theory *Problem-5*
  imports *Main*
    HOL−Number-Theory.Number-Theory
    Common.Future-Library
begin

A mathematical frog jumps along the number line. The frog starts at 1, and jumps according to the following rule: if the frog is at integer n, then it can jump to $n+1$ or $n+2^{m_n+1}$, where 2^{m_n} is the largest power of 2 that is a factor of n. Show that if $k \geq 2$, is a positive integer and i is a non-negative integer, then the minimum number of jumps needed to reach $2^i k$ is greater than the minimum number of jumps needed to reach 2^i.

definition *jump* :: *nat* ⇒ *nat* **where**
  jump n = *n* + *2*^(*multiplicity 2 n* + *1*)

reachable s n is true if the frog can reach integer n with s jumps.

inductive *reachable* :: *nat* ⇒ *nat* ⇒ *bool* **where**
  Start[*intro*]: *reachable 0 1* |
  Incr[*intro*]: *reachable s n* ⟹ *reachable* (*Suc s*) (*Suc n*) |
  Jump[*intro*]: *reachable s n* ⟹ *reachable* (*Suc s*) (*jump n*)

The *height* of a number is the minimum number of jumps required.

definition *height* :: *nat* ⇒ *nat* **where**
  height n = *arg-min id* (λ*s*. *reachable s n*)

0.1 Alternative definition of *jump*

jump 0 isn't really well-defined. We introduce an alternative definition:

definition *jump′* **where**
  jump′ n = (*if n* = *0 then 0 else jump n*)

lemma *jump′*: *n* ≠ *0* ⟹ *jump n* = *jump′ n*
  unfolding *jump-def jump′-def* **by** *simp*

lemma *jump′-0*[*simp*]: *jump′ 0* = *0* **by** (*simp add*: *jump′-def*)

inductive *reachable′* :: *nat* ⇒ *nat* ⇒ *bool* **where**
  Start[*intro*]: *reachable′ 0 1* |
  Incr[*intro*]: *reachable′ s n* ⟹ *reachable′* (*Suc s*) (*Suc n*) |
  Jump[*intro*]: *reachable′ s n* ⟹ *reachable′* (*Suc s*) (*jump′ n*)

lemma *zero-unreachable*:
  ¬*reachable s 0*
proof
  assume *reachable s 0*
  moreover have *jump n* > *0* **for** *n* **unfolding** *jump-def* **by** *simp*
  ultimately show *False*
    apply (*cases rule*: *reachable.cases*)
     apply *presburger*
    by (*metis not-less0*)
qed

lemma *reachable′*:
  reachable s n = *reachable′ s n*
proof
  assume *reachable s n*
  then show *reachable′ s n*
  proof (*induction rule*: *reachable.induct*)
    case (*Jump s n*)
    from ‹*reachable s n*› **have** *n* ≠ *0*
      by (*metis zero-unreachable*)
    with *Jump* **show** *?case* **by** (*auto simp add*: *jump′*)

qed (*intro reachable'.intros*)+
next
assume *reachable' s n*
then show *reachable s n*
proof (*induction rule*: *reachable'.induct*)
case (*Jump s n*)
from ‹*reachable s n*› **have** $n \neq 0$
by (*metis zero-unreachable*)
with *Jump* **show** *?case* **by** (*auto simp flip*: *jump'*)
qed (*intro reachable.intros*)+
qed

lemma *zero-unreachable'*: ¬ *reachable' s 0*
using *zero-unreachable* **by** (*simp add*: *reachable'*)

definition *height'* :: *nat* ⇒ *nat* **where**
height' n = *arg-min id* (λ*s*. *reachable' s n*)

lemma *height'*[*simp*]:
height n = *height' n*
unfolding *height-def height'-def* **by** (*simp add*: *reachable'*)

0.2 Properties of *jump'*

lemma *jump-even*[*simp*]: *jump'* (*2***n*) = *2* * *jump' n*
by (*auto simp add*: *jump'-def jump-def multiplicity-times-same*)

lemma *jump-odd*[*simp*]: *jump'* (*Suc* (*2***n*)) = *Suc* (*Suc* (*Suc* (*2***n*)))
unfolding *jump'-def jump-def*
by (*auto intro*: *not-dvd-imp-multiplicity-0*)

lemma *jump-gt0*[*simp*]: $n > 0 \Longrightarrow$ *jump' n* > 0
unfolding *jump'-def jump-def* **by** *auto*

0.3 Representing paths

lemma *reachable-height*:
$n \neq 0 \Longrightarrow$ *reachable'* (*height' n*) *n*
— This is not a tautology, we need to prove that the number is reachable at all.
unfolding *height'-def*
proof (*intro arg-min-natI*)
assume $n \neq 0$
then show *reachable'* ($n - 1$) *n*
proof (*induction n*)
case (*Suc n*)
then show *?case*
using *Start Incr* **by** (*cases n*) *auto*
qed *auto*
qed

Based on this, we define a notion of a path — a specific set of moves with a desired result.

datatype *move* = *Incr* | *Jump*
type-synonym *path* = *move list*

fun *make-move* :: *move* ⇒ *nat* ⇒ *nat* **where**
make-move Incr n = *Suc n* |
make-move Jump n = *jump' n*

fun *make-moves* :: *path* ⇒ *nat* ⇒ *nat* **where**
make-moves moves n = *fold make-move moves n*

lemma *path-exists*:

```
  assumes reachable' s n
  shows ∃path. make-moves path 1 = n ∧ length path = s
  using assms
proof induction
  case (Incr s n)
  then obtain path where make-moves path 1 = n and length path = s
    by auto
  hence make-moves (path @ [Incr]) 1 = Suc n and length (path @ [Incr]) = Suc s
    by auto
  then show ?case by blast
next
  case (Jump s n)
  then obtain path where make-moves path 1 = n and length path = s
    by auto
  moreover have n ≠ 0 using Jump.hyps zero-unreachable' by meson
  ultimately have make-moves (path @ [Jump]) 1 = jump' n and length (path @ [Jump]) = Suc s
    by auto
  then show ?case by blast
qed simp

lemma path-implies-reachable:
  reachable' (length path) (make-moves path 1)
proof (induction path rule: rev-induct)
  case Nil
  then show ?case using Start by auto
next
  case (snoc move path)
  then show ?case
    by (cases move) auto
qed

lemma heightI[intro]:
  assumes make-moves p 1 = x
  shows height' x ≤ length p
proof -
  from assms have reachable' (length p) x
    using path-implies-reachable by auto
  thus height' x ≤ length p
    unfolding height'-def using arg-min-nat-le[where m=id] by auto
qed

lemma heightD:
  assumes x ≠ 0
  obtains p where make-moves p 1 = x and length p = height' x
proof -
  from ‹x ≠ 0› have reachable' (height' x) x
    using reachable-height by simp
  with that show thesis using path-exists by auto
qed
```

0.4 The case for even k

From a path leading to k, we can derive a path to $k/2$ — for each step, look at the change to the current integer without the lowest bit. As it turns out, each of those can be mimicked in one move.

```
fun halfpath :: nat ⇒ path ⇒ path where
halfpath n [] = [] |
halfpath n (Incr # p) =
  (if even n
   then halfpath (Suc n) p
   else Incr # halfpath (Suc n) p) |
```

```
halfpath n (Jump # p) =
  (if even n
   then Jump # halfpath (jump' n) p
   else Incr # halfpath (jump' n) p)

lemma halfpath:
  assumes n ≠ 0  b < 2
    and make-moves p (2*n+b) = 2*v
  shows make-moves (halfpath (2*n+b) p) n = v
  using assms
proof (induction p arbitrary: n b)
  case Nil
  then show ?case by simp
next
  case (Cons move p)
  then consider b = 0 | b = 1 by fastforce
  thus ?case
  proof cases
    case 1
    then show ?thesis
    proof (cases move)
      case Incr
      from Cons.IH[of n 1] ‹n ≠ 0› Cons show ?thesis
        unfolding Incr 1 by simp
    next
      case Jump
      from Cons.IH[of jump' n 0] ‹n ≠ 0› Cons.prems show ?thesis
        unfolding Jump 1 by simp
    qed
  next
    case 2
    then show ?thesis
    proof (cases move)
      case Incr
      from Cons.IH[of Suc n 0] ‹n ≠ 0› Cons show ?thesis
        unfolding Incr 2 by simp
    next
      case Jump
      from Cons.IH[of Suc n 1] ‹n ≠ 0› Cons show ?thesis
        unfolding Jump 2 by simp
    qed
  qed
qed

lemma halfpath-shorter:
  length (halfpath n p) ≤ length p
  apply (induction p arbitrary: n)
   apply simp
  subgoal for move
    apply (cases move)
    by (auto simp add: le-SucI)
  done

lemma halfpath-hd:
  p ≠ [] ⟹ hd (halfpath 1 p) = Incr
  apply (cases p)
   apply simp
  subgoal for move
    apply (cases move)
    by auto
  done

corollary even-reduce:
```

assumes $x \neq 0$
shows *height'* $x <$ *height'* $(2{*}x)$
proof –
from *assms* **have** $2{*}x \neq 0$ **by** *simp*
then obtain *path* **where** *path*: *make-moves path* $1 = 2{*}x$ *length path* $=$ *height'* $(2{*}x)$
using *heightD* **by** *blast*
then have *path* $\neq$ [] **by** *auto*
then obtain *m p* **where** *m-p*: *path* $= m \# p$
using *list.exhaust* **by** *blast*
let *?path'* $=$ *halfpath* (*make-move m 1*) *p*
obtain *b* **where** $*$: *make-move m* $1 = 2 + b$ $b < 2$
by (*cases m*; *auto simp add*: *jump'-def jump-def*)
hence *make-moves p* $(2 + b) = 2{*}x$
using *path m-p* **by** *auto*
hence *make-moves ?path'* $1 = x$
using *halfpath*[**where** $n{=}1$] **and** $*$ **by** *auto*
hence *height'* $x \leq$ *length ?path'* **by** *auto*
also have ... $\leq$ *length p*
by (*fact halfpath-shorter*)
also have ... $<$ *length path*
using *m-p* **by** *auto*
also have ... $=$ *height'* $(2{*}x)$
by (*fact path*(2))
finally show *height'* $x <$ *height'* $(2{*}x)$.
qed

corollary *twopow-reduce*:
assumes $x \neq 0$
shows *height'* $x <$ *height'* $(2\hat{}(Suc\ i) * x)$
proof (*induction i*)
case (*Suc i*)
have *height'* $x <$ *height'* $(2\hat{}Suc\ i * x)$ **by** *fact*
also have ... $<$ *height'* $(2 * (2\hat{}Suc\ i * x))$ **using** *assms* **by** (*intro even-reduce*) *auto*
finally show *?case* **by** (*simp add*: *ac-simps*)
qed (*simp add*: *even-reduce assms*)

0.5 The case for odd k

The strategy here is similar, but we disregard some high-order bits instead.

lemma *jump-multiplicity*:
multiplicity 2 (*jump' x*) $=$ *multiplicity 2 x*
unfolding *jump'-def jump-def*
by (*auto intro*: *multiplicity-sum-lt simp del*: *power-Suc*)

lemma *jump-mod*:
[*jump'* $(x \bmod 2\hat{}n) =$ *jump' x*] $(\bmod 2\hat{}n)$
proof (*cases* $x \bmod 2\hat{}n = x$)
case *True*
then show *?thesis* **by** *simp*
next
case *False*
show *?thesis*
proof (*cases* $x \bmod 2\hat{}n = 0$)
case *True*
then have $2\hat{}n$ *dvd x* **by** *auto*
moreover have $x \neq 0$ **using** ‹$x \bmod 2\hat{}n \neq x$› ‹$x \bmod 2\hat{}n = 0$› **by** *simp*
ultimately have *multiplicity 2 x* $\geq n$
by (*auto intro*: *multiplicity-geI*)
with *jump-multiplicity* **have** *multiplicity 2* (*jump' x*) $\geq n$ **by** *auto*
hence $2\hat{}n$ *dvd jump' x*
by (*simp add*: *multiplicity-dvd'*)

```
    then show ?thesis
      using ‹x mod 2^n = 0› by (auto simp add: cong-def)
  next
    case False
    then have x ≠ 0 by (meson mod-0)

    have x = x mod 2^n + 2^n * (x div 2^n)
      by simp
    also have multiplicity 2 ... = multiplicity 2 (x mod 2^n)
    proof (intro multiplicity-sum-lt)
      show x mod 2^n ≠ 0 by fact

      have x div 2^n ≠ 0
      proof
        assume x div 2^n = 0
        then have x mod 2^n = x by presburger
        with ‹x mod 2^n ≠ x› show False..
      qed
      thus 2^n * (x div 2^n) ≠ 0 by simp

      have multiplicity 2 (x mod 2^n) < n
        using ‹x mod 2^n ≠ 0›
        by (metis dvd-div-eq-0-iff mod-div-trivial odd-one multiplicity-lessI)

      also have n ≤ multiplicity 2 (2^n * (x div 2^n))
        using ‹2^n * (x div 2^n) ≠ 0› dvd-triv-left multiplicity-geI odd-one by blast

      finally show multiplicity 2 (x mod 2^n) < multiplicity 2 (2^n * (x div 2^n)).
    qed
    finally have [2^(multiplicity 2 (x mod 2^n) + 1) = 2^(multiplicity 2 x + 1)] (mod 2^n)
      by simp

    have [jump (x mod 2^n) = jump x] (mod 2^n)
      unfolding jump-def by (intro cong-add, simp, fact)
    with ‹x ≠ 0› and ‹x mod 2^n ≠ 0›
    show [jump' (x mod 2^n) = jump' x] (mod 2^n)
      by (auto simp add: jump'-def)
  qed
qed

lemma jump-mod-rollover:
  assumes x < 2^n and jump' x ≥ 2^n
  shows [jump' x = 2^(multiplicity 2 x)] (mod 2^n)
proof -
  have x ≠ 0
  proof
    assume x = 0
    then have jump' x = 0 by simp
    with assms have 0 < 2^n and 0 ≥ 2^n by auto
    thus False by auto
  qed

  let ?v = (2::nat)^(multiplicity 2 x)
  have jump' x = x + 2*?v unfolding jump'-def jump-def using ‹x ≠ 0› by simp
  moreover have ?v dvd x by (fact multiplicity-dvd)
  ultimately have ?v dvd jump' x by auto

  show ?thesis
  proof (cases 2^n dvd ?v)
    case False
    then have ?v dvd 2^n
      by (simp add: dvd-power-iff-le)
    then obtain m where m: 2^n = ?v * m by auto
```

```
    moreover from ‹?v dvd x› obtain x' where x': x = ?v * x' by auto
    moreover from ‹?v dvd jump' x› obtain j' where j': jump' x = ?v * j' by auto
    ultimately have x' < m and j' ≥ m using assms
      by auto (metis nat-mult-less-cancel-disj)

    moreover have j' = x' + 2
    proof -
      from ‹jump' x = x + 2*?v› have ?v * j' = ?v*x' + ?v*2
        using x' j' by simp
      also have ... = ?v * (x' + 2) by simp
      finally show j' = x' + 2
        by (simp only: mult-cancel1, simp)
    qed

    ultimately have j' = m ∨ j' = m + 1 by auto

    moreover have j' ≠ m
    proof
      assume j' = m
      then have 2^n = jump' x
        using j' m by simp
      hence multiplicity 2 x = multiplicity (2::nat) (2^n)
        by (simp add: jump-multiplicity)
      also have ... = n by simp
      finally have ?v = 2^n by simp
      with ‹¬ 2^n dvd ?v› show False by simp
    qed

    ultimately have j' = m + 1 by auto
    hence jump' x = 2^n + ?v using j' m by auto
    thus ?thesis unfolding cong-def by simp
  next
    case True
    with ‹?v dvd jump' x› show ?thesis
      unfolding cong-def by auto
  qed
qed
```

Instead of constructing the new path explicitly, we inductively prove that one exists. To show that the resulting path is strictly shorter, we maintain the invariant that the path was constructed by removing steps from the original. Thus, since the number we end up on is different, the path cannot be equal and must be shorter.

```
inductive skipping :: path ⇒ path ⇒ bool where
  SkipEmpty[intro]: skipping [] p |
  SkipCons[intro]: skipping p q ⟹ skipping (m # p) (m # q) |
  SkipSemicons[intro]: skipping p q ⟹ skipping p (m # q)

lemma length-skipping:
  assumes skipping p q
  shows length p ≤ length q
  using assms by induction auto

lemma skipping-equal:
  assumes skipping p q and length p = length q
  shows p = q
  using assms
proof induction
  case (SkipSemicons p q m)
  hence length p ≤ length q by (intro length-skipping)
  hence length p < length (m # q) by simp
  with ‹length p = length (m # q)› have False by simp
  thus ?case..
```

```
qed auto

lemma skipping-self[simp]: skipping a a
  by (induction a) auto

lemma skipping-semiprepend[simp]:
  skipping a (p @ a)
  by (induction p) auto

lemma skip-append[intro]:
  assumes skipping p q
  shows skipping (p @ a) (q @ a)
  using assms
  by induction auto

lemma skip-semiappend[intro]:
  assumes skipping p q
  shows skipping p (q @ a)
  using assms
  by induction auto

lemma modpath:
  assumes make-moves p 0 = x
  shows ∃p'. make-moves p' 0 = x mod 2^k ∧ skipping p' p
  using assms
proof (induction p arbitrary: x k rule: rev-induct)
  case Nil
  then show ?case by force
next
  case (snoc m p)
  then show ?case
  proof (cases x mod 2^k = 0)
    case True
    then show ?thesis
      by (auto intro: exI[of - []])
  next
    case False
    then show ?thesis
    proof (cases m)
      case Incr
      with snoc have x-eq: x = Suc (make-moves p 0) by simp
      with ‹x mod 2^k ≠ 0› have ∗: x mod 2^k = Suc (make-moves p 0 mod 2^k)
        using mod-Suc by auto
     from snoc.IH obtain p' where make-moves p' 0 = make-moves p 0 mod 2^k and skipping
p' p
        by auto
      with ∗ have make-moves (p' @ [Incr]) 0 = x mod 2^k
        by auto
      thus ?thesis
        apply (intro exI[of - p' @ [Incr]])
        using ‹skipping p' p› ‹m = Incr› by auto
    next
      case Jump
      with snoc have x-eq: x = jump' (make-moves p 0) (is x = jump' ?x') by simp
      hence x-cong: [jump' (?x' mod 2^k) = x] (mod 2^k) by (simp add: jump-mod)
      then consider (nowrap) jump' (?x' mod 2^k) = x mod 2^k
        | (wrap) jump' (?x' mod 2^k) > 2^k
        by (smt ‹x mod 2^k ≠ 0› cong-def linorder-neqE-nat mod-less mod-self)

      then show ?thesis
      proof cases
        case nowrap
        from snoc.IH obtain p' where make-moves p' 0 = ?x' mod 2^k and skipping p' p by
```

```
auto
    with nowrap have make-moves (p' @ [Jump]) 0 = x mod 2^k by simp
    then show ?thesis
      apply (intro exI[of - p' @ [Jump]])
      using ‹skipping p' p› ‹m = Jump› by auto
  next
    let ?k' = multiplicity 2 (?x' mod 2^k)
    case wrap
    then have jump-mod2k: [jump' (?x' mod 2^k) = 2^?k'] (mod 2^k)
      by (intro jump-mod-rollover; auto)
    with x-cong have [x = 2^?k'] (mod 2^k)
      by (metis cong-trans cong-sym)
    have x-mod: x mod 2^k = 2^?k'
    proof -
      from ‹[x = 2^?k'] (mod 2^k)› have x mod 2^k = 2^?k' mod 2^k
        unfolding cong-def .
      hence x mod 2^k = 0 ∨ x mod 2^k = 2^?k'
        by (simp add: exp-mod-exp)
      with ‹x mod 2^k ≠ 0› show x mod 2^k = 2^?k'
        by simp
    qed
    hence ?k' < k
      by (metis mod-less-divisor nat-power-less-imp-less nat-zero-less-power-iff power-eq-0-iff
power-zero-numeral)
    hence Suc ?k' ≤ k by simp
    hence dvd: (2::nat)^Suc ?k' dvd 2^k
      using le-imp-power-dvd by blast

    have ?x' mod 2^Suc ?k' = 2^?k'
    proof -
      have 2^?k' dvd ?x' mod 2^k by (fact multiplicity-dvd)
      with ‹Suc ?k' ≤ k› have 2^?k' dvd ?x'
        by (meson Suc-leD dvd-mod-imp-dvd dvd-power-le gcd-nat.refl)
      hence 2^?k' dvd ?x' mod 2^Suc ?k'
        by (simp add: dvd-mod)
      then obtain w where w: ?x' mod 2^Suc ?k' = 2^?k' * w by blast
      have ?x' mod 2^Suc ?k' < 2^Suc ?k' by simp
      with w have 2^?k' * w < 2^Suc ?k' by simp
      hence w < 2 by simp
      hence w = 0 ∨ w = 1 by auto
      moreover have w ≠ 0
      proof
        assume w = 0
        with w have 2^Suc ?k' dvd ?x' by auto
        with dvd have 2^Suc ?k' dvd ?x' mod 2^k
          by (simp add: dvd-mod)
        hence ?k' ≥ Suc ?k'
        proof (intro multiplicity-geI)
          show ?x' mod 2^k ≠ 0
            by (metis jump'-0 not-less0 wrap)
        qed auto
        thus False by simp
      qed

      ultimately have w = 1 by simp
      with w show ?x' mod 2^Suc ?k' = 2^?k' by simp
    qed
    moreover obtain p' where make-moves p' 0 = ?x' mod 2^Suc ?k' and skipping p' p
      using snoc.IH by blast
    ultimately have make-moves p' 0 = x mod 2^k using x-mod by simp
    then show ?thesis
      apply (intro exI[of - p'])
      using ‹skipping p' p› by auto
```

```
      qed
    qed
  qed
qed

lemma strip-zero:
  assumes make-moves p 0 = x and x ≠ 0
  shows ∃p'. make-moves p' 1 = x ∧ length p' < length p
  using assms
proof (induction p)
  case (Cons m p)
  show ?case
  proof (cases m)
    case Incr
    with Cons show ?thesis by (intro exI[of - p]; auto)
  next
    case Jump
    with Cons.prems have make-moves p 0 = x by auto
    with Cons show ?thesis by auto
  qed
qed auto

lemma odd-height:
  assumes odd k and k > 1
  shows height' (2^i) < height' (2^i * k)
proof -
  from assms have 2^i * k ≠ 0
    by (simp add: odd-pos)
  with heightD obtain p where make-moves p 1 = 2^i * k
    and **: length p = height' (2^i * k)
    by blast
  then have p-result: make-moves (Incr # p) 0 = 2^i * k by simp
  with modpath obtain p' where make-moves p' 0 = 2^i * k mod 2^Suc i
    and skipping: skipping p' (Incr # p)
    by blast
  hence p'-result: make-moves p' 0 = 2^i
    using ‹odd k›
    by (simp add: odd-iff-mod-2-eq-one)
  from skipping and length-skipping have length p' ≤ Suc (length p)
    by force
  also have length p' ≠ Suc (length p)
  proof
    assume length p' = Suc (length p)
    hence p' = Incr # p by (intro skipping-equal; auto simp add: skipping)
    hence make-moves p' 0 = make-moves (Incr # p) 0 by simp
    hence k = 1 using p-result p'-result by simp
    with ‹k > 1› show False by simp
  qed
  finally have *: length p' ≤ length p by simp

  have (2::nat)^i ≠ 0 by simp
  with p'-result obtain p''
    where p'': make-moves p'' 1 = 2^i
      and length p'' < length p'
    using strip-zero by blast
  with * have length p'' < length p by simp
  moreover from p'' have height' (2^i) ≤ length p'' by auto
  ultimately show height' (2^i) < height' (2^i * k) using ** by simp
qed

theorem problem5:
  assumes k ≥ 2
  shows height (2^i) < height (2^i * k)
```

```
proof -
  let ?n = multiplicity 2 k
  obtain k' where k = 2^?n * k' and odd k'
    using multiplicity-decompose' assms
    by (metis not-numeral-le-zero odd-one)
  then have height' (2^i) < height' (2^i * k)
  proof (cases ?n)
    case 0
    hence k = k' using ‹k = 2^?n * k'› by simp
    hence odd k using ‹odd k'› by simp
    thus height' (2^i) < height' (2^i * k)
      using assms by (auto intro: odd-height)
  next
    case (Suc n)
    have height' (2^i) < height' (2^Suc n * 2^i)
      by (intro twopow-reduce; auto)
    also have ... = height' (2^(?n + i))
      by (simp add: power-add Suc ac-simps)
    also have ... ≤ height' (2^(?n + i) * k')
    proof (cases k' = 1)
      case True
      then show ?thesis by simp
    next
      case False
      then have height' (2^(?n + i)) < height' (2^(?n + i) * k')
        using ‹odd k'› by (intro odd-height; auto simp add: Suc-lessI odd-pos)
      then show ?thesis by simp
    qed
    also have ... = height' (2^i * (2^?n * k'))
      by (simp add: ac-simps power-add)
    finally show height' (2^i) < height' (2^i * k)
      using ‹k = 2^?n * k'› by simp
  qed
  thus ?thesis by simp
qed

end
```