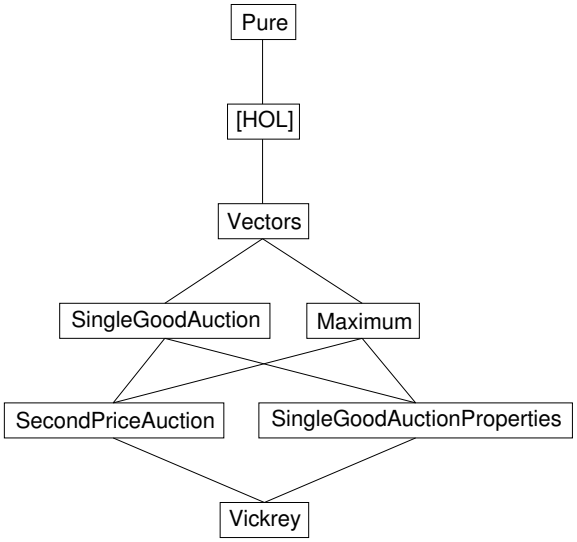
Auction Theory Toolbox

ForMaRE

April 3, 2013

Contents



1 Vickrey's Theorem: second price auctions are efficient, and truthful bidding is a weakly dominant strategy – copy to experiment with "case checking"

theory *Vickrey_CaseChecker*
imports *Complex_Main*
begin

2 Single good auctions

2.1 Preliminaries

type_synonym *participant* = *nat*
type_synonym *participants* = *nat set*

type_synonym *'a vector* = *participant* ⇒ *'a*

definition *non_negative_real_vector* :: *participants* ⇒ *real vector* ⇒ *bool*
  where *non_negative_real_vector N v* ⟷ (∀ *i* ∈ *N*. *v i* ≥ *0*)

definition *positive_real_vector* :: *participants* ⇒ *real vector* ⇒ *bool*
  where *positive_real_vector N v* ⟷ (∀ *i* ∈ *N*. *v i* > *0*)

convenience type synonyms for most of the basic concepts we are dealing with

type_synonym *valuations* = *real vector*
type_synonym *bids* = *real vector*
type_synonym *allocation* = *real vector*
type_synonym *payments* = *real vector*

Initially we'd like to formalise any single good auction as a relation of bids and outcome. Proving the well-definedness of an auction is then a separate step in the auction design process. It involves:

1. checking that the allocation and payments vectors actually meet our expectation of an allocation or payment, as defined by the *allocation_def* and *vickrey_payment* predicates below
2. checking that the relation actually is a function, i.e. that it is
   (a) left-total: "for any admissible bids ..."
   (b) right-unique: "... there is a unique outcome."

type_synonym *single_good_auction* = ((*participants* × *bids*) × (*allocation* × *payments*)) *set*

2.2 Valuation

definition *valuations* :: *participants* ⇒ *valuations* ⇒ *bool*
where *valuations N v* ⟷ *positive_real_vector N v*

2.3 Strategy (bids)

definition *bids* :: *participants* ⇒ *bids* ⇒ *bool*
where *bids N b* ⟷ *non_negative_real_vector N b*

lemma *valuation_is_bid*: *valuations N v* ⟹ *bids N v*
by (*auto simp add*: *valuations_def positive_real_vector_def bids_def non_negative_real_vector_def*)

2.4 Allocation

We employ the general definition of an allocation for a divisible single good. This is to allow for more possibilities of an auction to be not well-defined. Also, it is no longer the allocation that we model as a function of the bid, but instead we model the *auction* as a relation of bids to a (*allocation* × *payments*) outcome.

definition *allocation* :: *participants* ⇒ *allocation* ⇒ *bool*
where *allocation N x* ⟷ $(\sum\ i \in N\ .\ x\ i) = 1$

2.5 Payment

Same as with the *allocation* we now model this as a plain vector.

definition *vickrey_payment* :: *participants* ⇒ *payments* ⇒ *bool*
where *vickrey_payment N p* ⟷ $(\forall\, i \in N.\ p\ i \geq 0)$

2.6 Payoff

definition *payoff* :: *real* ⇒ *real* ⇒ *real* ⇒ *real*
where *payoff v x p* = *v* ∗ *x* − *p*

To give the auction designer flexibility (including the possibility to introduce mistakes), we only constrain the left hand side of the relation, as to cover admissible *bids*. This definition makes sure that whenever we speak of a single

good auction, there is a bid vector on the left hand side. In other words, this predicate returns false for relations having left hand side entries that are known not to be bid vectors. For this and other purposes it is more convenient to treat the auction as a predicate over all of its arguments, instead of a left-hand-side/right-hand-side relation.

definition *sga_pred* :: *participants* ⇒ *bids* ⇒ *allocation* ⇒ *payments* ⇒ *bool*
where
sga_pred N b x p ⟷ *bids N b*

We construct the relational version of an auction from the predicate version: given a predicate that defines an auction by telling us for all possible arguments whether they form an (input, outcome) pair according to the auction's definition, we construct a predicate that tells us whether all (input, outcome) pairs in a given relation satisfy that predicate, i.e. whether the given relation is an auction of the desired type.

definition *rel_sat_sga_pred* ::
(*participants* ⇒ *bids* ⇒ *allocation* ⇒ *payments* ⇒ *bool*) ⇒ *single_good_auction* ⇒ *bool*
where *rel_sat_sga_pred pred A* ⟷ (∀ ((*N*, *b*), (*x*, *p*)) ∈ *A* . *pred N b x p*)

A variant of *rel_sat_sga_pred*: We construct a predicate that tells us whether the given relation comprises all (input, outcome) pairs that satisfy the given auction predicate, i.e. whether the given relation comprises all possible auctions of the desired type.

definition *rel_all_sga_pred* ::
(*participants* ⇒ *bids* ⇒ *allocation* ⇒ *payments* ⇒ *bool*) ⇒ *single_good_auction* ⇒ *bool*
where *rel_all_sga_pred pred A* ⟷ (∀ *N b x p* . ((*N*, *b*), (*x*, *p*)) ∈ *A* ⟷ *pred N b x p*)

Now for the relational version of the single good auction:

definition *single_good_auction* :: *single_good_auction* ⇒ *bool*
where *single_good_auction A* = *rel_sat_sga_pred sga_pred A*

In the general case, by "well-defined outcome" we mean that the good gets properly allocated w.r.t. the definition of an *allocation*. We are not constraining the payments at this point.

definition *sga_well_defined_outcome_pred* :: *participants* ⇒ *bids* ⇒ *allocation* ⇒ *payments* ⇒ *bool*
where
sga_well_defined_outcome_pred N b x p ⟷ *allocation N x*

definition *sga_well_defined_outcome* :: *single_good_auction* ⇒ *bool*
where
sga_well_defined_outcome A ⟷
(∀ ((*N*::*participants*, *b*::*bids*), (*x*::*allocation*, *p*::*payments*)) ∈ *A* .
sga_well_defined_outcome_pred N b x p)

type_synonym *input_admissibility* = *participants* ⇒ *bids* ⇒ *bool*

Left-totality of an auction defined as a relation: for each admissible bid vector there exists some outcome (not necessarily unique).

definition *sga_left_total* :: *single_good_auction* ⇒ *input_admissibility* ⇒ *bool*
  where *sga_left_total A admissible* ⟷
    (∀ (*N* :: *participants*) (*b* :: *bids*) . *admissible N b* ⟶
      (∃ (*x* :: *allocation*) (*p* :: *payments*) . ((*N*, *b*), (*x*, *p*)) ∈ *A*))

type_synonym *outcome_equivalence* = *participants* ⇒ *bids* ⇒ *allocation* ⇒ *payments*
⇒ *allocation* ⇒ *payments* ⇒ *bool*

Right-uniqueness of an auction defined as a relation: for each admissible bid vector, if there is an outcome, it is unique.

definition *sga_right_unique* :: *single_good_auction* ⇒ *input_admissibility* ⇒ *outcome_equivalence*
⇒ *bool*
  where *sga_right_unique A admissible equivalent* ⟷
    (∀ (*N* :: *participants*) (*b* :: *bids*) . *admissible N b* ⟶
      (∀ (*x* :: *allocation*) (*x'* :: *allocation*) (*p* :: *payments*) (*p'* :: *payments*) .
        ((*N*, *b*), (*x*, *p*)) ∈ *A* ∧ ((*N*, *b*), (*x'*, *p'*)) ∈ *A* ⟶ *equivalent N b x p x' p'*))

definition *sga_function* :: *single_good_auction* ⇒ *input_admissibility* ⇒ *outcome_equivalence*
⇒ *bool*
  where *sga_function A admissible equivalent* ⟷
    sga_left_total A admissible ∧ *sga_right_unique A admissible equivalent*

2.7 Maximum

This subsection uses Isabelle's set maximum functions, wrapping them for our use.

definition *maximum_defined* :: *participants* ⇒ *bool*
  where *maximum_defined N* ⟷ *card N* > *0*

definition *maximum* :: *participants* ⇒ *real vector* ⇒ *real*
  where *maximum N y* = *Max* (*y* ‘ *N*)

lemma *maximum_equal*:
  fixes *N* :: *participants* **and** *y* :: *real vector* **and** *z* :: *real vector*
  assumes ∀ *i* ∈ *N*. *y i* = *z i*
  shows *maximum N y* = *maximum N z*
proof −
  have *y* ‘ *N* = *z* ‘ *N* **by** (*rule image_cong*) (*auto simp add*: *assms*)
  then show *?thesis* **unfolding** *maximum_def* **by** *simp*
qed

lemma *maximum_is_greater_or_equal*:
  fixes *N* :: *participants* **and** *y* :: *real vector* **and** *i* :: *participant*
  assumes *maximum_defined N*
    and *i* ∈ *N*
  shows *maximum N y* ≥ *y i*
 using *assms* **unfolding** *maximum_defined_def maximum_def* **by** (*simp add*: *card_gt_0_iff*)

lemma *maximum_is_component*:
  fixes *N* :: *participants* **and** *y* :: *real vector*
  assumes *defined*: *maximum_defined N*
  shows ∃ *i* ∈ *N*. *maximum N y* = *y i*
proof −

```
  let ?A = y ‘ N
  from defined have finite ?A and ?A ≠ {}
    unfolding maximum_defined_def by (simp_all add: card_gt_0_iff)
  then have Max ?A ∈ ?A by (rule Max_in)
  then obtain i where i ∈ N and Max ?A = y i by blast
  with maximum_def show ?thesis by auto
qed

lemma maximum_sufficient:
  fixes N :: participants and y :: real vector and m :: real
  assumes defined: maximum_defined N
    and greater_or_equal: ∀ i ∈ N. m ≥ y i
    and is_component: ∃ i ∈ N. m = y i
  shows maximum N y = m
  unfolding maximum_def
proof −
  let ?A = y ‘ N
  show Max ?A = m
  proof (rule Max_eqI)
    from defined show finite ?A
      unfolding maximum_defined_def by (simp add: card_gt_0_iff)
    show m ∈ ?A using is_component by blast
    fix a assume a ∈ ?A
    then show a ≤ m using greater_or_equal by blast
  qed
qed

definition arg_max_set :: participants ⇒ real vector ⇒ participants
  where arg_max_set N b = {i. i ∈ N ∧ maximum N b = b i}

lemma maximum_except_is_greater_or_equal:
  fixes N :: participants and y :: real vector and j :: participant and i :: participant
  assumes defined: maximum_defined N
    and i: i ∈ N ∧ i ≠ j
  shows maximum (N − {j}) y ≥ y i
proof −
  let ?M = N − {j}
  let ?A = y ‘ ?M
  from i have ∗: i ∈ ?M by simp
  with defined have finite ?A and ?A ≠ {}
    unfolding maximum_defined_def by (auto simp add: card_gt_0_iff)
  with ∗ have Max ?A ≥ y i by (auto simp add: Max_ge_iff)
  then show ?thesis unfolding maximum_def .
qed

lemma maximum_remaining_maximum:
  fixes N :: participants and y :: real vector and j :: participant
  assumes defined′: maximum_defined (N − {j})
    and j_max: y j = maximum N y
  shows y j ≥ maximum (N − {j}) y
proof −
  have y ‘ (N − {j}) ⊆ y ‘ N by blast
  with defined′ have maximum (N − {j}) y ≤ maximum N y
   unfolding maximum_def maximum_defined_def by (simp add: card_gt_0_iff Max_mono)
```

also note *j_max* [*symmetric*]
finally show *?thesis* **.**
qed

lemma *remaining_maximum_invariant*:
fixes N :: *participants* **and** y :: *real vector* **and** i :: *participant* **and** a :: *real*
shows *maximum* $(N - \{i\})$ $(y(i := a))$ = *maximum* $(N - \{i\})$ y
proof −
let *?M* = $N - \{i\}$
have y ‘ *?M* = $(y(i := a))$ ‘ *?M* **by** *auto*
then show *?thesis* **unfolding** *maximum_def* **by** *simp*
qed

2.8 Second price single good auctions and some of their properties

definition *second_price_auction_winner* ::
participants ⇒ *bids* ⇒ *allocation* ⇒ *payments* ⇒ *participant* ⇒ *bool*
where
second_price_auction_winner N b x p i ⟷
$i \in N \wedge i \in$ *arg_max_set N b* $\wedge$ *x i* = *1* $\wedge$ *p i* = *maximum* $(N - \{i\})$ *b*

definition *second_price_auction_loser* ::
participants ⇒ *allocation* ⇒ *payments* ⇒ *participant* ⇒ *bool*
where *second_price_auction_loser N x p i* ⟷ $i \in N \wedge$
x i = *0* $\wedge$
p i = *0*

definition *spa_pred* :: *participants* ⇒ *bids* ⇒ *allocation* ⇒ *payments* ⇒ *bool*
where
spa_pred N b x p ⟷
bids N b $\wedge$
$(\exists\, i \in N.$ *second_price_auction_winner N b x p i* $\wedge$
$(\forall\, j \in N\ .\ j \neq i \longrightarrow$ *second_price_auction_loser N x p j*$))$

definition *second_price_auction* :: *single_good_auction* ⇒ *bool*
where *second_price_auction A* = *rel_sat_sga_pred spa_pred A*

definition of a second price auction, projected to the allocation

lemma *spa_allocation* :
fixes N :: *participants* **and** b :: *bids*
and x :: *allocation* **and** p :: *payments*
assumes *spa_pred N b x p*
shows $\exists\ i \in N\ .\ x\ i = 1 \wedge (\forall\ j \in N\ .\ j \neq i \longrightarrow x\ j = 0)$
using *assms*
unfolding *spa_pred_def second_price_auction_def second_price_auction_winner_def second_price_auction_loser_def*
by *metis*

Our second price auction (*spa_pred*) is well-defined in that its outcome is an allocation.

lemma *spa_allocates* :
fixes N :: *participants* **and** b :: *bids*

and x :: *allocation* **and** p :: *payments*
assumes *spa*: *spa_pred N b x p*
and *finite*: *finite N*
shows *allocation N x*
proof −
from *spa* **obtain** i **where** *i_def*: $i \in N \wedge x\ i = 1$ **using** *spa_allocation* **by** *blast*

with *spa* **have** $\forall j \in N - \{i\}\ .\ x\ j = 0$ **using** *spa_allocation* **by** (*metis member_remove remove_def*)
then have $(\sum k \in N\ .\ x\ k) = 1$ **using** *finite i_def* **by** (*metis comm_monoid_add_class.add.right_neutral setsum.F_neutral' setsum.remove*)
then show *?thesis* **unfolding** *allocation_def* **by** *simp*
qed

lemma *spa_well_defined_sga* :
fixes N :: *participants* **and** b :: *bids*
and x :: *allocation* **and** p :: *payments*
assumes *spa*: *spa_pred N b x p*
and *finite*: *finite N*
shows *sga_well_defined_outcome_pred N b x p*
using *assms spa_allocates* **unfolding** *allocation_def sga_well_defined_outcome_pred_def*
by *simp*

definition of a second price auction, projected to the payments

lemma *spa_payments* :
fixes N :: *participants* **and** b :: *bids*
and x :: *allocation* **and** p :: *payments*
assumes *spa_pred N b x p*
shows $\exists\ i \in N\ .\ p\ i = maximum\ (N - \{i\})\ b \wedge (\forall\ j \in N\ .\ j \neq i \longrightarrow p\ j = 0)$
using *assms*
unfolding *spa_pred_def second_price_auction_def second_price_auction_winner_def second_price_auction_loser_def*
by *metis*

Our second price auction (*spa_pred*) is well-defined in that its outcome specifies non-negative payments for everyone.

lemma *spa_vickrey_payment* :
fixes N :: *participants* **and** b :: *bids*
and x :: *allocation* **and** p :: *payments*
assumes *spa*: *spa_pred N b x p*
and *card_N*: *card N* > 1
shows *vickrey_payment N p*
proof −
from *card_N* **have** *maximum_defined*: *maximum_defined N* **unfolding** *maximum_defined_def*
by *auto*
from *spa* **obtain** i **where** *i_range*: $i \in N$
and *i_pay*: $p\ i = maximum\ (N - \{i\})\ b$
and *losers_pay*: $\forall\ j \in N\ .\ j \neq i \longrightarrow p\ j = 0$
using *spa_payments* **by** *blast*

from *card_N* **and** *i_range* **obtain** k **where** *k_def*: $k \in N \wedge k \neq i$
by (*metis all_not_in_conv card.insert card_empty ex_least_nat_le finite.emptyI insertCI le0 monoid_add_class.add.right_neutral nonempty_iff not_less*)

from *k_def* **and** *maximum_defined* **have** *greater*: *maximum* $(N - \{i\})\ b \geq b\ k$
using *maximum_except_is_greater_or_equal* **by** *blast*
also have $\ldots \geq 0$ **using** *spa spa_pred_def second_price_auction_def bids_def non_negative_real_vector_def*
by (*smt greater k_def*)
with *i_pay* **and** *calculation* **have** $p\ i \geq 0$ **by** *simp*
with *i_range* **and** *losers_pay* **have** $\forall\ k \in N\ .\ p\ k \geq 0$ **by** *auto*
with *vickrey_payment_def* **show** *?thesis* **by** *blast*
qed

definition *spa_admissible_input* :: *participants* ⇒ *bids* ⇒ *bool*
where *spa_admissible_input N b* ⟷ *card N* > *1* ∧ *bids N b*

Our relational definition of second price auction is left-total.

lemma *spa_is_left_total* :
fixes *A* :: *single_good_auction*

assumes *spa*: *rel_all_sga_pred spa_pred A*
shows *sga_left_total A spa_admissible_input*
proof −
{
fix *N* :: *participants* **and** *b* :: *bids*
assume *admissible*: *spa_admissible_input N b*

then obtain *winner*::*participant* **where** *winner_def*: *winner* ∈ *N* ∧ *winner* ∈ *arg_max_set N b*
using *spa_admissible_input_def arg_max_set_def maximum_def maximum_def maximum_is_component maximum_defined_def*
by (*smt mem_Collect_eq*)

def *x* ≡ λ *i*::*participant* . *if i* = *winner then 1*::*real else 0*
def *p* ≡ λ *i*::*participant* . *if i* = *winner then maximum* $(N - \{i\})\ b$ *else 0*
from *x_def* **and** *p_def* **and** *winner_def* **and** *admissible*
have *spa_pred N b x p*
using *spa_admissible_input_def second_price_auction_winner_def second_price_auction_loser_def spa_pred_def*
by *auto*
with *spa* **have** ∃ (*x* :: *allocation*) (*p* :: *payments*) . ((*N*, *b*), (*x*, *p*)) ∈ *A*
using *spa_pred_def rel_all_sga_pred_def* **by** *auto*
}
then show *?thesis* **unfolding** *sga_left_total_def* **by** *blast*
qed

We consider two outcomes of a second price auction equivalent if

1. the bids of the participants to which the good is allocated are equal
2. the bids of the participants with positive payments are equal

This should be as weak as possible, as not to restate the definition.

definition *spa_equivalent_outcome* :: *participants* ⇒ *bids* ⇒ *allocation* ⇒ *payments* ⇒ *allocation* ⇒ *payments* ⇒ *bool*
where *spa_equivalent_outcome N b x p x′ p′* ⟷
b ‘ { *i* ∈ *N* . *x i* = *1* } = *b* ‘ { *i* ∈ *N* . *x′ i* = *1* } ∧
b ‘ { *i* ∈ *N* . *p i* > *0* } = *b* ‘ { *i* ∈ *N* . *p′ i* > *0* }

lemma *spa_is_sga_pred* :
fixes N :: *participants* **and** b :: *bids*
and x :: *allocation* **and** p :: *payments*
assumes *spa_pred N b x p*
shows *sga_pred N b x p*
using *assms*
unfolding *spa_pred_def sga_pred_def* **by** *blast*

lemma *spa_is_sga* :
fixes A :: *single_good_auction*
assumes *spa*: *second_price_auction A*
shows *single_good_auction A*
proof −
{
fix N :: *participants* **and** b :: *bids* **and** x :: *allocation* **and** p :: *payments*
assume $((N,\ b),\ (x,\ p)) \in A$
with *spa* **have** *spa_pred N b x p* **unfolding** *second_price_auction_def rel_sat_sga_pred_def*
by *blast*
then have *sga_pred N b x p* **using** *spa_is_sga_pred* **by** *blast*
}
then show *?thesis* **unfolding** *single_good_auction_def rel_sat_sga_pred_def* **by** *blast*
qed

lemma *spa_allocates_binary* :
fixes N :: *participants* **and** b :: *bids*
and x :: *allocation* **and** p :: *payments*
and i :: *participant*
assumes *spa_pred N b x p*
and $i \in N$
shows $x\ i = 0 \vee x\ i = 1$
using *assms*
unfolding *spa_pred_def second_price_auction_def second_price_auction_winner_def second_price_auction_loser_def*
by *fast*

lemma *allocated_implies_spa_winner*:
fixes N :: *participants* **and** b :: *bids*
and x :: *allocation* **and** p :: *payments*
and *winner* :: *participant*
assumes *spa_pred N b x p*
and *winner* $\in N$
and x *winner* = *1*
shows *second_price_auction_winner N b x p winner*
using *assms*
unfolding *spa_pred_def second_price_auction_def second_price_auction_winner_def second_price_auction_loser_def*
allocation_def
by (*metis zero_neq_one*)

```
lemma not_allocated_implies_spa_loser:
  fixes N :: participants and b :: bids
    and x :: allocation and p :: payments
    and loser :: participant
  assumes spa: spa_pred N b x p
    and range: loser ∈ N
    and loses: x loser = 0
  shows second_price_auction_loser N x p loser
proof −
  from loses have ¬ second_price_auction_winner N b x p loser
    unfolding second_price_auction_winner_def by simp
  with spa range show ?thesis
      unfolding spa_pred_def second_price_auction_def by fast
qed

lemma only_max_bidder_wins:
  fixes N :: participants and max_bidder :: participant
    and b :: bids and x :: allocation and p :: payments
  assumes defined: maximum_defined N
    and spa: spa_pred N b x p
    and range: max_bidder ∈ N
    and only_max_bidder: b max_bidder > maximum (N − {max_bidder}) b
  shows second_price_auction_winner N b x p max_bidder
proof −
  from spa have spa_unfolded:
    bids N b ∧ (∃ i. second_price_auction_winner N b x p i ∧
      (∀ j ∈ N. j ≠ i ⟶ second_price_auction_loser N x p j))
    unfolding spa_pred_def second_price_auction_def by blast
  {
    fix j :: participant
    assume j_not_max: j ∈ N ∧ j ≠ max_bidder
    have j ∉ arg_max_set N b
    proof
      assume j ∈ arg_max_set N b
      then have maximum: b j = maximum N b unfolding arg_max_set_def by simp

      from j_not_max range have b j ≤ maximum (N − {max_bidder}) b
        using defined maximum_except_is_greater_or_equal by simp
      with only_max_bidder have ∗: b j < b max_bidder by simp

      from defined range maximum have b j ≥ b max_bidder
        by (simp add: maximum_is_greater_or_equal)
      with ∗ show False by simp
    qed
  }
    with spa_unfolded show ?thesis
    by (auto simp add: second_price_auction_winner_def)
qed

lemma second_price_auction_winner_payoff:
  fixes N :: participants and v :: valuations and x :: allocation
    and b :: bids and p :: payments and winner :: participant
  assumes defined: maximum_defined N
    and spa: spa_pred N b x p
```

```
    and i_range: i ∈ N
    and wins: x i = 1
  shows payoff (v i) (x i) (p i) = v i − maximum (N − {i}) b
proof −
  have payoff (v i) (x i) (p i) = v i − p i
    using wins unfolding payoff_def by simp
  also have ... = v i − maximum (N − {i}) b
    using defined spa i_range wins
      using allocated_implies_spa_winner
    unfolding second_price_auction_winner_def
    by simp
  finally show ?thesis .
qed

lemma second_price_auction_loser_payoff:
  fixes N :: participants and v :: valuations and x :: allocation
    and b :: bids and p :: payments and loser :: participant
  assumes spa_pred N b x p
    and i ∈ N
    and x i = 0
  shows payoff (v i) (x i) (p i) = 0
  using assms not_allocated_implies_spa_loser
  unfolding second_price_auction_loser_def payoff_def by simp

lemma winners_payoff_on_deviation_from_valuation:
  fixes N :: participants and v :: valuations and x :: allocation
    and b :: bids and p :: payments and winner :: participant
  assumes maximum_defined N
    and spa_pred N b x p
    and i ∈ N
    and x i = 1
  shows payoff (v i) (x i) (p i) = v i − maximum (N − {i}) (b(i := v i))
  using assms second_price_auction_winner_payoff remaining_maximum_invariant
  by simp
```

3 Some properties that single good auctions can have

3.1 Efficiency

```
definition efficient :: participants ⇒ valuations ⇒ bids ⇒ single_good_auction ⇒
bool
  where
    efficient N v b A ⟷
      valuations N v ∧ bids N b ∧ single_good_auction A ∧
      (∀ x p . ((N, b), (x, p)) ∈ A
        (∗ TODO CL: Is there a way of not naming p, as we don't need it? ∗)
        ⟶
        (∀ i ∈ N. x i = 1 ⟶ i ∈ arg_max_set N v))
```

3.2 Equilibrium in weakly dominant strategies

```
definition equilibrium_weakly_dominant_strategy ::
```

```
  participants ⇒ valuations ⇒ bids ⇒ single_good_auction ⇒ bool where
  equilibrium_weakly_dominant_strategy N v b A ⟷
    valuations N v ∧ bids N b ∧ single_good_auction A ∧
    (∀ i ∈ N.
      (∀ whatever_bid . bids N whatever_bid ∧ whatever_bid i ≠ b i ⟶
        (let b' = whatever_bid(i := b i)
         in
         (∀ x p x' p' . ((N, whatever_bid), (x, p)) ∈ A ∧ ((N, b'), (x', p')) ∈ A
          ⟶
          payoff (v i) (x' i) (p' i) ≥ payoff (v i) (x i) (p i)))))
```

4 Vickrey's Theorem

4.1 Part 1: A second-price auction supports an equilibrium in weakly dominant strategies if all participants bid their valuation.

```
theorem vickreyA:
  fixes N :: participants and v :: valuations and A :: single_good_auction
  assumes card_N: card N > 1
  assumes val: valuations N v
  defines b ≡ v
  assumes spa: second_price_auction A
  shows equilibrium_weakly_dominant_strategy N v b A
proof −
  have defined: maximum_defined N using card_N
    unfolding maximum_defined_def by (auto simp: card_ge_0_finite)
  then have finite: finite N by (metis card_N card_infinite less_nat_zero_code)

  from val have bids: bids N b
    unfolding b_def by (rule valuation_is_bid)
  {
    fix i :: participant
    assume i_range: i ∈ N

    let ?M = N − {i}
    have defined': maximum_defined ?M
      using card_N i_range unfolding maximum_defined_def
      by (simp add: card_ge_0_finite card_Diff_singleton)

    fix whatever_bid :: bids
    assume alternative_is_bid: bids N whatever_bid


    let ?b = whatever_bid(i := b i)

    have is_bid: bids N ?b
      using bids alternative_is_bid
      unfolding bids_def non_negative_real_vector_def by simp

    let ?b_max = maximum N ?b
    let ?b_max' = maximum ?M ?b
```

```
  fix x :: allocation and x' :: allocation and p :: payments and p' :: payments
  assume outcome: ((N, whatever_bid), (x, p)) ∈ A
    and outcome': ((N, ?b), (x', p')) ∈ A

  from spa outcome have spa_pred: spa_pred N whatever_bid x p
    unfolding second_price_auction_def rel_sat_sga_pred_def by blast
  from spa outcome' have spa_pred': spa_pred N ?b x' p'
    unfolding second_price_auction_def rel_sat_sga_pred_def by blast

  from spa_pred finite have allocation: allocation N x using spa_allocates by blast
  from spa_pred' finite have allocation': allocation N x' using spa_allocates by blast
  from spa_pred card_N have pay: vickrey_payment N p using spa_vickrey_payment
by blast
  from spa_pred' card_N have pay': vickrey_payment N p' using spa_vickrey_payment
by blast

  have weak_dominance:
    payoff (v i) (x' i) (p' i) ≥ payoff (v i) (x i) (p i)
  proof cases
    assume alloc: x' i = 1
    with spa_pred' i_range
    have winner: second_price_auction_winner N ?b x' p' i
      by (rule allocated_implies_spa_winner)

    from winner have ?b i = ?b_max
      unfolding second_price_auction_winner_def arg_max_set_def by simp
    with defined' have ?b i ≥ ?b_max'
      by (rule maximum_remaining_maximum)

    from winner have p' i = ?b_max'
      unfolding second_price_auction_winner_def
      by simp

    have winner_payoff: payoff (v i) (x' i) (p' i) = v i − ?b_max'
      using defined spa_pred' i_range alloc
      by (rule second_price_auction_winner_payoff)

    have non_negative_payoff: payoff (v i) (x' i) (p' i) ≥ 0
    proof −
      from `?b i ≥ ?b_max'` have ?b i − ?b_max' ≥ 0 by simp
      with winner_payoff show ?thesis unfolding b_def by simp
    qed

    show ?thesis
    proof cases — case 1a of the short proof
      assume x i = 1
      with defined spa_pred alternative_is_bid i_range
      have payoff (v i) (x i) (p i) = v i − ?b_max'
        using winners_payoff_on_deviation_from_valuation unfolding b_def by simp
      also have ... = payoff (v i) (x' i) (p' i) using winner_payoff by simp
      finally show ?thesis by (rule eq_refl)
    next — case 1b of the short proof
      assume x i ≠ 1
```

```
      with spa_pred alternative_is_bid i_range have x i = 0
        using spa_allocates_binary by blast
      with spa_pred i_range
      have payoff (v i) (x i) (p i) = 0
        by (rule second_price_auction_loser_payoff)
      also have ... ≤ payoff (v i) (x' i) (p' i) using non_negative_payoff by simp
      finally show ?thesis .
    qed

  next — case 2 of the short proof
    assume non_alloc: x' i ≠ 1

    with spa_pred' i_range have x' i = 0
      using spa_allocates_binary by blast
    with spa_pred' i_range
    have loser_payoff: payoff (v i) (x' i) (p' i) = 0
      by (rule second_price_auction_loser_payoff)

    have i_bid_at_most_second: ?b i ≤ ?b_max'
    proof (rule ccontr)
      assume ¬ ?thesis
      then have ?b i > ?b_max' by simp
      with defined spa_pred' i_range
      have second_price_auction_winner N ?b x' p' i
        using only_max_bidder_wins by simp
      with non_alloc show False using second_price_auction_winner_def by simp
    qed

    show ?thesis
    proof cases — case 2a of the short proof
      assume x i = 1
      with defined spa_pred i_range
      have payoff (v i) (x i) (p i) = ?b i − ?b_max'
        using winners_payoff_on_deviation_from_valuation unfolding b_def by simp
      also have ... ≤ 0 using i_bid_at_most_second by simp
      also have ... = payoff (v i) (x' i) (p' i) using loser_payoff by simp
      finally show ?thesis .
    next — case 2b of the short proof
      assume x i ≠ 1

      with spa_pred alternative_is_bid i_range have x i = 0
        using spa_allocates_binary by blast
      with spa_pred i_range
      have payoff (v i) (x i) (p i) = 0
        by (rule second_price_auction_loser_payoff)
      also have ... = payoff (v i) (x' i) (p' i) using loser_payoff by simp
      finally show ?thesis by (rule eq_refl)
    qed
  qed
}
with spa val bids show ?thesis
  using spa_is_sga
  unfolding equilibrium_weakly_dominant_strategy_def
  by auto
```

qed

4.2 Part 2: A second-price auction is efficient if all participants bid their valuation.

theorem *vickreyB*:
fixes N :: *participants* **and** v :: *valuations* **and** A :: *single_good_auction*
assumes *val*: *valuations N v*
assumes *spa*: *second_price_auction A*
defines $b \equiv v$
shows *efficient N v b A*
proof −
from *val* **have** *bids*: *bids N v* **by** (*rule valuation_is_bid*)
{
fix k :: *participant* **and** x :: *allocation* **and** p :: *payments*

assume *range*: $k \in N$ **and** *outcome*: $((N, v), (x, p)) \in A$ **and** *wins*: $x\ k = 1$
from *outcome spa* **have** *spa_pred N v x p*
unfolding *second_price_auction_def rel_sat_sga_pred_def*
by *blast*
with *range* **and** *wins* **have** $k \in$ *arg_max_set N v*
using *allocated_implies_spa_winner*
unfolding *second_price_auction_winner_def* **by** *blast*
}
with *bids spa* **show** *?thesis*
using *val* **unfolding** *b_def efficient_def* **using** *spa_is_sga* **by** *blast*
qed

end