The Correctness of Herr Schmidmeier's Algorithm for the Top Cycle

Florian Hübler, MC Sr

December 9, 2020

Contents

1 Auxiliary facts

lemma *ex-max-if-finite*: *finite* $S \Longrightarrow S \neq \{\} \Longrightarrow \exists\, m{\in}S.\ \neg(\exists\, x{\in}S.\ x > (m{::}'a{::}order))$
by (*induction rule*: *finite.induct*) (*auto intro*: *order.strict-trans*)

lemma *ex-is-arg-max-if-finite*:
fixes f :: $'a \Rightarrow\ 'b$ :: *order*
shows *finite* $S \Longrightarrow S \neq \{\} \Longrightarrow \exists\, x.$ *is-arg-max* f $(\lambda x.\ x \in S)$ x
unfolding *is-arg-max-def* **using** *ex-max-if-finite*[*of* f ‘ S] **by** *auto*

definition *repeat* **where** *repeat* n *xs* = *concat* (*replicate* n *xs*)

lemma *repeat-eq-Nil-iff* [*simp*]: *repeat* n *xs* = [] $\longleftrightarrow$ $n = 0 \vee xs = []$
by (*induction* n) (*auto simp*: *repeat-def*)

lemma *hd-concat* [*simp*]: *xss* $\neq$ [] $\Longrightarrow$ *hd xss* $\neq$ [] $\Longrightarrow$ *hd* (*concat xss*) = *hd* (*hd* *xss*)
by (*cases xss*) *auto*

lemma *hd-repeat* [*simp*]: $n > 0 \Longrightarrow$ *hd* (*repeat* n *xs*) = *hd xs*
by (*cases* $n = 0 \vee xs = []$) (*auto simp*: *repeat-def*)

lemma *length-repeat* [*simp*]: *length* (*repeat n xs*) = *n* ∗ *length xs*
by (*induction n*) (*auto simp*: *repeat-def*)

lemma *repeat-0* [*simp*]: *repeat 0 xs* = []
by (*simp add*: *repeat-def*)

lemma *repeat-Nil* [*simp*]: *repeat n* [] = []
by (*simp add*: *repeat-def*)

lemma *repeat-Suc* [*simp*]: *repeat* (*Suc n*) *xs* = *xs* @ *repeat n xs*
by (*simp add*: *repeat-def*)

2 Tournaments

A tournament (i.e. a total and asymmetric relation) with players of type $'a$ is represented by a function *Dom* mapping players to their dominion.

Ffor simplicity, we assume that the type of players is finite and that the tournament contains all the players of the type.

locale *tournament* =
fixes *Dom* :: $'a$ :: *finite* ⇒ $'a$ *set*
assumes *total*: $x \neq y \Longrightarrow x \in Dom\ y \lor y \in Dom\ x$
assumes *asym*: $x \notin Dom\ y \lor y \notin Dom\ x$
begin

lemma *not-in-Dom-iff* [*simp*]: $x \notin Dom\ y \longleftrightarrow x = y \lor y \in Dom\ x$
using *total asym* **by** *force*

lemma *asym′* [*simp*]: $x \in Dom\ y \Longrightarrow y \notin Dom\ x$
using *asym* **by** *auto*

lemma *irrefl* [*simp*]: $x \notin Dom\ x$
using *asym*[*of x x*] **by** *simp*

lemmas *Dom-props* = *total antisym*

2.1 Basic concepts

definition *covers* :: $'a \Rightarrow 'a \Rightarrow bool$ (**infixl** *covers 50*)
where $x\ covers\ y \longleftrightarrow Dom\ y \subseteq Dom\ x$

definition *dominant* :: $'a\ set \Rightarrow bool$ **where**
$dominant\ X \longleftrightarrow X \neq \{\} \land (\forall x{\in}X.\ \forall y{\in}{-X}.\ y \in Dom\ x)$

definition *CO* :: $'a$ *set*
where $CO = \{x.\ is\text{-}arg\text{-}max\ (card \circ Dom)\ (\lambda\text{-}.\ True)\ x\}$

definition *UC* :: $'a$ *set*

where $UC = \{x.\ \neg(\exists\, y.\ y \neq x \wedge y\ covers\ x)\}$

definition TC :: $'a\ set$
where $TC = \bigcap\{X.\ dominant\ X\}$

lemma CO-$nonempty$: $CO \neq \{\}$
using ex-is-arg-max-if-$finite[of\ UNIV\ card \circ Dom]$
unfolding CO-def **by** $simp$

lemma CO-$subset$-UC: $CO \subseteq UC$
proof
fix x **assume** $x \in CO$
show $x \in UC$
proof $(rule\ ccontr)$
assume $x \notin UC$
then obtain y **where** $y \neq x\ y\ covers\ x$
by $(auto\ simp$: UC-$def)$
hence $insert\ x\ (Dom\ x) \subseteq Dom\ y$
using Dom-$props$ **by** $(cases\ x \in Dom\ y)\ (auto\ simp$: $covers$-$def)$
moreover have $x \notin Dom\ x$
by $auto$
ultimately have $Dom\ x \subset Dom\ y$
by $blast$
hence $card\ (Dom\ x) < card\ (Dom\ y)$
by $(intro\ psubset$-$card$-$mono)\ auto$
with $\langle x \in CO\rangle$ **show** $False$
by $(auto\ simp$: CO-$def\ is$-arg-max-$def)$
qed
qed

lemma UC-$subset$-$dominant$:
assumes $dominant\ X$
shows $UC \subseteq X$
proof
fix x **assume** $x \in UC$
show $x \in X$
proof $(rule\ ccontr)$
assume x: $x \notin X$
from $x\ assms$ **have** $Dom\ x \subseteq -X$
using $assms\ Dom$-$props$ **by** $(auto\ simp$: $dominant$-$def)$
moreover obtain y **where** $y \in X\ -X \subseteq Dom\ y$
using $assms\ Dom$-$props$ **unfolding** $dominant$-def **by** $fast$
ultimately have $y\ covers\ x$
by $(auto\ simp$: $covers$-$def)$
with $\langle x \in UC\rangle\ \langle x \notin X\rangle\ \langle y \in X\rangle$ **show** $False$
by $(auto\ simp$: UC-$def)$
qed
qed

lemma *dominant-UNIV* [*intro*]: *dominant UNIV*
  by (*auto simp*: *dominant-def*)

lemma *dominant-INT* [*intro*]:
  assumes $\bigwedge X.\ X \in F \Longrightarrow$ *dominant* X
  shows *dominant* $(\bigcap F)$
proof $-$
  have $CO \subseteq UC$
    by (*rule CO-subset-UC*)
  also have $UC \subseteq \bigcap F$
    using *UC-subset-dominant assms* **by** *auto*
  finally have $\bigcap F \neq \{\}$
    using *CO-nonempty* **by** *blast*
  with *assms* **show** *?thesis* **unfolding** *dominant-def*
    by *auto*
qed

lemma *dominant-Int* [*intro*]:
  assumes *dominant* X **and** *dominant* Y
  shows *dominant* $(X \cap Y)$
  using *dominant-INT*[*of* $\{X, Y\}$] *assms* **by** *auto*

lemma *dominant-subset-total*:
  assumes *dominant* X **and** *dominant* Y
  shows $X \subseteq Y \vee Y \subseteq X$
proof (*rule ccontr*)
  assume $\neg(X \subseteq Y \vee Y \subseteq X)$
  then obtain $x\ y$ **where** *xy*: $x \in X - Y\ y \in Y - X$
    by *auto*
  from *xy* **have** $y \in$ *Dom* x
    using ‹*dominant* X› **by** (*auto simp*: *dominant-def*)
  moreover from *xy* **have** $x \in$ *Dom* y
    using ‹*dominant* Y› **by** (*auto simp*: *dominant-def*)
  ultimately show *False*
    by *auto*
qed

lemma *dominant-TC*: *dominant TC*
  unfolding *TC-def* **by** *auto*

lemma *UC-subset-TC*: $UC \subseteq TC$
  using *dominant-TC UC-subset-dominant* **by** *blast*

2.2 Chains and cycles

A chain is a list of element such that each element (except for the last one) is defeated by its successor.

fun *chain* :: *'a list* $\Rightarrow$ *bool* **where**
  chain [] $\longleftrightarrow$ *True*

| *chain* $[x] \longleftrightarrow$ *True*
| *chain* $(x \# y \# xs) \longleftrightarrow x \in Dom\ y \wedge chain\ (y \# xs)$

lemma *chain-ConsD*: *chain* $(x \# xs) \Longrightarrow$ *chain xs*
by (*cases xs*) *auto*

lemma *chain-append-iff*: *chain* $(xs @ z \# ys) \longleftrightarrow chain\ (xs @ [z]) \wedge chain\ (z \# ys)$
proof (*induction xs*)
case (*Cons x xs*)
thus *?case* **by** (*cases xs*) *auto*
qed *auto*

A cycle is a chain where the last element is additionally defeated by the first one.

definition *cycle* :: *'a list* $\Rightarrow$ *bool*
where *cycle xs* $\longleftrightarrow$ *chain* $(xs @ [hd\ xs])$

lemma *cycle-Nil* [*simp*]: *cycle* []
by (*simp add*: *cycle-def*)

lemma *cycle-appendI* [*intro*]:
assumes *cycle xs* **and** *cycle ys*
assumes $xs = [] \vee ys = [] \vee hd\ xs = hd\ ys$
shows *cycle* $(xs @ ys)$
proof (*cases* $xs = [] \vee ys = []$)
case *False*
obtain *y ys'* **where** [*simp*]: $ys = y \# ys'$
using *False* **by** (*cases ys*) *auto*
from *assms* **have** *chain* $(xs @ hd\ xs \# (tl\ ys @ [hd\ ys]))$
using *False* **by** (*subst chain-append-iff*) (*auto simp*: *cycle-def*)
thus *?thesis*
using *False assms* **by** (*simp add*: *cycle-def*)
qed (*use assms* **in** *auto*)

lemma *cycle-repeatI* [*intro*]:
assumes *cycle xs*
shows *cycle* (*repeat n xs*)
proof (*cases* $xs = [] \vee n = 0$)
case *False*
hence $n > 0$ $xs \neq []$
by *auto*
thus *?thesis* **using** *assms*
by (*induction n rule*: *nat-induct-non-zero*) *auto*
qed *auto*

2.3 The two iterative algorithms for TC

The function *step* performs one iteration of Herr Schmidmeier's algorithm (i.e. it computes the union of all dominators of elements of X).

definition *step* **where**
$\quad step\ X = (\bigcup x{\in}X.\ -Dom\ x-\{x\})$

The function $step'$ performs one iteration of the MC Sr's algorithm, which takes *adds* all the dominators of elements in X to X.

definition $step'$ **where**
$\quad step'\ X = X \cup step\ X$

We show some fairly obvious properties of *step* and $step'$.

lemma *step-subset*: $dominant\ Y \Longrightarrow X \subseteq Y \Longrightarrow step\ X \subseteq Y$
  by (*auto simp*: *step-def dominant-def*)

lemma *steps-subset*: $dominant\ Y \Longrightarrow X \subseteq Y \Longrightarrow (step\ \hat{}\hat{}\ n)\ X \subseteq Y$
  by (*induction n*) (*simp-all add*: *step-subset*)

lemma *step′-subset*: $dominant\ Y \Longrightarrow X \subseteq Y \Longrightarrow step'\ X \subseteq Y$
  unfolding *step′-def* **using** *step-subset* **by** *blast*

lemma *step′-dominant*: $dominant\ X \Longrightarrow step'\ X = X$
  by (*auto simp*: *dominant-def step′-def step-def*)

lemma *step′-TC* [*simp*]: $step'\ TC = TC$
  by (*rule step′-dominant*) (*rule dominant-TC*)

lemma *steps-mono*: $X \subseteq Y \Longrightarrow (step\ \hat{}\hat{}\ n)\ X \subseteq (step\ \hat{}\hat{}\ n)\ Y$
proof (*induction n arbitrary*: $X\ Y$)
  case (*Suc n*)
  have $(step\ \hat{}\hat{}\ n)\ (step\ X) \subseteq (step\ \hat{}\hat{}\ n)\ (step\ Y)$
    using *Suc.prems* **by** (*intro Suc.IH*) (*auto simp*: *step-def*)
  thus *?case* **by** (*simp del*: *funpow.simps add*: *funpow-Suc-right*)
qed *auto*

In particular, iterating $step'$ n times is the same as the union of all results produced by iterating *step* up to n times.

lemma *funpow-step′-eq*: $(step'\ \hat{}\hat{}\ n)\ X = (\bigcup k{\le}n.\ (step\ \hat{}\hat{}\ k)\ X)$
proof (*induction n arbitrary*: X)
  case (*Suc n*)
  have $(step'\ \hat{}\hat{}\ Suc\ n)\ X = step'\ ((step'\ \hat{}\hat{}\ n)\ X)$
    by *simp*
  also have $(step'\ \hat{}\hat{}\ n)\ X = (\bigcup k{\le}n.\ (step\ \hat{}\hat{}\ k)\ X)$
    by (*rule Suc.IH*)
  also have $step'\ \ldots = (\bigcup k{\le}n.\ (step\ \hat{}\hat{}\ k)\ X) \cup step\ (\bigcup k{\le}n.\ (step\ \hat{}\hat{}\ k)\ X)$
    by (*simp add*: *step′-def*)
  also have $step\ (\bigcup k{\le}n.\ (step\ \hat{}\hat{}\ k)\ X) = (\bigcup k{\le}n.\ (step\ \hat{}\hat{}\ Suc\ k)\ X)$

by (*auto simp*: *step-def*)
also have ... = ($\bigcup k\in Suc`\{..n\}$. (*step* ^^ *k*) *X*)
by *blast*
also have ($\bigcup k\le n$. (*step* ^^ *k*) *X*) $\cup$ ... = ($\bigcup k\in\{..n\}\cup Suc`\{..n\}$. (*step* ^^ *k*) *X*)
by *blast*
also have $\{..n\}\cup Suc`\{..n\} = \{..Suc\ n\}$
by *force*
finally show *?case* **.**
qed *auto*

Auxiliary lemma: if we have a chain of length n from some element of X ending in some element y, then y will be in the result after iterating *step* on X n times.

lemma *steps-chain*:
assumes *chain* (*xs* @ [*y*]) **and** *hd* (*xs* @ [*y*]) $\in$ *X*
shows *y* $\in$ (*step* ^^ *length xs*) *X*
using *assms*
proof (*induction xs arbitrary*: *X*)
case (*Cons x xs*)
have *x* $\in$ *Dom* (*hd* (*xs* @ [*y*]))
using *Cons.prems* **by** (*cases xs*) *auto*
hence *hd* (*xs* @ [*y*]) $\in$ *step X*
using *Cons.prems Dom-props* **by** (*fastforce simp*: *step-def*)
hence *y* $\in$ (*step* ^^ *length xs*) (*step X*)
using *Cons.prems chain-ConsD* **by** (*intro Cons.IH*) *auto*
thus *?case*
by (*subst length-Cons*, *subst funpow-Suc-right*) *simp*
qed *auto*

Correctness lemma for the MC Sr's algorithm: eventually, applying *step′* does not change the result anymore. At that point, we have computed *TC*.

lemma *step′-stabilises*:
assumes $X \neq \{\}$ **and** $X \subseteq TC$
shows $\exists N.\ \forall n\ge N.$ (*step′* ^^ *n*) *X* = *TC*
using *assms*
proof (*induction card* ($-X$) *arbitrary*: *X rule*: *less-induct*)
case (*less X*)
show *?case*
proof (*cases step′ X* = *X*)
case *True*
hence *dominant X* **using** *less.prems*
by (*auto simp*: *step′-def step-def dominant-def*)
with ‹$X \subseteq TC$› **have** *X* = *TC*
by (*auto simp*: *TC-def*)
moreover have (*step′* ^^ *n*) *TC* = *TC* **for** *n*
by (*induction n*) *auto*
ultimately show *?thesis*
by *blast*

next
case *False*
hence $-(step'\ X) \subset -X$
by $(auto\ simp: step'\text{-}def)$
hence $card\ (-step'\ X) < card\ (-X)$
by $(intro\ psubset\text{-}card\text{-}mono)\ auto$
moreover have $step'\ X \neq \{\}$
using $less.prems$ **by** $(auto\ simp: step'\text{-}def)$
moreover have $step'\ X \subseteq TC$
using $step'\text{-}subset\ dominant\text{-}TC\ less.prems$ **by** $blast$
ultimately obtain N **where** $\forall\, n \geq N.\ (step'\ \hat{}\hat{}\ Suc\ n)\ \ X = TC$
using $less(1)[of\ step'\ X]$
by $(auto\ simp\ del: funpow.simps\ simp\ add: funpow\text{-}Suc\text{-}right)$
hence $\forall\, n \geq Suc\ N.\ (step'\ \hat{}\hat{}\ n)\ \ X = TC$
using $Suc\text{-}le\text{-}D$ **by** $blast$
thus $?thesis$ **..**
qed
qed

3 Main proof

Lemma 1: If $CO \neq TC$, then there exists an element $x \in CO$ that lies on a cycle of length 3 and a cycle of length 4.

lemma *cycle34*:
  assumes $CO \neq TC$
  shows $\exists x\ y\ w_1\ w_2.\ x \in CO \wedge cycle\ [x,\ y,\ w_1] \wedge cycle\ [x,\ y,\ w_1,\ w_2]$
proof −
  have $CO \subset TC$
    using *assms CO-subset-UC UC-subset-TC* **by** *blast*
  have $\neg dominant\ CO$
    using ‹$CO \subset TC$› **by** (*auto simp*: *TC-def*)
  then obtain $x\ y$ **where** $x \in CO$ **and** $y \notin CO$ **and** $x \in Dom\ y$
    using *CO-nonempty total* **unfolding** *dominant-def* **by** (*metis Compl-iff*)
  hence $y \in TC - CO$
    using ‹$CO \subset TC$› *dominant-TC*
    unfolding *dominant-def* **by** *auto*

We now show that there are at least two different elements $w_1,\ w_2 \in Dom\ x - Dom\ y$ and w.l.o.g. $w_1 \in Dom\ w_2$:

  obtain $w_1\ w_2$ **where**
    $w_1 \in Dom\ x - Dom\ y$ **and** $w_2 \in Dom\ x - Dom\ y$ **and**
    $w_1 \neq w_2$ **and** $w_1 \in Dom\ w_2$
  proof −
    have $card\ (Dom\ y - \{x\}) + 1 = card\ (Dom\ y)$
      using ‹$x \in Dom\ y$› **by** (*metis Suc-eq-plus1 card-Suc-Diff1 finite-code*)
    also from ‹$y \in TC - CO$› **have** $card\ (Dom\ y) < card\ (Dom\ x)$
      using ‹$x \in CO$› *less-linear* **by** (*fastforce simp*: *CO-def is-arg-max-def*)
    finally have $2 \leq card\ (Dom\ x) - card\ (Dom\ y - \{x\})$
      by *auto*
    also have $\ldots \leq card\ (Dom\ x - (Dom\ y - \{x\}))$
      using *diff-card-le-card-Diff* **by** (*intro diff-card-le-card-Diff*) *auto*
    also have $Dom\ x - (Dom\ y - \{x\}) = Dom\ x - Dom\ y$
      using *Dom-props* **by** *auto*
    finally have $card\ (Dom\ x - Dom\ y) \geq 2$
      by *auto*
    thus *?thesis*
      using *total that*
        by (*auto simp*: *card-le-Suc-iff numeral-2-eq-2*)
          (*metis Diff-iff insertCI*)
  qed

With that, we have our two cycles:

  hence $cycle\ [x,\ y,\ w_1]$ **and** $cycle\ [x,\ y,\ w_1,\ w_2]$
    using ‹$x \in Dom\ y$› **by** (*auto simp*: *cycle-def*)
  thus *?thesis*
    using ‹$x \in CO$› **by** (*auto simp*: *cycle-def*)
qed

Lemma 2: If $CO \neq TC$, there exists a Copeland winner x that is in every iteration of *step* on the initial set $\{x\}$ past the 6th one. (this is part of Corollary 1 by Herr Hübler)

```
lemma stable-element-exists:
  assumes CO ≠ TC
  shows   ∃x∈CO. ∀n≥6. x ∈ (step ^^ n) {x}
proof −
  from assms obtain x y w₁ w₂
    where x ∈ CO and cycle [x, y, w₁] and cycle [x, y, w₁, w₂]
    using cycle34 by auto

  have ∀n≥6. x ∈ (step ^^ n) {x}
  proof safe
    fix n :: nat
    assume n ≥ 6
    have ∃k l. n = 3 ∗ k + 4 ∗ l
      using ‹n ≥ 6› by presburger
    then obtain k l where kl: n = 3 ∗ k + 4 ∗ l
      by auto

    define xs where xs = repeat k [x, y, w₁] @ repeat l [x, y, w₁, w₂]
    have length-xs: length xs = n
      by (auto simp: xs-def kl)
    have [simp]: xs ≠ []
      using ‹n ≥ 6› length-xs by auto
    hence [simp]: hd xs = x
      by (cases k = 0; cases l = 0) (auto simp: xs-def hd-append)

    have cycle xs
      using ‹cycle [x, y, w₁]› and ‹cycle [x, y, w₁, w₂]›
      by (cases k = 0; cases l = 0) (auto simp: xs-def)
    hence x ∈ (step ^^ length xs) {x}
      by (intro steps-chain) (use ‹x ∈ CO› in ‹auto simp: cycle-def›)
    thus x ∈ (step ^^ n) {x}
      by (simp add: length-xs)
  qed
  thus ?thesis
    using ‹x ∈ CO› by blast
qed
```

Corollary 1: If $CO \neq TC$, Herr Schmidmeier's algorithm returns TC in finitely many steps.

```
corollary steps-converges-to-TC:
  assumes CO ≠ TC and CO ⊆ X and X ⊆ TC
  shows   ∃N. ∀n≥N. (step ^^ n) X = TC
proof −
  from ‹CO ≠ TC› obtain x where x: x ∈ CO ∀n≥6. x ∈ (step ^^ n) {x}
    using stable-element-exists by blast

  have ∃N. ∀n≥N. (step′ ^^ n) {x} = TC
    using x CO-subset-UC UC-subset-TC
    by (intro step′-stabilises) auto
  then obtain N where N: ∀n≥N. (step′ ^^ n) {x} = TC ..

  have (step ^^ n) X = TC if n: n ≥ N + 6 for n
  proof
    have TC = (step′ ^^ (n − 6)) {x}
      using N n by auto
    also have ... ⊆ (step ^^ n) {x}
    proof
      fix y assume y ∈ (step′ ^^ (n − 6)) {x}
      then obtain k where k: k ≤ n − 6 y ∈ (step ^^ k) {x}
        by (auto simp: funpow-step′-eq)
      have y ∈ (step ^^ k) {x}
        by fact
      also have (step ^^ k) {x} ⊆ (step ^^ k) ((step ^^ (n − 6 − k + 6)) {x})
        using x by (intro steps-mono) auto
      also have ... = (step ^^ (k + (n − 6 − k + 6))) {x}
        by (subst funpow-add) auto
      also have k + (n − 6 − k + 6) = n
        using k n by auto
      finally show y ∈ (step ^^ n) {x} .
    qed
    also have ... ⊆ (step ^^ n) X
      using x assms by (intro steps-mono) auto
    finally show TC ⊆ ... .
  next
    show (step ^^ n) X ⊆ TC
      using x CO-subset-UC UC-subset-TC assms
      by (intro steps-subset dominant-TC) auto
  qed
  thus ?thesis by blast
qed

end
```