Formalization of "Knight's Tour Revisited"

Lukas Koller

January 9, 2022

Abstract

This is a formalization of [1]. In [1] the existence of a Knight's path is proved for arbitrary $n \times m$-boards with $\min(n, m) \geq 5$. If $n \cdot m$ is even, then there exists a Knight's circuit.

A Knight's Path is a sequence of moves of a Knight on a chessboard s.t. the Knight visits every square of a chessboard exactly once. Finding a Knight's path is a an instance of the Hamiltonian path problem.

During the formalization two mistakes in the original proof in [1] were discovered. These mistakes are corrected in this formalization.

Contents

theory *KnightsTour*
  imports *Main*
begin

1 Introduction and Definitions

A Knight's path is a sequence of moves on a chessboard s.t. every step in sequence is a valid move for a Knight and that the Knight visits every square on the boards exactly once. A Knight is a chess figure that is only able to move two squares vertically and one square horizontally or two squares horizontally and one square vertically. Finding a Knight's path is an instance of the Hamiltonian Path Problem. A Knight's circuit is a Knight's path, where additionally the Knight can move from the last square to the first square of the path, forming a loop.

[1] proves the existence of a Knight's path on a $n\times m$-board for sufficiently large n and m. The main idea for the proof is to inductivly construct a Knight's path for the $n\times m$-board from a few pre-computed Knight's paths for small boards, i.e. 5×5, 8×6, ..., 8×9. The paths for small boards are transformed (i.e. transpose, mirror, translate) and concatenated to create paths for larger boards.

While formalizing the proofs I discovered two mistakes in the original proof in [1]: (i) the pre-computed path for the 6×6-board that ends in the upper-left (in Figure 2) and (ii) the pre-computed path for the 8×8-board that ends in the upper-left (in Figure 5) are incorrect. I.e. on the 6×6-board the Knight cannot step from square 26 to square 27; in the 8×8-board the Knight cannot step from square 27 to square 28. In this formalization I have replaced the two incorrect paths with correct paths.

A square on a board is identified by its coordinates.

type-synonym *square* = *int* × *int*

A board is represented as a set of squares. Note, that this allows boards to have an arbitrary shape and do not necessarily need to be rectangular.

type-synonym *board* = *square set*

A (rectangular) $(n\times m)$-board is the set of all squares (i,j) where $1 \le i \le n$

and $1 \leq j \leq m$. $(1,1)$ is the lower-left corner, and (n,m) is the upper-right corner.

definition *board* :: *nat* ⇒ *nat* ⇒ *board* **where**
  $board\ n\ m = \{(i,j)\ |i\ j.\ 1 \leq i \wedge i \leq int\ n \wedge 1 \leq j \wedge j \leq int\ m\}$

A path is a sequence of steps on a board. A path is represented by the list of visited squares on the board. Each square on the $(n \times m)$-board is identified by its coordinates (i,j).

type-synonym *path* = *square list*

A Knight can only move two squares vertically and one square horizontally or two squares horizontally and one square vertically. Thus, a knight at position (i,j) can only move to $(i \pm 1, j \pm 2)$ or $(i \pm 2, j \pm 1)$.

definition *valid-step* :: *square* ⇒ *square* ⇒ *bool* **where**
  $valid\text{-}step\ s_i\ s_j \equiv (case\ s_i\ of\ (i,j) \Rightarrow s_j \in \{(i+1,j+2),(i-1,j+2),(i+1,j-2),(i-1,j-2),$
  $(i+2,j+1),(i-2,j+1),(i+2,j-1),(i-2,j-1)\})$

Now we define an inductive predicate that characterizes a Knight's path. A square s_i can be pre-pended to a current Knight's path $s_j \# ps$ if (i) there is a valid step from the square s_i to the first square s_j of the current path and (ii) the square s_i has not been visited yet.

inductive *knights-path* :: *board* ⇒ *path* ⇒ *bool* **where**
  $knights\text{-}path\ \{s_i\}\ [s_i]$
$|\ s_i \notin b \Longrightarrow valid\text{-}step\ s_i\ s_j \Longrightarrow knights\text{-}path\ b\ (s_j \# ps) \Longrightarrow knights\text{-}path\ (b \cup \{s_i\})$
$(s_i \# s_j \# ps)$

code-pred *knights-path* **.**

A sequence is a Knight's circuit iff the sequence if a Knight's path and there is a valid step from the last square to the first square.

definition $knights\text{-}circuit\ b\ ps \equiv (knights\text{-}path\ b\ ps \wedge valid\text{-}step\ (last\ ps)\ (hd\ ps))$

2 Executable Checker for a Knight's Path

This section gives the implementation and correctness-proof for an executable checker for a knights-path wrt. the definition *knights-path*.

2.1 Implementation of an Executable Checker

fun *row-exec* :: *nat* ⇒ *int set* **where**
  $row\text{-}exec\ 0 = \{\}$
$|\ row\text{-}exec\ m = insert\ (int\ m)\ (row\text{-}exec\ (m-1))$

fun *board-exec-aux* :: *nat* ⇒ *int set* ⇒ *board* **where**
  $board\text{-}exec\text{-}aux\ 0\ M = \{\}$

| *board-exec-aux* k M = $\{(int\ k,j)\ |j.\ j \in M\} \cup$ *board-exec-aux* $(k-1)$ M

Compute a board.

fun *board-exec* :: *nat* ⇒ *nat* ⇒ *board* **where**
board-exec n m = *board-exec-aux* n (*row-exec* m)

fun *step-checker* :: *square* ⇒ *square* ⇒ *bool* **where**
step-checker (i,j) (i',j') =
$((i+1,j+2) = (i',j') \lor (i-1,j+2) = (i',j') \lor (i+1,j-2) = (i',j') \lor (i-1,j-2) = (i',j')$
$\lor (i+2,j+1) = (i',j') \lor (i-2,j+1) = (i',j') \lor (i+2,j-1) = (i',j') \lor (i-2,j-1) = (i',j'))$

fun *path-checker* :: *board* ⇒ *path* ⇒ *bool* **where**
path-checker b [] = *False*
| *path-checker* b $[s_i]$ = $(\{s_i\} = b)$
| *path-checker* b $(s_i \# s_j \# ps)$ = $(s_i \in b \land$ *step-checker* s_i s_j $\land$ *path-checker* $(b - \{s_i\})$ $(s_j \# ps))$

fun *circuit-checker* :: *board* ⇒ *path* ⇒ *bool* **where**
circuit-checker b ps = (*path-checker* b ps ∧ *step-checker* (*last* ps) (*hd* ps))

2.2 Correctness Proof of the Executable Checker

lemma *row-exec-leq*: $j \in$ *row-exec* $m \longleftrightarrow 1 \le j \land j \le int\ m$
by (*induction* m) *auto*

lemma *board-exec-aux-leq-mem*: $(i,j) \in$ *board-exec-aux* k $M \longleftrightarrow 1 \le i \land i \le int\ k \land j \in M$
by (*induction* k M *rule*: *board-exec-aux.induct*) *auto*

lemma *board-exec-leq*: $(i,j) \in$ *board-exec* n $m \longleftrightarrow 1 \le i \land i \le int\ n \land 1 \le j \land j \le int\ m$
using *board-exec-aux-leq-mem row-exec-leq* **by** *auto*

lemma *board-exec-correct*: *board* n m = *board-exec* n m
unfolding *board-def* **using** *board-exec-leq* **by** *auto*

lemma *step-checker-correct*: *step-checker* s_i s_j ⟷ *valid-step* s_i s_j
proof
assume *step-checker* s_i s_j
then show *valid-step* s_i s_j
unfolding *valid-step-def*
apply (*cases* s_i)
apply (*cases* s_j)
apply *auto*
done
next
assume *assms*: *valid-step* s_i s_j

then show *step-checker* s_i s_j
unfolding *valid-step-def* **by** *auto*
qed

lemma *step-checker-rev*: *step-checker* (i,j) (i',j') $\Longrightarrow$ *step-checker* (i',j') (i,j)
apply (*simp only*: *step-checker.simps*)
by (*elim disjE*) *auto*

lemma *knights-path-intro-rev*:
assumes $s_i \in b$ *valid-step* s_i s_j *knights-path* $(b - \{s_i\})$ $(s_j\#ps)$
shows *knights-path* b $(s_i\#s_j\#ps)$
using *assms*
proof −
assume *assms*: $s_i \in b$ *valid-step* s_i s_j *knights-path* $(b - \{s_i\})$ $(s_j\#ps)$
then have $s_i \notin (b - \{s_i\})$ $b - \{s_i\} \cup \{s_i\} = b$
by *auto*
then show *?thesis*
using *assms knights-path.intros*(*2*)[*of* s_i $b - \{s_i\}$] **by** *auto*
qed

Final correctness corollary for the executable checker *path-checker*.

lemma *path-checker-correct*: *path-checker b ps* $\longleftrightarrow$ *knights-path b ps*
proof
assume *path-checker b ps*
then show *knights-path b ps*
proof (*induction rule*: *path-checker.induct*)
case (*3* s_i s_j *xs b*)
then show *?case* **using** *step-checker-correct knights-path-intro-rev* **by** *auto*
qed (*auto intro*: *knights-path.intros*)
next
assume *knights-path b ps*
then show *path-checker b ps*
using *step-checker-correct*
by (*induction rule*: *knights-path.induct*) (*auto elim*: *knights-path.cases*)
qed

corollary *knights-path-exec-simp*: *knights-path* (*board n m*) *ps* $\longleftrightarrow$ *path-checker* (*board-exec n m*) *ps*
using *board-exec-correct path-checker-correct*[*symmetric*] **by** *simp*

lemma *circuit-checker-correct*: *circuit-checker b ps* $\longleftrightarrow$ *knights-circuit b ps*
unfolding *knights-circuit-def* **using** *path-checker-correct step-checker-correct* **by** *auto*

corollary *knights-circuit-exec-simp*:
knights-circuit (*board n m*) *ps* $\longleftrightarrow$ *circuit-checker* (*board-exec n m*) *ps*
using *board-exec-correct circuit-checker-correct*[*symmetric*] **by** *simp*

3 Basic Properties of *knights-path* and *knights-circuit*

lemma *board-leq-subset*: $n_1 \leq n_2 \wedge m_1 \leq m_2 \Longrightarrow$ *board* n_1 $m_1 \subseteq$ *board* n_2 m_2
  unfolding *board-def* **by** *auto*

lemma *finite-row-exec*: *finite* (*row-exec m*)
  by (*induction m*) *auto*

lemma *finite-board-exec-aux*: *finite M* $\Longrightarrow$ *finite* (*board-exec-aux n M*)
  by (*induction n*) *auto*

lemma *board-finite*: *finite* (*board n m*)
  using *finite-board-exec-aux finite-row-exec* **by** (*simp only*: *board-exec-correct*) *auto*

lemma *card-row-exec*: *card* (*row-exec m*) = *m*
proof (*induction m*)
  case (*Suc m*)
  have *int* (*Suc m*) $\notin$ *row-exec m*
    using *row-exec-leq* **by** *auto*
  then have *card* (*insert* (*int* (*Suc m*)) (*row-exec m*)) = *1* + *card* (*row-exec m*)
    using *card-Suc-eq* **by** (*metis Suc plus-1-eq-Suc row-exec.simps*(*1*))
  then have *card* (*row-exec* (*Suc m*)) = *1* + *card* (*row-exec m*)
    by *auto*
  then show *?case* **using** *Suc.IH* **by** *auto*
qed *auto*

lemma *set-comp-ins*:
  {(*k*,*j*) |*j*. *j* $\in$ *insert x M*} = *insert* (*k*,*x*) {(*k*,*j*) |*j*. *j* $\in$ *M*} (**is** *?Mi* = *?iM*)
proof
  show *?Mi* $\subseteq$ *?iM*
  proof
    fix *y* **assume** *y* $\in$ *?Mi*
    then obtain *j* **where** [*simp*]: *y* = (*k*,*j*) **and** *j* $\in$ *insert x M* **by** *blast*
    then have *j* = *x* $\vee$ *j* $\in$ *M* **by** *auto*
    then show *y* $\in$ *?iM* **by** (*elim disjE*) *auto*
  qed
next
  show *?iM* $\subseteq$ *?Mi*
  proof
    fix *y* **assume** *y* $\in$ *?iM*
    then obtain *j* **where** [*simp*]: *y* = (*k*,*j*) **and** *j* $\in$ *insert x M* **by** *blast*
    then have *j* = *x* $\vee$ *j* $\in$ *M* **by** *auto*
    then show *y* $\in$ *?Mi* **by** (*elim disjE*) *auto*
  qed
qed

lemma *finite-card-set-comp*: *finite M* $\Longrightarrow$ *card* {(*k*,*j*) |*j*. *j* $\in$ *M*} = *card M*
proof (*induction M rule*: *finite-induct*)
  case (*insert x M*)

then show *?case* **using** *set-comp-ins*[*of k x M*] **by** *auto*
qed *auto*

lemma *card-board-exec-aux*: *finite M* $\Longrightarrow$ *card* (*board-exec-aux k M*) = *k* $*$ *card M*
proof (*induction k*)
  case (*Suc k*)
  let *?M'*={(*int* (*Suc k*),*j*) |*j*. *j* $\in$ *M*}
  let *?rec-k*=*board-exec-aux k M*

  have *finite*: *finite ?M' finite ?rec-k*
    using *Suc finite-board-exec-aux* **by** *auto*
  then have *card-Un-simp*: *card* (*?M'* $\cup$ *?rec-k*) = *card ?M'* + *card ?rec-k*
    using *board-exec-aux-leq-mem card-Un-Int*[*of ?M' ?rec-k*] **by** *auto*

  have *card-M*: *card ?M'* = *card M*
    using *Suc finite-card-set-comp* **by** *auto*

  have *card* (*board-exec-aux* (*Suc k*) *M*) = *card ?M'* + *card ?rec-k*
    using *card-Un-simp* **by** *auto*
  also have ... = *card M* + *k* $*$ *card M*
    using *Suc card-M* **by** *auto*
  also have ... = (*Suc k*) $*$ *card M*
    by *auto*
  finally show *?case* **.**
qed *auto*

lemma *card-board*: *card* (*board n m*) = *n* $*$ *m*
proof −
  have *card* (*board n m*) = *card* (*board-exec-aux n* (*row-exec m*))
    using *board-exec-correct* **by** *auto*
  also have ... = *n* $*$ *m*
    using *card-row-exec card-board-exec-aux finite-row-exec* **by** *auto*
  finally show *?thesis* **.**
qed

lemma *knights-path-board-non-empty*: *knights-path b ps* $\Longrightarrow$ *b* $\neq$ {}
  by (*induction arbitrary*: *ps rule*: *knights-path.induct*) *auto*

lemma *knights-path-board-m-n-geq-1*: *knights-path* (*board n m*) *ps* $\Longrightarrow$ *min n m* $\geq$ *1*
  unfolding *board-def* **using** *knights-path-board-non-empty* **by** *fastforce*

lemma *knights-path-non-nil*: *knights-path b ps* $\Longrightarrow$ *ps* $\neq$ []
  by (*induction arbitrary*: *b rule*: *knights-path.induct*) *auto*

lemma *knights-path-set-eq*: *knights-path b ps* $\Longrightarrow$ *set ps* = *b*
  by (*induction rule*: *knights-path.induct*) *auto*

lemma *knights-path-subset*:

knights-path b_1 ps_1 $\Longrightarrow$ *knights-path* b_2 ps_2 $\Longrightarrow$ *set* $ps_1 \subseteq$ *set* $ps_2 \longleftrightarrow b_1 \subseteq b_2$
using *knights-path-set-eq* **by** *auto*

lemma *knights-path-board-unique*: *knights-path* b_1 *ps* $\Longrightarrow$ *knights-path* b_2 *ps* $\Longrightarrow$ $b_1 = b_2$
using *knights-path-set-eq* **by** *auto*

lemma *valid-step-neq*: *valid-step* s_i $s_j \Longrightarrow s_i \neq s_j$
unfolding *valid-step-def* **by** *auto*

lemma *valid-step-non-transitive*: *valid-step* s_i $s_j \Longrightarrow$ *valid-step* s_j $s_k \Longrightarrow \neg$*valid-step* s_i s_k
proof –
assume *assms*: *valid-step* s_i s_j *valid-step* s_j s_k
obtain i_i j_i i_j j_j i_k j_k **where** [*simp*]: $s_i = (i_i,j_i)$ $s_j = (i_j,j_j)$ $s_k = (i_k,j_k)$ **by** *force*
then have *step-checker* (i_i,j_i) (i_j,j_j) *step-checker* (i_j,j_j) (i_k,j_k)
using *assms step-checker-correct* **by** *auto*
then show $\neg$*valid-step* s_i s_k
apply (*simp add*: *step-checker-correct*[*symmetric*])
apply (*elim disjE*)
apply *auto*
done
qed

lemma *knights-path-distinct*: *knights-path b ps* $\Longrightarrow$ *distinct ps*
proof (*induction rule*: *knights-path.induct*)
case (*2* s_i *b* s_j *ps*)
then have $s_i \notin$ *set* (s_j # *ps*)
using *knights-path-set-eq valid-step-neq* **by** *blast*
then show *?case* **using** *2* **by** *auto*
qed *auto*

lemma *knights-path-length*: *knights-path b ps* $\Longrightarrow$ *length ps* = *card b*
using *knights-path-set-eq knights-path-distinct* **by** (*metis distinct-card*)

lemma *knights-path-take*:
assumes *knights-path b ps* $0 < k$ $k <$ *length ps*
shows *knights-path* (*set* (*take k ps*)) (*take k ps*)
using *assms*
proof (*induction arbitrary*: *k rule*: *knights-path.induct*)
case (*2* s_i *b* s_j *ps*)
then have $k = 1 \vee k = 2 \vee 2 < k$ **by** *force*
then show *?case*
using *2*
proof (*elim disjE*)
assume $k = 2$
then have *take k* $(s_i\#s_j\#ps) = [s_i,s_j]$ $s_i \notin \{s_j\}$ **using** *2 valid-step-neq* **by** *auto*

then show *?thesis* **using** *2 knights-path.intros* **by** *auto*
next
assume $2 < k$
then have *k-simps*: $k-2 = k-1-1$ $0 < k-2$ $k-2 <$ *length ps* **and**
take-simp1: *take k* $(s_i\#s_j\#ps) = s_i\#$*take* $(k-1)$ $(s_j\#ps)$ **and**
take-simp2: *take k* $(s_i\#s_j\#ps) = s_i\#s_j\#$*take* $(k-1-1)$ *ps*
using *assms 2 take-Cons'*[*of k* s_i $s_j\#ps$] *take-Cons'*[*of* $k-1$ s_j *ps*] **by** *auto*
then have *knights-path* (*set* (*take* $(k-1)$ $(s_j\#ps)$)) (*take* $(k-1)$ $(s_j\#ps)$)
using *2 k-simps* **by** *auto*
then have *kp*: *knights-path* (*set* (*take* $(k-1)$ $(s_j\#ps)$)) ($s_j\#$*take* $(k-2)$ *ps*)
using *take-Cons'*[*of* $k-1$ s_j *ps*] **by** (*auto simp*: *k-simps elim*: *knights-path.cases*)

have *no-mem*: $s_i \notin$ *set* (*take* $(k-1)$ $(s_j\#ps)$)
using *2 set-take-subset*[*of* $k-1$ $s_j\#ps$] *knights-path-set-eq* **by** *blast*
have *knights-path* (*set* (*take* $(k-1)$ $(s_j\#ps)$) $\cup \{s_i\}$) ($s_i\#s_j\#$*take* $(k-2)$ *ps*)
using *knights-path.intros*(*2*)[*OF no-mem* ‹*valid-step* s_i s_j› *kp*] **by** *auto*
then show *?thesis* **using** *k-simps take-simp2 knights-path-set-eq* **by** *metis*
qed (*auto intro*: *knights-path.intros*)
qed *auto*

lemma *knights-path-drop*:
assumes *knights-path b ps* $0 < k$ $k <$ *length ps*
shows *knights-path* (*set* (*drop k ps*)) (*drop k ps*)
using *assms*
proof (*induction arbitrary*: *k rule*: *knights-path.induct*)
case (*2* s_i *b* s_j *ps*)
then have $(k = 1 \wedge ps = []) \vee (k = 1 \wedge ps \neq []) \vee 1 < k$ **by** *force*
then show *?case*
using *2*
proof (*elim disjE*)
assume $k = 1 \wedge ps \neq []$
then show *?thesis* **using** *2 knights-path-set-eq* **by** *force*
next
assume $1 < k$
then have $0 < k-1$ $k-1 <$ *length* $(s_j\#ps)$ *drop k* $(s_i\#s_j\#ps) =$ *drop* $(k-1)$ $(s_j\#ps)$
using *assms 2 drop-Cons'*[*of k* s_i $s_j\#ps$] **by** *auto*
then show *?thesis*
using *2* **by** *auto*
qed (*auto intro*: *knights-path.intros*)
qed *auto*

A Knight's path can be split to form two new disjoint Knight's paths.

corollary *knights-path-split*:
assumes *knights-path b ps* $0 < k$ $k <$ *length ps*
shows
$\exists\, b_1\ b_2.$ *knights-path* b_1 (*take k ps*) $\wedge$ *knights-path* b_2 (*drop k ps*) $\wedge\ b_1 \cup b_2 = b \wedge b_1 \cap b_2 = \{\}$
using *assms*

proof −
  let *?b*$_1$=*set* (*take k ps*)
  let *?b*$_2$=*set* (*drop k ps*)
  have *kp1*: *knights-path ?b*$_1$ (*take k ps*) **and** *kp2*: *knights-path ?b*$_2$ (*drop k ps*)
    using *assms knights-path-take knights-path-drop* **by** *auto*
  have *union*: *?b*$_1$ ∪ *?b*$_2$ = *b*
    using *assms knights-path-set-eq* **by** (*metis append-take-drop-id set-append*)
  have *inter*: *?b*$_1$ ∩ *?b*$_2$ = {}
   using *assms knights-path-distinct* **by** (*metis append-take-drop-id distinct-append*)
  show *?thesis* **using** *kp1 kp2 union inter* **by** *auto*
qed

Append two disjoint Knight's paths.

corollary *knights-path-append*:
  assumes *knights-path* b_1 ps_1 *knights-path* b_2 ps_2 $b_1 \cap b_2 = \{\}$ *valid-step* (*last* ps_1) (*hd* ps_2)
  shows *knights-path* ($b_1 \cup b_2$) (ps_1 @ ps_2)
  using *assms*
proof (*induction arbitrary*: ps_2 b_2 *rule*: *knights-path.induct*)
  case (*1* s_i)
  then have $s_i \notin b_2$ $ps_2 \neq []$ *valid-step* s_i (*hd* ps_2) *knights-path* b_2 (*hd* ps_2#*tl* ps_2)

    using *knights-path-non-nil* **by** *auto*
  then have *knights-path* ($b_2 \cup \{s_i\}$) (s_i#*hd* ps_2#*tl* ps_2)
    using *knights-path.intros* **by** *blast*
  then show *?case* **using** ‹$ps_2 \neq []$› **by** *auto*
next
  case (*2* s_i b_1 s_j ps_1)
  then have $s_i \notin b_1 \cup b_2$ *valid-step* s_i s_j *knights-path* ($b_1 \cup b_2$) (s_j#ps_1@ps_2) **by** *auto*
  then have *knights-path* ($b_1 \cup b_2 \cup \{s_i\}$) ($s_i$#$s_j$#$ps_1$@$ps_2$)
    using *knights-path.intros* **by** *auto*
  then show *?case* **by** *auto*
qed

lemma *valid-step-rev*: *valid-step* s_i s_j ⟹ *valid-step* s_j s_i
  using *step-checker-correct step-checker-rev* **by** (*metis prod.exhaust-sel*)

Reverse a Knight's path.

corollary *knights-path-rev*:
  assumes *knights-path b ps*
  shows *knights-path b* (*rev ps*)
  using *assms*
proof (*induction rule*: *knights-path.induct*)
  case (*2* s_i *b* s_j *ps*)
  then have *knights-path* $\{s_i\}$ $[s_i]$ $b \cap \{s_i\} = \{\}$ *valid-step* (*last* (*rev* (s_j # *ps*))) (*hd* $[s_i]$)
    using *valid-step-rev* **by** (*auto intro*: *knights-path.intros*)
  then have *knights-path* ($b \cup \{s_i\}$) ((*rev* (s_j#*ps*))@$[s_i]$)

using *2 knights-path-append* **by** *blast*
then show *?case* **by** *auto*
qed (*auto intro*: *knights-path.intros*)

Reverse a Knight's circuit.

corollary *knights-circuit-rev*:
assumes *knights-circuit b ps*
shows *knights-circuit b* (*rev ps*)
using *assms knights-path-rev valid-step-rev*
unfolding *knights-circuit-def* **by** (*auto simp*: *hd-rev last-rev*)

lemma *knights-circuit-rotate1*:
assumes *knights-circuit b* ($s_i\#ps$)
shows *knights-circuit b* ($ps@[s_i]$)
proof (*cases ps* = [])
case *True*
then show *?thesis* **using** *assms* **by** *auto*
next
case *False*
have *kp1*: *knights-path b* ($s_i\#ps$) *valid-step* (*last* ($s_i\#ps$)) (*hd* ($s_i\#ps$))
using *assms* **unfolding** *knights-circuit-def* **by** *auto*
then have *kp-elim*: $s_i \notin (b - \{s_i\})$ *valid-step* s_i (*hd ps*) *knights-path* ($b - \{s_i\}$) *ps*
using ‹*ps* ≠ []› **by** (*auto elim*: *knights-path.cases*)
then have *vs'*: *valid-step* (*last* ($ps@[s_i]$)) (*hd* ($ps@[s_i]$))
using ‹*ps* ≠ []› *valid-step-rev* **by** *auto*

have *kp2*: *knights-path* $\{s_i\}$ $[s_i]$ $(b - \{s_i\}) \cap \{s_i\} = \{\}$
by (*auto intro*: *knights-path.intros*)

have *vs*: *valid-step* (*last ps*) (*hd* $[s_i]$)
using ‹*ps* ≠ []› ‹*valid-step* (*last* ($s_i\#ps$)) (*hd* ($s_i\#ps$))› **by** *auto*

have $(b - \{s_i\}) \cup \{s_i\} = b$
using *kp1 kp-elim knights-path-set-eq* **by** *force*
then show *?thesis*
unfolding *knights-circuit-def*
using *vs knights-path-append*[*OF* ‹*knights-path* ($b - \{s_i\}$) *ps*› *kp2*] *vs'* **by** *auto*
qed

A Knight's circuit can be rotated to start at any square on the board.

lemma *knights-circuit-rotate-to*:
assumes *knights-circuit b ps hd* (*drop k ps*) $= s_i$ $k <$ *length ps*
shows $\exists$ *ps'*. *knights-circuit b ps'* $\wedge$ *hd ps'* $= s_i$
using *assms*
proof (*induction k arbitrary*: *b ps*)

case (*Suc k*)
let $?s_j$=*hd ps*
let $?ps'$=*tl ps*
show *?case*
proof (*cases* s_i = $?s_j$)
case *True*
then show *?thesis* **using** *Suc* **by** *auto*
next
case *False*
then have $?ps' \neq []$
using *Suc* **by** (*metis drop-Nil drop-Suc drop-eq-Nil2 le-antisym nat-less-le*)
then have *knights-circuit b* ($?s_j \# ?ps'$)
using *Suc* **by** (*metis list.exhaust-sel tl-Nil*)
then have *knights-circuit b* ($?ps'$@[$?s_j$]) *hd* (*drop k* ($?ps'$@[$?s_j$])) = s_i
using *Suc knights-circuit-rotate1* **by** (*auto simp*: *drop-Suc*)
then show *?thesis* **using** *Suc* **by** *auto*
qed
qed *auto*

For positive boards (1,1) can only have (2,3) and (3,2) as a neighbour.

lemma *valid-step-1-1*:
assumes *valid-step* (*1*,*1*) (*i*,*j*) $i > 0$ $j > 0$
shows (*i*,*j*) = (*2*,*3*) $\vee$ (*i*,*j*) = (*3*,*2*)
using *assms* **unfolding** *valid-step-def* **by** *auto*

lemma *list-len-g-1-split*: *length xs* > *1* $\Longrightarrow \exists x_1\ x_2\ xs'.\ xs = x_1 \# x_2 \# xs'$
proof (*induction xs*)
case (*Cons x xs*)
then have *length xs* > *0* **by** *auto*
then have *length xs* $\geq$ *1* **by** *presburger*
then have *length xs* = *1* $\vee$ *length xs* > *1* **by** *auto*
then show *?case*
proof (*elim disjE*)
assume *length xs* = *1*
then obtain x_1 **where** [*simp*]: $xs = [x_1]$
using *length-Suc-conv*[*of xs 0*] **by** *auto*
then show *?thesis* **by** *auto*
next
assume *1* < *length xs*
then show *?thesis* **using** *Cons* **by** *auto*
qed
qed *auto*

lemma *list-len-g-3-split*: *length xs* > *3* $\Longrightarrow \exists x_1\ x_2\ xs'\ x_3.\ xs = x_1 \# x_2 \# xs' @ [x_3]$
proof (*induction xs*)
case (*Cons x xs*)
then have *length xs* = *3* $\vee$ *length xs* > *3* **by** *auto*
then show *?case*
proof (*elim disjE*)

assume *length xs* = *3*
then obtain x_1 xs_1 **where** [*simp*]: $xs = x_1\#xs_1$ *length* xs_1 = *2*
using *length-Suc-conv*[*of xs 2*] **by** *auto*
then obtain x_2 xs_2 **where** [*simp*]: $xs_1 = x_2\#xs_2$ *length* xs_2 = *1*
using *length-Suc-conv*[*of* xs_1 *1*] **by** *auto*
then obtain x_3 **where** [*simp*]: $xs_2 = [x_3]$
using *length-Suc-conv*[*of* xs_2 *0*] **by** *auto*
then show *?thesis* **by** *auto*
next
assume *length xs* > *3*
then show *?thesis* **using** *Cons* **by** *auto*
qed
qed *auto*

Any Knight's circuit on a positive board can be rotated to start with (1,1) and end with (3,2).

corollary *rotate-knights-circuit*:
assumes *knights-circuit* (*board n m*) *ps min n m* ≥ *5*
shows ∃*ps*. *knights-circuit* (*board n m*) *ps* ∧ *hd ps* = (*1*,*1*) ∧ *last ps* = (*3*,*2*)
using *assms*
proof −
let *?b*=*board n m*
have *knights-path ?b ps*
using *assms* **unfolding** *knights-circuit-def* **by** *auto*
then have (*1*,*1*) ∈ *set ps*
using *assms knights-path-set-eq* **by** (*auto simp*: *board-def*)
then obtain *k* **where** *hd* (*drop k ps*) = (*1*,*1*) *k* < *length ps*
by (*metis hd-drop-conv-nth in-set-conv-nth*)
then obtain ps_r **where** ps_r*-prems*: *knights-circuit ?b* ps_r *hd* ps_r = (*1*,*1*)
using *assms knights-circuit-rotate-to* **by** *blast*
then have *kp*: *knights-path ?b* ps_r **and** *valid-step* (*last* ps_r) (*1*,*1*)
unfolding *knights-circuit-def* **by** *auto*

have (*1*,*1*) ∈ *?b* (*1*,*2*) ∈ *?b* (*1*,*3*) ∈ *?b*
using *assms* **unfolding** *board-def* **by** *auto*
then have (*1*,*1*) ∈ *set* ps_r (*1*,*2*) ∈ *set* ps_r (*1*,*3*) ∈ *set* ps_r
using *kp knights-path-set-eq* **by** *auto*

have *3* < *card ?b*
using *assms board-leq-subset card-board*[*of 5 5*]
card-mono[*OF board-finite*[*of n m*], *of board 5 5*] **by** *auto*
then have *3* < *length* ps_r
using *knights-path-length kp* **by** *auto*
then obtain s_j ps' s_k **where** [*simp*]: $ps_r = (1,1)\#s_j\#ps'@[s_k]$
using ‹*hd* ps_r = (*1*,*1*)› *list-len-g-3-split*[*of* ps_r] **by** *auto*
have $s_j \neq s_k$
using *kp knights-path-distinct* **by** *force*

have *vs-*s_k: *valid-step* s_k (*1*,*1*)

using ‹*valid-step* (*last* ps_r) (*1*,*1*)› **by** *simp*

have *vs*-s_j: *valid-step* (*1*,*1*) s_j **and** *kp'*: *knights-path* (*?b* − {(*1*,*1*)}) (s_j#*ps'*@[s_k])
using *kp* **by** (*auto elim*: *knights-path.cases*)

have $s_j \in$ *set* ps_r $s_k \in$ *set* ps_r **by** *auto*
then have $s_j \in$ *?b* $s_k \in$ *?b*
using *kp knights-path-set-eq* **by** *blast+*
then have *0* < *fst* s_j ∧ *0* < *snd* s_j *0* < *fst* s_k ∧ *0* < *snd* s_k
unfolding *board-def* **by** *auto*
then have s_k = (*2*,*3*) ∨ s_k = (*3*,*2*) s_j = (*2*,*3*) ∨ s_j = (*3*,*2*)
using *vs*-s_k *vs*-s_j *valid-step-1-1 valid-step-rev* **by** (*metis prod.collapse*)+
then have s_k = (*3*,*2*) ∨ s_j = (*3*,*2*)
using ‹$s_j \neq s_k$› **by** *auto*
then show *?thesis*
proof (*elim disjE*)
assume s_k = (*3*,*2*)
then have *last* ps_r = (*3*,*2*) **by** *auto*
then show *?thesis* **using** ps_r*-prems* **by** *auto*
next
assume s_j = (*3*,*2*)
then have *vs*: *valid-step* (*last* ((*1*,*1*)#*rev* (s_j#*ps'*@[s_k]))) (*hd* ((*1*,*1*)#*rev* (s_j#*ps'*@[s_k])))
unfolding *valid-step-def* **by** *auto*

have *rev-simp*: *rev* (s_j#*ps'*@[s_k]) = s_k#(*rev ps'*)@[s_j] **by** *auto*

have *knights-path* (*?b* − {(*1*,*1*)}) (*rev* (s_j#*ps'*@[s_k]))
using *knights-path-rev*[*OF kp'*] **by** *auto*
then have (*1*,*1*) ∉ (*?b* − {(*1*,*1*)}) *valid-step* (*1*,*1*) s_k
knights-path (*?b* − {(*1*,*1*)}) (s_k#(*rev ps'*)@[s_j])
using *assms vs*-s_k *valid-step-rev* **by** (*auto simp*: *rev-simp*)
then have *knights-path* (*?b* − {(*1*, *1*)} ∪ {(*1*, *1*)}) ((*1*,*1*)#s_k#(*rev ps'*)@[s_j])
using *knights-path.intros*(*2*)[*of* (*1*,*1*) *?b* − {(*1*,*1*)} s_k (*rev ps'*)@[s_j]] **by** *auto*
then have *knights-path ?b* ((*1*,*1*)#*rev* (s_j#*ps'*@[s_k]))
using *assms* **by** (*simp add*: *board-def insert-absorb rev-simp*)
then have *knights-circuit ?b* ((*1*,*1*)#*rev* (s_j#*ps'*@[s_k]))
unfolding *knights-circuit-def* **using** *vs* **by** *auto*
then show *?thesis*
using ‹s_j = (*3*,*2*)› **by** *auto*
qed
qed

4 Transposing Paths and Boards

4.1 Implementation of Path and Board Transposition

definition *transpose-square* s_i = (*case* s_i *of* (*i*,*j*) ⇒ (*j*,*i*))

fun *transpose* :: *path* ⇒ *path* **where**
  transpose [] = []
| *transpose* (s_i#*ps*) = (*transpose-square* s_i)#*transpose ps*

definition *transpose-board* :: *board* ⇒ *board* **where**
  transpose-board b ≡ {(*j*,*i*) |*i j*. (*i*,*j*) ∈ *b*}

4.2 Correctness of Path and Board Transposition

lemma *transpose2*: *transpose-square* (*transpose-square* s_i) = s_i
  unfolding *transpose-square-def* **by** (*auto split*: *prod.splits*)

lemma *transpose-nil*: *ps* = [] ⟷ *transpose ps* = []
  using *transpose.elims* **by** *blast*

lemma *transpose-length*: *length ps* = *length* (*transpose ps*)
  by (*induction ps*) *auto*

lemma *hd-transpose*: *ps* ≠[] ⟹ *hd* (*transpose ps*) = *transpose-square* (*hd ps*)
  by (*induction ps*) (*auto simp*: *transpose-square-def*)

lemma *last-transpose*: *ps* ≠[] ⟹ *last* (*transpose ps*) = *transpose-square* (*last ps*)
proof (*induction ps*)
  case (*Cons* s_i *ps*)
  then show *?case*
  proof (*cases ps* = [])
    case *True*
    then show *?thesis* **using** *Cons* **by** (*auto simp*: *transpose-square-def*)
  next
    case *False*
    then show *?thesis* **using** *Cons transpose-nil* **by** *auto*
  qed
qed *auto*

lemma *take-transpose*:
  shows *take k* (*transpose ps*) = *transpose* (*take k ps*)
proof (*induction ps arbitrary*: *k*)
  case *Nil*
  then show *?case* **by** *auto*
next
  case (*Cons* s_i *ps*)
  then obtain *i j* **where** s_i = (*i*,*j*) **by** *force*
  then have *k* = *0* ∨ *k* > *0* **by** *auto*
  then show *?case*
  proof (*elim disjE*)
    assume *k* > *0*
    then show *?thesis* **using** *Cons.IH* **by** (*auto simp*: ‹s_i = (*i*,*j*)› *take-Cons'*)
  qed *auto*
qed

```
lemma drop-transpose:
  shows drop k (transpose ps) = transpose (drop k ps)
proof (induction ps arbitrary: k)
  case Nil
  then show ?case by auto
next
  case (Cons s_i ps)
  then obtain i j where s_i = (i,j) by force
  then have k = 0 ∨ k > 0 by auto
  then show ?case
  proof (elim disjE)
    assume k > 0
    then show ?thesis using Cons.IH by (auto simp: ‹s_i = (i,j)› drop-Cons')
  qed auto
qed

lemma transpose-board-correct: s_i ∈ b ⟷ (transpose-square s_i) ∈ transpose-board
b
  unfolding transpose-board-def transpose-square-def by (auto split: prod.splits)

lemma transpose-board: transpose-board (board n m) = board m n
 unfolding board-def using transpose-board-correct by (auto simp: transpose-square-def)

lemma insert-transpose-board:
  insert (transpose-square s_i) (transpose-board b) = transpose-board (insert s_i b)
  unfolding transpose-board-def transpose-square-def by (auto split: prod.splits)

lemma transpose-board2: transpose-board (transpose-board b) = b
  unfolding transpose-board-def by auto

lemma transpose-union: transpose-board (b_1 ∪ b_2) = transpose-board b_1 ∪ trans-
pose-board b_2
  unfolding transpose-board-def by auto

lemma transpose-valid-step:
  valid-step s_i s_j ⟷ valid-step (transpose-square s_i) (transpose-square s_j)
  unfolding valid-step-def transpose-square-def by (auto split: prod.splits)

lemma transpose-knights-path':
  assumes knights-path b ps
  shows knights-path (transpose-board b) (transpose ps)
  using assms
proof (induction rule: knights-path.induct)
  case (1 s_i)
 then have transpose-board {s_i} = {transpose-square s_i} transpose [s_i] = [transpose-square
s_i]
   using transpose-board-correct by (auto simp: transpose-square-def split: prod.splits)
  then show ?case by (auto intro: knights-path.intros)
```

next
  case (*2* s_i *b* s_j *ps*)
  then have *prems*: *transpose-square* $s_i \notin$ *transpose-board b*
      valid-step (*transpose-square* s_i) (*transpose-square* s_j)
      and *transpose* (s_j#*ps*) = *transpose-square* s_j#*transpose ps*
    using *2 transpose-board-correct transpose-valid-step* **by** *auto*
  then show *?case*
    using *2 knights-path.intros*(*2*)[*OF prems*] *insert-transpose-board* **by** *auto*
qed

corollary *transpose-knights-path*:
  assumes *knights-path* (*board n m*) *ps*
  shows *knights-path* (*board m n*) (*transpose ps*)
  using *assms transpose-knights-path′*[*of board n m ps*] **by** (*auto simp*: *transpose-board*)

corollary *transpose-knights-circuit*:
  assumes *knights-circuit* (*board n m*) *ps*
  shows *knights-circuit* (*board m n*) (*transpose ps*)
  using *assms*
proof −
  have *knights-path* (*board n m*) *ps* **and** *vs*: *valid-step* (*last ps*) (*hd ps*)
    using *assms* **unfolding** *knights-circuit-def* **by** *auto*
  then have *kp-t*: *knights-path* (*board m n*) (*transpose ps*) **and** *ps* ≠ []
    using *transpose-knights-path knights-path-non-nil* **by** *auto*
  then have *valid-step* (*last* (*transpose ps*)) (*hd* (*transpose ps*))
    using *vs hd-transpose last-transpose transpose-valid-step* **by** *auto*
  then show *?thesis* **using** *kp-t* **by** (*auto simp*: *knights-circuit-def*)
qed

5 Mirroring Paths and Boards

5.1 Implementation of Path and Board Mirroring

abbreviation *min1 ps* ≡ *Min* ((*fst*) ‘ *set ps*)
abbreviation *max1 ps* ≡ *Max* ((*fst*) ‘ *set ps*)
abbreviation *min2 ps* ≡ *Min* ((*snd*) ‘ *set ps*)
abbreviation *max2 ps* ≡ *Max* ((*snd*) ‘ *set ps*)

definition *mirror1-square* :: *int* ⇒ *square* ⇒ *square* **where**
  mirror1-square n s_i = (*case* s_i *of* (*i*,*j*) ⇒ (*n*−*i*,*j*))

fun *mirror1-aux* :: *int* ⇒ *path* ⇒ *path* **where**
  mirror1-aux n [] = []
| *mirror1-aux n* (s_i#*ps*) = (*mirror1-square n* s_i)#*mirror1-aux n ps*

definition *mirror1 ps* = *mirror1-aux* (*max1 ps* + *min1 ps*) *ps*

definition *mirror1-board* :: *int* ⇒ *board* ⇒ *board* **where**

```
  mirror1-board n b ≡ {mirror1-square n s_i |s_i. s_i ∈ b}

definition mirror2-square :: int ⇒ square ⇒ square where
  mirror2-square m s_i = (case s_i of (i,j) ⇒ (i,m−j))

fun mirror2-aux :: int ⇒ path ⇒ path where
  mirror2-aux m [] = []
| mirror2-aux m (s_i#ps) = (mirror2-square m s_i)#mirror2-aux m ps

definition mirror2 ps = mirror2-aux (max2 ps + min2 ps) ps

definition mirror2-board :: int ⇒ board ⇒ board where
  mirror2-board m b ≡ {mirror2-square m s_i |s_i. s_i ∈ b}
```

5.2 Correctness of Path and Board Mirroring

```
lemma mirror1-board-id: mirror1-board (int n+1) (board n m) = board n m (is -
= ?b)
proof
  show mirror1-board (int n+1) ?b ⊆ ?b
  proof
    fix s_i'
    assume assms: s_i' ∈ mirror1-board (int n+1) ?b
    then obtain i' j' where [simp]: s_i' = (i',j') by force
    then have (i',j') ∈ mirror1-board (int n+1) ?b
      using assms by auto
    then obtain i j where (i,j) ∈ ?b mirror1-square (int n+1) (i,j) = (i',j')
      unfolding mirror1-board-def by auto
    then have 1 ≤ i ∧ i ≤ int n 1 ≤ j ∧ j ≤ int m i'=(int n+1)−i j'=j
      unfolding board-def mirror1-square-def by auto
    then have 1 ≤ i' ∧ i' ≤ int n 1 ≤ j' ∧ j' ≤ int m
      by auto
    then show s_i' ∈ ?b
      unfolding board-def by auto
  qed
next
  show ?b ⊆ mirror1-board (int n+1) ?b
  proof
    fix s_i
    assume assms: s_i ∈ ?b
    then obtain i j where [simp]: s_i = (i,j) by force
    then have (i,j) ∈ ?b
      using assms by auto
    then have 1 ≤ i ∧ i ≤ int n 1 ≤ j ∧ j ≤ int m
      unfolding board-def by auto
    then obtain i' j' where i'=(int n+1)−i j'=j by auto
    then have (i',j') ∈ ?b mirror1-square (int n+1) (i',j') = (i,j)
      using ‹1 ≤ i ∧ i ≤ int n› ‹1 ≤ j ∧ j ≤ int m›
      unfolding mirror1-square-def by (auto simp: board-def)
```

```
    then show s_i ∈ mirror1-board (int n+1) ?b
      unfolding mirror1-board-def by force
  qed
qed

lemma mirror2-board-id: mirror2-board (int m+1) (board n m) = board n m (is -
= ?b)
proof
  show mirror2-board (int m+1) ?b ⊆ ?b
  proof
    fix s_i'
    assume assms: s_i' ∈ mirror2-board (int m+1) ?b
    then obtain i' j' where [simp]: s_i' = (i',j') by force
    then have (i',j') ∈ mirror2-board (int m+1) ?b
      using assms by auto
    then obtain i j where (i,j) ∈ ?b mirror2-square (int m+1) (i,j) = (i',j')
      unfolding mirror2-board-def by auto
    then have 1 ≤ i ∧ i ≤ int n 1 ≤ j ∧ j ≤ int m i'=i j'=(int m+1)−j
      unfolding board-def mirror2-square-def by auto
    then have 1 ≤ i' ∧ i' ≤ int n 1 ≤ j' ∧ j' ≤ int m
      by auto
    then show s_i' ∈ ?b
      unfolding board-def by auto
  qed
next
  show ?b ⊆ mirror2-board (int m+1) ?b
  proof
    fix s_i
    assume assms: s_i ∈ ?b
    then obtain i j where [simp]: s_i = (i,j) by force
    then have (i,j) ∈ ?b
      using assms by auto
    then have 1 ≤ i ∧ i ≤ int n 1 ≤ j ∧ j ≤ int m
      unfolding board-def by auto
    then obtain i' j' where i'=i j'=(int m+1)−j by auto
    then have (i',j') ∈ ?b mirror2-square (int m+1) (i',j') = (i,j)
      using ‹1 ≤ i ∧ i ≤ int n› ‹1 ≤ j ∧ j ≤ int m›
      unfolding mirror2-square-def by (auto simp: board-def)
    then show s_i ∈ mirror2-board (int m+1) ?b
      unfolding mirror2-board-def by force
  qed
qed

lemma knights-path-min1: knights-path (board n m) ps ⟹ min1 ps = 1
proof −
  assume assms: knights-path (board n m) ps
  then have min n m ≥ 1
    using knights-path-board-m-n-geq-1 by auto
  then have (1,1) ∈ board n m and ge-1: ∀ (i,j) ∈ board n m. i ≥ 1
```

```
    unfolding board-def by auto
  then have finite: finite ((fst) ‘ board n m) and
        non-empty: (fst) ‘ board n m ≠ {} and
        mem-1: 1 ∈ (fst) ‘ board n m
    using board-finite by auto (metis fstI image-eqI)
  then have Min ((fst) ‘ board n m) = 1
    using ge-1 by (auto simp: Min-eq-iff)
  then show ?thesis
    using assms knights-path-set-eq by auto
qed

lemma knights-path-min2: knights-path (board n m) ps ⟹ min2 ps = 1
proof −
  assume assms: knights-path (board n m) ps
  then have min n m ≥ 1
    using knights-path-board-m-n-geq-1 by auto
  then have (1,1) ∈ board n m and ge-1: ∀(i,j) ∈ board n m. j ≥ 1
    unfolding board-def by auto
  then have finite: finite ((snd) ‘ board n m) and
        non-empty: (snd) ‘ board n m ≠ {} and
        mem-1: 1 ∈ (snd) ‘ board n m
    using board-finite by auto (metis sndI image-eqI)
  then have Min ((snd) ‘ board n m) = 1
    using ge-1 by (auto simp: Min-eq-iff)
  then show ?thesis
    using assms knights-path-set-eq by auto
qed

lemma knights-path-max1: knights-path (board n m) ps ⟹ max1 ps = int n
proof −
  assume assms: knights-path (board n m) ps
  then have min n m ≥ 1
    using knights-path-board-m-n-geq-1 by auto
  then have (int n,1) ∈ board n m and leq-n: ∀(i,j) ∈ board n m. i ≤ int n
    unfolding board-def by auto
  then have finite: finite ((fst) ‘ board n m) and
        non-empty: (fst) ‘ board n m ≠ {} and
        mem-1: int n ∈ (fst) ‘ board n m
    using board-finite by auto (metis fstI image-eqI)
  then have Max ((fst) ‘ board n m) = int n
    using leq-n by (auto simp: Max-eq-iff)
  then show ?thesis
    using assms knights-path-set-eq by auto
qed

lemma knights-path-max2: knights-path (board n m) ps ⟹ max2 ps = int m
proof −
  assume assms: knights-path (board n m) ps
  then have min n m ≥ 1
```

using *knights-path-board-m-n-geq-1* **by** *auto*
then have $(1,int\ m) \in board\ n\ m$ **and** *leq-m*: $\forall (i,j) \in board\ n\ m.\ j \leq int\ m$
unfolding *board-def* **by** *auto*
then have *finite*: *finite* $((snd)$ ' $board\ n\ m)$ **and**
non-empty: (snd) ' $board\ n\ m \neq \{\}$ **and**
mem-1: $int\ m \in (snd)$ ' $board\ n\ m$
using *board-finite* **by** *auto* (*metis sndI image-eqI*)
then have *Max* $((snd)$ ' $board\ n\ m) = int\ m$
using *leq-m* **by** (*auto simp*: *Max-eq-iff*)
then show *?thesis*
using *assms knights-path-set-eq* **by** *auto*
qed

lemma *mirror1-aux-nil*: $ps = [] \longleftrightarrow mirror1\text{-}aux\ m\ ps = []$
using *mirror1-aux.elims* **by** *blast*

lemma *mirror1-nil*: $ps = [] \longleftrightarrow mirror1\ ps = []$
unfolding *mirror1-def* **using** *mirror1-aux-nil* **by** *blast*

lemma *mirror2-aux-nil*: $ps = [] \longleftrightarrow mirror2\text{-}aux\ m\ ps = []$
using *mirror2-aux.elims* **by** *blast*

lemma *mirror2-nil*: $ps = [] \longleftrightarrow mirror2\ ps = []$
unfolding *mirror2-def* **using** *mirror2-aux-nil* **by** *blast*

lemma *length-mirror1-aux*: *length ps* = *length* (*mirror1-aux n ps*)
by (*induction ps*) *auto*

lemma *length-mirror1*: *length ps* = *length* (*mirror1 ps*)
unfolding *mirror1-def* **using** *length-mirror1-aux* **by** *auto*

lemma *length-mirror2-aux*: *length ps* = *length* (*mirror2-aux n ps*)
by (*induction ps*) *auto*

lemma *length-mirror2*: *length ps* = *length* (*mirror2 ps*)
unfolding *mirror2-def* **using** *length-mirror2-aux* **by** *auto*

lemma *mirror1-board-iff*: $s_i \notin b \longleftrightarrow mirror1\text{-}square\ n\ s_i \notin mirror1\text{-}board\ n\ b$
unfolding *mirror1-board-def mirror1-square-def* **by** (*auto split*: *prod.splits*)

lemma *mirror2-board-iff*: $s_i \notin b \longleftrightarrow mirror2\text{-}square\ n\ s_i \notin mirror2\text{-}board\ n\ b$
unfolding *mirror2-board-def mirror2-square-def* **by** (*auto split*: *prod.splits*)

lemma *insert-mirror1-board*:
$insert\ (mirror1\text{-}square\ n\ s_i)\ (mirror1\text{-}board\ n\ b) = mirror1\text{-}board\ n\ (insert\ s_i\ b)$
unfolding *mirror1-board-def mirror1-square-def* **by** (*auto split*: *prod.splits*)

lemma *insert-mirror2-board*:
$insert\ (mirror2\text{-}square\ n\ s_i)\ (mirror2\text{-}board\ n\ b) = mirror2\text{-}board\ n\ (insert\ s_i\ b)$

unfolding *mirror2-board-def mirror2-square-def* **by** (*auto split*: *prod.splits*)

lemma $(i::int) = i'+1 \Longrightarrow n-i=n-(i'+1)$
by *auto*

lemma *valid-step-mirror1*:
valid-step s_i s_j $\longleftrightarrow$ *valid-step* (*mirror1-square n* s_i) (*mirror1-square n* s_j)
proof
assume *assms*: *valid-step* s_i s_j
obtain $i\ j\ i'\ j'$ **where** [*simp*]: $s_i = (i,j)$ $s_j = (i',j')$ **by** *force*
then have *valid-step* $(n-i,j)$ $(n-i',j')$
using *assms* **unfolding** *valid-step-def*
apply *simp*
apply (*elim disjE*)
apply *auto*
done
then show *valid-step* (*mirror1-square n* s_i) (*mirror1-square n* s_j)
unfolding *mirror1-square-def* **by** *auto*
next
assume *assms*: *valid-step* (*mirror1-square n* s_i) (*mirror1-square n* s_j)
obtain $i\ j\ i'\ j'$ **where** [*simp*]: $s_i = (i,j)$ $s_j = (i',j')$ **by** *force*
then have *valid-step* (i,j) (i',j')
using *assms* **unfolding** *valid-step-def mirror1-square-def*
apply *simp*
apply (*elim disjE*)
apply *auto*
done
then show *valid-step* s_i s_j
unfolding *mirror1-square-def* **by** *auto*
qed

lemma *valid-step-mirror2*:
valid-step s_i s_j $\longleftrightarrow$ *valid-step* (*mirror2-square m* s_i) (*mirror2-square m* s_j)
proof
assume *assms*: *valid-step* s_i s_j
obtain $i\ j\ i'\ j'$ **where** [*simp*]: $s_i = (i,j)$ $s_j = (i',j')$ **by** *force*
then have *valid-step* $(i,m-j)$ $(i',m-j')$
using *assms* **unfolding** *valid-step-def*
apply *simp*
apply (*elim disjE*)
apply *auto*
done
then show *valid-step* (*mirror2-square m* s_i) (*mirror2-square m* s_j)
unfolding *mirror2-square-def* **by** *auto*
next
assume *assms*: *valid-step* (*mirror2-square m* s_i) (*mirror2-square m* s_j)
obtain $i\ j\ i'\ j'$ **where** [*simp*]: $s_i = (i,j)$ $s_j = (i',j')$ **by** *force*
then have *valid-step* (i,j) (i',j')
using *assms* **unfolding** *valid-step-def mirror2-square-def*

apply *simp*
apply (*elim disjE*)
apply *auto*
done
then show *valid-step* s_i s_j
unfolding *mirror1-square-def* **by** *auto*
qed

lemma *hd-mirror1*:
assumes *knights-path* (*board n m*) *ps hd ps* = (*i*,*j*)
shows *hd* (*mirror1 ps*) = (*int n*+*1*−*i*,*j*)
using *assms*
proof −
have *hd* (*mirror1 ps*) = *hd* (*mirror1-aux* (*int n*+*1*) *ps*)
unfolding *mirror1-def* **using** *assms knights-path-min1 knights-path-max1* **by** *auto*
also have ... = *hd* (*mirror1-aux* (*int n*+*1*) ((*hd ps*)#(*tl ps*)))
using *assms knights-path-non-nil* **by** (*metis list.collapse*)
also have ... = (*int n*+*1*−*i*,*j*)
using *assms* **by** (*auto simp*: *mirror1-square-def*)
finally show *?thesis* **.**
qed

lemma *last-mirror1-aux*:
assumes *ps* ≠ [] *last ps* = (*i*,*j*)
shows *last* (*mirror1-aux n ps*) = (*n*−*i*,*j*)
using *assms*
proof (*induction ps*)
case (*Cons* s_i *ps*)
then show *?case*
using *mirror1-aux-nil Cons* **by** (*cases ps* = []) (*auto simp*: *mirror1-square-def*)
qed *auto*

lemma *last-mirror1*:
assumes *knights-path* (*board n m*) *ps last ps* = (*i*,*j*)
shows *last* (*mirror1 ps*) = (*int n*+*1*−*i*,*j*)
unfolding *mirror1-def* **using** *assms last-mirror1-aux knights-path-non-nil*
by (*simp add*: *knights-path-max1 knights-path-min1*)

lemma *hd-mirror2*:
assumes *knights-path* (*board n m*) *ps hd ps* = (*i*,*j*)
shows *hd* (*mirror2 ps*) = (*i*,*int m*+*1*−*j*)
using *assms*
proof −
have *hd* (*mirror2 ps*) = *hd* (*mirror2-aux* (*int m*+*1*) *ps*)
unfolding *mirror2-def* **using** *assms knights-path-min2 knights-path-max2* **by** *auto*
also have ... = *hd* (*mirror2-aux* (*int m*+*1*) ((*hd ps*)#(*tl ps*)))
using *assms knights-path-non-nil* **by** (*metis list.collapse*)

```
  also have ... = (i,int m+1−j)
    using assms by (auto simp: mirror2-square-def)
  finally show ?thesis .
qed

lemma last-mirror2-aux:
  assumes ps ≠ [] last ps = (i,j)
  shows last (mirror2-aux m ps) = (i,m−j)
  using assms
proof (induction ps)
  case (Cons s_i ps)
  then show ?case
    using mirror2-aux-nil Cons by (cases ps = []) (auto simp: mirror2-square-def)
qed auto

lemma last-mirror2:
  assumes knights-path (board n m) ps last ps = (i,j)
  shows last (mirror2 ps) = (i,int m+1−j)
  unfolding mirror2-def using assms last-mirror2-aux knights-path-non-nil
  by (simp add: knights-path-max2 knights-path-min2)

lemma mirror1-aux-knights-path:
  assumes knights-path b ps
  shows knights-path (mirror1-board n b) (mirror1-aux n ps)
  using assms
proof (induction rule: knights-path.induct)
  case (1 s_i)
  then have mirror1-board n {s_i} = {mirror1-square n s_i}
    unfolding mirror1-board-def by blast
  then show ?case by (auto intro: knights-path.intros)
next
  case (2 s_i b s_j ps)
  then have prems: mirror1-square n s_i ∉ mirror1-board n b
           valid-step (mirror1-square n s_i) (mirror1-square n s_j)
           and mirror1-aux n (s_j#ps) = mirror1-square n s_j#mirror1-aux n ps
    using 2 mirror1-board-iff valid-step-mirror1 by auto
  then show ?case
    using 2 knights-path.intros(2)[OF prems] insert-mirror1-board by auto
qed

corollary mirror1-knights-path:
  assumes knights-path (board n m) ps
  shows knights-path (board n m) (mirror1 ps)
  using assms
proof −
  have [simp]: min1 ps = 1 max1 ps = int n
    using assms knights-path-min1 knights-path-max1 by auto
  then have mirror1-board (int n+1) (board n m) = (board n m)
    using mirror1-board-id by auto
```

then have *knights-path* (*board n m*) (*mirror1-aux* (*int n+1*) *ps*)
using *assms mirror1-aux-knights-path*[*of board n m ps int n+1*] **by** *auto*
then show *?thesis* **unfolding** *mirror1-def* **by** *auto*
qed

lemma *mirror2-aux-knights-path*:
assumes *knights-path b ps*
shows *knights-path* (*mirror2-board n b*) (*mirror2-aux n ps*)
using *assms*
proof (*induction rule*: *knights-path.induct*)
case (*1* s_i)
then have *mirror2-board n* $\{s_i\}$ = {*mirror2-square n* s_i}
unfolding *mirror2-board-def* **by** *blast*
then show *?case* **by** (*auto intro*: *knights-path.intros*)
next
case (*2* s_i *b* s_j *ps*)
then have *prems*: *mirror2-square n* $s_i \notin$ *mirror2-board n b*
valid-step (*mirror2-square n* s_i) (*mirror2-square n* s_j)
and *mirror2-aux n* (s_j#*ps*) = *mirror2-square n* s_j#*mirror2-aux n ps*
using *2 mirror2-board-iff valid-step-mirror2* **by** *auto*
then show *?case*
using *2 knights-path.intros*(*2*)[*OF prems*] *insert-mirror2-board* **by** *auto*
qed

corollary *mirror2-knights-path*:
assumes *knights-path* (*board n m*) *ps*
shows *knights-path* (*board n m*) (*mirror2 ps*)
proof −
have [*simp*]: *min2 ps* = *1 max2 ps* = *int m*
using *assms knights-path-min2 knights-path-max2* **by** *auto*
then have *mirror2-board* (*int m+1*) (*board n m*) = (*board n m*)
using *mirror2-board-id* **by** *auto*
then have *knights-path* (*board n m*) (*mirror2-aux* (*int m+1*) *ps*)
using *assms mirror2-aux-knights-path*[*of board n m ps int m+1*] **by** *auto*
then show *?thesis* **unfolding** *mirror2-def* **by** *auto*
qed

5.3 Rotate Knight's Paths

Transposing (*transpose*) and mirroring (along first axis *mirror1*) a Knight's path preserves the Knight's path's property. Tranpose+Mirror1 equals a 90deg-clockwise turn.

corollary *rot90-knights-path*:
assumes *knights-path* (*board n m*) *ps*
shows *knights-path* (*board m n*) (*mirror1* (*transpose ps*))
using *assms transpose-knights-path mirror1-knights-path* **by** *auto*

lemma *hd-rot90-knights-path*:
assumes *knights-path* (*board n m*) *ps hd ps* = (*i*,*j*)

shows *hd* (*mirror1* (*transpose ps*)) = (*int m*+*1*−*j*,*i*)
using *assms*
proof −
have *hd* (*transpose ps*) = (*j*,*i*) *knights-path* (*board m n*) (*transpose ps*)
using *assms knights-path-non-nil hd-transpose transpose-knights-path*
by (*auto simp*: *transpose-square-def*)
then show *?thesis* **using** *hd-mirror1* **by** *auto*
qed

lemma *last-rot90-knights-path*:
assumes *knights-path* (*board n m*) *ps last ps* = (*i*,*j*)
shows *last* (*mirror1* (*transpose ps*)) = (*int m*+*1*−*j*,*i*)
using *assms*
proof −
have *last* (*transpose ps*) = (*j*,*i*) *knights-path* (*board m n*) (*transpose ps*)
using *assms knights-path-non-nil last-transpose transpose-knights-path*
by (*auto simp*: *transpose-square-def*)
then show *?thesis* **using** *last-mirror1* **by** *auto*
qed

6 Translating Paths and Boards

When constructing knight's paths for larger boards multiple knight's paths for smaller boards are concatenated. To concatenate paths the the coordinates in the path need to be translated. Therefore, simple auxiliary functions are provided.

6.1 Implementation of Path and Board Translation

Translate the coordinates for a path by (k_1,k_2).

fun *trans-path* :: *int* × *int* ⇒ *path* ⇒ *path* **where**
trans-path (k_1,k_2) [] = []
| *trans-path* (k_1,k_2) ((*i*,*j*)#*xs*) = (i+k_1,j+k_2)#(*trans-path* (k_1,k_2) *xs*)

Translate the coordinates of a board by (k_1,k_2).

definition *trans-board* :: *int* × *int* ⇒ *board* ⇒ *board* **where**
trans-board t b ≡ (*case t of* (k_1,k_2) ⇒ {(i+k_1,j+k_2)|*i j*. (*i*,*j*) ∈ *b*})

6.2 Correctness of Path and Board Translation

lemma *trans-path-length*: *length ps* = *length* (*trans-path* (k_1,k_2) *ps*)
by (*induction ps*) *auto*

lemma *trans-path-non-nil*: *ps* ≠ [] ⟹ *trans-path* (k_1,k_2) *ps* ≠ []
by (*induction ps*) *auto*

lemma *trans-path-correct*: $(i,j) \in set\ ps \longleftrightarrow (i+k_1,j+k_2) \in set\ (trans\text{-}path\ (k_1,k_2)\ ps)$
proof (*induction ps*)
  case (*Cons* s_i *ps*)
  then show *?case* **by** (*cases* s_i) *auto*
qed *auto*

lemma *trans-path-non-nil-last*:
  $ps \neq [] \Longrightarrow last\ (trans\text{-}path\ (k_1,k_2)\ ps) = last\ (trans\text{-}path\ (k_1,k_2)\ ((i,j)\#ps))$
  using *trans-path-non-nil* **by** (*induction ps*) *auto*

lemma *hd-trans-path*:
  assumes $ps \neq []$ $hd\ ps = (i,j)$
  shows $hd\ (trans\text{-}path\ (k_1,k_2)\ ps) = (i+k_1,j+k_2)$
  using *assms* **by** (*induction ps*) *auto*

lemma *last-trans-path*:
  assumes $ps \neq []$ $last\ ps = (i,j)$
  shows $last\ (trans\text{-}path\ (k_1,k_2)\ ps) = (i+k_1,j+k_2)$
  using *assms*
proof (*induction ps*)
  case (*Cons* s_i *ps*)
  then show *?case*
    using *trans-path-non-nil-last*[*symmetric*]
    apply (*cases* s_i)
    apply (*cases* $ps = []$)
    apply *auto*
    done
qed (*auto*)

lemma *take-trans*:
  shows $take\ k\ (trans\text{-}path\ (k_1,k_2)\ ps) = trans\text{-}path\ (k_1,k_2)\ (take\ k\ ps)$
proof (*induction ps arbitrary*: *k*)
  case *Nil*
  then show *?case* **by** *auto*
next
  case (*Cons* s_i *ps*)
  then obtain *i j* **where** $s_i = (i,j)$ **by** *force*
  then have $k = 0 \vee k > 0$ **by** *auto*
  then show *?case*
  proof (*elim disjE*)
    assume $k > 0$
    then show *?thesis* **using** *Cons.IH* **by** (*auto simp*: ‹$s_i = (i,j)$› *take-Cons'*)
  qed *auto*
qed

lemma *drop-trans*:
  shows $drop\ k\ (trans\text{-}path\ (k_1,k_2)\ ps) = trans\text{-}path\ (k_1,k_2)\ (drop\ k\ ps)$
proof (*induction ps arbitrary*: *k*)

  case *Nil*
  then show *?case* **by** *auto*
next
  case (*Cons* s_i *ps*)
  then obtain *i j* **where** $s_i = (i,j)$ **by** *force*
  then have $k = 0 \vee k > 0$ **by** *auto*
  then show *?case*
  proof (*elim disjE*)
    assume $k > 0$
    then show *?thesis* **using** *Cons.IH* **by** (*auto simp*: ‹$s_i = (i,j)$› *drop-Cons′*)
  qed *auto*
qed

lemma *trans-board-correct*: $(i,j) \in b \longleftrightarrow (i+k_1,j+k_2) \in$ *trans-board* (k_1,k_2) b
  unfolding *trans-board-def* **by** *auto*

lemma *board-subset*: $n_1 \le n_2 \Longrightarrow m_1 \le m_2 \Longrightarrow$ *board* n_1 $m_1 \subseteq$ *board* n_2 m_2
  unfolding *board-def* **by** *auto*

Board concatenation

corollary *board-concat*:
  shows *board* n m_1 $\cup$ *trans-board* $(0,\mathit{int}\ m_1)$ (*board* n m_2) $=$ *board* n (m_1+m_2)
(**is** *?b1* $\cup$ *?b2* $=$ *?b*)
proof
  show *?b1* $\cup$ *?b2* $\subseteq$ *?b* **unfolding** *board-def trans-board-def* **by** *auto*
next
  show *?b* $\subseteq$ *?b1* $\cup$ *?b2*
  proof
    fix x
    assume $x \in$ *?b*
    then obtain *i j* **where** *x-split*: $x = (i,j)$ $1 \le i \wedge i \le \mathit{int}\ n$ $1 \le j \wedge j \le \mathit{int}$ (m_1+m_2)
      unfolding *board-def* **by** *auto*
    then have $j \le \mathit{int}\ m_1 \vee (\mathit{int}\ m_1 < j \wedge j \le \mathit{int}\ (m_1+m_2))$ **by** *auto*
    then show $x \in$ *?b1* $\cup$ *?b2*
    proof
      assume $j \le \mathit{int}\ m_1$
      then show $x \in$ *?b1* $\cup$ *?b2* **using** *x-split* **unfolding** *board-def* **by** *auto*
    next
      assume *asm*: $\mathit{int}\ m_1 < j \wedge j \le \mathit{int}\ (m_1+m_2)$
      then have $(i,j-\mathit{int}\ m_1) \in$ *board* n m_2 **using** *x-split* **unfolding** *board-def* **by** *auto*
      then show $x \in$ *?b1* $\cup$ *?b2*
        using *x-split asm trans-board-correct*[*of i j*$-$*int* m_1 *board* n m_2 *0 int* m_1] **by** *auto*
    qed
  qed
qed

lemma *transpose-trans-board*:
  transpose-board (*trans-board* (k_1,k_2) *b*) = *trans-board* (k_2,k_1) (*transpose-board b*)
  unfolding *transpose-board-def trans-board-def* **by** *blast*

corollary *board-concatT*:
  shows *board* n_1 *m* ∪ *trans-board* (*int* n_1,*0*) (*board* n_2 *m*) = *board* (n_1+n_2) *m* (**is** *?b*$_1$ ∪ *?b*$_2$ = *?b*)
proof −
  let *?b*$_1$*T*=*board m* n_1
  let *?b*$_2$*T*=*trans-board* (*0*,*int* n_1) (*board m* n_2)
  have *?b*$_1$ ∪ *?b*$_2$ = *transpose-board* (*?b*$_1$*T* ∪ *?b*$_2$*T*)
    using *transpose-board2 transpose-union transpose-board transpose-trans-board* **by** *auto*
  also have ... = *transpose-board* (*board m* (n_1+n_2))
    using *board-concat* **by** *auto*
  also have ... = *board* (n_1+n_2) *m*
    using *transpose-board* **by** *auto*
  finally show *?thesis* **.**
qed

lemma *trans-valid-step*:
  valid-step (*i*,*j*) (*i'*,*j'*) ⟹ *valid-step* (*i*+k_1,*j*+k_2) (*i'*+k_1,*j'*+k_2)
  unfolding *valid-step-def* **by** *auto*

Translating a path and a boards preserves the validity.

lemma *trans-knights-path*:
  assumes *knights-path b ps*
  shows *knights-path* (*trans-board* (k_1,k_2) *b*) (*trans-path* (k_1,k_2) *ps*)
  using *assms*
proof (*induction rule*: *knights-path.induct*)
  case (*2* s_i *b* s_j *xs*)
  then obtain *i j i' j'* **where** *split*: s_i = (*i*,*j*) s_j = (*i'*,*j'*) **by** *force*
  let *?s*$_i$=(*i*+k_1,*j*+k_2)
  let *?s*$_j$=(*i'*+k_1,*j'*+k_2)
  let *?xs*=*trans-path* (k_1,k_2) *xs*
  let *?b*=*trans-board* (k_1,k_2) *b*
  have *simps*: *trans-path* (k_1,k_2) (s_i#s_j#*xs*) = *?s*$_i$#*?s*$_j$#*?xs*
      ?b ∪ {*?s*$_i$} = *trans-board* (k_1,k_2) (*b* ∪ {s_i})
    unfolding *trans-board-def* **using** *split* **by** *auto*
  have *?s*$_i$ ∉ *?b valid-step ?s*$_i$ *?s*$_j$ *knights-path ?b* (*?s*$_j$#*?xs*)
    using *2 split trans-valid-step* **by** (*auto simp*: *trans-board-def*)
  then have *knights-path* (*?b* ∪ {*?s*$_i$}) (*?s*$_i$#*?s*$_j$#*?xs*)
    using *knights-path.intros* **by** *auto*
  then show *?case* **using** *simps* **by** *auto*
qed (*auto simp*: *trans-board-def intro*: *knights-path.intros*)

Predicate that indicates if two squares s_i and s_j are adjacent in *ps*.

definition *step-in* :: *path* ⇒ *square* ⇒ *square* ⇒ *bool* **where**
  step-in ps s_i s_j ≡ (∃ *k*. *0* < *k* ∧ *k* < *length ps* ∧ *last* (*take k ps*) = s_i ∧ *hd* (*drop*

k ps) = s_j)

lemma *step-in-Cons*: *step-in ps* s_i s_j $\Longrightarrow$ *step-in* (s_k#*ps*) s_i s_j
proof −
  assume *step-in ps* s_i s_j
  then obtain *k* **where** $0 < k \wedge k <$ *length ps last* (*take k ps*) = s_i *hd* (*drop k ps*) = s_j
    unfolding *step-in-def* **by** *auto*
  then have $0 < k+1 \wedge k+1 <$ *length* (s_k#*ps*)
      last (*take* (*k*+*1*) (s_k#*ps*)) = s_i *hd* (*drop* (*k*+*1*) (s_k#*ps*)) = s_j
    by *auto*
  then show *?thesis*
    by (*auto simp*: *step-in-def*)
qed

lemma *step-in-append*: *step-in ps* s_i s_j $\Longrightarrow$ *step-in* (*ps*@*ps'*) s_i s_j
proof −
  assume *step-in ps* s_i s_j
  then obtain *k* **where** $0 < k \wedge k <$ *length ps last* (*take k ps*) = s_i *hd* (*drop k ps*) = s_j
    unfolding *step-in-def* **by** *auto*
  then have $0 < k \wedge k <$ *length* (*ps*@*ps'*)
      last (*take k* (*ps*@*ps'*)) = s_i *hd* (*drop k* (*ps*@*ps'*)) = s_j
    by *auto*
  then show *?thesis*
    by (*auto simp*: *step-in-def*)
qed

lemma *step-in-prepend*: *step-in ps* s_i s_j $\Longrightarrow$ *step-in* (*ps'*@*ps*) s_i s_j
  using *step-in-Cons* **by** (*induction ps' arbitrary*: *ps*) *auto*

lemma *step-in-valid-step*: *knights-path b ps* $\Longrightarrow$ *step-in ps* s_i s_j $\Longrightarrow$ *valid-step* s_i s_j
proof −
  assume *assms*: *knights-path b ps step-in ps* s_i s_j
  then obtain *k* **where** *k-prems*: $0 < k \wedge k <$ *length ps last* (*take k ps*) = s_i *hd* (*drop k ps*) = s_j
    unfolding *step-in-def* **by** *auto*
  then have $k = 1 \vee k > 1$ **by** *auto*
  then show *?thesis*
  proof (*elim disjE*)
    assume $k = 1$
    then obtain *ps'* **where** *ps* = s_i#s_j#*ps'*
      using *k-prems list-len-g-1-split* **by** *fastforce*
    then show *?thesis*
      using *assms* **by** (*auto elim*: *knights-path.cases*)
  next
    assume $k > 1$
    then have $0 < k-1 \wedge k-1 <$ *length ps*

using *k-prems* **by** *auto*
then obtain b **where** *knights-path b* (*drop* ($k-1$) *ps*)
using *assms knights-path-split* **by** *blast*

obtain ps' **where** *drop* ($k-1$) $ps = s_i\#s_j\#ps'$
using *k-prems* ‹$0 < k - 1 \wedge k - 1 <$ *length ps*›
by (*metis Cons-nth-drop-Suc Suc-diff-1 hd-drop-conv-nth last-snoc take-hd-drop*)
then show *?thesis*
using ‹*knights-path b* (*drop* ($k-1$) *ps*)› **by** (*auto elim*: *knights-path.cases*)
qed
qed

lemma *trans-step-in*:
step-in ps (i,j) (i',j') $\Longrightarrow$ *step-in* (*trans-path* (k_1,k_2) *ps*) $(i+k_1,j+k_2)$ $(i'+k_1,j'+k_2)$
proof $-$
let $?ps'$=*trans-path* (k_1,k_2) *ps*
assume *step-in ps* (i,j) (i',j')
then obtain k **where** $0 < k \wedge k <$ *length ps last* (*take k ps*) $= (i,j)$ *hd* (*drop k ps*) $= (i',j')$
unfolding *step-in-def* **by** *auto*
then have *take k ps* $\neq$ [] *drop k ps* $\neq$ [] **by** *fastforce+*
then have $0 < k \wedge k <$ *length* $?ps'$
last (*take k* $?ps'$) $= (i+k_1,j+k_2)$ *hd* (*drop k* $?ps'$) $= (i'+k_1,j'+k_2)$
using *trans-path-length*
last-trans-path[*OF* ‹*take k ps* $\neq$ []› ‹*last* (*take k ps*) $= (i,j)$›] *take-trans*
hd-trans-path[*OF* ‹*drop k ps* $\neq$ []› ‹*hd* (*drop k ps*) $= (i',j')$›] *drop-trans*
by *auto*
then show *?thesis*
by (*auto simp*: *step-in-def*)
qed

lemma *transpose-step-in*:
step-in ps s_i s_j $\Longrightarrow$ *step-in* (*transpose ps*) (*transpose-square* s_i) (*transpose-square* s_j)
(**is** - $\Longrightarrow$ *step-in* $?psT$ $?s_iT$ $?s_jT$)
proof $-$
assume *step-in ps* s_i s_j
then obtain k **where**
k-prems: $0 < k$ $k <$ *length ps last* (*take k ps*) $= s_i$ *hd* (*drop k ps*) $= s_j$
unfolding *step-in-def* **by** *auto*
then have *non-nil*: *take k ps* $\neq$ [] *drop k ps* $\neq$ [] **by** *fastforce+*
have *take k* $?psT$ = *transpose* (*take k ps*) *drop k* $?psT$ = *transpose* (*drop k ps*)
using *take-transpose drop-transpose* **by** *auto*
then have *last* (*take k* $?psT$) $= ?s_iT$ *hd* (*drop k* $?psT$) $= ?s_jT$
using *non-nil k-prems hd-transpose last-transpose* **by** *auto*
then show *step-in* $?psT$ $?s_iT$ $?s_jT$
unfolding *step-in-def* **using** *k-prems transpose-length* **by** *auto*
qed

lemma *hd-take*: $0 < k \Longrightarrow hd\ xs = hd\ (take\ k\ xs)$
  by (*induction xs*) *auto*

lemma *last-drop*: $k < length\ xs \Longrightarrow last\ xs = last\ (drop\ k\ xs)$
  by (*induction xs*) *auto*

6.3 Concatenate Knight's Paths and Circuits

Concatenate two knight's path on a $n \times m$-board along the 2nd axis if the first path contains the step $s_i \to s_j$ and there are valid steps $s_i \to hd\ ps_2'$ and $s_j \to last\ ps_2'$, where ps_2' is ps_2 is translated by m_1. An arbitrary step in ps_2 is preserved.

corollary *knights-path-split-concat-si-prev*:
  assumes $knights\text{-}path\ (board\ n\ m_1)\ ps_1\ knights\text{-}path\ (board\ n\ m_2)\ ps_2$
    $step\text{-}in\ ps_1\ s_i\ s_j\ hd\ ps_2 = (i_h,j_h)\ last\ ps_2 = (i_l,j_l)\ step\text{-}in\ ps_2\ (i,j)\ (i',j')$
    $valid\text{-}step\ s_i\ (i_h,int\ m_1+j_h)\ valid\text{-}step\ (i_l,int\ m_1+j_l)\ s_j$
  shows $\exists ps.\ knights\text{-}path\ (board\ n\ (m_1+m_2))\ ps \wedge hd\ ps = hd\ ps_1$
    $\wedge\ last\ ps = last\ ps_1 \wedge step\text{-}in\ ps\ (i,int\ m_1+j)\ (i',int\ m_1+j')$
  using *assms*
proof −
  let $?b_1=board\ n\ m_1$
  let $?b_2=board\ n\ m_2$
  let $?ps_2'=trans\text{-}path\ (0,int\ m_1)\ ps_2$
  let $?b'=trans\text{-}board\ (0,int\ m_1)\ ?b_2$
  have *kp2'*: $knights\text{-}path\ ?b'\ ?ps_2'$ **using** *assms trans-knights-path* **by** *auto*
  then have $?ps_2' \neq []$ **using** *knights-path-non-nil* **by** *auto*

  obtain k **where** *k-prems*:
    $0 < k\ k < length\ ps_1\ last\ (take\ k\ ps_1) = s_i\ hd\ (drop\ k\ ps_1) = s_j$
    using *assms* **unfolding** *step-in-def* **by** *auto*
  let $?ps=(take\ k\ ps_1)\ @\ ?ps_2'\ @\ (drop\ k\ ps_1)$
  obtain $b_1\ b_2$ **where** *b-prems*: $knights\text{-}path\ b_1\ (take\ k\ ps_1)\ knights\text{-}path\ b_2\ (drop$
$k\ ps_1)$
      $b_1 \cup b_2 = ?b_1\ b_1 \cap b_2 = \{\}$
    using *assms* ‹$0 < k$› ‹$k < length\ ps_1$› *knights-path-split* **by** *blast*

  have $hd\ ?ps_2' = (i_h,int\ m_1+j_h)\ last\ ?ps_2' = (i_l,int\ m_1+j_l)$
    using *assms knights-path-non-nil hd-trans-path last-trans-path* **by** *auto*
  then have $hd\ ?ps_2' = (i_h,int\ m_1+j_h)\ last\ ((take\ k\ ps_1)\ @\ ?ps_2') = (i_l,int\ m_1+j_l)$
    using ‹$?ps_2' \neq []$› **by** *auto*
  then have *vs*: $valid\text{-}step\ (last\ (take\ k\ ps_1))\ (hd\ ?ps_2')$
      $valid\text{-}step\ (last\ ((take\ k\ ps_1)\ @\ ?ps_2'))\ (hd\ (drop\ k\ ps_1))$
    using *assms k-prems* **by** *auto*

  have $?b_1 \cap ?b' = \{\}$ **unfolding** *board-def trans-board-def* **by** *auto*
  then have $b_1 \cap ?b' = \{\} \wedge (b_1 \cup ?b') \cap b_2 = \{\}$ **using** *b-prems* **by** *blast*
  then have *inter-empty*: $b_1 \cap ?b' = \{\}\ (b_1 \cup ?b') \cap b_2 = \{\}$ **by** *auto*

have $\textit{knights-path}\ (b_1 \cup \textit{?b}')\ ((\textit{take}\ k\ \textit{ps}_1)\ @\ \textit{?ps}_2')$
  using *kp2′ b-prems inter-empty vs knights-path-append* **by** *auto*
then have $\textit{knights-path}\ (b_1 \cup \textit{?b}' \cup b_2)\ \textit{?ps}$
  using *b-prems inter-empty vs knights-path-append*[**where** $\textit{ps}_1 = (\textit{take}\ k\ \textit{ps}_1)\ @\ \textit{?ps}_2'$] **by** *auto*
then have $\textit{knights-path}\ (\textit{?b}_1 \cup \textit{?b}')\ \textit{?ps}$
  using *b-prems Un-commute Un-assoc* **by** *metis*
then have *kp*: $\textit{knights-path}\ (\textit{board}\ n\ (m_1+m_2))\ \textit{?ps}$
  using *board-concat*[*of* $n\ m_1\ m_2$] **by** *auto*

have *hd*: $\textit{hd}\ \textit{?ps} = \textit{hd}\ \textit{ps}_1$
  using *assms* ‹$0 < k$› *knights-path-non-nil hd-take* **by** *auto*

have *last*: $\textit{last}\ \textit{?ps} = \textit{last}\ \textit{ps}_1$
  using *assms* ‹$k < \textit{length}\ \textit{ps}_1$› *knights-path-non-nil last-drop* **by** *auto*

have *m-simps*: $j+\textit{int}\ m_1 = \textit{int}\ m_1+j\ \ j'+\textit{int}\ m_1 = \textit{int}\ m_1+j'$ **by** *auto*
have *si*: $\textit{step-in}\ \textit{?ps}\ (i,\textit{int}\ m_1+j)\ (i',\textit{int}\ m_1+j')$
  using *assms step-in-append*[*OF step-in-prepend*[*OF trans-step-in*],
    of $\textit{ps}_2\ i\ j\ i'\ j'\ \textit{take}\ k\ \textit{ps}_1\ 0\ \textit{int}\ m_1\ \textit{drop}\ k\ \textit{ps}_1$]
  by (*auto simp*: *m-simps*)

show *?thesis* **using** *kp hd last si* **by** *auto*
qed

lemma *len1-hd-last*: $\textit{length}\ \textit{xs} = 1 \Longrightarrow \textit{hd}\ \textit{xs} = \textit{last}\ \textit{xs}$
  by (*induction xs*) *auto*

Weaker version of $[\![\textit{knights-path}\ (\textit{board}\ \textit{?n}\ \textit{?m}_1)\ \textit{?ps}_1;\ \textit{knights-path}\ (\textit{board}\ \textit{?n}\ \textit{?m}_2)\ \textit{?ps}_2;\ \textit{step-in}\ \textit{?ps}_1\ \textit{?s}_i\ \textit{?s}_j;\ \textit{hd}\ \textit{?ps}_2 = (\textit{?i}_h,\ \textit{?j}_h);\ \textit{last}\ \textit{?ps}_2 = (\textit{?i}_l,\ \textit{?j}_l);\ \textit{step-in}\ \textit{?ps}_2\ (\textit{?i},\ \textit{?j})\ (\textit{?i}',\ \textit{?j}');\ \textit{valid-step}\ \textit{?s}_i\ (\textit{?i}_h,\ \textit{int}\ \textit{?m}_1 + \textit{?j}_h);\ \textit{valid-step}\ (\textit{?i}_l,\ \textit{int}\ \textit{?m}_1 + \textit{?j}_l)\ \textit{?s}_j]\!] \Longrightarrow \exists \textit{ps}.\ \textit{knights-path}\ (\textit{board}\ \textit{?n}\ (\textit{?m}_1 + \textit{?m}_2))\ \textit{ps} \wedge \textit{hd}\ \textit{ps} = \textit{hd}\ \textit{?ps}_1 \wedge \textit{last}\ \textit{ps} = \textit{last}\ \textit{?ps}_1 \wedge \textit{step-in}\ \textit{ps}\ (\textit{?i},\ \textit{int}\ \textit{?m}_1 + \textit{?j})\ (\textit{?i}',\ \textit{int}\ \textit{?m}_1 + \textit{?j}')$.

corollary *knights-path-split-concat*:
  assumes $\textit{knights-path}\ (\textit{board}\ n\ m_1)\ \textit{ps}_1\ \ \textit{knights-path}\ (\textit{board}\ n\ m_2)\ \textit{ps}_2$
    $\textit{step-in}\ \textit{ps}_1\ s_i\ s_j\ \ \textit{hd}\ \textit{ps}_2 = (i_h,j_h)\ \ \textit{last}\ \textit{ps}_2 = (i_l,j_l)$
    $\textit{valid-step}\ s_i\ (i_h,\textit{int}\ m_1+j_h)\ \ \textit{valid-step}\ (i_l,\textit{int}\ m_1+j_l)\ s_j$
  shows $\exists \textit{ps}.\ \textit{knights-path}\ (\textit{board}\ n\ (m_1+m_2))\ \textit{ps} \wedge \textit{hd}\ \textit{ps} = \textit{hd}\ \textit{ps}_1 \wedge \textit{last}\ \textit{ps} = \textit{last}\ \textit{ps}_1$
proof −
  have $\textit{length}\ \textit{ps}_2 = 1 \vee \textit{length}\ \textit{ps}_2 > 1$
    using *assms knights-path-non-nil* **by** (*meson length-0-conv less-one linorder-neqE-nat*)
  then show *?thesis*
  proof (*elim disjE*)
    let $\textit{?s}_k = (i_h,\textit{int}\ m_1+j_h)$
    assume $\textit{length}\ \textit{ps}_2 = 1$

    then have $(i_h,j_h) = (i_l,j_l)$

using *assms len1-hd-last* **by** *metis*
then have *valid-step* s_i *?*s_k *valid-step ?*s_k s_j *valid-step* s_i s_j
using *assms step-in-valid-step* **by** *auto*
then show *?thesis*
using *valid-step-non-transitive* **by** *blast*
next
assume *length* $ps_2 > 1$
then obtain i_1 j_1 i_2 j_2 ps_2' **where** $ps_2 = (i_1,j_1)\#(i_2,j_2)\#ps_2'$
using *list-len-g-1-split* **by** *fastforce*
then have *last* (*take 1* ps_2) = (i_1,j_1) *hd* (*drop 1* ps_2) = (i_2,j_2) **by** *auto*
then have *step-in* ps_2 (i_1,j_1) (i_2,j_2) **using** ‹*length* $ps_2 > 1$› **by** (*auto simp*: *step-in-def*)
then show *?thesis*
using *assms knights-path-split-concat-si-prev* **by** *blast*
qed
qed

Concatenate two knight's path on a $n\times m$-board along the 1st axis.

corollary *knights-path-split-concatT*:
assumes *knights-path* (*board* n_1 *m*) ps_1 *knights-path* (*board* n_2 *m*) ps_2
step-in ps_1 s_i s_j *hd* $ps_2 = (i_h,j_h)$ *last* $ps_2 = (i_l,j_l)$
valid-step s_i (*int* n_1+i_h,j_h) *valid-step* (*int* n_1+i_l,j_l) s_j
shows $\exists$ *ps*. *knights-path* (*board* (n_1+n_2) *m*) *ps* $\wedge$ *hd ps* = *hd* ps_1 $\wedge$ *last ps* = *last* ps_1
using *assms*
proof −
let *?*ps_1T=*transpose* ps_1
let *?*ps_2T=*transpose* ps_2
have *kps*: *knights-path* (*board m* n_1) *?*ps_1T *knights-path* (*board m* n_2) *?*ps_2T
using *assms transpose-knights-path* **by** *auto*

let *?*s_iT=*transpose-square* s_i
let *?*s_jT=*transpose-square* s_j
have *si*: *step-in ?*ps_1T *?*s_iT *?*s_jT
using *assms transpose-step-in* **by** *auto*

have $ps_1 \neq []$ $ps_2 \neq []$
using *assms knights-path-non-nil* **by** *auto*
then have *hd-last2*: *hd ?*$ps_2T = (j_h,i_h)$ *last ?*$ps_2T = (j_l,i_l)$
using *assms hd-transpose last-transpose* **by** (*auto simp*: *transpose-square-def*)

have *vs*: *valid-step ?*s_iT (j_h,*int* n_1+i_h) *valid-step* (j_l,*int* n_1+i_l) *?*s_jT
using *assms transpose-valid-step* **by** (*auto simp*: *transpose-square-def split*: *prod.splits*)

then obtain *ps* **where**
ps-prems: *knights-path* (*board m* (n_1+n_2)) *ps hd ps* = *hd ?*ps_1T *last ps* = *last ?*ps_1T
using *knights-path-split-concat*[*OF kps si hd-last2 vs*] **by** *auto*

then have *ps* $\neq$ [] **using** *knights-path-non-nil* **by** *auto*
let *?psT=transpose ps*
have *knights-path* (*board* (n_1+n_2) *m*) *?psT hd ?psT* = *hd* ps_1 *last ?psT* = *last* ps_1
using ‹ps_1 $\neq$ []› ‹*ps* $\neq$ []› *ps-prems transpose-knights-path hd-transpose last-transpose*
by (*auto simp*: *transpose2*)
then show *?thesis* **by** *auto*
qed

Concatenate two Knight's path along the 2nd axis. There is a valid step from the last square in the first Knight's path ps_1 to the first square in the second Knight's path ps_2.

corollary *knights-path-concat*:
assumes *knights-path* (*board n* m_1) ps_1 *knights-path* (*board n* m_2) ps_2
hd ps_2 = (i_h,j_h) *valid-step* (*last* ps_1) (i_h,*int* m_1+j_h)
shows *knights-path* (*board n* (m_1+m_2)) (ps_1 @ (*trans-path* (*0*,*int* m_1) ps_2))
proof –
let *?*ps_2′=*trans-path* (*0*,*int* m_1) ps_2
let *?b=trans-board* (*0*,*int* m_1) (*board n* m_2)
have *inter-empty*: *board n* m_1 $\cap$ *?b* = {}
unfolding *board-def trans-board-def* **by** *auto*
have *hd ?*ps_2′ = (i_h,*int* m_1+j_h)
using *assms knights-path-non-nil hd-trans-path* **by** *auto*
then have *kp*: *knights-path* (*board n* m_1) ps_1 *knights-path ?b ?*ps_2′ **and**
vs: *valid-step* (*last* ps_1) (*hd ?*ps_2′)
using *assms trans-knights-path* **by** *auto*
then show *knights-path* (*board n* (m_1+m_2)) (ps_1 @ *?*ps_2′)
using *knights-path-append*[*OF kp inter-empty vs*] *board-concat* **by** *auto*
qed

Concatenate two Knight's path along the 2nd axis. The first Knight's path end in (*2*,m_1–*1*) (lower-right) and the second Knight's paths start in (*1*,*1*) (lower-left).

corollary *knights-path-lr-concat*:
assumes *knights-path* (*board n* m_1) ps_1 *knights-path* (*board n* m_2) ps_2
last ps_1 = (*2*,*int* m_1–*1*) *hd* ps_2 = (*1*,*1*)
shows *knights-path* (*board n* (m_1+m_2)) (ps_1 @ (*trans-path* (*0*,*int* m_1) ps_2))
proof –
have *valid-step* (*last* ps_1) (*1*,*int* m_1+*1*)
using *assms* **unfolding** *valid-step-def* **by** *auto*
then show *?thesis*
using *assms knights-path-concat* **by** *auto*
qed

Concatenate two Knight's circuits along the 2nd axis. In the first Knight's path the squares (*2*,m_1–*1*) and (*4*,m_1) are adjacent and the second Knight's cirucit starts in (*1*,*1*) (lower-left) and end in (*3*,*2*).

corollary *knights-circuit-lr-concat*:
assumes *knights-circuit* (*board n* m_1) ps_1 *knights-circuit* (*board n* m_2) ps_2
step-in ps_1 (*2*,*int* m_1−*1*) (*4*,*int* m_1)
hd ps_2 = (*1*,*1*) *last* ps_2 = (*3*,*2*) *step-in* ps_2 (*2*,*int* m_2−*1*) (*4*,*int* m_2)
shows ∃ *ps*. *knights-circuit* (*board n* (m_1+m_2)) *ps* ∧ *step-in ps* (*2*,*int* (m_1+m_2)−*1*) (*4*,*int* (m_1+m_2))
proof −
have *kp1*: *knights-path* (*board n* m_1) ps_1 **and** *kp2*: *knights-path* (*board n* m_2) ps_2

and *vs*: *valid-step* (*last* ps_1) (*hd* ps_1)
using *assms* **unfolding** *knights-circuit-def* **by** *auto*

have *m-simps*: *int* m_1 + (*int* m_2−*1*) = *int* (m_1+m_2)−*1* *int* m_1 + *int* m_2 = *int* (m_1+m_2) **by** *auto*

have *valid-step* (*2*,*int* m_1−*1*) (*1*,*int* m_1+*1*) *valid-step* (*3*,*int* m_1+*2*) (*4*,*int* m_1)
unfolding *valid-step-def* **by** *auto*
then obtain *ps* **where** *knights-path* (*board n* (m_1+m_2)) *ps hd ps* = *hd* ps_1 *last ps* = *last* ps_1 **and**
si: *step-in ps* (*2*,*int* (m_1+m_2)−*1*) (*4*,*int* (m_1+m_2))
using *assms kp1 kp2*
knights-path-split-concat-si-prev[*of n* m_1 ps_1 m_2 ps_2 (*2*,*int* m_1−*1*)
(*4*,*int* m_1) *1 1 3 2 2 int* m_2−*1 4 int* m_2]
by (*auto simp only*: *m-simps*)
then have *knights-circuit* (*board n* (m_1+m_2)) *ps*
using *vs* **by** (*auto simp*: *knights-circuit-def*)
then show *?thesis*
using *si* **by** *auto*
qed

7 Parsing Paths

In this section functions are implemented to parse and construct paths. The parser converts the matrix representation ((*nat list*) *list*) used in [1] to a path (*path*).

for debugging

fun *test-path* :: *path* ⇒ *bool* **where**
test-path (s_i#s_j#*xs*) = (*step-checker* s_i s_j ∧ *test-path* (s_j#*xs*))
| *test-path* - = *True*

fun *f-opt* :: (′*a* ⇒ ′*a*) ⇒ ′*a option* ⇒ ′*a option* **where**
f-opt - *None* = *None*
| *f-opt f* (*Some a*) = *Some* (*f a*)

fun *add-opt-fst-sq* :: *int* ⇒ *square option* ⇒ *square option* **where**
add-opt-fst-sq - *None* = *None*
| *add-opt-fst-sq k* (*Some* (*i*,*j*)) = *Some* (*k*+*i*,*j*)

fun *find-k-in-col* :: *nat* ⇒ *nat list* ⇒ *int option* **where**
find-k-in-col k [] = *None*
| *find-k-in-col k* (*c*#*cs*) = (*if c* = *k then Some 1 else f-opt* ((+) *1*) (*find-k-in-col k cs*))

fun *find-k-sqr* :: *nat* ⇒ (*nat list*) *list* ⇒ *square option* **where**
find-k-sqr k [] = *None*
| *find-k-sqr k* (*r*#*rs*) = (*case find-k-in-col k r of*
None ⇒ *f-opt* (λ(*i*,*j*). (*i*+*1*,*j*)) (*find-k-sqr k rs*)
| *Some j* ⇒ *Some* (*1*,*j*))

Auxiliary function to easily parse pre-computed boards from paper.

fun *to-sqrs* :: *nat* ⇒ (*nat list*) *list* ⇒ *path option* **where**
to-sqrs 0 rs = *Some* []
| *to-sqrs k rs* = (*case find-k-sqr k rs of*
None ⇒ *None*
| *Some* s_i ⇒ *f-opt* (λ*ps*. *ps*@[s_i]) (*to-sqrs* (*k*−*1*) *rs*))

fun *num-elems* :: (*nat list*) *list* ⇒ *nat* **where**
num-elems (*r*#*rs*) = *length r* ∗ *length* (*r*#*rs*)

Convert a matrix (*nat list list*) to a path (*path*). With this function we implicitly define the lower-left corner to be (*1*,*1*) and the upper-right corner to be (*n*,*m*).

definition *to-path rs* ≡ *to-sqrs* (*num-elems rs*) (*rev rs*)

Example

value *to-path*
[[*3*,*22*,*13*,*16*,*5*],
[*12*,*17*,*4*,*21*,*14*],
[*23*,*2*,*15*,*6*,*9*],
[*18*,*11*,*8*,*25*,*20*],
[*1*,*24*,*19*,*10*,*7*::*nat*]]

8 Knight's Paths for $5\times m$-Boards

Given here are knight's paths, *kp5xmlr* and *kp5xmur*, for the ($5\times m$)-board that start in the lower-left corner for $m \in \{5,6,7,8,9\}$. The path *kp5xmlr* ends in the lower-right corner, whereas the path *kp5xmur* ends in the upper-right corner. The tables show the visited squares numbered in ascending order.

abbreviation *b5x5* ≡ *board 5 5*

A Knight's path for the (5×5)-board that starts in the lower-left and ends in the lower-right.

3	22	13	16	5
12	17	4	21	14
23	2	15	6	9
18	11	8	25	20
1	24	19	10	7

abbreviation *kp5x5lr* ≡ *the* (*to-path*
  [[*3*,*22*,*13*,*16*,*5*],
  [*12*,*17*,*4*,*21*,*14*],
  [*23*,*2*,*15*,*6*,*9*],
  [*18*,*11*,*8*,*25*,*20*],
  [*1*,*24*,*19*,*10*,*7*]])
lemma *kp-5x5-lr*: *knights-path b5x5 kp5x5lr*
  by (*simp only*: *knights-path-exec-simp*) *eval*

lemma *kp-5x5-lr-hd*: *hd kp5x5lr* = (*1*,*1*) **by** *eval*

lemma *kp-5x5-lr-last*: *last kp5x5lr* = (*2*,*4*) **by** *eval*

lemma *kp-5x5-lr-non-nil*: *kp5x5lr* ≠ [] **by** *eval*

A Knight's path for the (5×5)-board that starts in the lower-left and ends in the upper-right.

7	12	15	20	5
16	21	6	25	14
11	8	13	4	19
22	17	2	9	24
1	10	23	18	3

abbreviation *kp5x5ur* ≡ *the* (*to-path*
  [[*7*,*12*,*15*,*20*,*5*],
  [*16*,*21*,*6*,*25*,*14*],
  [*11*,*8*,*13*,*4*,*19*],
  [*22*,*17*,*2*,*9*,*24*],
  [*1*,*10*,*23*,*18*,*3*]])
lemma *kp-5x5-ur*: *knights-path b5x5 kp5x5ur*
  by (*simp only*: *knights-path-exec-simp*) *eval*

lemma *kp-5x5-ur-hd*: *hd kp5x5ur* = (*1*,*1*) **by** *eval*

lemma *kp-5x5-ur-last*: *last kp5x5ur* = (*4*,*4*) **by** *eval*

lemma *kp-5x5-ur-non-nil*: *kp5x5ur* ≠ [] **by** *eval*

abbreviation *b5x6* ≡ *board 5 6*

A Knight's path for the (5×6)-board that starts in the lower-left and ends in the lower-right.

7	14	21	28	5	12
22	27	6	13	20	29
15	8	17	24	11	4
26	23	2	9	30	19
1	16	25	18	3	10

abbreviation *kp5x6lr* ≡ *the* (*to-path*
  [[*7*,*14*,*21*,*28*,*5*,*12*],
  [*22*,*27*,*6*,*13*,*20*,*29*],
  [*15*,*8*,*17*,*24*,*11*,*4*],
  [*26*,*23*,*2*,*9*,*30*,*19*],
  [*1*,*16*,*25*,*18*,*3*,*10*]])
lemma *kp-5x6-lr*: *knights-path b5x6 kp5x6lr*
  by (*simp only*: *knights-path-exec-simp*) *eval*

lemma *kp-5x6-lr-hd*: *hd kp5x6lr* = (*1*,*1*) **by** *eval*

lemma *kp-5x6-lr-last*: *last kp5x6lr* = (*2*,*5*) **by** *eval*

lemma *kp-5x6-lr-non-nil*: *kp5x6lr* ≠ [] **by** *eval*

A Knight's path for the (5×6)-board that starts in the lower-left and ends in the upper-right.

3	10	29	20	5	12
28	19	4	11	30	21
9	2	17	24	13	6
18	27	8	15	22	25
1	16	23	26	7	14

abbreviation *kp5x6ur* ≡ *the* (*to-path*
  [[*3*,*10*,*29*,*20*,*5*,*12*],
  [*28*,*19*,*4*,*11*,*30*,*21*],
  [*9*,*2*,*17*,*24*,*13*,*6*],
  [*18*,*27*,*8*,*15*,*22*,*25*],
  [*1*,*16*,*23*,*26*,*7*,*14*]])
lemma *kp-5x6-ur*: *knights-path b5x6 kp5x6ur*
  by (*simp only*: *knights-path-exec-simp*) *eval*

lemma *kp-5x6-ur-hd*: *hd kp5x6ur* = (*1*,*1*) **by** *eval*

lemma *kp-5x6-ur-last*: *last kp5x6ur* = (*4*,*5*) **by** *eval*

lemma *kp-5x6-ur-non-nil*: *kp5x6ur* $\neq$ [] **by** *eval*

abbreviation *b5x7* $\equiv$ *board 5 7*

A Knight's path for the (5×7)-board that starts in the lower-left and ends in the lower-right.

3	12	21	30	5	14	23
20	29	4	13	22	31	6
11	2	19	32	7	24	15
28	33	10	17	26	35	8
1	18	27	34	9	16	25

abbreviation *kp5x7lr* $\equiv$ *the* (*to-path*
[[*3*,*12*,*21*,*30*,*5*,*14*,*23*],
[*20*,*29*,*4*,*13*,*22*,*31*,*6*],
[*11*,*2*,*19*,*32*,*7*,*24*,*15*],
[*28*,*33*,*10*,*17*,*26*,*35*,*8*],
[*1*,*18*,*27*,*34*,*9*,*16*,*25*]])
lemma *kp-5x7-lr*: *knights-path b5x7 kp5x7lr*
by (*simp only*: *knights-path-exec-simp*) *eval*

lemma *kp-5x7-lr-hd*: *hd kp5x7lr* = (*1*,*1*) **by** *eval*

lemma *kp-5x7-lr-last*: *last kp5x7lr* = (*2*,*6*) **by** *eval*

lemma *kp-5x7-lr-non-nil*: *kp5x7lr* $\neq$ [] **by** *eval*

A Knight's path for the (5×7)-board that starts in the lower-left and ends in the upper-right.

3	32	11	34	5	26	13
10	19	4	25	12	35	6
31	2	33	20	23	14	27
18	9	24	29	16	7	22
1	30	17	8	21	28	15

abbreviation *kp5x7ur* $\equiv$ *the* (*to-path*
[[*3*,*32*,*11*,*34*,*5*,*26*,*13*],
[*10*,*19*,*4*,*25*,*12*,*35*,*6*],
[*31*,*2*,*33*,*20*,*23*,*14*,*27*],
[*18*,*9*,*24*,*29*,*16*,*7*,*22*],
[*1*,*30*,*17*,*8*,*21*,*28*,*15*]])
lemma *kp-5x7-ur*: *knights-path b5x7 kp5x7ur*
by (*simp only*: *knights-path-exec-simp*) *eval*

lemma *kp-5x7-ur-hd*: *hd kp5x7ur* = (*1*,*1*) **by** *eval*

lemma *kp-5x7-ur-last*: *last kp5x7ur* = (*4*,*6*) **by** *eval*

lemma *kp-5x7-ur-non-nil*: *kp5x7ur* ≠ [] **by** *eval*

abbreviation *b5x8* ≡ *board 5 8*

A Knight's path for the (*5*×*8*)-board that starts in the lower-left and ends in the lower-right.

3	12	37	26	5	14	17	28
34	23	4	13	36	27	6	15
11	2	35	38	25	16	29	18
22	33	24	9	20	31	40	7
1	10	21	32	39	8	19	30

abbreviation *kp5x8lr* ≡ *the* (*to-path*
  [[*3*,*12*,*37*,*26*,*5*,*14*,*17*,*28*],
  [*34*,*23*,*4*,*13*,*36*,*27*,*6*,*15*],
  [*11*,*2*,*35*,*38*,*25*,*16*,*29*,*18*],
  [*22*,*33*,*24*,*9*,*20*,*31*,*40*,*7*],
  [*1*,*10*,*21*,*32*,*39*,*8*,*19*,*30*]])
lemma *kp-5x8-lr*: *knights-path b5x8 kp5x8lr*
  by (*simp only*: *knights-path-exec-simp*) *eval*

lemma *kp-5x8-lr-hd*: *hd kp5x8lr* = (*1*,*1*) **by** *eval*

lemma *kp-5x8-lr-last*: *last kp5x8lr* = (*2*,*7*) **by** *eval*

lemma *kp-5x8-lr-non-nil*: *kp5x8lr* ≠ [] **by** *eval*

A Knight's path for the (*5*×*8*)-board that starts in the lower-left and ends in the upper-right.

33	8	17	38	35	6	15	24
18	37	34	7	16	25	40	5
9	32	29	36	39	14	23	26
30	19	2	11	28	21	4	13
1	10	31	20	3	12	27	22

abbreviation *kp5x8ur* ≡ *the* (*to-path*
  [[*33*,*8*,*17*,*38*,*35*,*6*,*15*,*24*],
  [*18*,*37*,*34*,*7*,*16*,*25*,*40*,*5*],
  [*9*,*32*,*29*,*36*,*39*,*14*,*23*,*26*],
  [*30*,*19*,*2*,*11*,*28*,*21*,*4*,*13*],
  [*1*,*10*,*31*,*20*,*3*,*12*,*27*,*22*]])
lemma *kp-5x8-ur*: *knights-path b5x8 kp5x8ur*

by (*simp only*: *knights-path-exec-simp*) *eval*

lemma *kp-5x8-ur-hd*: *hd kp5x8ur* = (*1*,*1*) **by** *eval*

lemma *kp-5x8-ur-last*: *last kp5x8ur* = (*4*,*7*) **by** *eval*

lemma *kp-5x8-ur-non-nil*: *kp5x8ur* ≠ [] **by** *eval*

abbreviation *b5x9* ≡ *board 5 9*

A Knight's path for the (*5*×*9*)-board that starts in the lower-left and ends in the lower-right.

9	4	11	16	23	42	33	36	25
12	17	8	3	32	37	24	41	34
5	10	15	20	43	22	35	26	29
18	13	2	7	38	31	28	45	40
1	6	19	14	21	44	39	30	27

abbreviation *kp5x9lr* ≡ *the* (*to-path*
[[*9*,*4*,*11*,*16*,*23*,*42*,*33*,*36*,*25*],
[*12*,*17*,*8*,*3*,*32*,*37*,*24*,*41*,*34*],
[*5*,*10*,*15*,*20*,*43*,*22*,*35*,*26*,*29*],
[*18*,*13*,*2*,*7*,*38*,*31*,*28*,*45*,*40*],
[*1*,*6*,*19*,*14*,*21*,*44*,*39*,*30*,*27*]])
lemma *kp-5x9-lr*: *knights-path b5x9 kp5x9lr*
by (*simp only*: *knights-path-exec-simp*) *eval*

lemma *kp-5x9-lr-hd*: *hd kp5x9lr* = (*1*,*1*) **by** *eval*

lemma *kp-5x9-lr-last*: *last kp5x9lr* = (*2*,*8*) **by** *eval*

lemma *kp-5x9-lr-non-nil*: *kp5x9lr* ≠ [] **by** *eval*

A Knight's path for the (*5*×*9*)-board that starts in the lower-left and ends in the upper-right.

9	4	11	16	27	32	35	40	25
12	17	8	3	36	41	26	45	34
5	10	15	20	31	28	33	24	39
18	13	2	7	42	37	22	29	44
1	6	19	14	21	30	43	38	23

abbreviation *kp5x9ur* ≡ *the* (*to-path*
[[*9*,*4*,*11*,*16*,*27*,*32*,*35*,*40*,*25*],
[*12*,*17*,*8*,*3*,*36*,*41*,*26*,*45*,*34*],
[*5*,*10*,*15*,*20*,*31*,*28*,*33*,*24*,*39*],

$[18,13,2,7,42,37,22,29,44]$,
$[1,6,19,14,21,30,43,38,23]])$

lemma *kp-5x9-ur*: *knights-path b5x9 kp5x9ur*
by (*simp only*: *knights-path-exec-simp*) *eval*

lemma *kp-5x9-ur-hd*: *hd kp5x9ur* = $(1,1)$ **by** *eval*

lemma *kp-5x9-ur-last*: *last kp5x9ur* = $(4,8)$ **by** *eval*

lemma *kp-5x9-ur-non-nil*: *kp5x9ur* $\neq$ [] **by** *eval*

lemmas *kp-5xm-lr* =
kp-5x5-lr kp-5x5-lr-hd kp-5x5-lr-last kp-5x5-lr-non-nil
kp-5x6-lr kp-5x6-lr-hd kp-5x6-lr-last kp-5x6-lr-non-nil
kp-5x7-lr kp-5x7-lr-hd kp-5x7-lr-last kp-5x7-lr-non-nil
kp-5x8-lr kp-5x8-lr-hd kp-5x8-lr-last kp-5x8-lr-non-nil
kp-5x9-lr kp-5x9-lr-hd kp-5x9-lr-last kp-5x9-lr-non-nil

lemmas *kp-5xm-ur* =
kp-5x5-ur kp-5x5-ur-hd kp-5x5-ur-last kp-5x5-ur-non-nil
kp-5x6-ur kp-5x6-ur-hd kp-5x6-ur-last kp-5x6-ur-non-nil
kp-5x7-ur kp-5x7-ur-hd kp-5x7-ur-last kp-5x7-ur-non-nil
kp-5x8-ur kp-5x8-ur-hd kp-5x8-ur-last kp-5x8-ur-non-nil
kp-5x9-ur kp-5x9-ur-hd kp-5x9-ur-last kp-5x9-ur-non-nil

For every $5 \times m$-board with $m \geq 5$ there exists a knight's path that starts in $(1,1)$ (bottom-left) and ends in $(2,m-1)$ (bottom-right).

lemma *knights-path-5xm-lr-exists*:
assumes $m \geq 5$
shows $\exists$ *ps*. *knights-path* (*board 5 m*) *ps* $\wedge$ *hd ps* = $(1,1)$ $\wedge$ *last ps* = $(2,int\ m-1)$
using *assms*
proof (*induction m rule*: *less-induct*)
case (*less m*)
then have $m \in \{5,6,7,8,9\} \vee 5 \leq m-5$ **by** *auto*
then show *?case*
proof (*elim disjE*)
assume $m \in \{5,6,7,8,9\}$
then show *?thesis* **using** *kp-5xm-lr* **by** *fastforce*
next
assume *m-ge*: $5 \leq m-5$
then obtain ps_1 **where** ps_1*-IH*: *knights-path* (*board* 5 $(m-5)$) ps_1 *hd* ps_1 = $(1,1)$
last $ps_1 = (2,int\ (m-5)-1)$ $ps_1 \neq []$
using *less.IH*[*of* $m-5$] *knights-path-non-nil* **by** *auto*

let $?ps_2$=*kp5x5lr*
let $?ps_2{}'=ps_1$ @ *trans-path* $(0,int\ (m-5))$ $?ps_2$
have *knights-path b5x5* $?ps_2$ *hd* $?ps_2 = (1,\ 1)$ $?ps_2 \neq []$ *last* $?ps_2 = (2,4)$
using *kp-5xm-lr* **by** *auto*

then have *1*: *knights-path* (*board 5 m*) *?ps*$_2$*'*
  using *m-ge ps*$_1$*-IH knights-path-lr-concat*[*of 5 m*−*5 ps*$_1$ *5 ?ps*$_2$] **by** *auto*

have *2*: *hd ?ps*$_2$*'* = (*1*,*1*) **using** *ps*$_1$*-IH* **by** *auto*

have *last* (*trans-path* (*0*,*int* (*m*−*5*)) *?ps*$_2$) = (*2*,*int m*−*1*)
  using *m-ge last-trans-path*[*OF* ‹*?ps*$_2$ ≠ []› ‹*last ?ps*$_2$ = (*2*,*4*)›] **by** *auto*
then have *3*: *last ?ps*$_2$*'* = (*2*,*int m*−*1*)
  using *last-appendR*[*OF trans-path-non-nil*[*OF* ‹*?ps*$_2$ ≠ []›],*symmetric*] **by** *metis*

show *?thesis* **using** *1 2 3* **by** *auto*
qed
qed

For every $5 \times m$-board with $m \geq 5$ there exists a knight's path that starts in $(1,1)$ (bottom-left) and ends in $(4,m-1)$ (top-right).

lemma *knights-path-5xm-ur-exists*:
  assumes $m \geq 5$
  shows ∃ *ps*. *knights-path* (*board 5 m*) *ps* ∧ *hd ps* = (*1*,*1*) ∧ *last ps* = (*4*,*int m*−*1*)
  using *assms*
proof −
  have *m* ∈ {*5*,*6*,*7*,*8*,*9*} ∨ *5* ≤ *m*−*5* **using** *assms* **by** *auto*
  then show *?thesis*
  proof (*elim disjE*)
    assume *m* ∈ {*5*,*6*,*7*,*8*,*9*}
    then show *?thesis* **using** *kp-5xm-ur* **by** *fastforce*
  next
    assume *m-ge*: *5* ≤ *m*−*5*
    then obtain *ps*$_1$ **where** *ps-prems*: *knights-path* (*board 5* (*m*−*5*)) *ps*$_1$ *hd ps*$_1$ = (*1*,*1*)
                last ps$_1$ = (*2*,*int* (*m*−*5*)−*1*) *ps*$_1$ ≠ []
      using *knights-path-5xm-lr-exists*[*of* (*m*−*5*)] *knights-path-non-nil* **by** *auto*
    let *?ps*$_2$=*kp5x5ur*
    let *?ps'*=*ps*$_1$ @ *trans-path* (*0*,*int* (*m*−*5*)) *?ps*$_2$
    have *knights-path b5x5 ?ps*$_2$ *hd ?ps*$_2$ = (*1*, *1*) *?ps*$_2$ ≠ []
        last ?ps$_2$ = (*4*,*4*)
      using *kp-5xm-ur* **by** *auto*
    then have *1*: *knights-path* (*board 5 m*) *?ps'*
      using *m-ge ps-prems knights-path-lr-concat*[*of 5 m*−*5 ps*$_1$ *5 ?ps*$_2$] **by** *auto*

    have *2*: *hd ?ps'* = (*1*,*1*) **using** *ps-prems* **by** *auto*

    have *last* (*trans-path* (*0*,*int* (*m*−*5*)) *?ps*$_2$) = (*4*,*int m*−*1*)
      using *m-ge last-trans-path*[*OF* ‹*?ps*$_2$ ≠ []› ‹*last ?ps*$_2$ = (*4*,*4*)›] **by** *auto*
    then have *3*: *last ?ps'* = (*4*,*int m*−*1*)
      using *last-appendR*[*OF trans-path-non-nil*[*OF* ‹*?ps*$_2$ ≠ []›],*symmetric*] **by** *metis*

show *?thesis* **using** *1 2 3* **by** *auto*
qed
qed

$5 \le ?m \Longrightarrow \exists ps.\ knights\text{-}path\ (board\ 5\ ?m)\ ps \wedge hd\ ps = (1,\ 1) \wedge last\ ps = (2,\ int\ ?m - 1)$ and $5 \le ?m \Longrightarrow \exists ps.\ knights\text{-}path\ (board\ 5\ ?m)\ ps \wedge hd\ ps = (1,\ 1) \wedge last\ ps = (2,\ int\ ?m - 1)$ formalize Lemma 1 from [1].

lemmas *knights-path-5xm-exists* = *knights-path-5xm-lr-exists knights-path-5xm-ur-exists*

9 Knight's Paths and Circuits for $6\times m$-Boards

abbreviation *b6x5* ≡ *board 6 5*

A Knight's path for the (6×5)-board that starts in the lower-left and ends in the upper-left.

10	19	4	29	12
3	30	11	20	5
18	9	24	13	28
25	2	17	6	21
16	23	8	27	14
1	26	15	22	7

abbreviation *kp6x5ul* ≡ *the* (*to-path*
[[*10*,*19*,*4*,*29*,*12*],
[*3*,*30*,*11*,*20*,*5*],
[*18*,*9*,*24*,*13*,*28*],
[*25*,*2*,*17*,*6*,*21*],
[*16*,*23*,*8*,*27*,*14*],
[*1*,*26*,*15*,*22*,*7*]])
lemma *kp-6x5-ul*: *knights-path b6x5 kp6x5ul*
by (*simp only*: *knights-path-exec-simp*) *eval*

lemma *kp-6x5-ul-hd*: *hd kp6x5ul* = (*1*,*1*) **by** *eval*

lemma *kp-6x5-ul-last*: *last kp6x5ul* = (*5*,*2*) **by** *eval*

lemma *kp-6x5-ul-non-nil*: *kp6x5ul* ≠ [] **by** *eval*

A Knight's circuit for the (6×5)-board.

16	9	6	27	18
7	26	17	14	5
10	15	8	19	28
25	30	23	4	13
22	11	2	29	20
1	24	21	12	3

abbreviation *kc6x5* ≡ *the* (*to-path*
[[*16*,*9*,*6*,*27*,*18*],
[*7*,*26*,*17*,*14*,*5*],
[*10*,*15*,*8*,*19*,*28*],
[*25*,*30*,*23*,*4*,*13*],
[*22*,*11*,*2*,*29*,*20*],
[*1*,*24*,*21*,*12*,*3*]])
lemma *kc-6x5*: *knights-circuit b6x5 kc6x5*
by (*simp only*: *knights-circuit-exec-simp*) *eval*

lemma *kc-6x5-hd*: *hd kc6x5* = (*1*,*1*) **by** *eval*

lemma *kc-6x5-non-nil*: *kc6x5* ≠ [] **by** *eval*

abbreviation *b6x6* ≡ *board 6 6*

The path given for the 6×6-board that ends in the upper-left is wrong. The Knight cannot move from square 26 to square 27.

14	23	6	28	12	21
7	36	13	22	5	27
24	15	29	35	20	11
30	8	17	26	34	4
16	25	2	32	10	19
1	31	9	18	3	33

abbreviation *kp6x6ul-false* ≡ *the* (*to-path*
[[*14*,*23*,*6*,*28*,*12*,*21*],
[*7*,*36*,*13*,*22*,*5*,*27*],
[*24*,*15*,*29*,*35*,*20*,*11*],
[*30*,*8*,*17*,*26*,*34*,*4*],
[*16*,*25*,*2*,*32*,*10*,*19*],
[*1*,*31*,*9*,*18*,*3*,*33*]])

lemma ¬*knights-path b6x6 kp6x6ul-false*
by (*simp only*: *knights-path-exec-simp*) *eval*

I have computed a correct Knight's path for the 6×6-board that ends in the upper-left. A Knight's path for the (6×6)-board that starts in the lower-left and ends in the upper-left.

8	25	10	21	6	23
11	36	7	24	33	20
26	9	34	3	22	5
35	12	15	30	19	32
14	27	2	17	4	29
1	16	13	28	31	18

abbreviation *kp6x6ul* ≡ *the* (*to-path*
[[*8*,*25*,*10*,*21*,*6*,*23*],
[*11*,*36*,*7*,*24*,*33*,*20*],
[*26*,*9*,*34*,*3*,*22*,*5*],
[*35*,*12*,*15*,*30*,*19*,*32*],
[*14*,*27*,*2*,*17*,*4*,*29*],
[*1*,*16*,*13*,*28*,*31*,*18*]])
lemma *kp-6x6-ul*: *knights-path b6x6 kp6x6ul*
by (*simp only*: *knights-path-exec-simp*) *eval*

lemma *kp-6x6-ul-hd*: *hd kp6x6ul* = (*1*,*1*) **by** *eval*

lemma *kp-6x6-ul-last*: *last kp6x6ul* = (*5*,*2*) **by** *eval*

lemma *kp-6x6-ul-non-nil*: *kp6x6ul* ≠ [] **by** *eval*

A Knight's circuit for the (6×6)-board.

4	25	34	15	18	7
35	14	5	8	33	16
24	3	26	17	6	19
13	36	23	30	9	32
22	27	2	11	20	29
1	12	21	28	31	10

abbreviation *kc6x6* ≡ *the* (*to-path*
[[*4*,*25*,*34*,*15*,*18*,*7*],
[*35*,*14*,*5*,*8*,*33*,*16*],
[*24*,*3*,*26*,*17*,*6*,*19*],
[*13*,*36*,*23*,*30*,*9*,*32*],
[*22*,*27*,*2*,*11*,*20*,*29*],
[*1*,*12*,*21*,*28*,*31*,*10*]])
lemma *kc-6x6*: *knights-circuit b6x6 kc6x6*
by (*simp only*: *knights-circuit-exec-simp*) *eval*

lemma *kc-6x6-hd*: *hd kc6x6* = (*1*,*1*) **by** *eval*

lemma *kc-6x6-non-nil*: *kc6x6* ≠ [] **by** *eval*

abbreviation *b6x7* ≡ *board 6 7*

A Knight's path for the (6×7)-board that starts in the lower-left and ends in the upper-left.

18	23	8	39	16	25	6
9	42	17	24	7	40	15
22	19	32	41	38	5	26
33	10	21	28	31	14	37
20	29	2	35	12	27	4
1	34	11	30	3	36	13

abbreviation *kp6x7ul* ≡ *the* (*to-path*
[[*18*,*23*,*8*,*39*,*16*,*25*,*6*],
[*9*,*42*,*17*,*24*,*7*,*40*,*15*],
[*22*,*19*,*32*,*41*,*38*,*5*,*26*],
[*33*,*10*,*21*,*28*,*31*,*14*,*37*],
[*20*,*29*,*2*,*35*,*12*,*27*,*4*],
[*1*,*34*,*11*,*30*,*3*,*36*,*13*]])
lemma *kp-6x7-ul*: *knights-path b6x7 kp6x7ul*
by (*simp only*: *knights-path-exec-simp*) *eval*

lemma *kp-6x7-ul-hd*: *hd kp6x7ul* = (*1*,*1*) **by** *eval*

lemma *kp-6x7-ul-last*: *last kp6x7ul* = (*5*,*2*) **by** *eval*

lemma *kp-6x7-ul-non-nil*: *kp6x7ul* ≠ [] **by** *eval*

A Knight's circuit for the (*6*×*7*)-board.

26	37	8	17	28	31	6
9	18	27	36	7	16	29
38	25	10	19	30	5	32
11	42	23	40	35	20	15
24	39	2	13	22	33	4
1	12	41	34	3	14	21

abbreviation *kc6x7* ≡ *the* (*to-path*
[[*26*,*37*,*8*,*17*,*28*,*31*,*6*],
[*9*,*18*,*27*,*36*,*7*,*16*,*29*],
[*38*,*25*,*10*,*19*,*30*,*5*,*32*],
[*11*,*42*,*23*,*40*,*35*,*20*,*15*],
[*24*,*39*,*2*,*13*,*22*,*33*,*4*],
[*1*,*12*,*41*,*34*,*3*,*14*,*21*]])
lemma *kc-6x7*: *knights-circuit b6x7 kc6x7*
by (*simp only*: *knights-circuit-exec-simp*) *eval*

lemma *kc-6x7-hd*: *hd kc6x7* = (*1*,*1*) **by** *eval*

lemma *kc-6x7-non-nil*: *kc6x7* ≠ [] **by** *eval*

abbreviation *b6x8* ≡ *board 6 8*

A Knight's path for the (6×8)-board that starts in the lower-left and ends in the upper-left.

18	31	8	35	16	33	6	45
9	48	17	32	7	46	15	26
30	19	36	47	34	27	44	5
37	10	21	28	43	40	25	14
20	29	2	39	12	23	4	41
1	38	11	22	3	42	13	24

abbreviation *kp6x8ul* ≡ *the* (*to-path*
  [[*18*,*31*,*8*,*35*,*16*,*33*,*6*,*45*],
  [*9*,*48*,*17*,*32*,*7*,*46*,*15*,*26*],
  [*30*,*19*,*36*,*47*,*34*,*27*,*44*,*5*],
  [*37*,*10*,*21*,*28*,*43*,*40*,*25*,*14*],
  [*20*,*29*,*2*,*39*,*12*,*23*,*4*,*41*],
  [*1*,*38*,*11*,*22*,*3*,*42*,*13*,*24*]])
lemma *kp-6x8-ul*: *knights-path b6x8 kp6x8ul*
  by (*simp only*: *knights-path-exec-simp*) *eval*

lemma *kp-6x8-ul-hd*: *hd kp6x8ul* = (*1*,*1*) **by** *eval*

lemma *kp-6x8-ul-last*: *last kp6x8ul* = (*5*,*2*) **by** *eval*

lemma *kp-6x8-ul-non-nil*: *kp6x8ul* ≠ [] **by** *eval*

A Knight's circuit for the (6×8)-board.

30	35	8	15	28	39	6	13
9	16	29	36	7	14	27	38
34	31	10	23	40	37	12	5
17	48	33	46	11	22	41	26
32	45	2	19	24	43	4	21
1	18	47	44	3	20	25	42

abbreviation *kc6x8* ≡ *the* (*to-path*
  [[*30*,*35*,*8*,*15*,*28*,*39*,*6*,*13*],
  [*9*,*16*,*29*,*36*,*7*,*14*,*27*,*38*],
  [*34*,*31*,*10*,*23*,*40*,*37*,*12*,*5*],
  [*17*,*48*,*33*,*46*,*11*,*22*,*41*,*26*],
  [*32*,*45*,*2*,*19*,*24*,*43*,*4*,*21*],
  [*1*,*18*,*47*,*44*,*3*,*20*,*25*,*42*]])
lemma *kc-6x8*: *knights-circuit b6x8 kc6x8*
  by (*simp only*: *knights-circuit-exec-simp*) *eval*

lemma *kc-6x8-hd*: *hd kc6x8* = (*1*,*1*) **by** *eval*

lemma *kc-6x8-non-nil*: *kc6x8* ≠ [] **by** *eval*

abbreviation *b6x9* ≡ *board 6 9*

A Knight's path for the (*6*×*9*)-board that starts in the lower-left and ends in the upper-left.

22	45	10	53	20	47	8	35	18
11	54	21	46	9	36	19	48	7
44	23	42	37	52	49	32	17	34
41	12	25	50	27	38	29	6	31
24	43	2	39	14	51	4	33	16
1	40	13	26	3	28	15	30	5

abbreviation *kp6x9ul* ≡ *the* (*to-path*
  [[*22*,*45*,*10*,*53*,*20*,*47*,*8*,*35*,*18*],
  [*11*,*54*,*21*,*46*,*9*,*36*,*19*,*48*,*7*],
  [*44*,*23*,*42*,*37*,*52*,*49*,*32*,*17*,*34*],
  [*41*,*12*,*25*,*50*,*27*,*38*,*29*,*6*,*31*],
  [*24*,*43*,*2*,*39*,*14*,*51*,*4*,*33*,*16*],
  [*1*,*40*,*13*,*26*,*3*,*28*,*15*,*30*,*5*]])
lemma *kp-6x9-ul*: *knights-path b6x9 kp6x9ul*
  by (*simp only*: *knights-path-exec-simp*) *eval*

lemma *kp-6x9-ul-hd*: *hd kp6x9ul* = (*1*,*1*) **by** *eval*

lemma *kp-6x9-ul-last*: *last kp6x9ul* = (*5*,*2*) **by** *eval*

lemma *kp-6x9-ul-non-nil*: *kp6x9ul* ≠ [] **by** *eval*

A Knight's circuit for the (*6*×*9*)-board.

14	49	4	51	24	39	6	29	22
3	52	13	40	5	32	23	42	7
48	15	50	25	38	41	28	21	30
53	2	37	12	33	26	31	8	43
16	47	54	35	18	45	10	27	20
1	36	17	46	11	34	19	44	9

abbreviation *kc6x9* ≡ *the* (*to-path*
  [[*14*,*49*,*4*,*51*,*24*,*39*,*6*,*29*,*22*],
  [*3*,*52*,*13*,*40*,*5*,*32*,*23*,*42*,*7*],
  [*48*,*15*,*50*,*25*,*38*,*41*,*28*,*21*,*30*],

[53,2,37,12,33,26,31,8,43],
[16,47,54,35,18,45,10,27,20],
[1,36,17,46,11,34,19,44,9]])

lemma *kc-6x9*: *knights-circuit b6x9 kc6x9*
by (*simp only*: *knights-circuit-exec-simp*) *eval*

lemma *kc-6x9-hd*: *hd kc6x9* = (*1*,*1*) **by** *eval*

lemma *kc-6x9-non-nil*: *kc6x9* ≠ [] **by** *eval*

lemmas *kp-6xm-ul* =
kp-6x5-ul kp-6x5-ul-hd kp-6x5-ul-last kp-6x5-ul-non-nil
kp-6x6-ul kp-6x6-ul-hd kp-6x6-ul-last kp-6x6-ul-non-nil
kp-6x7-ul kp-6x7-ul-hd kp-6x7-ul-last kp-6x7-ul-non-nil
kp-6x8-ul kp-6x8-ul-hd kp-6x8-ul-last kp-6x8-ul-non-nil
kp-6x9-ul kp-6x9-ul-hd kp-6x9-ul-last kp-6x9-ul-non-nil

lemmas *kc-6xm* =
kc-6x5 kc-6x5-hd kc-6x5-non-nil
kc-6x6 kc-6x6-hd kc-6x6-non-nil
kc-6x7 kc-6x7-hd kc-6x7-non-nil
kc-6x8 kc-6x8-hd kc-6x8-non-nil
kc-6x9 kc-6x9-hd kc-6x9-non-nil

For every $6 \times m$-board with $m \geq 5$ there exists a knight's path that starts in $(1,1)$ (bottom-left) and ends in $(5,2)$ (top-left).

lemma *knights-path-6xm-ul-exists*:
assumes $m \geq 5$
shows ∃ *ps*. *knights-path* (*board 6 m*) *ps* ∧ *hd ps* = (*1*,*1*) ∧ *last ps* = (*5*,*2*)
using *assms*
proof (*induction m rule*: *less-induct*)
case (*less m*)
then have *m* ∈ {*5*,*6*,*7*,*8*,*9*} ∨ *5* ≤ *m*−*5* **by** *auto*
then show *?case*
proof (*elim disjE*)
assume *m* ∈ {*5*,*6*,*7*,*8*,*9*}
then show *?thesis* **using** *kp-6xm-ul* **by** *fastforce*
next
let $?ps_1$=*kp6x5ul*
let $?b_1$=*board 6 5*
have ps_1-*prems*: *knights-path* $?b_1$ $?ps_1$ *hd* $?ps_1$ = (*1*,*1*) *last* $?ps_1$ = (*5*,*2*)
using *kp-6xm-ul* **by** *auto*
assume *m-ge*: *5* ≤ *m*−*5*
then obtain ps_2 **where** ps_2-*IH*: *knights-path* (*board 6* (*m*−*5*)) ps_2 *hd* ps_2 = (*1*,*1*)
last ps_2 = (*5*,*2*)
using *less.IH*[*of m*−*5*] *knights-path-non-nil* **by** *auto*

have *27* < *length* $?ps_1$ *last* (*take 27* $?ps_1$) = (*2*,*4*) *hd* (*drop 27* $?ps_1$) = (*4*,*5*)

by *eval+*
  then have *step-in* $?ps_1$ $(2,4)$ $(4,5)$
    unfolding *step-in-def* **using** *zero-less-numeral* **by** *blast*
  then have *step-in* $?ps_1$ $(2,4)$ $(4,5)$
      valid-step $(2,4)$ $(1,int\ 5+1)$
      valid-step $(5,int\ 5+2)$ $(4,5)$
    unfolding *valid-step-def* **by** *auto*
  then show *?thesis*
   using ‹$5 \leq m-5$› ps_1*-prems* ps_2*-IH* *knights-path-split-concat*[*of* 6 5 $?ps_1$ $m-5$ ps_2] **by** *auto*
 qed
qed

For every $6 \times m$-board with $m \geq 5$ there exists a knight's circuit.

lemma *knights-circuit-6xm-exists*:
 assumes $m \geq 5$
 shows $\exists ps.$ *knights-circuit* (*board* 6 *m*) *ps*
 using *assms*
proof −
 have $m \in \{5,6,7,8,9\} \vee 5 \leq m-5$ **using** *assms* **by** *auto*
 then show *?thesis*
 proof (*elim disjE*)
  assume $m \in \{5,6,7,8,9\}$
  then show *?thesis* **using** *kc-6xm* **by** *fastforce*
 next
  let $?ps_1$=*rev kc6x5*
  have *knights-circuit b6x5* $?ps_1$ *last* $?ps_1 = (1,1)$
    using *kc-6xm knights-circuit-rev* **by** (*auto simp*: *last-rev*)
  then have ps_1*-prems*: *knights-path b6x5* $?ps_1$ *valid-step* (*last* $?ps_1$) (*hd* $?ps_1$)
    unfolding *knights-circuit-def* **using** *valid-step-rev* **by** *auto*
  assume *m-ge*: $5 \leq m-5$
  then obtain ps_2 **where** *ps2-prems*: *knights-path* (*board* 6 $(m-5)$) ps_2 *hd* $ps_2 = (1,1)$
          last $ps_2 = (5,2)$
    using *knights-path-6xm-ul-exists*[*of* $(m-5)$] *knights-path-non-nil* **by** *auto*

  have $2 <$ *length* $?ps_1$ *last* (*take* 2 $?ps_1$) $= (2,4)$ *hd* (*drop* 2 $?ps_1$) $= (4,5)$ **by** *eval+*
  then have *step-in* $?ps_1$ $(2,4)$ $(4,5)$
    unfolding *step-in-def* **using** *zero-less-numeral* **by** *blast*
  then have *step-in* $?ps_1$ $(2,4)$ $(4,5)$
      valid-step $(2,4)$ $(1,int\ 5+1)$
      valid-step $(5,int\ 5+2)$ $(4,5)$
    unfolding *valid-step-def* **by** *auto*
  then have $\exists ps.$ *knights-path* (*board* 6 *m*) *ps* $\wedge$ *hd ps* = *hd* $?ps_1$ $\wedge$ *last ps* = *last* $?ps_1$
    using *m-ge* ps_1*-prems* *ps2-prems* *knights-path-split-concat*[*of* 6 5 $?ps_1$ $m-5$ ps_2] **by** *auto*
  then show *?thesis* **using** ps_1*-prems* **by** (*auto simp*: *knights-circuit-def*)

qed
qed

$5 \leq ?m \Longrightarrow \exists ps.\ knights\text{-}path\ (board\ 6\ ?m)\ ps \wedge hd\ ps = (1,\ 1) \wedge last\ ps = (5,\ 2)$ and $5 \leq ?m \Longrightarrow \exists ps.\ knights\text{-}circuit\ (board\ 6\ ?m)\ ps$ formalize Lemma 2 from [1].

lemmas *knights-path-6xm-exists* = *knights-path-6xm-ul-exists knights-circuit-6xm-exists*

10 Knight's Paths and Circuits for $8 \times m$-Boards

abbreviation *b8x5* ≡ *board 8 5*

A Knight's path for the (8×5)-board that starts in the lower-left and ends in the upper-left.

28	7	22	39	26
23	40	27	6	21
8	29	38	25	14
37	24	15	20	5
16	9	30	13	34
31	36	33	4	19
10	17	2	35	12
1	32	11	18	3

abbreviation *kp8x5ul* ≡ *the* (*to-path*
  [[*28*,*7*,*22*,*39*,*26*],
  [*23*,*40*,*27*,*6*,*21*],
  [*8*,*29*,*38*,*25*,*14*],
  [*37*,*24*,*15*,*20*,*5*],
  [*16*,*9*,*30*,*13*,*34*],
  [*31*,*36*,*33*,*4*,*19*],
  [*10*,*17*,*2*,*35*,*12*],
  [*1*,*32*,*11*,*18*,*3*]])
lemma *kp-8x5-ul*: *knights-path b8x5 kp8x5ul*
  by (*simp only*: *knights-path-exec-simp*) *eval*

lemma *kp-8x5-ul-hd*: *hd kp8x5ul* = (*1*,*1*) **by** *eval*

lemma *kp-8x5-ul-last*: *last kp8x5ul* = (*7*,*2*) **by** *eval*

lemma *kp-8x5-ul-non-nil*: *kp8x5ul* ≠ [] **by** *eval*

A Knight's circuit for the (8×5)-board.

26	7	28	15	24
31	16	25	6	29
8	27	30	23	14
17	32	39	34	5
38	9	18	13	22
19	40	33	4	35
10	37	2	21	12
1	20	11	36	3

abbreviation *kc8x5* ≡ *the* (*to-path*
  [[*26*,*7*,*28*,*15*,*24*],
  [*31*,*16*,*25*,*6*,*29*],
  [*8*,*27*,*30*,*23*,*14*],
  [*17*,*32*,*39*,*34*,*5*],
  [*38*,*9*,*18*,*13*,*22*],
  [*19*,*40*,*33*,*4*,*35*],
  [*10*,*37*,*2*,*21*,*12*],
  [*1*,*20*,*11*,*36*,*3*]])
lemma *kc-8x5*: *knights-circuit b8x5 kc8x5*
  by (*simp only*: *knights-circuit-exec-simp*) *eval*

lemma *kc-8x5-hd*: *hd kc8x5* = (*1*,*1*) **by** *eval*

lemma *kc-8x5-last*: *last kc8x5* = (*3*,*2*) **by** *eval*

lemma *kc-8x5-non-nil*: *kc8x5* ≠ [] **by** *eval*

lemma *kc-8x5-si*: *step-in kc8x5* (*2*,*4*) (*4*,*5*) (**is** *step-in ?ps - -*)
proof −
  have *0* < (*21*::*nat*) *21* < *length ?ps last* (*take 21 ?ps*) = (*2*,*4*) *hd* (*drop 21 ?ps*) = (*4*,*5*)
    by *eval*+
  then show *?thesis* **unfolding** *step-in-def* **by** *blast*
qed

abbreviation *b8x6* ≡ *board 8 6*

A Knight's path for the (*8*×*6*)-board that starts in the lower-left and ends in the upper-left.

42	11	26	9	34	13
25	48	43	12	27	8
44	41	10	33	14	35
47	24	45	20	7	28
40	19	32	3	36	15
23	46	21	6	29	4
18	39	2	31	16	37
1	22	17	38	5	30

abbreviation *kp8x6ul* ≡ *the* (*to-path*
  [[*42*,*11*,*26*,*9*,*34*,*13*],
  [*25*,*48*,*43*,*12*,*27*,*8*],
  [*44*,*41*,*10*,*33*,*14*,*35*],
  [*47*,*24*,*45*,*20*,*7*,*28*],
  [*40*,*19*,*32*,*3*,*36*,*15*],
  [*23*,*46*,*21*,*6*,*29*,*4*],
  [*18*,*39*,*2*,*31*,*16*,*37*],
  [*1*,*22*,*17*,*38*,*5*,*30*]])
lemma *kp-8x6-ul*: *knights-path b8x6 kp8x6ul*
  by (*simp only*: *knights-path-exec-simp*) *eval*

lemma *kp-8x6-ul-hd*: *hd kp8x6ul* = (*1*,*1*) **by** *eval*

lemma *kp-8x6-ul-last*: *last kp8x6ul* = (*7*,*2*) **by** *eval*

lemma *kp-8x6-ul-non-nil*: *kp8x6ul* ≠ [] **by** *eval*

A Knight's circuit for the (*8*×*6*)-board. I have reversed circuit s.t. the circuit steps from (*2*,*5*) to (*4*,*6*) and not the other way around. This makes the proofs easier.

8	29	24	45	12	37
25	46	9	38	23	44
30	7	28	13	36	11
47	26	39	10	43	22
6	31	4	27	14	35
3	48	17	40	21	42
32	5	2	19	34	15
1	18	33	16	41	20

abbreviation *kc8x6* ≡ *the* (*to-path*
  [[*8*,*29*,*24*,*45*,*12*,*37*],
  [*25*,*46*,*9*,*38*,*23*,*44*],
  [*30*,*7*,*28*,*13*,*36*,*11*],
  [*47*,*26*,*39*,*10*,*43*,*22*],
  [*6*,*31*,*4*,*27*,*14*,*35*],

[*3*,*48*,*17*,*40*,*21*,*42*],
[*32*,*5*,*2*,*19*,*34*,*15*],
[*1*,*18*,*33*,*16*,*41*,*20*]])
lemma *kc-8x6*: *knights-circuit b8x6 kc8x6*
by (*simp only*: *knights-circuit-exec-simp*) *eval*

lemma *kc-8x6-hd*: *hd kc8x6* = (*1*,*1*) **by** *eval*

lemma *kc-8x6-non-nil*: *kc8x6* ≠ [] **by** *eval*

lemma *kc-8x6-si*: *step-in kc8x6* (*2*,*5*) (*4*,*6*) (**is** *step-in ?ps* - -)
proof −
have *0* < (*34*::*nat*) *34* < *length ?ps*
last (*take 34 ?ps*) = (*2*,*5*) *hd* (*drop 34 ?ps*) = (*4*,*6*) **by** *eval*+
then show *?thesis* **unfolding** *step-in-def* **by** *blast*
qed

abbreviation *b8x7* ≡ *board 8 7*

A Knight's path for the (*8*×*7*)-board that starts in the lower-left and ends in the upper-left.

38	19	6	55	46	21	8
5	56	39	20	7	54	45
18	37	4	47	34	9	22
3	48	35	40	53	44	33
36	17	52	49	32	23	10
51	2	29	14	41	26	43
16	13	50	31	28	11	24
1	30	15	12	25	42	27

abbreviation *kp8x7ul* ≡ *the* (*to-path*
[[*38*,*19*,*6*,*55*,*46*,*21*,*8*],
[*5*,*56*,*39*,*20*,*7*,*54*,*45*],
[*18*,*37*,*4*,*47*,*34*,*9*,*22*],
[*3*,*48*,*35*,*40*,*53*,*44*,*33*],
[*36*,*17*,*52*,*49*,*32*,*23*,*10*],
[*51*,*2*,*29*,*14*,*41*,*26*,*43*],
[*16*,*13*,*50*,*31*,*28*,*11*,*24*],
[*1*,*30*,*15*,*12*,*25*,*42*,*27*]])
lemma *kp-8x7-ul*: *knights-path b8x7 kp8x7ul*
by (*simp only*: *knights-path-exec-simp*) *eval*

lemma *kp-8x7-ul-hd*: *hd kp8x7ul* = (*1*,*1*) **by** *eval*

lemma *kp-8x7-ul-last*: *last kp8x7ul* = (*7*,*2*) **by** *eval*

lemma *kp-8x7-ul-non-nil*: *kp8x7ul* ≠ [] **by** *eval*

A Knight's circuit for the (8×7)-board. I have reversed circuit s.t. the circuit steps from $(2,6)$ to $(4,7)$ and not the other way around. This makes the proofs easier.

36	31	18	53	20	29	44
17	54	35	30	45	52	21
32	37	46	19	8	43	28
55	16	7	34	27	22	51
38	33	26	47	6	9	42
3	56	15	12	25	50	23
14	39	2	5	48	41	10
1	4	13	40	11	24	49

abbreviation *kc8x7* ≡ *the* (*to-path*
  [[*36*,*31*,*18*,*53*,*20*,*29*,*44*],
  [*17*,*54*,*35*,*30*,*45*,*52*,*21*],
  [*32*,*37*,*46*,*19*,*8*,*43*,*28*],
  [*55*,*16*,*7*,*34*,*27*,*22*,*51*],
  [*38*,*33*,*26*,*47*,*6*,*9*,*42*],
  [*3*,*56*,*15*,*12*,*25*,*50*,*23*],
  [*14*,*39*,*2*,*5*,*48*,*41*,*10*],
  [*1*,*4*,*13*,*40*,*11*,*24*,*49*]])
lemma *kc-8x7*: *knights-circuit b8x7 kc8x7*
  by (*simp only*: *knights-circuit-exec-simp*) *eval*

lemma *kc-8x7-hd*: *hd kc8x7* = (*1*,*1*) **by** *eval*

lemma *kc-8x7-non-nil*: *kc8x7* ≠ [] **by** *eval*

lemma *kc-8x7-si*: *step-in kc8x7* (*2*,*6*) (*4*,*7*) (**is** *step-in ?ps - -*)
proof −
  have *0* < (*41*::*nat*) *41* < *length ?ps*
    last (*take 41 ?ps*) = (*2*,*6*) *hd* (*drop 41 ?ps*) = (*4*,*7*) **by** *eval+*
  then show *?thesis* **unfolding** *step-in-def* **by** *blast*
qed

abbreviation *b8x8* ≡ *board 8 8*

The path given for the 8×8-board that ends in the upper-left is wrong. The Knight cannot move from square 27 to square 28.

24	11	37	9	26	21	39	7
36	64	24	22	38	8	27	20
12	23	10	53	58	49	6	28
63	35	61	50	55	52	19	40
46	13	54	57	48	59	29	5
34	62	47	60	51	56	41	18
14	45	2	32	16	43	4	30
1	33	15	44	3	31	17	42

abbreviation *kp8x8ul-false* ≡ *the* (*to-path*
  [[*24*,*11*,*37*,*9*,*26*,*21*,*39*,*7*],
  [*36*,*64*,*25*,*22*,*38*,*8*,*27*,*20*],
  [*12*,*23*,*10*,*53*,*58*,*49*,*6*,*28*],
  [*63*,*35*,*61*,*50*,*55*,*52*,*19*,*40*],
  [*46*,*13*,*54*,*57*,*48*,*59*,*29*,*5*],
  [*34*,*62*,*47*,*60*,*51*,*56*,*41*,*18*],
  [*14*,*45*,*2*,*32*,*16*,*43*,*4*,*30*],
  [*1*,*33*,*15*,*44*,*3*,*31*,*17*,*42*]])

lemma ¬*knights-path b8x8 kp8x8ul-false*
  by (*simp only*: *knights-path-exec-simp*) *eval*

I have computed a correct Knight's path for the $8{\times}8$-board that ends in the upper-left.

38	41	36	27	32	43	20	25
35	64	39	42	21	26	29	44
40	37	6	33	28	31	24	19
5	34	63	14	7	22	45	30
62	13	4	9	58	49	18	23
3	10	61	52	15	8	57	46
12	53	2	59	48	55	50	17
1	60	11	54	51	16	47	56

abbreviation *kp8x8ul* ≡ *the* (*to-path*
  [[*38*,*41*,*36*,*27*,*32*,*43*,*20*,*25*],
  [*35*,*64*,*39*,*42*,*21*,*26*,*29*,*44*],
  [*40*,*37*,*6*,*33*,*28*,*31*,*24*,*19*],
  [*5*,*34*,*63*,*14*,*7*,*22*,*45*,*30*],
  [*62*,*13*,*4*,*9*,*58*,*49*,*18*,*23*],
  [*3*,*10*,*61*,*52*,*15*,*8*,*57*,*46*],
  [*12*,*53*,*2*,*59*,*48*,*55*,*50*,*17*],
  [*1*,*60*,*11*,*54*,*51*,*16*,*47*,*56*]])

lemma *kp-8x8-ul*: *knights-path b8x8 kp8x8ul*
  by (*simp only*: *knights-path-exec-simp*) *eval*

lemma *kp-8x8-ul-hd*: *hd kp8x8ul* = (*1*,*1*) **by** *eval*

lemma *kp-8x8-ul-last*: *last kp8x8ul* = (*7*,*2*) **by** *eval*

lemma *kp-8x8-ul-non-nil*: *kp8x8ul* ≠ [] **by** *eval*

A Knight's circuit for the (*8*×*8*)-board.

48	13	30	9	56	45	28	7
31	10	47	50	29	8	57	44
14	49	12	55	46	59	6	27
11	32	37	60	51	54	43	58
36	15	52	63	38	61	26	5
33	64	35	18	53	40	23	42
16	19	2	39	62	21	4	25
1	34	17	20	3	24	41	22

abbreviation *kc8x8* ≡ *the* (*to-path*
  [[*48*,*13*,*30*,*9*,*56*,*45*,*28*,*7*],
  [*31*,*10*,*47*,*50*,*29*,*8*,*57*,*44*],
  [*14*,*49*,*12*,*55*,*46*,*59*,*6*,*27*],
  [*11*,*32*,*37*,*60*,*51*,*54*,*43*,*58*],
  [*36*,*15*,*52*,*63*,*38*,*61*,*26*,*5*],
  [*33*,*64*,*35*,*18*,*53*,*40*,*23*,*42*],
  [*16*,*19*,*2*,*39*,*62*,*21*,*4*,*25*],
  [*1*,*34*,*17*,*20*,*3*,*24*,*41*,*22*]])
lemma *kc-8x8*: *knights-circuit b8x8 kc8x8*
  by (*simp only*: *knights-circuit-exec-simp*) *eval*

lemma *kc-8x8-hd*: *hd kc8x8* = (*1*,*1*) **by** *eval*

lemma *kc-8x8-non-nil*: *kc8x8* ≠ [] **by** *eval*

lemma *kc-8x8-si*: *step-in kc8x8* (*2*,*7*) (*4*,*8*) (**is** *step-in ?ps* - -)
proof −
  have *0* < (*4*::*nat*) *4* < *length ?ps*
      last (*take 4 ?ps*) = (*2*,*7*) *hd* (*drop 4 ?ps*) = (*4*,*8*) **by** *eval*+
  then show *?thesis* **unfolding** *step-in-def* **by** *blast*
qed

abbreviation *b8x9* ≡ *board 8 9*

A Knight's path for the (*8*×*9*)-board that starts in the lower-left and ends in the upper-left.

32	47	6	71	30	45	8	43	26
5	72	31	46	7	70	27	22	9
48	33	4	29	64	23	44	25	42
3	60	35	62	69	28	41	10	21
34	49	68	65	36	63	24	55	40
59	2	61	16	67	56	37	20	11
50	15	66	57	52	13	18	39	54
1	58	51	14	17	38	53	12	19

abbreviation *kp8x9ul* ≡ *the* (*to-path*
[[*32*,*47*,*6*,*71*,*30*,*45*,*8*,*43*,*26*],
[*5*,*72*,*31*,*46*,*7*,*70*,*27*,*22*,*9*],
[*48*,*33*,*4*,*29*,*64*,*23*,*44*,*25*,*42*],
[*3*,*60*,*35*,*62*,*69*,*28*,*41*,*10*,*21*],
[*34*,*49*,*68*,*65*,*36*,*63*,*24*,*55*,*40*],
[*59*,*2*,*61*,*16*,*67*,*56*,*37*,*20*,*11*],
[*50*,*15*,*66*,*57*,*52*,*13*,*18*,*39*,*54*],
[*1*,*58*,*51*,*14*,*17*,*38*,*53*,*12*,*19*]])
lemma *kp-8x9-ul*: *knights-path b8x9 kp8x9ul*
by (*simp only*: *knights-path-exec-simp*) *eval*

lemma *kp-8x9-ul-hd*: *hd kp8x9ul* = (*1*,*1*) **by** *eval*

lemma *kp-8x9-ul-last*: *last kp8x9ul* = (*7*,*2*) **by** *eval*

lemma *kp-8x9-ul-non-nil*: *kp8x9ul* ≠ [] **by** *eval*

A Knight's circuit for the (*8*×*9*)-board.

42	19	38	5	36	21	34	7	60
39	4	41	20	63	6	59	22	33
18	43	70	37	58	35	68	61	8
3	40	49	64	69	62	57	32	23
50	17	44	71	48	67	54	9	56
45	2	65	14	27	12	29	24	31
16	51	72	47	66	53	26	55	10
1	46	15	52	13	28	11	30	25

abbreviation *kc8x9* ≡ *the* (*to-path*
[[*42*,*19*,*38*,*5*,*36*,*21*,*34*,*7*,*60*],
[*39*,*4*,*41*,*20*,*63*,*6*,*59*,*22*,*33*],
[*18*,*43*,*70*,*37*,*58*,*35*,*68*,*61*,*8*],
[*3*,*40*,*49*,*64*,*69*,*62*,*57*,*32*,*23*],
[*50*,*17*,*44*,*71*,*48*,*67*,*54*,*9*,*56*],
[*45*,*2*,*65*,*14*,*27*,*12*,*29*,*24*,*31*],
[*16*,*51*,*72*,*47*,*66*,*53*,*26*,*55*,*10*],

```
  [1,46,15,52,13,28,11,30,25]])
lemma kc-8x9: knights-circuit b8x9 kc8x9
  by (simp only: knights-circuit-exec-simp) eval

lemma kc-8x9-hd: hd kc8x9 = (1,1) by eval

lemma kc-8x9-non-nil: kc8x9 ≠ [] by eval

lemma kc-8x9-si: step-in kc8x9 (2,8) (4,9) (is step-in ?ps - -)
proof −
  have 0 < (55::nat) 55 < length ?ps
       last (take 55 ?ps) = (2,8) hd (drop 55 ?ps) = (4,9) by eval+
  then show ?thesis unfolding step-in-def by blast
qed

lemmas kp-8xm-ul =
  kp-8x5-ul kp-8x5-ul-hd kp-8x5-ul-last kp-8x5-ul-non-nil
  kp-8x6-ul kp-8x6-ul-hd kp-8x6-ul-last kp-8x6-ul-non-nil
  kp-8x7-ul kp-8x7-ul-hd kp-8x7-ul-last kp-8x7-ul-non-nil
  kp-8x8-ul kp-8x8-ul-hd kp-8x8-ul-last kp-8x8-ul-non-nil
  kp-8x9-ul kp-8x9-ul-hd kp-8x9-ul-last kp-8x9-ul-non-nil

lemmas kc-8xm =
  kc-8x5 kc-8x5-hd kc-8x5-last kc-8x5-non-nil kc-8x5-si
  kc-8x6 kc-8x6-hd kc-8x6-non-nil kc-8x6-si
  kc-8x7 kc-8x7-hd kc-8x7-non-nil kc-8x7-si
  kc-8x8 kc-8x8-hd kc-8x8-non-nil kc-8x8-si
  kc-8x9 kc-8x9-hd kc-8x9-non-nil kc-8x9-si
```

For every $8 \times m$-board with $m \geq 5$ there exists a knight's circuit.

```
lemma knights-circuit-8xm-exists:
  assumes m ≥ 5
  shows ∃ps. knights-circuit (board 8 m) ps ∧ step-in ps (2,int m−1) (4,int m)
  using assms
proof (induction m rule: less-induct)
  case (less m)
  then have m ∈ {5,6,7,8,9} ∨ 5 ≤ m−5 by auto
  then show ?case
  proof (elim disjE)
    assume m ∈ {5,6,7,8,9}
    then show ?thesis using kc-8xm by fastforce
  next
    let ?ps2=kc8x5
    let ?b2=board 8 5
    have ps2-prems: knights-circuit ?b2 ?ps2 hd ?ps2 = (1,1) last ?ps2 = (3,2)
      using kc-8xm by auto
    have 21 < length ?ps2 last (take 21 ?ps2) = (2,int 5−1) hd (drop 21 ?ps2) =
(4,int 5)
      by eval+
```

then have *si*: *step-in ?ps*$_2$ (*2*,*int 5*−*1*) (*4*,*int 5*)
  unfolding *step-in-def* **using** *zero-less-numeral* **by** *blast*
assume *m-ge*: *5* ≤ *m*−*5*
then obtain *ps*$_1$ **where** *ps*$_1$*-IH*: *knights-circuit* (*board 8* (*m*−*5*)) *ps*$_1$
step-in ps$_1$ (*2*,*int* (*m*−*5*)−*1*) (*4*,*int* (*m*−*5*))
  using *less.IH*[*of m*−*5*] *knights-path-non-nil* **by** *auto*
then show *?thesis*
  using *m-ge ps*$_2$*-prems si knights-circuit-lr-concat*[*of 8 m*−*5 ps*$_1$ *5 ?ps*$_2$] **by** *auto*
qed
qed

For every *8*×*m*-board with *m* ≥ *5* there exists a knight's path that starts in (*1*,*1*) (bottom-left) and ends in (*7*,*2*) (top-left).

lemma *knights-path-8xm-ul-exists*:
assumes *m* ≥ *5*
shows ∃*ps*. *knights-path* (*board 8 m*) *ps* ∧ *hd ps* = (*1*,*1*) ∧ *last ps* = (*7*,*2*)
using *assms*
proof −
have *m* ∈ {*5*,*6*,*7*,*8*,*9*} ∨ *5* ≤ *m*−*5* **using** *assms* **by** *auto*
then show *?thesis*
proof (*elim disjE*)
assume *m* ∈ {*5*,*6*,*7*,*8*,*9*}
then show *?thesis* **using** *kp-8xm-ul* **by** *fastforce*
next
let *?ps*$_1$=*kp8x5ul*
have *ps*$_1$*-prems*: *knights-path b8x5 ?ps*$_1$ *hd ?ps*$_1$ = (*1*,*1*) *last ?ps*$_1$ = (*7*,*2*)
  using *kp-8xm-ul* **by** *auto*
assume *m-ge*: *5* ≤ *m*−*5*
then have *b-prems*: *5* ≤ *min 8* (*m*−*5*)
  unfolding *board-def* **by** *auto*

obtain *ps*$_2$ **where** *knights-circuit* (*board 8* (*m*−*5*)) *ps*$_2$
  using *m-ge knights-circuit-8xm-exists*[*of* (*m*−*5*)] *knights-path-non-nil* **by** *auto*
then obtain *ps*$_2$*'* **where** *ps*$_2$*'-prems'*: *knights-circuit* (*board 8* (*m*−*5*)) *ps*$_2$*'*
hd ps$_2$*'* = (*1*,*1*) *last ps*$_2$*'* = (*3*,*2*)
  using *b-prems* ‹*5* ≤ *min 8* (*m*−*5*)› *rotate-knights-circuit* **by** *blast*
then have *ps*$_2$*'-path*: *knights-path* (*board 8* (*m*−*5*)) (*rev ps*$_2$*'*)
valid-step (*last ps*$_2$*'*) (*hd ps*$_2$*'*) *hd* (*rev ps*$_2$*'*) = (*3*,*2*) *last* (*rev ps*$_2$*'*) = (*1*,*1*)
  unfolding *knights-circuit-def* **using** *knights-path-rev* **by** (*auto simp*: *hd-rev last-rev*)

have *34* < *length ?ps*$_1$ *last* (*take 34 ?ps*$_1$) = (*4*,*5*) *hd* (*drop 34 ?ps*$_1$) = (*2*,*4*) **by** *eval+*
then have *step-in ?ps*$_1$ (*4*,*5*) (*2*,*4*)
  unfolding *step-in-def* **using** *zero-less-numeral* **by** *blast*
then have *step-in ?ps*$_1$ (*4*,*5*) (*2*,*4*)
valid-step (*4*,*5*) (*3*,*int 5*+*2*)
valid-step (*1*,*int 5*+*1*) (*2*,*4*)

unfolding *valid-step-def* **by** *auto*
then have $\exists$ *ps. knights-path* (*board 8 m*) *ps* $\wedge$ *hd ps* = *hd ?ps*$_1$ $\wedge$ *last ps* = *last ?ps*$_1$
using *m-ge ps*$_1$*-prems ps*$_2$*'-prems' ps*$_2$*'-path*
knights-path-split-concat[*of 8 5 ?ps*$_1$ *m*−*5 rev ps*$_2$*'*] **by** *auto*
then show *?thesis* **using** *ps*$_1$*-prems* **by** *auto*
qed
qed

$5 \leq ?m \Longrightarrow \exists ps.$ *knights-circuit* (*board 8 ?m*) *ps* $\wedge$ *step-in ps* (*2*, *int ?m* − *1*) (*4*, *int ?m*) and $5 \leq ?m \Longrightarrow \exists ps.$ *knights-path* (*board 8 ?m*) *ps* $\wedge$ *hd ps* = (*1*, *1*) $\wedge$ *last ps* = (*7*, *2*) formalize Lemma 3 from [1].

lemmas *knights-path-8xm-exists* = *knights-circuit-8xm-exists knights-path-8xm-ul-exists*

11 Knight's Paths and Circuits for $n \times m$-Boards

In this section the desired theorems are proved. The proof uses the previous lemmas to construct paths and circuits for arbitrary $n \times m$-boards.

A Knight's path for the (*5*×*5*)-board that starts in the lower-left and ends in the upper-left.

7	20	9	14	5
10	25	6	21	16
19	8	15	4	13
24	11	2	17	22
1	18	23	12	3

abbreviation *kp5x5ul* ≡ *the* (*to-path*
[[*7*,*20*,*9*,*14*,*5*],
[*10*,*25*,*6*,*21*,*16*],
[*19*,*8*,*15*,*4*,*13*],
[*24*,*11*,*2*,*17*,*22*],
[*1*,*18*,*23*,*12*,*3*]])
lemma *kp-5x5-ul*: *knights-path b5x5 kp5x5ul*
by (*simp only*: *knights-path-exec-simp*) *eval*

A Knight's path for the (*5*×*7*)-board that starts in the lower-left and ends in the upper-left.

17	14	25	6	19	8	29
26	35	18	15	28	5	20
13	16	27	24	7	30	9
34	23	2	11	32	21	4
1	12	33	22	3	10	31

abbreviation *kp5x7ul* ≡ *the* (*to-path*
[[*17*,*14*,*25*,*6*,*19*,*8*,*29*],
[*26*,*35*,*18*,*15*,*28*,*5*,*20*],
[*13*,*16*,*27*,*24*,*7*,*30*,*9*],
[*34*,*23*,*2*,*11*,*32*,*21*,*4*],
[*1*,*12*,*33*,*22*,*3*,*10*,*31*]])
lemma *kp-5x7-ul*: *knights-path b5x7 kp5x7ul*
by (*simp only*: *knights-path-exec-simp*) *eval*

A Knight's path for the (5×9)-board that starts in the lower-left and ends in the upper-left.

7	12	37	42	5	18	23	32	27
38	45	6	11	36	31	26	19	24
13	8	43	4	41	22	17	28	33
44	39	2	15	10	35	30	25	20
1	14	9	40	3	16	21	34	29

abbreviation *kp5x9ul* ≡ *the* (*to-path*
[[*7*,*12*,*37*,*42*,*5*,*18*,*23*,*32*,*27*],
[*38*,*45*,*6*,*11*,*36*,*31*,*26*,*19*,*24*],
[*13*,*8*,*43*,*4*,*41*,*22*,*17*,*28*,*33*],
[*44*,*39*,*2*,*15*,*10*,*35*,*30*,*25*,*20*],
[*1*,*14*,*9*,*40*,*3*,*16*,*21*,*34*,*29*]])
lemma *kp-5x9-ul*: *knights-path b5x9 kp5x9ul*
by (*simp only*: *knights-path-exec-simp*) *eval*

abbreviation *b7x7* ≡ *board 7 7*

A Knight's path for the (7×7)-board that starts in the lower-left and ends in the upper-left.

9	30	19	42	7	32	17
20	49	8	31	18	43	6
29	10	41	36	39	16	33
48	21	38	27	34	5	44
11	28	35	40	37	26	15
22	47	2	13	24	45	4
1	12	23	46	3	14	25

abbreviation *kp7x7ul* ≡ *the* (*to-path*
[[*9*,*30*,*19*,*42*,*7*,*32*,*17*],
[*20*,*49*,*8*,*31*,*18*,*43*,*6*],
[*29*,*10*,*41*,*36*,*39*,*16*,*33*],
[*48*,*21*,*38*,*27*,*34*,*5*,*44*],
[*11*,*28*,*35*,*40*,*37*,*26*,*15*],

[*22*,*47*,*2*,*13*,*24*,*45*,*4*],
[*1*,*12*,*23*,*46*,*3*,*14*,*25*]])

lemma *kp-7x7-ul*: *knights-path b7x7 kp7x7ul*
by (*simp only*: *knights-path-exec-simp*) *eval*

abbreviation *b7x9* ≡ *board 7 9*

A Knight's path for the (*7*×*9*)-board that starts in the lower-left and ends in the upper-left.

59	4	17	50	37	6	19	30	39
16	63	58	5	18	51	38	7	20
3	60	49	36	57	42	29	40	31
48	15	62	43	52	35	56	21	8
61	2	13	26	45	28	41	32	55
14	47	44	11	24	53	34	9	22
1	12	25	46	27	10	23	54	33

abbreviation *kp7x9ul* ≡ *the* (*to-path*
[[*59*,*4*,*17*,*50*,*37*,*6*,*19*,*30*,*39*],
[*16*,*63*,*58*,*5*,*18*,*51*,*38*,*7*,*20*],
[*3*,*60*,*49*,*36*,*57*,*42*,*29*,*40*,*31*],
[*48*,*15*,*62*,*43*,*52*,*35*,*56*,*21*,*8*],
[*61*,*2*,*13*,*26*,*45*,*28*,*41*,*32*,*55*],
[*14*,*47*,*44*,*11*,*24*,*53*,*34*,*9*,*22*],
[*1*,*12*,*25*,*46*,*27*,*10*,*23*,*54*,*33*]])

lemma *kp-7x9-ul*: *knights-path b7x9 kp7x9ul*
by (*simp only*: *knights-path-exec-simp*) *eval*

abbreviation *b9x7* ≡ *board 9 7*

A Knight's path for the (*9*×*7*)-board that starts in the lower-left and ends in the upper-left.

5	20	53	48	7	22	31
52	63	6	21	32	55	8
19	4	49	54	47	30	23
62	51	46	33	56	9	58
3	18	61	50	59	24	29
14	43	34	45	28	57	10
17	2	15	60	35	38	25
42	13	44	27	40	11	36
1	16	41	12	37	26	39

abbreviation *kp9x7ul* ≡ *the* (*to-path*
[[*5*,*20*,*53*,*48*,*7*,*22*,*31*],

[*52*,*63*,*6*,*21*,*32*,*55*,*8*],
[*19*,*4*,*49*,*54*,*47*,*30*,*23*],
[*62*,*51*,*46*,*33*,*56*,*9*,*58*],
[*3*,*18*,*61*,*50*,*59*,*24*,*29*],
[*14*,*43*,*34*,*45*,*28*,*57*,*10*],
[*17*,*2*,*15*,*60*,*35*,*38*,*25*],
[*42*,*13*,*44*,*27*,*40*,*11*,*36*],
[*1*,*16*,*41*,*12*,*37*,*26*,*39*]])

lemma *kp-9x7-ul*: *knights-path b9x7 kp9x7ul*
by (*simp only*: *knights-path-exec-simp*) *eval*

abbreviation *b9x9* ≡ *board 9 9*

A Knight's path for the (9×9)-board that starts in the lower-left and ends in the upper-left.

13	26	39	52	11	24	37	50	9
40	81	12	25	38	51	10	23	36
27	14	53	58	63	68	73	8	49
80	41	64	67	72	57	62	35	22
15	28	59	54	65	74	69	48	7
42	79	66	71	76	61	56	21	34
29	16	77	60	55	70	75	6	47
78	43	2	31	18	45	4	33	20
1	30	17	44	3	32	19	46	5

abbreviation *kp9x9ul* ≡ *the* (*to-path*
[[*13*,*26*,*39*,*52*,*11*,*24*,*37*,*50*,*9*],
[*40*,*81*,*12*,*25*,*38*,*51*,*10*,*23*,*36*],
[*27*,*14*,*53*,*58*,*63*,*68*,*73*,*8*,*49*],
[*80*,*41*,*64*,*67*,*72*,*57*,*62*,*35*,*22*],
[*15*,*28*,*59*,*54*,*65*,*74*,*69*,*48*,*7*],
[*42*,*79*,*66*,*71*,*76*,*61*,*56*,*21*,*34*],
[*29*,*16*,*77*,*60*,*55*,*70*,*75*,*6*,*47*],
[*78*,*43*,*2*,*31*,*18*,*45*,*4*,*33*,*20*],
[*1*,*30*,*17*,*44*,*3*,*32*,*19*,*46*,*5*]])

lemma *kp-9x9-ul*: *knights-path b9x9 kp9x9ul*
by (*simp only*: *knights-path-exec-simp*) *eval*

The following lemma is a sub-proof used in Lemma 4 in [1]. I moved the sub-proof out to a separate lemma.

lemma *knights-circuit-exists-even-n-gr10*:
assumes *even n* $n \geq 10$ $m \geq 5$
$\exists ps.$ *knights-path* (*board* $(n-5)$ *m*) *ps* ∧ *hd ps* = (*int* $(n-5)$,*1*)
∧ *last ps* = (*int* $(n-5)-1$,*int* $m-1$)
shows ∃*ps*. *knights-circuit* (*board m n*) *ps*
using *assms*

proof −
  let *?b*$_2$=*board* (*n*−*5*) *m*
  assume $n \geq 10$
  then obtain *ps*$_2$ **where** *ps*$_2$*-prems*: *knights-path ?b*$_2$ *ps*$_2$ *hd ps*$_2$ = (*int* (*n*−*5*),*1*)

    last ps$_2$ = (*int* (*n*−*5*)−*1*,*int m*−*1*)
    using *assms* **by** *auto*
  let *?ps*$_2$*-m2*=*mirror2 ps*$_2$
  have *ps*$_2$*-m2-prems*: *knights-path ?b*$_2$ *?ps*$_2$*-m2 hd ?ps*$_2$*-m2* = (*int* (*n*−*5*),*int m*)
    last ?ps$_2$*-m2* = (*int* (*n*−*5*)−*1*,*2*)
    using *ps*$_2$*-prems mirror2-knights-path hd-mirror2 last-mirror2* **by** *auto*

  obtain *ps*$_1$ **where** *ps*$_1$*-prems*: *knights-path* (*board 5 m*) *ps*$_1$ *hd ps*$_1$ = (*1*,*1*)*last ps*$_1$ = (*2*,*int m*−*1*)
    using *assms knights-path-5xm-exists* **by** *auto*
  let *?ps*$_1$*'*=*trans-path* (*int* (*n*−*5*),*0*) *ps*$_1$
  let *?b*$_1$*'*=*trans-board* (*int* (*n*−*5*),*0*) (*board 5 m*)
  have *ps*$_1$*'-prems*: *knights-path ?b*$_1$*' ?ps*$_1$*' hd ?ps*$_1$*'* = (*int* (*n*−*5*)+*1*,*1*)
    last ?ps$_1$*'* = (*int* (*n*−*5*)+*2*,*int m*−*1*)
  using *ps*$_1$*-prems trans-knights-path knights-path-non-nil hd-trans-path last-trans-path* **by** *auto*

  let *?ps*=*?ps*$_1$*'*@*?ps*$_2$*-m2*
  let *?psT*=*transpose ?ps*

  have $n-5 \geq 5$ **using** ‹$n \geq 10$› **by** *auto*
  have *inter*: *?b*$_1$*'* ∩ *?b*$_2$ = {}
    unfolding *trans-board-def board-def* **using** ‹$n-5 \geq 5$› **by** *auto*
  have *union*: *?b*$_1$*'* ∪ *?b*$_2$ = *board n m*
    using ‹$n-5 \geq 5$› *board-concatT*[*of n*−*5 m 5*] **by** *auto*

  have *vs*: *valid-step* (*last ?ps*$_1$*'*) (*hd ?ps*$_2$*-m2*) **and** *valid-step* (*last ?ps*$_2$*-m2*) (*hd ?ps*$_1$*'*)
    unfolding *valid-step-def* **using** *ps*$_1$*'-prems ps*$_2$*-m2-prems* **by** *auto*
  then have *vs-c*: *valid-step* (*last ?ps*) (*hd ?ps*)
    using *ps*$_1$*'-prems ps*$_2$*-m2-prems knights-path-non-nil* **by** *auto*

  have *knights-path* (*board n m*) *?ps*
    using *ps*$_1$*'-prems ps*$_2$*-m2-prems inter vs union knights-path-append*[*of ?b*$_1$*' ?ps*$_1$*' ?b*$_2$ *?ps*$_2$*-m2*]
    by *auto*
  then have *knights-circuit* (*board n m*) *?ps*
    unfolding *knights-circuit-def* **using** *vs-c* **by** *auto*
  then show *?thesis* **using** *transpose-knights-circuit* **by** *auto*
qed

For every $n \times m$-board with $min\ n\ m \geq 5$ and odd n there exists a Knight's path that starts in $(n,1)$ (top-left) and ends in $(n-1,m-1)$ (top-right).

This lemma formalizes Lemma 4 from [1]. Formalizing the proof of this lemma was quite challenging as a lot of details on how to exactly combine the boards are left out in the original proof in [1].

```
lemma knights-path-odd-n-exists:
  assumes odd n min n m ≥ 5
  shows ∃ps. knights-path (board n m) ps ∧ hd ps = (int n,1) ∧ last ps = (int
n−1,int m−1)
  using assms
proof −
  obtain x where x = n + m by auto
  then show ?thesis
    using assms
  proof (induction x arbitrary: n m rule: less-induct)
    case (less x)
    then have m = 5 ∨ m = 6 ∨ m = 7 ∨ m = 8 ∨ m = 9 ∨ m ≥ 10 by auto
    then show ?case
    proof (elim disjE)
      assume [simp]: m = 5
      have odd n n ≥ 5 using less by auto
      then have n = 5 ∨ n = 7 ∨ n = 9 ∨ n−5 ≥ 5 by presburger
      then show ?thesis
      proof (elim disjE)
        assume [simp]: n = 5
        let ?ps=mirror1 (transpose kp5x5ul)
        have kp: knights-path (board n m) ?ps
          using kp-5x5-ul rot90-knights-path by auto
        have hd ?ps = (int n,1) last ?ps = (int n−1,int m−1)
          by (simp only: ‹m = 5› ‹n = 5› | eval)+
        then show ?thesis using kp by auto
      next
        assume [simp]: n = 7
        let ?ps=mirror1 (transpose kp5x7ul)
        have kp: knights-path (board n m) ?ps
          using kp-5x7-ul rot90-knights-path by auto
        have hd ?ps = (int n,1) last ?ps = (int n−1,int m−1)
          by (simp only: ‹m = 5› ‹n = 7› | eval)+
        then show ?thesis using kp by auto
      next
        assume [simp]: n = 9
        let ?ps=mirror1 (transpose kp5x9ul)
        have kp: knights-path (board n m) ?ps
          using kp-5x9-ul rot90-knights-path by auto
        have hd ?ps = (int n,1) last ?ps = (int n−1,int m−1)
          by (simp only: ‹m = 5› ‹n = 9› | eval)+
        then show ?thesis using kp by auto
      next
        let ?b2=board m (n−5)
        assume n−5 ≥ 5
        then have ∃ps. knights-circuit ?b2 ps
        proof −
```

have *n−5 = 6 ∨ n−5 = 8 ∨ n−5 ≥ 10*
using *‹n−5 ≥ 5› less* **by** *presburger*
then show *?thesis*
proof (*elim disjE*)
assume *n−5 = 6*
then obtain *ps* **where** *knights-circuit* (*board* (*n−5*) *m*) *ps*
using *knights-path-6xm-exists*[*of m*] **by** *auto*
then show *?thesis*
using *transpose-knights-circuit* **by** *auto*
next
assume *n−5 = 8*
then obtain *ps* **where** *knights-circuit* (*board* (*n−5*) *m*) *ps*
using *knights-path-8xm-exists*[*of m*] **by** *auto*
then show *?thesis*
using *transpose-knights-circuit* **by** *auto*
next
assume *n−5 ≥ 10*
then show *?thesis*
using *less less.IH*[*of n−10+m n−10 m*]
knights-circuit-exists-even-n-gr10[*of n−5 m*] **by** *auto*
qed
qed
then obtain ps_2 **where** *knights-circuit* $?b_2$ ps_2 *hd* ps_2 = (*1*,*1*) *last* ps_2 = (*3*,*2*)
using *‹n−5 ≥ 5› rotate-knights-circuit*[*of m n−5*] **by** *auto*
then have *rev-*ps_2*-prems*: *knights-path* $?b_2$ (*rev* ps_2) *valid-step* (*last* ps_2) (*hd* ps_2)
hd (*rev* ps_2) = (*3*,*2*) *last* (*rev* ps_2) = (*1*,*1*)
unfolding *knights-circuit-def* **using** *knights-path-rev* **by** (*auto simp*: *hd-rev last-rev*)

let $?ps_1$=*kp5x5ul*
have ps_1*-prems*: *knights-path* (*board 5 5*) $?ps_1$ *hd* $?ps_1$ = (*1*,*1*) *last* $?ps_1$ = (*4*,*2*)
using *kp-5x5-ul* **by** *simp eval+*

have *16 < length* $?ps_1$ *last* (*take 16* $?ps_1$) = (*4*,*5*) *hd* (*drop 16* $?ps_1$) = (*2*,*4*) **by** *eval+*
then have *si*: *step-in* $?ps_1$ (*4*,*5*) (*2*,*4*)
unfolding *step-in-def* **using** *zero-less-numeral* **by** *blast*

have *vs*: *valid-step* (*4*,*5*) (*3*,*int 5+2*) *valid-step* (*1*,*int 5+1*) (*2*,*4*)
unfolding *valid-step-def* **by** *auto*

obtain *ps* **where** *knights-path* (*board m n*) *ps hd ps* = (*1*,*1*) *last ps* = (*4*,*2*)
using *‹n−5 ≥ 5›* ps_1*-prems rev-*ps_2*-prems si vs*
knights-path-split-concat[*of 5 5* $?ps_1$ *n−5 rev* ps_2 (*4*,*5*) (*2*,*4*)] **by** *auto*
then show *?thesis*
using *rot90-knights-path hd-rot90-knights-path last-rot90-knights-path* **by**

fastforce
qed
next
assume [*simp*]: *m* = *6*
then obtain *ps* **where**
ps-prems: *knights-path* (*board m n*) *ps hd ps* = (*1*,*1*) *last ps* = (*int m*−*1*,*2*)
using *less knights-path-6xm-exists*[*of n*] **by** *auto*
let *?ps′*=*mirror1* (*transpose ps*)
have *knights-path* (*board n m*) *?ps′ hd ?ps′* = (*int n*,*1*) *last ?ps′* = (*int n*−*1*,*int m*−*1*)
using *ps-prems rot90-knights-path hd-rot90-knights-path last-rot90-knights-path* **by** *auto*
then show *?thesis* **by** *auto*
next
assume [*simp*]: *m* = *7*
have *odd n n* ≥ *5* **using** *less* **by** *auto*
then have *n* = *5* ∨ *n* = *7* ∨ *n* = *9* ∨ *n*−*5* ≥ *5* **by** *presburger*
then show *?thesis*
proof (*elim disjE*)
assume [*simp*]: *n* = *5*
let *?ps*=*mirror1 kp5x7lr*
have *kp*: *knights-path* (*board n m*) *?ps*
using *kp-5x7-lr mirror1-knights-path* **by** *auto*
have *hd ?ps* = (*int n*,*1*) *last ?ps* = (*int n*−*1*,*int m*−*1*)
by (*simp only*: ‹*m* = *7*› ‹*n* = *5*› | *eval*)+
then show *?thesis* **using** *kp* **by** *auto*
next
assume [*simp*]: *n* = *7*
let *?ps*=*mirror1* (*transpose kp7x7ul*)
have *kp*: *knights-path* (*board n m*) *?ps*
using *kp-7x7-ul rot90-knights-path* **by** *auto*
have *hd ?ps* = (*int n*,*1*) *last ?ps* = (*int n*−*1*,*int m*−*1*)
by (*simp only*: ‹*m* = *7*› ‹*n* = *7*› | *eval*)+
then show *?thesis* **using** *kp* **by** *auto*
next
assume [*simp*]: *n* = *9*
let *?ps*=*mirror1* (*transpose kp7x9ul*)
have *kp*: *knights-path* (*board n m*) *?ps*
using *kp-7x9-ul rot90-knights-path* **by** *auto*
have *hd ?ps* = (*int n*,*1*) *last ?ps* = (*int n*−*1*,*int m*−*1*)
by (*simp only*: ‹*m* = *7*› ‹*n* = *9*› | *eval*)+
then show *?thesis* **using** *kp* **by** *auto*
next
let $?b_2$=*board m* (*n*−*5*)
let $?b_2T$=*board* (*n*−*5*) *m*
assume *n*−*5* ≥ *5*
then have ∃*ps*. *knights-circuit* $?b_2$ *ps*
proof −
have *n*−*5* = *6* ∨ *n*−*5* = *8* ∨ *n*−*5* ≥ *10*

using ‹$n-5 \geq 5$› *less* **by** *presburger*
then show *?thesis*
proof (*elim disjE*)
assume $n-5 = 6$
then obtain *ps* **where** *knights-circuit* (*board* ($n-5$) *m*) *ps*
using *knights-path-6xm-exists*[*of m*] **by** *auto*
then show *?thesis*
using *transpose-knights-circuit* **by** *auto*
next
assume $n-5 = 8$
then obtain *ps* **where** *knights-circuit* (*board* ($n-5$) *m*) *ps*
using *knights-path-8xm-exists*[*of m*] **by** *auto*
then show *?thesis*
using *transpose-knights-circuit* **by** *auto*
next
assume $n-5 \geq 10$
then show *?thesis*
using *less less.IH*[*of* $n-10+m$ $n-10$ *m*]
knights-circuit-exists-even-n-gr10[*of* $n-5$ *m*] **by** *auto*
qed
qed
then obtain ps_2 **where** ps_2*-prems*: *knights-circuit* $?b_2$ ps_2 *hd* $ps_2 = (1,1)$
last $ps_2 = (3,2)$
using ‹$n-5 \geq 5$› *rotate-knights-circuit*[*of m* $n-5$] **by** *auto*
let $?ps_2T$=*transpose* ps_2
have ps_2T*-prems*: *knights-path* $?b_2T$ $?ps_2T$ *hd* $?ps_2T = (1,1)$ *last* $?ps_2T =$ $(2,3)$
using ps_2*-prems transpose-knights-path knights-path-non-nil hd-transpose last-transpose*
unfolding *knights-circuit-def transpose-square-def* **by** *auto*

let $?ps_1$=*kp5x7lr*
have ps_1*-prems*: *knights-path b5x7* $?ps_1$ *hd* $?ps_1 = (1,1)$ *last* $?ps_1 = (2,6)$
using *kp-5x7-lr* **by** *simp eval+*

have $29 <$ *length* $?ps_1$ *last* (*take* 29 $?ps_1$) $= (4,2)$ *hd* (*drop* 29 $?ps_1$) $=$ $(5,4)$ **by** *eval+*
then have *si*: *step-in* $?ps_1$ $(4,2)$ $(5,4)$
unfolding *step-in-def* **using** *zero-less-numeral* **by** *blast*

have *vs*: *valid-step* $(4,2)$ (*int* $5+1,1$) *valid-step* (*int* $5+2,3$) $(5,4)$
unfolding *valid-step-def* **by** *auto*

obtain *ps* **where** *knights-path* (*board n m*) *ps hd ps* $= (1,1)$ *last ps* $= (2,6)$
using ‹$n-5 \geq 5$› ps_1*-prems* ps_2T*-prems si vs*
knights-path-split-concatT[*of 5 m* $?ps_1$ $n-5$ $?ps_2T$ $(4,2)$ $(5,4)$] **by** *auto*
then show *?thesis*
using *mirror1-knights-path hd-mirror1 last-mirror1* **by** *fastforce*
qed

next
assume [*simp*]: $m = 8$
then obtain *ps* **where** *ps-prems*: *knights-path* (*board m n*) *ps hd ps* = (*1*,*1*)
last ps = (*int m*−*1*,*2*)
using *less knights-path-8xm-exists*[*of n*] **by** *auto*
let *?ps'*=*mirror1* (*transpose ps*)
have *knights-path* (*board n m*) *?ps' hd ?ps'* = (*int n*,*1*) *last ?ps'* = (*int n*−*1*,*int m*−*1*)
using *ps-prems rot90-knights-path hd-rot90-knights-path last-rot90-knights-path* **by** *auto*
then show *?thesis* **by** *auto*
next
assume [*simp*]: $m = 9$
have *odd n* $n \geq 5$ **using** *less* **by** *auto*
then have $n = 5 \lor n = 7 \lor n = 9 \lor n-5 \geq 5$ **by** *presburger*
then show *?thesis*
proof (*elim disjE*)
assume [*simp*]: $n = 5$
let *?ps*=*mirror1 kp5x9lr*
have *kp*: *knights-path* (*board n m*) *?ps*
using *kp-5x9-lr mirror1-knights-path* **by** *auto*
have *hd ?ps* = (*int n*,*1*) *last ?ps* = (*int n*−*1*,*int m*−*1*)
by (*simp only*: ‹$m = 9$› ‹$n = 5$› | *eval*)+
then show *?thesis* **using** *kp* **by** *auto*
next
assume [*simp*]: $n = 7$
let *?ps*=*mirror1* (*transpose kp9x7ul*)
have *kp*: *knights-path* (*board n m*) *?ps*
using *kp-9x7-ul rot90-knights-path* **by** *auto*
have *hd ?ps* = (*int n*,*1*) *last ?ps* = (*int n*−*1*,*int m*−*1*)
by (*simp only*: ‹$m = 9$› ‹$n = 7$› | *eval*)+
then show *?thesis* **using** *kp* **by** *auto*
next
assume [*simp*]: $n = 9$
let *?ps*=*mirror1* (*transpose kp9x9ul*)
have *kp*: *knights-path* (*board n m*) *?ps*
using *kp-9x9-ul rot90-knights-path* **by** *auto*
have *hd ?ps* = (*int n*,*1*) *last ?ps* = (*int n*−*1*,*int m*−*1*)
by (*simp only*: ‹$m = 9$› ‹$n = 9$› | *eval*)+
then show *?thesis* **using** *kp* **by** *auto*
next
let $?b_2$=*board m* (*n*−*5*)
let $?b_2T$=*board* (*n*−*5*) *m*
assume $n-5 \geq 5$
then have ∃*ps*. *knights-circuit* $?b_2$ *ps*
proof −
have $n-5 = 6 \lor n-5 = 8 \lor n-5 \geq 10$
using ‹$n-5 \geq 5$› *less* **by** *presburger*
then show *?thesis*

proof (*elim disjE*)
assume $n-5 = 6$
then obtain *ps* **where** *knights-circuit* (*board* ($n-5$) *m*) *ps*
using *knights-path-6xm-exists*[*of m*] **by** *auto*
then show *?thesis*
using *transpose-knights-circuit* **by** *auto*
next
assume $n-5 = 8$
then obtain *ps* **where** *knights-circuit* (*board* ($n-5$) *m*) *ps*
using *knights-path-8xm-exists*[*of m*] **by** *auto*
then show *?thesis*
using *transpose-knights-circuit* **by** *auto*
next
assume $n-5 \geq 10$
then show *?thesis*
using *less less.IH*[*of* $n-10+m$ $n-10$ *m*]
knights-circuit-exists-even-n-gr10[*of* $n-5$ *m*] **by** *auto*
qed
qed
then obtain ps_2 **where** ps_2*-prems*: *knights-circuit* $?b_2$ ps_2 *hd* $ps_2 = (1,1)$
last $ps_2 = (3,2)$
using ‹$n-5 \geq 5$› *rotate-knights-circuit*[*of m* $n-5$] **by** *auto*
let $?ps_2T$=*transpose* (*rev* ps_2)
have ps_2T*-prems*: *knights-path* $?b_2T$ $?ps_2T$ *hd* $?ps_2T = (2,3)$ *last* $?ps_2T = (1,1)$
using ps_2*-prems knights-path-rev transpose-knights-path knights-path-non-nil*

hd-transpose last-transpose
unfolding *knights-circuit-def transpose-square-def* **by** (*auto simp*: *hd-rev last-rev*)

let $?ps_1$=*kp5x9lr*
have ps_1*-prems*: *knights-path b5x9* $?ps_1$ *hd* $?ps_1 = (1,1)$ *last* $?ps_1 = (2,8)$
using *kp-5x9-lr* **by** *simp eval+*

have $16 <$ *length* $?ps_1$ *last* (*take 16* $?ps_1$) $= (5,4)$ *hd* (*drop 16* $?ps_1$) $= (4,2)$ **by** *eval+*
then have *si*: *step-in* $?ps_1$ $(5,4)$ $(4,2)$
unfolding *step-in-def* **using** *zero-less-numeral* **by** *blast*

have *vs*: *valid-step* $(5,4)$ (*int* $5+2,3$) *valid-step* (*int* $5+1,1$) $(4,2)$
unfolding *valid-step-def* **by** *auto*

obtain *ps* **where** *knights-path* (*board n m*) *ps hd ps* $= (1,1)$ *last ps* $= (2,8)$
using ‹$n-5 \geq 5$› ps_1*-prems* ps_2T*-prems si vs*
knights-path-split-concatT[*of 5 m* $?ps_1$ $n-5$ $?ps_2T$ $(5,4)$ $(4,2)$] **by** *auto*
then show *?thesis*
using *mirror1-knights-path hd-mirror1 last-mirror1* **by** *fastforce*
qed

next
let *?b*$_1$*=board n 5*
let *?b*$_2$*=board n* (*m*−*5*)
assume $m \geq 10$
then have $n+5 < x\ 5 \leq min\ n\ 5\ n+(m-5) < x\ 5 \leq min\ n\ (m-5)$
using *less* **by** *auto*
then obtain ps_1 ps_2 **where** *kp-prems*:
knights-path ?b$_1$ ps_1 *hd* ps_1 = (*int n*,*1*) *last* ps_1 = (*int n*−*1*,*4*)
knights-path (*board n* (*m*−*5*)) ps_2 *hd* ps_2 = (*int n*,*1*) *last* ps_2 = (*int n*−*1*,*int* (*m*−*5*)−*1*)
using *less.prems less.IH*[*of n*+*5 n 5*] *less.IH*[*of n*+(*m*−*5*) *n m*−*5*] **by** *auto*
let *?ps=ps*$_1$*@trans-path* (*0*,*int 5*) ps_2
have *valid-step* (*last* ps_1) (*int n*,*int 5*+*1*)
unfolding *valid-step-def* **using** *kp-prems* **by** *auto*
then have *knights-path* (*board n m*) *?ps hd ?ps* = (*int n*,*1*) *last ?ps* = (*int n*−*1*,*int m*−*1*)
using ‹$m \geq 10$› *kp-prems knights-path-concat*[*of n 5* ps_1 *m*−*5* ps_2]
knights-path-non-nil trans-path-non-nil last-trans-path **by** *auto*
then show *?thesis* **by** *auto*
qed
qed
qed

Auxiliary lemma that constructs a Knight's circuit if $m \geq 5$ and $n \geq 10 \wedge$ *even n*.

lemma *knights-circuit-exists-n-even-gr-10*:
assumes $n \geq 10 \wedge$ *even n* $m \geq 5$
shows ∃*ps*. *knights-circuit* (*board n m*) *ps*
using *assms*
proof −
obtain ps_1 **where** ps_1*-prems*: *knights-path* (*board 5 m*) ps_1 *hd* ps_1 = (*1*,*1*)
last ps_1 = (*2*,*int m*−*1*)
using *assms knights-path-5xm-exists* **by** *auto*
let *?ps*$_1$*'=trans-path* (*int* (*n*−*5*),*0*) ps_1
let *?b5xm'=trans-board* (*int* (*n*−*5*),*0*) (*board 5 m*)
have ps_1*'-prems*: *knights-path ?b5xm' ?ps*$_1$*' hd ?ps*$_1$*'* = (*int* (*n*−*5*)+*1*,*1*)
last ?ps$_1$*'* = (*int* (*n*−*5*)+*2*,*int m*−*1*)
using ps_1*-prems trans-knights-path knights-path-non-nil hd-trans-path last-trans-path* **by** *auto*

assume $n \geq 10 \wedge$ *even n*
then have *odd* (*n*−*5*) *min* (*n*−*5*) $m \geq 5$ **using** *assms* **by** *auto*
then obtain ps_2 **where** ps_2*-prems*: *knights-path* (*board* (*n*−*5*) *m*) ps_2 *hd* ps_2 = (*int* (*n*−*5*),*1*)
last ps_2 = (*int* (*n*−*5*)−*1*,*int m*−*1*)
using *knights-path-odd-n-exists*[*of n*−*5 m*] **by** *auto*
let *?ps*$_2$*'=mirror2* ps_2
have ps_2*'-prems*: *knights-path* (*board* (*n*−*5*) *m*) *?ps*$_2$*' hd ?ps*$_2$*'* = (*int* (*n*−*5*),*int m*)

```
    last ?ps2' = (int (n−5)−1,2)
   using ps2-prems mirror2-knights-path hd-mirror2 last-mirror2 by auto

 have inter: ?b5xm' ∩ board (n−5) m = {}
   unfolding trans-board-def board-def by auto

 have union: board n m = ?b5xm' ∪ board (n−5) m
   using ‹n ≥ 10 ∧ even n› board-concatT[of n−5 m 5] by auto

 have vs: valid-step (last ?ps1') (hd ?ps2') valid-step (last ?ps2') (hd ?ps1')
   using ps1'-prems ps2'-prems unfolding valid-step-def by auto

 let ?ps=?ps1' @ ?ps2'
 have last ?ps = last ?ps2' hd ?ps = hd ?ps1'
   using ps1'-prems ps2'-prems knights-path-non-nil by auto
 then have vs-c: valid-step (last ?ps) (hd ?ps)
   using vs by auto

 have knights-path (board n m) ?ps
   using ps1'-prems ps2'-prems inter union vs knights-path-append by auto
 then show ?thesis
   using vs-c unfolding knights-circuit-def by blast
qed
```

Final Theorem 1: For every $n \times m$-board with $min\ n\ m \geq 5$ and $n*m$ even there exists a Knight's circuit.

```
theorem knights-circuit-exists:
 assumes min n m ≥ 5 even (n∗m)
 shows ∃ps. knights-circuit (board n m) ps
 using assms
proof −
 have n = 6 ∨ m = 6 ∨ n = 8 ∨ m = 8 ∨ (n ≥ 10 ∧ even n) ∨ (m ≥ 10 ∧
even m)
   using assms by auto
 then show ?thesis
 proof (elim disjE)
   assume n = 6
   then show ?thesis
     using assms knights-path-6xm-exists by auto
 next
   assume m = 6
   then obtain ps where knights-circuit (board m n) ps
     using assms knights-path-6xm-exists by auto
   then show ?thesis
     using transpose-knights-circuit by auto
 next
   assume n = 8
   then show ?thesis
     using assms knights-path-8xm-exists by auto
```

```
  next
    assume m = 8
    then obtain ps where knights-circuit (board m n) ps
      using assms knights-path-8xm-exists by auto
    then show ?thesis
      using transpose-knights-circuit by auto
  next
    assume n ≥ 10 ∧ even n
    then show ?thesis
      using assms knights-circuit-exists-n-even-gr-10 by auto
  next
    assume m ≥ 10 ∧ even m
    then obtain ps where knights-circuit (board m n) ps
      using assms knights-circuit-exists-n-even-gr-10 by auto
    then show ?thesis
      using transpose-knights-circuit by auto
  qed
qed
```

Final Theorem 2: for every $n \times m$-board with *min n m* $\geq$ *5* there exists a Knight's path.

```
theorem knights-path-exists:
  assumes min n m ≥ 5
  shows ∃ps. knights-path (board n m) ps
  using assms
proof −
  have odd n ∨ odd m ∨ even (n∗m) by simp
  then show ?thesis
  proof (elim disjE)
    assume odd n
    then show ?thesis
      using assms knights-path-odd-n-exists by auto
  next
    assume odd m
    then obtain ps where knights-path (board m n) ps
      using assms knights-path-odd-n-exists by auto
    then show ?thesis
      using transpose-knights-path by auto
  next
    assume even (n∗m)
    then show ?thesis
      using assms knights-circuit-exists by (auto simp: knights-circuit-def)
  qed
qed
```

THE END

end

References

[1] P. Cull and J. D. Curtins. Knight's tour revisited. *Fibonacci Quarterly*, 16:276–285, 1978.