Galois Energy Games

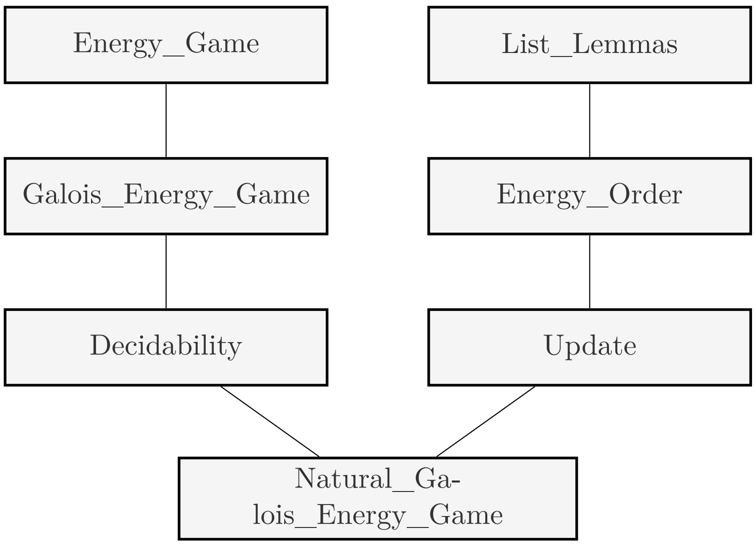
Caroline Lemke

March 26, 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 [4] 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[5] 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.



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.
Galois energy games over well-founded bounded join-semilattices are formalized 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.
The file List_Lemmas.thy contains a few simple observations about lists, specifically when using `those`. This file's contents can be found in the appendix.
In Energy_Order.thy we introduce the energies, i.e. vectors with entries in the extended natural numbers, and the component-wise order. There we establish that this order is a well-founded bounded join-semilattice.
In Update.thy we define a superset of Bisping's updates. These are partial functions of energy vectors updating each component by subtracting or adding one, replacing it with the minimum of some components or not changing it. In particular, we observe that these functions are monotonic and have upward-closed domains. Further, we introduce a generalisation of Bisping's inversion and relate it to the updates using Galois connections.
In Natural_Galois_Energy_Game.thy we formalise galois energy games over the previously defined with a fixed dimension. Afterwards, we formalise a subclass of such games where all edges of the game graph are labeled with a representation of the previously discussed updates (and thereby formalise Bisping's declining energy games). Finally, we establish the subclass-relationships and thereby conclude decidability.

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'))"
⟨proof⟩

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"
  ⟨proof⟩

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⟩

lemma valid_play_nth:
  assumes "⋀i. enat (Suc i) < llength p
                ⟶ weight (lnth p i) (lnth p (Suc i)) ≠ None"
  shows "valid_play p"
  ⟨proof⟩
```

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)"
  ⟨proof⟩

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"
⟨proof⟩

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⟩
```

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"
  ⟨proof⟩
```

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"
⟨proof⟩
```

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"
  ⟨proof⟩
```

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"
  ⟨proof⟩

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"
  ⟨proof⟩
```

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⟩

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⟩
```

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)"
  ⟨proof⟩
```

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⟩
```

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⟩
```

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⟩
```

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⟩

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⟩

lemma winning_budget_ind_implies_nonpos:
  assumes "winning_budget_ind e g"
  shows "nonpos_winning_budget e g"
⟨proof⟩
```

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⟩

end
end
```

3 Galois Energy Games

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

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

```
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 energies :: "'energy set" and
        order :: "'energy ⇒ 'energy ⇒ bool" (infix "e≤" 80)and
        energy_sup :: "'energy set ⇒ 'energy"
      assumes
        energy_order: "ordering order (λe e'. order e e' ∧ e ≠ e')" and
        energy_wqo: "wqo_on order energies" and
        bounded_join_semilattice: "⋀ set s'. set ⊆ energies ⟹ finite set
        ⟹ energy_sup set ∈ energies
            ∧ (∀s. s ∈ set ⟶ order s (energy_sup set))
            ∧ (s' ∈ energies ∧ (∀s. s ∈ set ⟶ order s s') ⟶ order (energy_sup
set) s')" and
        upward_closed_energies: "⋀e e'. e ∈ energies ⟹ e e≤ e' ⟹ e' ∈ energies"
and
        upd_well_defined: "⋀p p' e. weight p p' ≠ None
        ⟹ application (the (weight p p')) e ≠ None ⟹ e ∈ energies
        ⟹ (the (application (the (weight p p')) e)) ∈ energies" and
        inv_well_defined: "⋀p p' e. weight p p' ≠ None ⟹ e ∈ energies
        ⟹ (inverse_application (the (weight p p')) e) ≠ None
        ∧ (the (inverse_application (the (weight p p')) e)) ∈ energies
        ∧ application (the (weight p p')) (the (inverse_application (the (weight
p p')) e)) ≠ None" and
        domain_upw_closed: "⋀p p' e e'. weight p p' ≠ None ⟹ order e e'
        ⟹ application (the (weight p p')) e ≠ None
        ⟹ application (the (weight p p')) e' ≠ None" and
        galois: "⋀p p' e e'. weight p p' ≠ None
        ⟹ application (the (weight p p')) e' ≠ None
        ⟹ e ∈ energies ⟹ e' ∈ energies
        ⟹ order (the (inverse_application (the (weight p p')) e)) e' = order 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)"

abbreviation energy_l:: "'energy ⇒ 'energy ⇒ bool" (infix "e<" 80) where
  "energy_l e e' ≡  e e≤ e' ∧ e ≠ e'"

lemma leq_up_inv:
   "⋀p p' e. weight p p' ≠ None ⟹ e∈energies
    ⟹ order e (the (application (the (weight p p')) (the (inverse_application
(the (weight p p')) e))))"
```
⟨*proof*⟩

```
lemma inv_up_leq:
  "⋀p p' e. weight p p' ≠ None ⟹ e∈energies
  ⟹ application (the (weight p p')) e ≠ None
  ⟹ the (inverse_application (the (weight p p')) (the (application (the (weight
p p')) e))) e≤ e"
```
⟨*proof*⟩

```
lemma updates_monotonic:
  "⋀p p' e e'. weight p p' ≠ None ⟹e∈energies ⟹ 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*⟩

```
lemma inverse_monotonic:
  "⋀p p' e e'. weight p p' ≠ None ⟹ e∈energies ⟹ e e≤ e'
  ⟹ inverse_application (the (weight p p')) e ≠ None
  ⟹ the( inverse_application (the (weight p p')) e) e≤ the (inverse_application
(the (weight p p')) e')"
```
⟨*proof*⟩

Properties of the partial order.

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

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

lemma enumerate_arbitrary_in:
  shows "infinite A ⟹ enumerate_arbitrary A i ∈ A"
```
⟨*proof*⟩

```
lemma enumerate_arbitrary_neq:
  shows "infinite A ⟹ i < j
        ⟹ enumerate_arbitrary A i ≠ enumerate_arbitrary A j"
```
⟨*proof*⟩

```
lemma energy_Min_finite:
  assumes "A ⊆ energies"
  shows "finite (energy_Min A)"
```
⟨*proof*⟩

```
fun enumerate_decreasing :: "'energy set ⇒ nat ⇒ 'energy"  where
```

```
  "enumerate_decreasing A 0 = (SOME a. a ∈ A)" |
  "enumerate_decreasing A (Suc n)
    = (SOME x. (x ∈ A ∧ x e< enumerate_decreasing A n))"

lemma energy_Min_not_empty:
  assumes "A ≠ {}" and "A ⊆ energies"
  shows "energy_Min A ≠ {}"
⟨proof⟩

lemma energy_Min_contains_smaller:
  assumes "a ∈ A" and "A ⊆ energies"
  shows "∃b ∈ energy_Min A. b e≤ a"
⟨proof⟩

lemma energy_sup_leq_energy_sup:
  assumes "A ≠ {}" and "⋀a. a∈ A ⟹ ∃b∈ B. order a b" and
          "⋀a. a∈ A ⟹ a∈ energies" and "finite A" and "finite B" and "B ⊆ energies"
        shows "order (energy_sup A) (energy_sup B)"
⟨proof⟩
```

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 "e∈energies ∧ g ∉ attacker
            ∧ (∀g'. (weight g g' ≠ None) ⟶
                    ((application (the (weight g g')) e)≠ None
                     ∧ (winning_budget_len (the (application (the (weight g g'))
e))) g'))" |
 attacker: "winning_budget_len e g" if "e∈energies ∧ g ∈ attacker
            ∧ (∃g'. (weight g g' ≠ None)
                    ∧ (application (the (weight g g')) e)≠ None
                    ∧ (winning_budget_len (the (application (the (weight g g'))
e)) g'))"
```

We first restate the upward-closure of winning budgets.

```
lemma upwards_closure_wb_len:
  assumes "winning_budget_len e g" and "e e≤ e'"
  shows "winning_budget_len e' g"
⟨proof⟩
```

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 . e'∈energies}"

lemma winning_bugget_len_is_wb:
  assumes "nonpos_winning_budget = winning_budget"
  shows "winning_budget_len e g = (winning_budget e g ∧  e ∈energies)"
⟨proof⟩

end
end
```

4 Decidability of Galois Energy Games

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

In this theory we give a proof of decidability for Galois energy games (over vectors of naturals). We do this by providing a proof of correctness of the simplifyed version of Bisping's Algorithm to calculate minimal attacker winning budgets. We further formalise the key argument for its termination.

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

4.1 Minimal Attacker Winning Budgets as Pareto Fronts

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

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

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

```
lemma minimal_winning_budget_finite:
  shows "⋀g. finite (a_win_min g)"
⟨proof⟩
```

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

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

By definition minimal winning budgets are possible Pareto fronts.

```
lemma a_win_min_in_pareto:
  shows "a_win_min ∈ possible_pareto"
  ⟨proof⟩
```

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⟩

lemma pareto_partial_order:
  shows "reflp_on possible_pareto (≼)" and
        "transp_on possible_pareto (≼)" and
        "antisymp_on possible_pareto (≼)"
⟨proof⟩
```

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⟩

lemma pareto_directed_complete:
  shows "directed_complete possible_pareto (≼)"
  ⟨proof⟩

lemma pareto_minimal_element:
  shows "(λg. {}) ≼ F"
  ⟨proof⟩
```

4.2 Proof of Decidability

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

```
definition iteration:: "('position ⇒ 'energy set) ⇒ ('position ⇒ 'energy set)"
where
  "iteration F g ≡ (if g ∈ attacker
                    then energy_Min {inv_upd (the (weight g g')) e' | e' g'.
                        e' ∈ energies ∧ weight g g' ≠ None ∧ e' ∈ F g'}
                    else energy_Min {energy_sup
                         {inv_upd (the (weight g g')) (e_index g') | g'.
                         weight g g' ≠ None} | e_index. ∀g'. weight g g' ≠ None
                         ⟶(e_index g')∈energies ∧ e_index g' ∈ F g'})"
```

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"
  ⟨proof⟩

lemma iteration_monotonic:
  assumes "F ∈ possible_pareto" and "F' ∈ possible_pareto" and "F ≼ F'"
  shows "iteration F ≼ iteration F'"
  ⟨proof⟩

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')"
  ⟨proof⟩

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⟩

lemma iteration_scott_continuous:
  shows "scott_continuous possible_pareto (≼) possible_pareto (≼) iteration"
⟨proof⟩
```

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⟩
```

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⟩
```

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
              {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 ⟹ e ∈ energies"
⟨proof⟩

lemma a_win_min_is_minS:
  shows "energy_Min {e. S e g} = a_win_min g"
⟨proof⟩
```

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⟩
```

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⟩
```

4.3 Applying Kleene's Fixed Point Theorem

We now establish compatablity with Complete_Non_Orders.thy.

```
sublocale attractive possible_pareto pareto_order
  ⟨proof⟩

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⟩
```

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⟩

end
end
```

5 Vectors of (extended) Naturals as Energies

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

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

```
type_synonym energy = "enat list"

definition energy_leq:: "energy ⇒ 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⟩
```

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⟩
```

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 ≠ {}"
⟨proof⟩

lemma energy_Min_contains_smaller:
  assumes "a ∈ A"
  shows "∃b ∈ energy_Min A. b e≤ a"
⟨proof⟩
```

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⟩
```

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⟩

lemma enumerate_arbitrary_neq:
  shows "infinite A ⟹ i < j
        ⟹ enumerate_arbitrary A i ≠ enumerate_arbitrary A j"
⟨proof⟩

lemma energy_Min_finite:
  assumes "⋀e. e∈ A ⟹ length e = n"
  shows "finite (energy_Min A)"
⟨proof⟩
```

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⟩
```

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⟩

lemma empty_Sup_is_zero:
  assumes "i < n"
  shows "(energy_sup n {}) ! i = 0"
⟨proof⟩

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⟩

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⟩

lemma upd_domain_upward_closed:
  assumes "apply_update u e ≠ None" and "e e≤ e'"
  shows "apply_update u e' ≠ None"
⟨proof⟩
```

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')"
  ⟨proof⟩
```

6.2 Bisping's Inversion

The “inverse” of an update u is a function mapping energies e to $\min\{e' \mid e \le u(e')\}$ 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"
  ⟨proof⟩

lemma inv_not_none:
  assumes "length u = length e"
  shows "apply_inv_update u e ≠ None"
  ⟨proof⟩

lemma inv_not_none_then:
  assumes "apply_inv_update u e ≠ None"
  shows "(apply_update u (inv_upd u e)) ≠ None"
⟨proof⟩
```

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')"
  ⟨proof⟩
```

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]"
  ⟨proof⟩

lemma not_right_inverse:
  shows "∃u. ∃e. apply_inv_update u (upd u e) ≠ Some e"
  ⟨proof⟩

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]"
  ⟨proof⟩

lemma not_left_inverse:
  shows "∃u. ∃e. apply_update u (inv_upd u e) ≠ Some e"
  ⟨proof⟩
```

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))"
  ⟨proof⟩


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⟩
```

We now show that`apply_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"
  ⟨proof⟩
```

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⟩

end
```

7 Galois Energy Games over Naturals

```
theory Natural_Galois_Energy_Game
  imports Energy_Game Energy_Order Decidability Update
begin
```

We now define Galois energy games over vectors of naturals with the component-wise order. We formalise this in this theory as an `energy_game` with a fixed dimension. In particular, we assume all updates to have an upward-closed domain (as `domain_upw_closed`) and be length-preserving (as `len_appl`). We assume the latter for the inversion of updates too (as `len_inv_appl`) and assume that the inversion of an update is a total mapping from energies to the domain of the update (as `domain_inv`).

```
locale natural_galois_energy_game = energy_game attacker weight application
  for   attacker ::  "'position set" and
        weight :: "'position ⇒ 'position ⇒ 'label option" and
        application :: "'label ⇒ energy ⇒ energy option" and
        inverse_application :: "'label ⇒ energy ⇒ energy option"
+
  fixes dimension :: "nat"
  assumes
    domain_upw_closed: "⋀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 len_appl: "⋀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 len_inv_appl: "⋀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_applica
(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_applicatio
(the (weight p p')) e)) e≤ e' = e e≤ (the (application (the (weight p p')) e'))"


sublocale natural_galois_energy_game ⊆ galois_energy_game attacker weight application
inverse_application "{e::energy. length e = dimension}" energy_leq "λs. energy_sup
dimension s"
⟨proof⟩

locale natural_galois_energy_game_decidable = natural_galois_energy_game attacker
weight application inverse_application dimension
  for attacker ::  "'position set" and
      weight :: "'position ⇒ 'position ⇒ 'label option" and
      application :: "'label ⇒ energy ⇒ energy option" and
      inverse_application :: "'label ⇒ energy ⇒ energy option" and
      dimension :: "nat"
+
assumes nonpos_eq_pos: "nonpos_winning_budget = winning_budget" and
        finite_positions: "finite positions"

sublocale natural_galois_energy_game_decidable ⊆ galois_energy_game_decidable attacker
weight application inverse_application "{e::energy. length e = dimension}" energy_leq
"λs. energy_sup dimension s"
⟨proof⟩
```

Bisping's only considers declining energy games over vectors of naturals. We generalise this by considering all valid updates. We formalise this in this theory as an `energy_game` with a fixed dimension and show that such games are Galois energy games.

```
locale bispings_energy_game = energy_game attacker weight apply_update
  for attacker ::  "'position set" and
      weight :: "'position ⇒ 'position ⇒ update option"
+
  fixes dimension :: "nat"
  assumes
    valid_updates: "∀p. ∀p'. ((weight p p' ≠ None )
                    ⟶ ((length (the (weight p p')) = dimension)
                    ∧ valid_update (the (weight p p'))))"

sublocale bispings_energy_game ⊆ natural_galois_energy_game attacker weight apply_update
apply_inv_update dimension
⟨proof⟩

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 ⊆ natural_galois_energy_game_decidable
attacker weight apply_update apply_inv_update dimension
⟨proof⟩

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

[5] 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"
⟨proof⟩

lemma the_those_n:
  assumes "those (l:: 'a option list) ≠ None" and "(n::nat) < length l"
  shows "(the (those l)) ! n = the (l ! n)"
  ⟨proof⟩

lemma those_all_Some:
  assumes "those l ≠ None" and "n < length l"
  shows "(l ! n)≠None"
  ⟨proof⟩

lemma nth_map_enumerate:
  shows "n < length xs ⟹ (map f (List.enumerate 0 xs))!n = f((List.enumerate 0
xs)!n)"
⟨proof⟩

lemma those_map_not_None:
  assumes "∀n< length xs. f (xs ! n) ≠ None"
  shows "those (map f xs) ≠ None"
⟨proof⟩

lemma last_len:
  assumes "length xs = Suc n"
  shows "last xs = xs ! n"
  ⟨proof⟩

end
```