simp add: ‹length e' = dimension›)
                  have "energy_leq e' (the (application (the (weight g g')) (the
(inverse_application (the (weight g g')) e'))))" using upd_inv_increasing
                    using A2 ‹length e' = dimension› 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

    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 ‹length e = dimension› using winning_budget_len.simps by blast
        from A1 A2 ‹winning_budget_len 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') )"
          using winning_budget_len.simps
          by blast
        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')" by auto
        hence "length (the (application (the (weight g g')) e)) = dimension"
          using ‹length e = dimension› upd_preserves_legth 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 "energy_leq (the (inverse_application (the (weight g g')) (the (application
(the (weight g g')) e)))) e" using inv_upd_decreasing
          using G
          using ‹length e = dimension› 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 "∃e'. weight g g' ≠ None ∧ winning_budget_len e' g' ∧ e = the (inverse_applicat
(the (weight g g')) e') "
          proof
            show "weight g g' ≠ None ∧ winning_budget_len (the (application (the
(weight g g')) e)) g' ∧ 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
    ⟷ e ∈ energy_Min {e. ∃g' e'. weight g g' ≠ None ∧ winning_budget_len e'
g' ∧ e=(inv_upd (the (weight g g')) e')}"
    proof-
      have len: "⋀e. e∈ {e. ∃g' e'. weight g g' ≠ None ∧ winning_budget_len e'
g'  ∧ e=(the (inverse_application (the (weight g g')) e'))} ⟹ length e = dimension"
      proof-
        fix e
        assume "e∈ {e. ∃g' e'. weight g g' ≠ None ∧ winning_budget_len e' g'  ∧
e=(the (inverse_application (the (weight g g')) e'))}"
        hence "∃g' e'. weight g g' ≠ None ∧ winning_budget_len e' g'  ∧ e=(the
(inverse_application (the (weight g g')) e'))" by auto
        from this obtain g' e' where eg: "weight g g' ≠ None ∧ winning_budget_len
e' g'  ∧ e=(the (inverse_application (the (weight g g')) e'))" by auto
        hence "weight g g' ≠ None" by auto
        from eg have "length e' = dimension" using winning_budget_len.simps by blast

        thus "length e = dimension" using eg ‹length e' = dimension›
          using inv_preserves_length by blast
      qed

      show ?thesis
      proof
        assume "minimal_winning_budget e g"
        hence A: "winning_budget_len e g ∧ (∀e'. e' ≠ e ⟶ e' e≤ e ⟶ ¬ winning_budget_len
e' g)" using energy_Min_def by auto
        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 min_winning_budget_is_inv_a ‹g ∈ attacker›
          by (simp add: ‹minimal_winning_budget e g›)

        have "⋀x. x ∈ {e. ∃g' e'. weight g g' ≠ None ∧ winning_budget_len e' g'
∧ e=(the (inverse_application (the (weight g g')) e'))} ∧ energy_leq x e ⟹ e=x"
        proof-
          fix x
          assume X: "x ∈ {e. ∃g' e'. weight g g' ≠ None ∧ winning_budget_len e'
g' ∧ e=(the (inverse_application (the (weight g g')) e'))} ∧ energy_leq x e"
          have "winning_budget_len x g"
          proof
            show "length x = dimension ∧
              g ∈ attacker ∧
              (∃g'. weight g g' ≠ None ∧
```

```
            application (the (weight g g')) x ≠ None ∧ winning_budget_len (the
(application (the (weight g g')) x)) g')"
            proof
              show "length x = dimension" using len X by blast
              show "g ∈ attacker ∧
                (∃g'. weight g g' ≠ None ∧
                application (the (weight g g')) x ≠ None ∧ winning_budget_len (the
(application (the (weight g g')) x)) g')"
              proof
                show "g ∈ attacker" using ‹g ∈ attacker›.

                from X have "∃g' e'.
              weight g g' ≠ None ∧
              winning_budget_len e' g' ∧ x = inv_upd (the (weight g g')) e'"
                  by blast
                from this obtain g' e' where X: "weight g g' ≠ None ∧
              winning_budget_len e' g' ∧ x = inv_upd (the (weight g g')) e'" by
auto

                show "∃g'. weight g g' ≠ None ∧
         apply_w g g' x ≠ None ∧ winning_budget_len (upd (the (weight g g')) x)
g'"
                proof
                  show "weight g g' ≠ None ∧
         apply_w g g' x ≠ None ∧ winning_budget_len (upd (the (weight g g')) x)
g'"
                  proof
                    show "weight g g' ≠ None" using X by simp
                    show "apply_w g g' x ≠ None ∧ winning_budget_len (upd (the
(weight g g')) x) g'"
                    proof

                      have "e' e≤ (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 domain_inv  ‹weight g g' ≠ None›
                        by (metis X winning_budget_len.simps)
                      thus "apply_w g g' x ≠ None"
                        using X domain_inv
                        using nonpos_eq_pos winning_bugget_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 ⟶ energy_leq e' e ⟶ ¬ winning_budget_
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''. energy_leq e'' e ∧ winning_budget_len e'' g}" using
energy_leq.refl by simp
          hence notempty: "{} ≠ {e''. energy_leq e'' e ∧ winning_budget_len e''
g}" by auto
          have "⋀e''. e'' ∈ {e''. energy_leq e'' e ∧ winning_budget_len e'' g}
⟹ length e'' = dimension"
            using winning_budget_len.simps by blast
          hence "{} ≠ energy_Min {e''. energy_leq e'' e ∧ winning_budget_len e''
g}" using energy_Min_not_empty notempty
            by (smt (verit, ccfv_threshold) emptyE mem_Collect_eq)
          hence "∃e''. e'' ∈ energy_Min {e''. energy_leq e'' e ∧ winning_budget_len
e'' g}" by auto
          from this obtain e'' where "e'' ∈ energy_Min {e''. energy_leq e'' e ∧
winning_budget_len e'' g}" by auto
          hence X: "energy_leq e'' e ∧ winning_budget_len e'' g ∧ (∀e'. e'∈ {e''.
energy_leq e'' e ∧ winning_budget_len e'' g} ⟶ e'' ≠ e' ⟶ ¬ energy_leq e' e'')"
            using energy_Min_def by simp

          have "(∀e' ≠ e''. energy_leq e' e'' ⟶ ¬ winning_budget_len e' g)"
          proof
            fix e'
            show " e' ≠ e'' ⟶ energy_leq e' e'' ⟶ ¬ winning_budget_len e' g"
            proof
              assume " e' ≠ e''"
              show "energy_leq e' e'' ⟶ ¬ winning_budget_len e' g"
              proof
                assume " energy_leq e' e''"
                hence "length e' = dimension " using energy_leq_def X
                  using ‹⋀e''. e'' ∈ {e''. energy_leq e'' e ∧ winning_budget_len
e'' g} ⟹ length e'' = dimension› by blast
                from ‹energy_leq e' e''› have "energy_leq e' e" using X energy_leq.trans
by blast
                show "¬ winning_budget_len e' g"
                proof
                  assume "winning_budget_len e' g"
                  hence "e'∈{e''. energy_leq e'' e ∧ winning_budget_len e'' g ∧
length e'' = dimension}" using ‹length e' = dimension› ‹energy_leq e' e› by auto
```

```
                  hence "¬ energy_leq e' e''" using X ‹e' ≠ e''› by simp
                  thus "False" using ‹energy_leq e' e''› by simp
                qed
              qed
            qed
          qed
          hence E: "energy_leq e'' e ∧ winning_budget_len e'' g ∧ (∀e' ≠ e''. energy_leq
e' e'' ⟶ ¬ winning_budget_len e' g)" using X
            by meson
          hence "energy_leq e'' e ∧ minimal_winning_budget e'' g" 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 min_winning_budget_is_inv_a X ‹g ∈ attacker› by simp
          hence "e'' ∈ {e. ∃g' e'. weight g g' ≠ None ∧ winning_budget_len e' g'
∧ e=(the (inverse_application (the (weight g g')) e'))}" by auto
          hence "e=e''" using ‹g ∈ attacker› X energy_Min_def E
            by (smt (verit, best) ‹e ∈ energy_Min {e. ∃g' e'. weight g g' ≠ None
∧ winning_budget_len e' g' ∧ e = the (inverse_application (the (weight g g')) e')}›
mem_Collect_eq)
          thus "(∀e'. e' ≠ e ⟶ energy_leq e' e ⟶ ¬ 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: "⋀e g. g ∈ attacker ⟹ minimal_winning_budget
e g ⟹ ∃g' e'. weight g g' ≠ None ∧ minimal_winning_budget 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 ∧ minimal_winning_budget 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
        from A1 A2 have "∀e' ≠ e. energy_leq e' e ⟶ ¬ 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' ∧ energy_leq e e'}" using energy_leq.refl
by auto
        have "⋀e'. e' ∈ {e. winning_budget_len e g' ∧ energy_leq e e'} ⟹ length
e' = dimension" using winning_budget_len.simps by blast
        hence "energy_Min {e. winning_budget_len e g' ∧ energy_leq e e'} ≠ {}"
using ‹e' ∈ {e. winning_budget_len e g' ∧ energy_leq e e'}› energy_Min_not_empty
          by (metis (no_types, lifting) empty_iff mem_Collect_eq winning_budget_len.cases)
        hence "∃e''. e'' ∈ energy_Min {e. winning_budget_len e g' ∧ energy_leq
e e'}" by auto
        from this obtain e'' where "e'' ∈ energy_Min {e. winning_budget_len e g'
∧ energy_leq e e'}" by auto
```

```
        hence "minimal_winning_budget e'' g'" using energy_Min_def
          by (smt (verit, del_insts) energy_leq.strict_trans1 mem_Collect_eq)

        have "energy_leq e'' e'" using ‹e'' ∈ energy_Min {e. winning_budget_len
e g' ∧ energy_leq e e'}› energy_Min_def by auto
        hence "energy_leq (the (inverse_application (the (weight g g')) e'')) (the
(inverse_application (the (weight g g')) e'))"
          using inverse_monotonic
          using G domain_inv energy_leq_def nonpos_eq_pos winning_bugget_len_is_wb
by auto
        hence "energy_leq (the (inverse_application (the (weight g g')) e'')) e"
using G by auto
        hence "e=(the (inverse_application (the (weight g g')) e''))" using ‹minimal_winning_bu
e'' g'› ‹∀e' ≠ e. energy_leq e' e ⟶ ¬ winning_budget_len e' g›
          by (metis (no_types, lifting) G A1 attacker_inv_in_winning_budget energy_Min_def
mem_Collect_eq)
        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 meson
        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 "length e_min = dimension" using winning_budget_len.simps
            by (metis E'G' energy_leq_def)
```

```
            thus " inverse_application (the (weight g g')) e_min ≠ None"
              using domain_inv ‹weight g g' ≠ None› by auto
            show "length e_min = dimension"
              by (simp add: ‹length e_min = dimension›)
          qed
          hence "inv_upd (the (weight g g')) e_min e< e" using ‹e'' e< e› E'G'
            using energy_leq.trans
            by (metis energy_leq.asym)

          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 dimension {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 dimension {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 dimension {strat g'| g'. weight g g'
≠ None}) g"
      proof (rule winning_budget_len.intros(1))
        have "(∀g'. weight g g' ≠ None ⟶
          application (the (weight g g')) (energy_sup dimension {strat g' |g'. weight
```

```
g g' ≠ None}) ≠ None ∧
          winning_budget_len (the (application (the (weight g g')) (energy_sup dimension
{strat g' |g'. weight g g' ≠ None}))) g') "
        proof
          fix g'
          show "weight g g' ≠ None ⟶
          application (the (weight g g')) (energy_sup dimension {strat g' |g'. weight
g g' ≠ None}) ≠ None ∧
          winning_budget_len (the (application (the (weight g g')) (energy_sup dimension
{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 "length e = dimension" using winning_budget_len.simps by blast
            hence "inverse_application (the (weight g g')) e ≠ None" using domain_inv
‹weight g g' ≠ None› by simp

            have leq: "energy_leq (strat g') (energy_sup dimension {strat g' |g'.
weight g g' ≠ None})" using energy_sup_in ‹weight g g' ≠ None›
              by (metis (mono_tags, lifting) E ‹length e = dimension› inv_preserves_length
mem_Collect_eq )

            show "application (the (weight g g')) (energy_sup dimension {strat g'
|g'. weight g g' ≠ None}) ≠ None ∧
            winning_budget_len (the (application (the (weight g g')) (energy_sup
dimension {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› domain_inv
                using ‹length e = dimension› ‹weight g g' ≠ None› by presburger
              finally have "application (the (weight g g')) (strat g') ≠ None" .
              thus "application (the (weight g g')) (energy_sup dimension {strat
g' |g'. weight g g' ≠ None}) ≠ None"
                using leq domain_upw_closed
                using ‹weight g g' ≠ None› by blast

              have "energy_leq 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))› ‹length
e = dimension› ‹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 "energy_leq (the (application (the (weight g g')) (strat g')))
(the (application (the (weight g g')) (energy_sup dimension {strat g' |g'. weight
g g' ≠ None})))"
```

```
                using updates_monotonic
                by (smt (verit, del_insts) Collect_cong E ‹application (the (weight
g g')) (strat g') ≠ None› ‹length e = dimension› ‹weight g g' ≠ None› inv_preserves_length
leq)
              thus "winning_budget_len (the (application (the (weight g g')) (energy_sup
dimension {strat g' |g'. weight g g' ≠ None}))) g'"
                using W upwards_closure_wb_len by blast
            qed
          qed
        qed

        thus "length (energy_sup dimension {strat g' |g'. weight g g' ≠ None}) =
dimension ∧ g ∉ attacker ∧
          (∀g'. weight g g' ≠ None ⟶
          application (the (weight g g')) (energy_sup dimension {strat g' |g'. weight
g g' ≠ None}) ≠ None ∧
          winning_budget_len (the (application (the (weight g g')) (energy_sup dimension
{strat g' |g'. weight g g' ≠ None}))) g') "
          using A1 energy_sup_def
          by (simp add: Ex_list_of_length length_map map_nth)
      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 dimension {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'})
                  ∧ e = (energy_sup dimension {strat g'| g'. weight g g' ≠ None})"
      proof-
        from A2 have "length e =dimension" using winning_budget_len.simps energy_Min_def
          by (metis (no_types, lifting) mem_Collect_eq)
        from A1 A2 have W: "(∀g'. weight g g' ≠ None ⟶
                  application (the (weight g g')) e ≠ None ∧
                  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: "∀g'.  strat g' = the ((inverse_application (the (weight
g g'))) (the (application (the (weight g g')) e)))"
        have A: "(∀g'. weight g g' ≠ None ⟶ strat g' ∈ {the (inverse_application
(the (weight g g')) e) |e. winning_budget_len e g'})"
        proof
          fix g'
          show "weight g g' ≠ None ⟶ strat g' ∈ {the (inverse_application (the
(weight g g')) e) |e. winning_budget_len e g'}"
          proof
            assume "weight g g' ≠ None"
            hence "winning_budget_len (the (application (the (weight g g')) e))
g'" using W by auto
            thus "strat g' ∈ {the (inverse_application (the (weight g g')) e) |e.
winning_budget_len e g'}" using S by blast
```

```
          qed
        qed
        hence W: "winning_budget_len (energy_sup dimension {strat g' |g'. weight
g g' ≠ None}) g" using sup_inv_in_winning_budget A1 by simp
        have "⋀g'. weight g g' ≠ None ⟹ energy_leq (strat g') e"
        proof-
          fix g'
          assume "weight g g' ≠ None"
          hence "application (the (weight g g')) e ≠ 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' ≠ None› have "strat g' = the ((inverse_application
(the (weight g g'))) (the (application (the (weight g g')) e)))" using S by auto
          thus "energy_leq (strat g') e" using inv_upd_decreasing  ‹application
(the (weight g g')) e ≠ None›
            using ‹length e = dimension› ‹weight g g' ≠ None› by presburger
        qed
        hence "energy_leq (energy_sup dimension {strat g' |g'. weight g g' ≠ None})
e" using energy_sup_leq ‹length e =dimension›
          by (smt (verit) mem_Collect_eq)
        hence "e = energy_sup dimension {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 ⟹ length e =
dimension ⟹ ((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 dimension {strat g'| g'. weight g g' ≠ None})})
        ⟷ minimal_winning_budget e g)"
    proof-
      fix e g
      show "g ∉ attacker ⟹
          length e = dimension ⟹
          (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 dimension {strat g' |g'. weight g g' ≠ None}})
=
          minimal_winning_budget e g"
      proof-
        assume A1: "g ∉ attacker" and A2: "length e = dimension"
        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 dimension {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'})
                 ∧ e'' = (energy_sup dimension {strat g'| g'. weight g g' ≠ None})}"
          show "minimal_winning_budget e g"
            unfolding energy_Min_def
          proof
            show "e ∈ {e. winning_budget_len e g} ∧ (∀e'∈{e. winning_budget_len
e g}. e ≠ e' ⟶ ¬ e' e≤ e)"
            proof
              show "e ∈ {e. winning_budget_len e g}"
              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 dimension {strat g'| g'. weight g g' ≠ 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 "length e = dimension" using winning_budget_len.simps by blast
              hence "e∈ {e''. energy_leq e'' e ∧ winning_budget_len e'' g ∧ length
e'' = dimension}" using W energy_leq.refl ‹g ∉ attacker› by simp
              hence "{e''. energy_leq e'' e ∧ winning_budget_len e'' g ∧ length
e'' = dimension} ≠ {}" by auto
              hence "energy_Min {e''. energy_leq e'' e ∧ winning_budget_len e''
g ∧ length e'' = dimension} ≠ {}" using energy_Min_not_empty
                by (metis (mono_tags, lifting) mem_Collect_eq)
              hence "∃e''. e'' ∈ energy_Min {e''. energy_leq e'' e ∧ winning_budget_len
e'' g ∧ length e'' = dimension}" by auto
              from this obtain e'' where "e'' ∈ energy_Min {e''. energy_leq e''
e ∧ winning_budget_len e'' g ∧ length e'' = dimension}" by auto
              hence X: "energy_leq e'' e ∧ winning_budget_len e'' g ∧ length e''
= dimension
                      ∧ ( ∀e'. e'∈{e''. energy_leq e'' e ∧ winning_budget_len e''
g ∧ length e'' = dimension }⟶ e'' ≠ e' ⟶ ¬ energy_leq e' e'')" using energy_Min_def
                by simp
              have "(∀e' ≠ e''. energy_leq e' e'' ⟶ ¬ winning_budget_len e' g)"

              proof
                fix e'
                show " e' ≠ e'' ⟶ energy_leq e' e'' ⟶ ¬ winning_budget_len
e' g"
                proof
                  assume " e' ≠ e''"
                  show "energy_leq e' e'' ⟶ ¬ winning_budget_len e' g"
                  proof
                    assume " energy_leq e' e''"
                    hence "length e' = dimension " using energy_leq_def X by auto
                    from ‹energy_leq e' e''› have "energy_leq e' e" using X energy_leq.trans
```

```
by blast
                    show "¬ winning_budget_len e' g"
                    proof
                      assume "winning_budget_len e' g"
                      hence "e'∈{e''. energy_leq e'' e ∧ winning_budget_len e''
g ∧ length e'' = dimension}" using ‹length e' = dimension› ‹energy_leq e' e› by
auto
                      hence "¬ energy_leq e' e''" using X ‹e' ≠ e''› by simp
                      thus "False" using ‹energy_leq e' e''› by simp
                    qed
                  qed
                qed
              qed
              hence "energy_leq e'' e ∧ winning_budget_len e'' g ∧ (∀e' ≠ e''.
energy_leq e' e'' ⟶ ¬ winning_budget_len e' g)" using X
                by meson
              hence E: "energy_leq 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 dimension {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 dimension {strat g'| g'. weight g g' ≠ None})}"
            unfolding energy_Min_def
          proof
            from assumption have "length e = dimension" 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 dimension {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 dimension {strat g' |g'. weight g g' ≠ None}}.
          e ≠ e' ⟶ ¬ energy_leq 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 dimension {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 dimension {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 dimension {strat g' |g'. weight g g' ≠ None}}.
       e ≠ e' ⟶ ¬ energy_leq 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 dimension {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 dimension {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 dimension {strat g' |g'. weight g g' ≠ None}"
by auto
                  hence "length e'  = dimension" using energy_sup_def
                    by (simp add: length_map)
                  show "e ≠ e' ⟶ ¬ energy_leq 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 "length e''=dimension" 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 "energy_leq (strat g') e'" using S energy_sup_in ‹weight
g g' ≠ None›
                          by (smt (verit) E ‹length e'' = dimension› inv_preserves_length
mem_Collect_eq)
                        have "application (the (weight g g')) (strat g') ≠ None"
using E domain_inv domain_inv ‹length e''=dimension›
                          by (metis ‹weight g g' ≠ None› )
                        thus "application (the (weight g g')) e' ≠ None" using domain_upw_close
‹energy_leq (strat g') e'›
                          using ‹weight g g' ≠ None› by blast
                        have "energy_leq e'' (the (application (the (weight g g'))
(strat g')))" using E upd_inv_increasing
                          using ‹length e'' = dimension› ‹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 ‹energy_leq (strat g') e'› have "energy_leq (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 (smt (verit) Collect_cong E ‹length e'' = dimension›
‹weight g g' ≠ None› inv_preserves_length)
                        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 ‹length e'  = dimension›
                    by blast
                  thus "¬ energy_leq 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 dimension {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 dimension {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. energy_leq 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 dimension {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 dimension {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 dimension {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')) ∧ winning_budget_len (strat_e g') g'"
            using Strat_E by (smt (verit, del_insts) some_eq_ex)
        qed

        have exists: "⋀g'. weight g g' ≠ None ⟹ ∃e. e∈ energy_Min {e. winning_budget_len
e g' ∧ energy_leq e (strat_e g')}"
        proof-
          fix g'
          assume "weight g g' ≠ None "
          hence notempty: "strat_e g' ∈ {e. winning_budget_len e g' ∧ energy_leq
e (strat_e g')}" using Strat_E energy_leq.refl by auto
          have "∀e ∈ {e. winning_budget_len e g' ∧ energy_leq e (strat_e g')}.
length e = dimension"
            using winning_budget_len.cases by auto
          hence "{} ≠ energy_Min {e. winning_budget_len e g' ∧ energy_leq e (strat_e
g')}"
            using energy_Min_not_empty notempty
            by (smt (verit) empty_iff)
```

```
          thus "∃e. e∈ energy_Min {e. winning_budget_len e g' ∧ energy_leq e (strat_e
g')}" by auto
        qed

        define strat' where "strat' ≡ λg'. the (inverse_application (the (weight
g g')) (SOME e. e∈ energy_Min {e. winning_budget_len e g' ∧ energy_leq e (strat_e
g')}))"

        have "(∀g'. weight g g' ≠ None ⟶ strat' g' ∈ {the (inverse_application
(the (weight g g')) e) | e. minimal_winning_budget e g'})
                   ∧ e = (energy_sup dimension {strat' g'| g'. weight g g' ≠ None})"
        proof
          show win: "∀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' ∈ {the (inverse_application
(the (weight g g')) e) |e. minimal_winning_budget e g'}"
            proof
              assume "weight g g' ≠ None"
              hence "strat' g' = the (inverse_application (the (weight g g')) (SOME
e. e∈ energy_Min {e. winning_budget_len e g' ∧ energy_leq e (strat_e g')}))"
                using strat'_def by auto
              hence  "∃e. e∈ energy_Min {e. winning_budget_len e g' ∧ energy_leq
e (strat_e g')} ∧ strat' g' = the (inverse_application (the (weight g g')) e)"
                using exists ‹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' ∧ energy_leq e (strat_e g')} ∧ strat' g' = the (inverse_application (the (weight
g g')) e)" by auto
              hence "minimal_winning_budget e g'" using energy_Min_def
                by (smt (verit) energy_leq.strict_trans1 mem_Collect_eq)
              thus "strat' g' ∈ {the (inverse_application (the (weight g g')) e)
|e. minimal_winning_budget e g'}" using E
                by blast
            qed
          qed

          have "(⋀g'. weight g g' ≠ None ⟹
           strat' g' ∈ {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 dimension {strat' g' |g'. weight
g g' ≠ None}) g"
            using sup_inv_in_winning_budget A1 A2 by simp

          have "energy_leq (energy_sup dimension {strat' g' |g'. weight g g' ≠ None})
(energy_sup dimension {strat g'| g'. weight g g' ≠ None})"
          proof(cases " {g'. weight g g' ≠ None} = {}")
            case True
            then show ?thesis using energy_sup_def
              using energy_leq.refl by auto
          next
            case False
            show ?thesis
```

```
            proof(rule energy_sup_leq_energy_sup)
              show "{strat' g' |g'. weight g g' ≠ None} ≠ {}" using False by simp

              have A: "⋀a. a ∈ {strat' g' |g'. weight g g' ≠ None} ⟹ ∃b∈{strat
g' |g'. weight g g' ≠ None}. energy_leq a b ∧ length a = dimension"
              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

                have "(strat' g') = (the (inverse_application (the (weight g g'))
                  (SOME e. e ∈ energy_Min {e. winning_budget_len e g' ∧ energy_leq
e (strat_e g')})))" using strat'_def by auto
                hence  "∃e. e∈ energy_Min {e. winning_budget_len e g' ∧ energy_leq
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' ∧ energy_leq e (strat_e g')} ∧ strat' g' = the (inverse_application (the (weight
g g')) e)" by auto
                hence "energy_leq e (strat_e g')" using energy_Min_def by auto

                hence "length a = dimension " using ‹a = strat' g' ∧ weight g g'
≠ None› energy_Min_def
                  by (metis (no_types, lifting) E inv_preserves_length mem_Collect_eq
winning_budget_len.cases)

                have leq: "energy_leq (the (inverse_application (the (weight g g'))
e)) (the (inverse_application (the (weight g g')) (strat_e g')))"
                proof(rule inverse_monotonic)
                  show "energy_leq e (strat_e g')" using ‹energy_leq 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' ∧ energy_leq e (strat_e
g')}" using energy_Min_def
                    by auto
                  hence "winning_budget_len e g'"
                    by simp
                  thus "length e = dimension"
                    using winning_budget_len.simps
                    by blast
                  thus "inverse_application (the (weight g g')) e ≠ None"
                    using domain_inv ‹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 "energy_leq (strat' g') (strat g')" using leq E by simp
                hence "∃b∈{strat g' |g'. weight g g' ≠ None}. energy_leq a b" using
‹a = strat' g' ∧ weight g g' ≠ None› by auto
                thus "∃b∈{strat g' |g'. weight g g' ≠ None}. energy_leq a b ∧ length
a = dimension" using ‹length a = dimension› by simp
              qed
```

```
              thus "⋀a. a ∈ {strat' g' |g'. weight g g' ≠ None} ⟹ ∃b∈{strat
g' |g'. weight g g' ≠ None}. energy_leq a b" by simp
              show "⋀a. a ∈ {strat' g' |g'. weight g g' ≠ None} ⟹ length a =
dimension " using A by simp
            qed
          qed
          thus "e = energy_sup dimension {strat' g' |g'. weight g g' ≠ None}" using
‹g ∉ attacker› Strat win
            by (metis (no_types, lifting) ‹∀e'. e' ≠ e ⟶ energy_leq 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 dimension {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 dimension {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 dimension {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 dimension {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 dimension {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 dimension {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 dimension {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 dimension {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 dimension {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 dimension {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 dimension {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 dimension {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 dimension {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 dimension {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 dimension {strat g' |g'. weight g g' ≠ None}}"
by auto
      hence "length e = dimension"
        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 dimension {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 dimension {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 dimension {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 dimension {strat g' |g'. weight g g' ≠ None}" by
auto

        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: "⋀g'. 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 meson
          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} ⟹ length a =
dimension"
        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 "length x = dimension" 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 "length a = dimension"
            using ‹length x = dimension› inv_preserves_length  ‹a= strat' g' ∧
weight g g' ≠ None› by simp
        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 dimension {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 dimension {strat g' |g'. weight g g' ≠ None}" by auto
          hence "e = energy_sup dimension {}" using True by simp

          hence "⋀i. i < dimension ⟹ e!i = 0" using empty_Sup_is_zero
            by simp
          then show ?thesis using ‹e' e< e› energy_leq_def
            using ‹length e = dimension› energy_leq.antisym by auto
        next
          case False
          hence notempty: "{strat' g' |g'. weight g g' ≠ None} ≠ {}" by auto

          have "⋀g'. weight g g' ≠ None ⟹ strat' g' e≤ strat g'"
          proof-
```

```
            fix g'
            assume "weight g g' ≠ None"
            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 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)
            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 "length x = dimension" using winning_budget_len.simps energy_Min_def
              by (metis (mono_tags, lifting) mem_Collect_eq)
            hence "strat' g' e≤  inv_upd (the (weight g g')) (the_e g')" using inverse_monotoni
X

              by (metis ‹weight g g' ≠ None› domain_inv)

            have "strat g' ∈ {inv_upd (the (weight g g')) e |e. winning_budget_len
e g'}" using Strat ‹weight g g' ≠ 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 "(⋀a. a ∈ {strat' g' |g'. weight g g' ≠ None} ⟹ ∃b∈{strat g'
|g'. weight g g' ≠ None}. a e≤ b)" by auto
          hence "energy_sup dimension {strat' g' |g'. weight g g' ≠ None} e≤ e'"

            using notempty len Strat energy_sup_leq_energy_sup
            by presburger
          hence le: "energy_sup dimension {strat' g' |g'. weight g g' ≠ None} e<
e" using ‹e' e< e›
            using energy_leq.asym energy_leq.trans by blast

          have "energy_sup dimension {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 dimension {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_budge
x 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 dimension {strat' g' |g'. weight g g' ≠ None} =
(energy_sup dimension {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 dimension {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›  ‹length e = dimension›
by blast
    qed
  qed


  have "⋀g e. e ∈ a_win_min g ⟹ length e = dimension"
    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 ∧ length e = dimension)" 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'. length e'
= dimension ∧
                      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'})
                     ∧ e'' = (energy_sup dimension {strat g'| g'. weight g g' ≠ None})}"
      using minimal_winning_budget_defender False by simp
    hence "a_win_min g = energy_Min {energy_sup dimension {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 dimension {inv_upd (the (weight g g'))
(e_index g') | g'. weight g g' ≠ None} |
           e_index. ∀g'. weight g g' ≠ None ⟶ (length (e_index g') = dimension
∧ 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'})
                     ∧ e'' = (energy_sup dimension {strat g'| g'. weight g g' ≠ None})}
        ={energy_sup dimension {inv_upd (the (weight g g')) (e_index g') | g'. weight
g g' ≠ None} |
           e_index. ∀g'. weight g g' ≠ None ⟶ (length (e_index g') = dimension
∧ 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'})
                     ∧ e'' = (energy_sup dimension {strat g'| g'. weight g g' ≠ None})}
            ⊆{energy_sup dimension {inv_upd (the (weight g g')) (e_index g') | g'.
weight g g' ≠ None} |
           e_index. ∀g'. weight g g' ≠ None ⟶(length (e_index g') = dimension
∧ 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'})
                  ∧ e'' = (energy_sup dimension {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 dimension {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'})
                  ∧ e = (energy_sup dimension {strat g'| g'. weight g g' ≠ None})"
          by auto
        define e_index where "e_index ≡ λg'. (SOME e''. e'' ∈ a_win_min g' ∧ strat
g' = the (inverse_application (the (weight g g')) e''))"
        hence index: "⋀g'. weight g g' ≠ None ⟹ (e_index g') ∈ a_win_min g' ∧
strat g' = the (inverse_application (the (weight g g')) (e_index g'))"
        proof-
          fix g'
          have I: "e_index g' = (SOME e''. e'' ∈ a_win_min g' ∧ strat g' = the (inverse_applica
(the (weight g g')) e''))"
            using e_index_def by simp
          assume "weight g g' ≠ None"
          hence "strat g' ∈ {the (inverse_application (the (weight g g')) e) | e.
minimal_winning_budget e g'}"
            using S by simp
          hence "strat g' ∈ {the (inverse_application (the (weight g g')) e) | e.
e ∈ a_win_min g'}" by simp
          hence "∃e''. e'' ∈ a_win_min g' ∧ strat g' = the (inverse_application
(the (weight g g')) e'')" by auto
          thus "(e_index g') ∈ a_win_min g' ∧ 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 ∈ {energy_sup dimension {inv_upd (the (weight g g')) (e_index g')
| g'. weight g g' ≠ None} |
           e_index. ∀g'. weight g g' ≠ None ⟶ (length (e_index g') = dimension
∧ e_index g' ∈ a_win_min g')}"
        proof
          show "∃e_index. e = energy_sup dimension {inv_upd (the (weight g g'))
(e_index g') |g'. weight g g' ≠ None} ∧
       (∀g'. weight g g' ≠ None ⟶ (length (e_index g') = dimension ∧ e_index
g' ∈ a_win_min g'))"
          proof
            show "e = energy_sup dimension {inv_upd (the (weight g g')) (e_index
g') |g'. weight g g' ≠ None} ∧
       (∀g'. weight g g' ≠ None ⟶ (length (e_index g') = dimension ∧ e_index
g' ∈ a_win_min g'))"
            proof
              show "e = energy_sup dimension {inv_upd (the (weight g g')) (e_index
g') |g'. weight g g' ≠ None}"
                using index S
                by (smt (verit) Collect_cong)
```

```
              have "∀g'. weight g g' ≠ None ⟶ e_index g' ∈ a_win_min g'"
                using index by simp
              thus "∀g'. weight g g' ≠ None ⟶ (length (e_index g') = dimension
∧ e_index g' ∈ a_win_min g')"
                using D by meson
            qed
          qed
        qed
      qed
      show "{energy_sup dimension {inv_upd (the (weight g g')) (e_index g') | g'.
weight g g' ≠ None} |
           e_index. ∀g'. weight g g' ≠ None ⟶ (length (e_index g') = dimension
∧ e_index g' ∈ a_win_min g')}
          ⊆{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 dimension {strat g'| g'. weight g g' ≠ None})}"
      proof
        fix e
        assume "e ∈ {energy_sup dimension {inv_upd (the (weight g g')) (e_index
g') | g'. weight g g' ≠ None} |
           e_index. ∀g'. weight g g' ≠ None ⟶ (length (e_index g') = dimension
∧ e_index g' ∈ a_win_min g')}"
        from this obtain e_index where I: "e = energy_sup dimension {inv_upd (the
(weight g g')) (e_index g') | g'. weight g g' ≠ None} ∧ (∀g'. weight g g' ≠ None
⟶ e_index g' ∈ a_win_min g')"
          by blast
        define strat where "strat ≡ λg'. inv_upd (the (weight g g')) (e_index g')"

        show "e ∈{e''. ∃strat. (∀g'. weight g g' ≠ None ⟶ strat g' ∈ {the (inverse_applicat
(the (weight g g')) e) | e. minimal_winning_budget e g'})
                 ∧ e'' = (energy_sup dimension {strat g'| g'. weight g g' ≠ None})}"
        proof
          show "∃strat.
       (∀g'. weight g g' ≠ None ⟶
            strat g' ∈ {inv_upd (the (weight g g')) e |e. minimal_winning_budget
e g'}) ∧
       e = energy_sup dimension {strat g' |g'. weight g g' ≠ None}"
          proof
            show "(∀g'. weight g g' ≠ None ⟶
            strat g' ∈ {inv_upd (the (weight g g')) e |e. minimal_winning_budget
e g'}) ∧
       e = energy_sup dimension {strat g' |g'. weight g g' ≠ None}"
            proof
              show "∀g'. weight g g' ≠ None ⟶
        strat g' ∈ {inv_upd (the (weight g g')) e |e. minimal_winning_budget e
g'}"
                using I strat_def by blast
              show "e = energy_sup dimension {strat g' |g'. weight g g' ≠ None}"
using I strat_def
                by blast
            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 ∈ possible_pareto" and "F ≼ a_win_min"
  shows "iteration F ≼ a_win_min"
proof-
  have "iteration F ≼ 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 ⇒ 'position ⇒ bool" where
  "S e g" if "g ∉ attacker ∧ (∃index. e = (energy_sup dimension
              {inv_upd (the (weight g g')) (index g')| g'. weight g g' ≠ None})
              ∧ (∀g'.  weight g g' ≠ None ⟶  S (index g') g'))" |
  "S e g" if "g ∈ attacker ∧ (∃g'.( weight g g' ≠ None
              ∧ (∃e'. S e' g' ∧ e = inv_upd (the (weight g g')) e')))"

lemma length_S:
  shows "⋀e g. S e g ⟹ length e = dimension"
proof-
  fix e g
  assume "S e g"
  thus "length e = dimension"
  proof(rule S.induct)
    show "⋀g e. g ∉ attacker ∧
           (∃index.
               e =
               energy_sup dimension
                {inv_upd (the (weight g g')) (index g') |g'. weight g g' ≠ None}
∧
               (∀g'. weight g g' ≠ None ⟶ S (index g') g' ∧ length (index g')
= dimension)) ⟹
           length e = dimension"
    proof-
      fix e g
      assume "g ∉ attacker ∧
           (∃index.
               e =
               energy_sup dimension
                {inv_upd (the (weight g g')) (index g') |g'. weight g g' ≠ None}
∧
               (∀g'. weight g g' ≠ None ⟶ S (index g') g' ∧ length (index g')
= dimension))"
      from this obtain index where "e =
               energy_sup dimension
                {inv_upd (the (weight g g')) (index g') |g'. weight g g' ≠ None}"
by auto
      thus "length e = dimension" using energy_sup_def by simp
```

```
    qed

    show "⋀g e. g ∈ attacker ∧
           (∃g'. weight g g' ≠ None ∧
                 (∃e'. (S e' g' ∧ length e' = dimension) ∧
                       e = inv_upd (the (weight g g')) e')) ⟹
           length e = dimension"
    proof-
      fix e g
      assume "g ∈ attacker ∧
           (∃g'. weight g g' ≠ None ∧
                 (∃e'. (S e' g' ∧ length e' = dimension) ∧
                       e = inv_upd (the (weight g g')) e'))"
      from this obtain g' e' where "weight g g' ≠ None" and "(S e' g' ∧ length
e' = dimension) ∧
                       e = inv_upd (the (weight g g')) e'" by auto
      thus "length e = dimension" using inv_preserves_length  by simp
    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 dimension
                {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 dimension
                {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 dimension
                {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 "length e = dimension ∧
    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 "length e = dimension"
                using E energy_sup_def
                by simp
              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 dimension {inv_upd (the (weight
g g')) (index g') |g'. weight g g' ≠ None}" by simp
                      have leq: "inv_upd (the (weight g g')) (index g') e≤ e"
                        unfolding E proof(rule energy_sup_in)
                        show "inv_upd (the (weight g g')) (index g')
                              ∈ {inv_upd (the (weight g g')) (index g') |g'. weight
g g' ≠ None}" using ‹weight g g' ≠ None› by auto
                        show "length (inv_upd (the (weight g g')) (index g')) =
dimension" using inv_preserves_length  ‹weight g g' ≠ None› ‹S (index g') g' ∧
index g' ∈ a_win g'› winning_budget_len.simps
                          by (metis mem_Collect_eq)
                      qed

                      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 domain_inv  ‹weight g g' ≠ None› ‹S (index g')
g' ∧ index g' ∈ a_win g'› winning_budget_len.simps
                          by (metis domain_inv mem_Collect_eq)
```

```
                        show "inv_upd (the (weight g g')) (index g') e≤ e" using
leq by simp
                      qed

                      have "index g' e≤ upd (the (weight g g')) e"
                      proof(rule energy_leq.trans)
                        show "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
                          by (metis ‹weight g g' ≠ None› mem_Collect_eq)
                        show "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›
                          by (metis ‹S (index g') g' ∧ index g' ∈ a_win g'› domain_inv
inv_preserves_length length_S)
                      qed

                      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 domain_inv domain_inv
            by (metis length_S upd_inv_increasing)
          show "winning_budget_len e g"
          proof(rule winning_budget_len.intros(2))
            show "length e = dimension ∧
    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"length e' = dimension" using ‹(S e' g' ∧ e' ∈ a_win g') ∧ e
= inv_upd (the (weight g g')) e'› winning_budget_len.simps
```

```
                by blast
              show "length e = dimension"
                using ‹(S e' g' ∧ e' ∈ a_win g') ∧ e = inv_upd (the (weight g g'))
e'› inv_preserves_length ‹length e' = dimension› ‹weight g g' ≠ None›
                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 domain_inv
domain_inv
                        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
      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_bugget_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))} ∧
        length e' = dimension}"
            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))))
                 ∧ length e' = dimension"
            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 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: "length (upd (the (weight g' (the (s e' g')))) e') =
dimension" 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› upd_preserves_legth option.distinct(1))
                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 domain_inv 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_leq.trans
                  using y_len by blast
                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_sin
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_prems: "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 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_prems
                                  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: "length (upd (the (weight g' g'')) e') = dimension"
using y_len
                         using ‹apply_w g' g'' e' ≠ None› upd_preserves_legth
                         using ‹weight g' g'' ≠ None› 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 domain_inv 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 leq: "energy_sup dimension {inv_upd (the (weight g' g'')) (index
g'')| g''. weight g' g'' ≠ None} e≤ e'"
                using energy_sup_leq
                by (smt (z3) ‹y = (g', e')› ‹y ∈ {(g', e'). (g', e') ∈ reachable_positions
s g e ∧ length e' = dimension}› case_prodD mem_Collect_eq)

              have "S (energy_sup dimension {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}" unfolding
energy_Min_def finite_positions
    by (smt (verit) Collect_cong energy_leq.refl energy_leq.strict_trans1 mem_Collect_eq)

  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. {}) ∈ possible_pareto"
          using funpow_simps_right(1) 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_right(1)
by simp
      qed
```

```
    next
      case (Suc n)
      have "(iteration ^^ (Suc n)) (λg. {}) = iteration ((iteration ^^ n) (λg. {}))"

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

  show "pareto_sup {(iteration ^^ n) (λg. {}) |. n} ∈ possible_pareto"
  proof(rule pareto_sup_is_sup)
    show "{(iteration ^^ n) (λg. {}) |. 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. {}) |. 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. {}) |. n}"
    unfolding pareto_order_def proof
    fix g
    show "∀e. e ∈ Smin g ⟶
              (∃e'. e' ∈ pareto_sup {(iteration ^^ n) (λg. {}) |. n} g ∧ e' e≤ e)"
    proof
      fix e
      show "e ∈ Smin g ⟶
          (∃e'. e' ∈ pareto_sup {(iteration ^^ n) (λg. {}) |. n} g ∧ e' 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. {}) |. n} g ∧ e' e≤ e"
        proof(rule S.induct)
          show "⋀g e. g ∉ attacker ∧
           (∃index.
               e =
               energy_sup dimension
                {inv_upd (the (weight g g')) (index g') |g'. weight g g' ≠ None}
∧
               (∀g'. weight g g' ≠ None ⟶
                     S (index g') g' ∧
                     (∃e'. e' ∈ pareto_sup {(iteration ^^ n) (λg. {}) |. n} g'
∧
                           e' e≤ index g'))) ⟹
           ∃e'. e' ∈ pareto_sup {(iteration ^^ n) (λg. {}) |. n} g ∧ e' e≤ e"
          proof-
            fix e g
            assume A: "g ∉ attacker ∧
           (∃index.
               e =
```

```
              energy_sup dimension
               {inv_upd (the (weight g g')) (index g') |g'. weight g g' ≠ None}
∧
              (∀g'. weight g g' ≠ None ⟶
                    S (index g') g' ∧
                    (∃e'. e' ∈ pareto_sup {(iteration ^^ n) (λg. {}) |. n} g'
∧
                          e' e≤ index g')))"
           from this obtain index where "e =
              energy_sup dimension
               {inv_upd (the (weight g g')) (index g') |g'. weight g g' ≠ None}"
and
              "∀g'. weight g g' ≠ None ⟶
                    S (index g') g' ∧
                    (∃e'. e' ∈ pareto_sup {(iteration ^^ n) (λg. {}) |. n} g'
∧
                          e' e≤ index g')" by auto

           define index' where "index' ≡ λg'. SOME e'. e' ∈ pareto_sup {(iteration
^^ n) (λg. {}) |. n} g' ∧
                          e' e≤ index g'"

           have "⋀g'. weight g g' ≠ None ⟹ ∃e'. e' ∈ pareto_sup {(iteration
^^ n) (λg. {}) |. n} g' ∧
                          e' e≤ index g'" using ‹∀g'. weight g g' ≠ None ⟶
                    S (index g') g' ∧
                    (∃e'. e' ∈ pareto_sup {(iteration ^^ n) (λg. {}) |. n} g'
∧
                          e' e≤ index g')› by simp
           hence "⋀g'. weight g g' ≠ None ⟹ index' g' ∈ pareto_sup {(iteration
^^ n) (λg. {}) |. n} g' ∧
                          index' g' e≤ index g'" unfolding index'_def using some_eq_ex
             by (metis (mono_tags, lifting))
           hence "⋀g'. weight g g' ≠ None ⟹ ∃F. F ∈ {(iteration ^^ n) (λg.
{}) |. n} ∧ index' g' ∈ F g'"
             unfolding pareto_sup_def using energy_Min_def by simp
           hence index'_len: "⋀g'. weight g g' ≠ None ⟹ length (index' g') =
dimension" using possible_pareto_def
             by (metis ‹⋀g'. weight g g' ≠ None ⟹ index' g' ∈ pareto_sup {(iteration
^^ n) (λg. {}) |. n} g' ∧ index' g' e≤ index g'› ‹∀g'. weight g g' ≠ None ⟶
S (index g') g' ∧ (∃e'. e' ∈ pareto_sup {(iteration ^^ n) (λg. {}) |. n} g' ∧ e'
e≤ index g')› energy_leq_def length_S)

           define index_F where "index_F = (λg'. (SOME F. (F ∈ {(iteration ^^ n)
(λg. {}) |. n} ∧ index' g' ∈ F g')))"
           have IF: "⋀g'. weight g g' ≠ None ⟹ index_F g' ∈ {(iteration ^^ n)
(λg. {}) |. n} ∧ index' g' ∈ index_F g' g'"
             unfolding index_F_def using some_eq_ex ‹⋀g'. weight g g' ≠ None ⟹
∃F. F ∈ {(iteration ^^ n) (λg. {}) |. n} ∧ index' g' ∈ F g'›
             by (metis (mono_tags, lifting))

           have "∃F. (F∈ {(iteration ^^ n) (λg. {}) |. n} ∧ (∀g'. weight g g'
≠ None ⟶ index_F g' ≼ F))"
           proof-
             define P' where "P' = {index_F g'| g'. weight g g' ≠ None}"
             have "∃F'. F' ∈ {(iteration ^^ n) (λg. {}) |. n} ∧ (∀F. F ∈ P' ⟶
```

```
F ≼ F')"
              proof(rule finite_directed_set_upper_bound)
                show "⋀F F'.
       F ∈ {(iteration ^^ n) (λg. {}) |. n} ⟹
       F' ∈ {(iteration ^^ n) (λg. {}) |. n} ⟹
       ∃F''. F'' ∈ {(iteration ^^ n) (λg. {}) |. n} ∧ F ≼ F'' ∧ F' ≼ F''"
                proof-
                  fix F F'
                  assume "F ∈ {(iteration ^^ n) (λg. {}) |. n}" and "F' ∈ {(iteration
^^ n) (λg. {}) |. n}"
                  from this obtain n m where "F = (iteration ^^ n) (λg. {})" and
"F' = (iteration ^^ m)(λg. {})" by auto
                  show "∃F''. F'' ∈ {(iteration ^^ n) (λg. {}) |. n} ∧ F ≼ F''
∧ F' ≼ F''"
                  proof
                    show "((iteration ^^ (max n m)) (λg. {})) ∈ {(iteration ^^ n)
(λg. {}) |. n} ∧ F ≼ ((iteration ^^ (max n m)) (λg. {})) ∧ F' ≼ ((iteration ^^
(max n m)) (λg. {}))"
                    proof-
                      have "⋀i j. i ≤ j ⟹ ((iteration ^^ i) (λg. {})) ≼ ((iteration
^^ j) (λg. {}))"
                      proof-
                        fix i j
                        show " i ≤ j ⟹ ((iteration ^^ i) (λg. {})) ≼ ((iteration
^^ j) (λg. {}))"
                        proof-
                          assume "i ≤ j"
                          thus "(iteration ^^ i) (λg. {}) ≼ (iteration ^^ j) (λg.
{})"
                          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
                        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 dimension {inv_upd (the (weight g g')) (e_index
g')| g'. weight g g' ≠ None} e≤ energy_sup dimension {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 empty_Sup_is_zero
                using energy_leq.order_iff_strict by fastforce
            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 "length (e_index g') = dimension" using index'_len ‹weight
g g' ≠ None› energy_leq_def
                      by auto
                    thus "inverse_application (the (weight g g')) (e_index g') ≠
None"

                      using domain_inv ‹weight g g' ≠ None›
                      by auto
                    show "length (e_index g') = dimension"
                      using ‹length (e_index g') = dimension› 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
                show "⋀a. a ∈ {inv_upd (the (weight g g')) (e_index g') |g'. weight
g g' ≠ None} ⟹
         length a = dimension"
```

```
                  using inv_preserves_length  index'_len ‹⋀g'. weight g g' ≠ None
⟹ e_index g' ∈ F' g' ∧ e_index g' e≤ index' g'›
                  using energy_leq_def by force
              qed
            qed

            have sup_leq2: "energy_sup dimension {inv_upd (the (weight g g')) (index'
g')|g'. weight g g' ≠ None} e≤ energy_sup dimension {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 empty_Sup_is_zero
                using energy_leq.order_iff_strict by fastforce
            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_leq_def
                    by (meson domain_inv 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} ⟹
         length a = dimension"
                  using inv_preserves_length  index'_len by blast
              qed
            qed


            have "⋀g'. weight g g' ≠ None ⟹ length (e_index g') = dimension"

              using index'_len ‹⋀g'. weight g g' ≠ None ⟹ e_index g' ∈ F' g'
∧ e_index g' e≤ index' g'› energy_leq_def by simp
            hence "energy_sup dimension {inv_upd (the (weight g g')) (e_index g')|g'.
weight g g' ≠ None} ∈ {energy_sup dimension
                     {inv_upd (the (weight g g')) (e_index g') |g'. weight g
g' ≠ None} |
```

```
                          e_index.
                          ∀g'. weight g g' ≠ None ⟶
                               length (e_index g') = dimension ∧ e_index g' ∈ F' g'}"
                 using ‹⋀g'. weight g g' ≠ None ⟹ e_index g' ∈ F' g' ∧ e_index
g' e≤ index' g'›
                 by blast
               hence "∃em. em ∈  energy_Min
                         {energy_sup dimension
                           {inv_upd (the (weight g g')) (e_index g') |g'. weight g
g' ≠ None} |
                          e_index.
                          ∀g'. weight g g' ≠ None ⟶
                               length (e_index g') = dimension ∧ e_index g' ∈ F' g'}

                     ∧ em e≤ energy_sup dimension {inv_upd (the (weight g g')) (e_index
g')| g'. weight g g' ≠ None}"
                 using energy_Min_contains_smaller by meson
               hence "∃em. em∈ iteration F' g ∧ em e≤ energy_sup dimension {inv_upd
(the (weight g g')) (e_index g')| g'. weight g g' ≠ None}"
                 unfolding iteration_def using A
                 by simp
               from this obtain em where EM: "em ∈ iteration F' g ∧ em e≤ energy_sup
dimension {inv_upd (the (weight g g')) (e_index g')| g'. weight g g' ≠ None}"
                 by auto
               from F' have F': "iteration F' ∈ {(iteration ^^ n) (λg. {}) |. n}" using
funpow.simps image_iff rangeE
                 by (smt (z3) UNIV_I comp_eq_dest_lhs)
               hence "em ∈ {e. ∃F. F ∈ {(iteration ^^ n) (λg. {}) |. n} ∧ e ∈ F g}"

                 using EM by auto
               hence "∃em'. em' ∈ pareto_sup {(iteration ^^ n) (λg. {}) |. n} g ∧
em' e≤ em"
                 unfolding pareto_sup_def using F' energy_Min_contains_smaller by meson
               from this obtain em' where EM': "em' ∈ pareto_sup {(iteration ^^ n)
(λg. {}) |. n} g ∧ em' e≤ em" by auto
               hence "em' e≤ e" using EM sup_leq1 sup_leq2 ‹e =
                    energy_sup dimension
                     {inv_upd (the (weight g g')) (index g') |g'. weight g g' ≠ None}›
energy_leq.trans
                 by blast
               thus " ∃e'. e' ∈ pareto_sup {(iteration ^^ n) (λg. {}) |. n} g ∧ e'
e≤ e"
                 using EM' by auto
             qed

             show "⋀g e. 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')) ⟹
              ∃e'. e' ∈ pareto_sup {(iteration ^^ n) (λg. {}) |. n} g ∧ e' e≤ 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 "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 "length e' = dimension" using length_S ‹weight g g' ≠ None›
‹S e' g'› by auto
              hence "length e''=  dimension" using ‹e'' e≤ e'› energy_leq_def by
auto

              thus"inverse_application (the (weight g g')) e'' ≠ None"
                using domain_inv ‹weight g g' ≠ None›
                by blast
              show "length e'' = dimension"
                by (simp add: ‹length e'' = dimension›)
            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'.
           length e' = dimension ∧ 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'›
              using energy_leq_def by auto

            define Fn where "Fn =  (iteration ^^ n) (λg. {})"

            have "∃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''' 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. ∃F. F ∈ {(iteration ^^ n) (λg. {}) |. n} ∧ e ∈ F
g}" by blast
            hence "∃em. em ∈ pareto_sup {(iteration ^^ n) (λg. {}) |. n} g ∧ em
e≤ e'''"
```

```
              unfolding pareto_sup_def using energy_Min_contains_smaller
              by meson
            from this obtain em where EM: "em ∈ pareto_sup {(iteration ^^ n) (λg.
{}) |. n} g ∧ em e≤ 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≤
e"
              proof
                show "em ∈ pareto_sup {(iteration ^^ n) (λg. {}) |. n} g" using
EM by simp
                have "inv_upd (the (weight g g')) e'' 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≤ e" using E'''
                  using energy_leq.trans by blast
                thus "em e≤ e" using EM energy_leq.trans by blast
              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) energy_leq.trans mem_Collect_eq)
        hence "e = e'" using ‹e' ∈ (iteration ^^ (Suc n)) (λg. {}) g ∧ e' e≤ e›
‹e'' ∈ a_win_min g ∧ e'' e≤ e'›
          using energy_leq.strict_iff_not by auto
        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
```

5.3 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
```

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 ≡ (∀i D. u ! i = min_set D
                                    ⟶ D ≠ {} ∧ D ⊆ {x. x < length u})"
```

Now the application of updates `apply_update` will be defined.

```
fun apply_component::"nat ⇒ update_component ⇒ energy ⇒ 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 (nths e A))"|
  "apply_component i plus_one e = Some ((e ! i)+1)"

fun apply_update:: "update ⇒ energy ⇒ energy option"  where
  "apply_update u e = (if (length u = length e)
          then (those (map (λi. apply_component i (u ! i) e) [0..<length e]))
          else  None)"

abbreviation "upd u e ≡ 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 ≠ None"
  shows "length (upd u e) = length e"
proof -
  from assms have "apply_update u e = those (map (λn. apply_component n (u ! n)
e) [0..<length e])" by auto
  thus ?thesis using assms len_those
    using length_map length_upt by force
qed

lemma apply_to_comp_n:
  assumes "apply_update u e ≠ 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..<length e])))!i" using apply_update.simps
    using assms by auto
  also have "... = the ((map (λn. apply_component n (u ! n) e) [0..<length e])!
i)" using the_those_n
    by (metis (no_types, lifting) apply_update.simps assms(1) assms(2) calculation
length_map map_nth)
  also have "... = the ( apply_component i (u ! i) e)" using nth_map
    by (simp add: assms(2) calculation linordered_semidom_class.add_diff_inverse
not_less_zero nth_map_upt)
  finally show ?thesis.
qed

lemma upd_domain_upward_closed:
  assumes "apply_update u e ≠ None" and "e e≤ e'"
  shows "apply_update u e' ≠ None"
proof
  assume "apply_update u e' = None"
  from assms have "length u = length e'" unfolding apply_update.simps energy_leq_def
by metis
  hence A: "apply_update u e' = those (map (λn. apply_component n (u ! n) e') [0..<length
e'])" using apply_update.simps by simp
  hence "those (map (λn. apply_component n (u ! n) e') [0..<length e']) = None"
using ‹apply_update u e' = None› by simp
  hence "¬ (∀n < length e'. (λn. apply_component n (u ! n) e') ([0..<length e']
! n) ≠ None)" using those_map_not_None
    by (metis (no_types, lifting) length_map map_nth)
  hence "∃n< length e'. (λn. apply_component n (u ! n) e') ([0..<length e'] ! n)
= None" by auto
  from this obtain n where "n< length e'" and "(λn. apply_component n (u ! n) e')
([0..<length e'] ! n) = None" by auto
  hence "apply_component n (u ! n) e' = None" by simp
  hence "u ! n = minus_one" using apply_component.elims by blast
  hence  " e' ! n = 0" using ‹apply_component n (u ! n) e' = None› apply_component.elims
    by fastforce
  hence "e ! n = 0" using assms(2) energy_leq_def ‹n < length e'› by auto
  hence "(λn. apply_component n (u ! n) e) ([0..<length e] ! n) = None" using ‹u
! n = minus_one› apply_component.simps(2)
    using ‹n < length e'› assms(2) energy_leq_def by auto
  hence "those (map (λn. apply_component n (u ! n) e) [0..<length e]) = None" using
those.simps option.map_sel  ‹n < length e'›
    by (smt (verit, ccfv_SIG) ‹length u = length e'› apply_update.simps assms(1)
length_map map_nth nth_map those_all_Some)
  hence "apply_update u e = None" by simp
  thus "False" using assms(1) by simp
qed
```

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≤ e'" and "valid_update u"
  shows "(upd u e) 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..<length
e]) ! n ≠ None" using  ‹n < length e› those_all_Some apply_update.simps
          by (metis (no_types, lifting) length_map map_nth)
        hence "(apply_component n (u ! n) e) = (map (λn. apply_component n (u !
n) e) [0..<length e]) ! n " using ‹n < length e›
          by simp
        hence "(apply_component n (u ! n) e) ≠ None" using A by simp
        hence "e ! n >0 " using minus_one apply_component.elims by auto
        hence  "(e ! n) -1 ≤ (e' ! n) -1" using ‹e ! n ≤ e' ! n› by (metis eSuc_minus_1
iadd_Suc ileI1 le_iff_add)

        from ‹e ! n >0› have "e' ! n > 0" using assms(2) energy_leq_def
          using ‹e ! n ≤ e' ! n› by auto
        have A: "the (apply_update u e') ! n = the (apply_component n (u ! n) e')"
using apply_to_comp_n ‹apply_update u e' ≠ None›
          using ‹n < length e› assms(2) energy_leq_def by auto
        have "the (apply_component n (u ! n) e' )= (e' ! n) -1" using minus_one
‹e' ! n >0›
          by simp
        hence "the (apply_update u e') ! n  = (e' ! n) -1" using A by simp
        then show ?thesis using ‹(e ! n) -1 ≤ (e' ! n) -1›
          using ‹0 < e ! n› ‹the (apply_update u e) ! n = the (apply_component
n (u ! n) e)› minus_one by auto
      next
        case min_set
```

```
        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 (nths e A))) ≤ e! a" proof
          fix a
          assume "a ∈ A"
          hence "e!a ∈ set (nths e A)" using set_nths nths_def
            using ‹A ⊆ {x. x < length e}› in_mono by fastforce
          thus "Min (set (nths e A)) ≤ e ! a " using Min_le by simp
        qed
        hence "∀a∈ A. (Min (set (nths e A))) ≤ e'! a" using ‹∀a ∈ A. e!a ≤ e'!a›
          using dual_order.trans by blast
        hence "∀x ∈ (set (nths e' A)). (Min (set (nths e A))) ≤ x" using set_nths
          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 (nths e' A) ≠ {}" using ‹A ≠{}› set_nths
          by (smt (verit, best) Collect_empty_eq Collect_mem_eq Collect_mono_iff)

        hence "(nths e' A) ≠ []" by simp
        from ‹A≠{}› have "set (nths e A) ≠ {}" using set_nths ‹A ⊆ {x. x < length
e}› Collect_empty_eq ‹n < length e› ‹u ! n = min_set A›
          by (smt (verit, best) ‹set (nths e' A) ≠ {}› assms(2) energy_leq_def)
        hence "(nths e A) ≠ []" by simp
        hence "(min_list (nths e A)) = Min (set (nths e A))" using min_list_Min
by auto
        also have "... ≤ Min (set (nths e' A))"
          using ‹∀x ∈ (set (nths e' A)). (Min (set (nths e A))) ≤ x›
          by (simp add: ‹nths e' A ≠ []›)
        finally have "(min_list (nths e A)) ≤ min_list (nths e' A)" using min_list_Min
‹(nths 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
```

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 $e = (e_0, ..., e_{n-1})$ 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 $i \in u_j \subseteq \{0, ..., n-1\}$. Note that this generalises the inversion proposed by Bisping [1].

```
fun apply_inv_component::"nat ⇒ update ⇒ energy ⇒ enat" where
  "apply_inv_component i u e = Max (set (map (λ(j,up).
          (case up of zero ⇒ (if i=j then (e ! i) else 0) |
                minus_one ⇒ (if i=j then (e ! i)+1 else 0) |
                min_set A ⇒ (if i∈A then (e ! j) else 0) |
                plus_one ⇒ (if i=j then (e ! i)-1 else 0)))
          (List.enumerate 0 u)))"

fun apply_inv_update:: "update ⇒ energy ⇒ energy option" where
  "apply_inv_update u e = (if (length u = length e)
                    then Some (map (λi. apply_inv_component i u e) [0..<length e])
                    else None)"

abbreviation "inv_upd u e ≡ 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 | |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 ≠ None"
  using assms apply_inv_update.simps by simp

lemma inv_not_none_then:
  assumes "apply_inv_update u e ≠ None"
  shows "(apply_update u (inv_upd u e)) ≠ None"
proof -
  have len: "length u = length (the (apply_inv_update u e))" using assms apply_inv_update.simps
len_those
    by auto
  have "∀n<length u. (apply_component n (u ! n) (the (apply_inv_update u e)))≠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_componen
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 (λn. 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≤ e'" and "length u = length e'"
  shows "(inv_upd u e) 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 ⇒ (if i=j then (nth e i) else 0) | minus_one
⇒ if i = j then e ! i + 1 else 0
                          | min_set A ⇒ if i ∈ A then e ! j else 0 |
              plus_one ⇒ (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)
"∃ A. (u !j) = min_set A" | (plus_one) "(u!j) = plus_one"
          by (meson update_component.exhaust)

        thus "(∃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(cases)
          case zero
          hence "x= (if i=j then (nth e i) else 0)" using X by simp
          also have "... ≤ (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 "... = (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: zero)
          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
        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 "...≤ (e' ! i) +1" using assms
              using True ‹j < length u› energy_leq_def by auto
            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: minus_one True)
             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
          next
            case False
            hence "x = 0 " using X by simp
            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: minus_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
        next
          case min_set
          from this obtain A where A: "u ! j = min_set A " by auto
          hence X: "x = (if i ∈ A then e ! j else 0)" using X by auto
          then show ?thesis proof(cases "i ∈ A")
            case True
            hence "x=e ! j" using X by simp
            also have "...≤ e'!j" using assms
              using ‹j < length u› energy_leq_def by auto
            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: A True)
            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
          next
            case False
            hence "x=0" using X 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: 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: nths_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: nths_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 (nths (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 ⇒ (if j∈A then (nth e m) else 0)|
                plus_one ⇒ (if j=m then (nth e j)-1 else 0))) (n, u ! n)"
              using ‹min_set A = u ! n› by simp
            hence "e!n ∈ (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)))"
              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 ≤ (inv_upd u e)!j"
              using fin Max_le_iff
              using ‹inv_upd u e ! j = Max (set (map (λ(k, y). case y of zero ⇒(if
j=k then (nth e j) else 0) | minus_one ⇒ if j = k then e ! j + 1 else 0 | min_set
A ⇒ if j ∈ A then e ! k else 0 | plus_one ⇒ if j = k then e ! j - 1 else 0) (List.enumerate
0 u)))› by fastforce
          qed

          have "A ≠ {} ∧ A ⊆ {x. x < length u}" using assms
            by (simp add: ‹min_set A = u ! n›)
          hence "A ≠ {} ∧ A ⊆ {x. x < length (inv_upd u e)}" using assms
            by auto

          have "set (nths (inv_upd u e) A) = {(inv_upd u e) ! i |i. i < length (inv_upd
u e) ∧ i ∈ A}"
            using set_nths by metis
          hence not_empty: "(set (nths (inv_upd u e) A)) ≠ {}"
            using ‹A ≠ {} ∧ A ⊆ {x. x < length (inv_upd u e)}›
            by (smt (z3) Collect_empty_eq equals0I in_mono mem_Collect_eq)
          hence "(nths (inv_upd u e) A) ≠ []"
            by blast
          hence min_eq_Min: "min_list (nths (inv_upd u e) A) = Min (set (nths (inv_upd
u e) A))"
            using min_list_Min by blast

          have "finite (set (nths (inv_upd u e) A))" using assms ‹min_set A = u !
n›
            by simp
          hence "(e!n ≤ Min (set (nths (inv_upd u e) A))) = (∀a∈(set (nths (inv_upd
u e) A)). e!n ≤ a)"
            using not_empty Min_ge_iff by auto

          have "e!n ≤ Min (set (nths (inv_upd u e) A))"
            unfolding ‹(e!n ≤ Min (set (nths (inv_upd u e) A))) = (∀a∈(set (nths
(inv_upd u e) A)). e!n ≤ a)›
          proof
            fix x
            assume "x ∈ set (nths (inv_upd u e) A)"
            hence "x∈ {(inv_upd u e) ! i |i. i < length (inv_upd u e) ∧ i ∈ A}"
              using set_nths
              by metis
            hence "∃j. j ∈ A ∧ x = (inv_upd u e)!j"
              by blast
```

```
          thus "e ! n ≤ x " using leq
            by auto
        qed

        hence "e!n ≤ (min_list (nths (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: "∀e'∈(possible_inv u e). (inv_upd u e) e≤ e'"
  proof
    fix e'
    assume "e' ∈ possible_inv u e"
    hence "energy_leq e (the (apply_update u e'))" by auto
    hence B: "∀n < length e. e! n ≤ (the (apply_update u e')) ! n" unfolding energy_leq_def
by auto

    from‹e' ∈ possible_inv u e› have "apply_update u e' ≠ None" by simp
    have geq_0:  "⋀i. i < length u ⟹ u!i = minus_one ⟹ e'!i >0"
    proof-
      fix i
      assume "i < length u" and "u!i = minus_one"
      have " e'!i =0 ⟹ 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' ≠ None› have "those (map (λi. apply_component i
(u ! i) e') [0..<length e'])≠ None" unfolding apply_update.simps
          by meson
        hence "(map (λi. apply_component i (u ! i) e') [0..<length e']) ! i ≠ None"
using those_all_Some
          by (metis ‹apply_update u e' ≠ None› ‹i < length u› apply_update.simps
length_map map_nth)
        hence "(λi. apply_component i (u ! i) e') ([0..<length e'] ! i) ≠ None"
using nth_map
          by (metis ‹apply_update u e' ≠ None› ‹i < length u› apply_update.simps
length_map map_nth)
        hence "apply_component i (u ! i) e' ≠ None"
          by (metis ‹apply_update u e' ≠ 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' ∈ possible_inv u e› energy_leq_def len_appl
len_inv_appl mem_Collect_eq)
      show "∀n<length (the (apply_inv_update u e)). the (apply_inv_update u e) !
n ≤ 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..<length e]) ! n" using apply_inv_update.simps
            by (metis assms(1) option.sel)
          also have "... = apply_inv_component n u e"
            by (metis ‹n < length (the (apply_inv_update u e))› assms(1) len_inv_appl
minus_nat.diff_0 nth_map_upt plus_nat.add_0)
          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
          finally have inv_max: "the (apply_inv_update u e) ! n = 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)))".

          from B have "e ! n ≤ (the (apply_update u e')) ! n" using ‹n < length
(the (apply_inv_update u e))›
            using assms(1) len_inv_appl by auto
          hence upd_v: "e ! n ≤ the (apply_component n (u ! n) e')" using apply_to_comp_n
            using ‹length (the (apply_inv_update u e)) = length e'› ‹n < length
(the (apply_inv_update u e))› ‹e' ∈ possible_inv u e› by auto

          have Max_iff: "(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))) ≤ e' ! n)
                = (∀a∈ (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))). a ≤ e' ! n)"
          proof(rule Max_le_iff)
            show "finite (set (map (λ(m, y). case y 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)))"
              by simp
            show "set (map (λ(m, y). case y 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)) ≠ {}
"
              by (metis (no_types, lifting) ‹n < length (inv_upd u e)› assms(1)
empty_iff len_inv_appl length_enumerate length_map nth_mem)
```

```
          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 (nths e' A)) ≠ {}" using set_nths True assms
                  by (smt (verit) Collect_empty_eq ‹length (inv_upd u e) = length
e'› ‹n < length (inv_upd u e)›)
                hence "(nths 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 (nths e' A)))"
                  by simp
                also have "... = (min_list (nths e' A))"
                  by simp
                also have "... = Min (set (nths e' A))" using min_list_Min ‹(nths
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_nths 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


have apply_inv_is_possible_inv: "⋀u v. length u = length v ⟹ valid_update u
⟹ inv_upd u v ∈ possible_inv u v"
using leq_up_inv inv_not_none_then inv_not_none by blast

show "energy_Min (possible_inv u e) ⊆ {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)} ⊆ 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) ⊆ {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 thatapply_inv_update u is decreasing.

```
lemma inv_up_leq:
  assumes "apply_update u e ≠ None" and "valid_update u"
  shows "(inv_upd u (upd u e)) e≤ 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..<length (the (apply_update u e))]) ! n " using
apply_inv_update.simps
          using ‹length (the (apply_update u e)) = length u› ‹length e = length
u› option.sel by auto
        hence A: "the (apply_inv_update u (the (apply_update u e))) !n = apply_inv_component
n u (the (apply_update u e))"
          by (metis ‹length (the (apply_inv_update u (the (apply_update u e))))
= length e› ‹length (the (apply_update u e)) = length u› ‹length e = length u›
‹n < length (the (apply_inv_update u (the (apply_update u e))))› diff_diff_left
length_upt nth_map nth_upt plus_nat.add_0 zero_less_diff)
        have "apply_inv_component n u (the (apply_update u e)) ≤ e ! n" proof-

          have "∀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"
          proof
            fix x
            assume X: "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 the (apply_update u e) !
n + 1 else 0
                          | min_set A ⇒ if n ∈ A then 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))"
            hence "∃m < length (List.enumerate 0 u). x =  (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 the (apply_update u e) !
n + 1 else 0
                            | min_set A ⇒ if n ∈ A then 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)) ! 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 (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 the (apply_update u e) !
n + 1 else 0
                            | min_set A ⇒ if n ∈ A then 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)) ! m" by auto
          hence "x = (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 the (apply_update u e) !
n + 1 else 0
                            | min_set A ⇒ if n ∈ A then 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) ! m)" using nth_map ‹m < length (List.enumerate
0 u)›
            by simp
          hence X: "x= (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 the (apply_update u e) !
n + 1 else 0
                            | min_set A ⇒ if n ∈ A then the (apply_update u e) !
m else 0 |
                  plus_one ⇒ (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_l
length_enumerate nth_enumerate_eq)



          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
          thus "x ≤ 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 (nths e A))" using ‹u ! m = min_set
A› apply_component.simps by simp
                also have "... = Min (set (nths 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_nths)
                also have "... ≤ e!n" using True Min_le ‹A≠{} ∧ A ⊆ {x. x <
length e}›
                  using List.finite_set ‹n < length e› set_nths 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 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')) ⟹ 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 ≠ 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' ∈ 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 Bisping's (Declining) Energy Games

```
theory Bispings_Energy_Game
  imports Energy_Game Update Decidability
begin
```

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 ⊆ 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 ⟹ inv_upd (the (weight p p')) e e≤ e' = e e≤ 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 ⇒ 'position ⇒ update option" and
      dimension :: "nat"
+
```

```
assumes nonpos_eq_pos: "nonpos_winning_budget = winning_budget" and
        finite_positions: "finite positions"

sublocale bispings_energy_game_decidable ⊆ 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] B. Bisping, D. N. Jansen, and U. Nestmann. Deciding all behavioral equivalences at once: A game for linear-time-branching-time spectroscopy. *Logical Methods in Computer Science*, 18(3), 2022.

[3] B. Bisping and U. Nestmann. A game for linear-time–branching-time spectroscopy. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 12651 of *Lecture Notes in Computer Science*, pages 3–19. Springer International Publishing, 2021.

[4] M. Erné, 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.

[5] C. Stirling. Bisimulation, modal logic and model checking games. *Logic Journal of IGPL*, 7(1):103–124, 1999.

[6] R. J. van Glabbeek. The linear time - branching time spectrum. In J. C. M. Baeten and J. W. Klop, editors, *Theories of Concurrency: Unification and Extension (CONCUR)*, volume 458 of *Lecture Notes in Computer Science*, pages 278–297. Springer Berlin Heidelberg, 1990.

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.prems 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.prems(1) 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.prems(1) 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.prems(2) 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.prems(1) 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.prems(1) 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.prems(2) 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.prems 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.prems 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
```