Fun With Functions

Tobias Nipkow

December 14, 2021

Abstract

This is a collection of cute puzzles of the form “Show that if a function satisfies the following constraints, it must be . . . ” Please add further examples to this collection!

Apart from the one about factorial, they all come from the delightful booklet by Terence Tao [1] but go back to Math Olympiads and similar events.

Please add further examples of this kind, either directly or by sending them to me. Let us make this a growing body of *fun*!

theory *FunWithFunctions* **imports** *Complex-Main* **begin**

See [1]. Was first brought to our attention by Herbert Ehler who provided a similar proof.

theorem *identity1*: **fixes** f :: *nat* ⇒ *nat*
assumes *fff*: $\bigwedge n.\ f(f(n)) < f(Suc(n))$
shows $f(n) = n$
proof −
  { **fix** m n **have** *key*: $n \le m \Longrightarrow n \le f(m)$
    proof(*induct n arbitrary*: m)
      case *0* **show** *?case* **by** *simp*
    next
      case (*Suc n*)
      hence $m \neq 0$ **by** *simp*
      then obtain k **where** [*simp*]: $m = Suc\ k$ **by** (*metis not0-implies-Suc*)
      have $n \le f(k)$ **using** *Suc* **by** *simp*
      hence $n \le f(f(k))$ **using** *Suc* **by** *simp*
      also have $\ldots < f(m)$ **using** *fff* **by** *simp*
      finally show *?case* **by** *simp*
    qed }
  hence $\bigwedge n.\ n \le f(n)$ **by** *simp*
  hence $\bigwedge n.\ f(n) < f(Suc\ n)$ **by**(*metis fff order-le-less-trans*)
  hence $f(n) < n{+}1$ **by** (*metis fff lift-Suc-mono-less-iff*[*of f*] *Suc-eq-plus1*)
  with ‹$n \le f(n)$› **show** $f\ n = n$ **by** *arith*
qed

See [1]. Possible extension: Should also hold if the range of f is the reals!

```
lemma identity2: fixes f :: nat ⇒ nat
assumes f(k) = k and k ≥ 2
and f-times: ⋀m n. f(m∗n) = f(m)∗f(n)
and f-mono: ⋀m n. m<n ⟹ f m < f n
shows f(n) = n
proof −
  have 0: f(0) = 0
   by (metis f-mono f-times mult-1-right mult-is-0 nat-less-le nat-mult-eq-cancel-disj
not-less-eq)
  have 1: f(1) = 1
   by (metis f-mono f-times gr-implies-not0 mult-eq-self-implies-10 nat-mult-1-right
zero-less-one)
  have 2: f 2 = 2
  proof −
    have 2 + (k − 2) = k using ‹k ≥ 2› by arith
    hence f(2) ≤ 2
      using mono-nat-linear-lb[of f 2 k − 2,OF f-mono] ‹f k = k›
      by simp
    thus f 2 = 2 using 1 f-mono[of 1 2] by arith
  qed
  show ?thesis
  proof(induct rule:less-induct)
    case (less i)
    show ?case
    proof cases
      assume i≤1 thus ?case using 0 1 by (auto simp add:le-Suc-eq)
    next
      assume ~i≤1
      show ?case
      proof cases
        assume i mod 2 = 0
        hence ∃k. i=2∗k by arith
        then obtain k where i = 2∗k ..
        hence 0 < k and k<i using ‹~i≤1› by arith+
        hence f(k) = k using less(1) by blast
        thus f(i) = i using ‹i = 2∗k› by(simp add:f-times 2)
      next
        assume i mod 2 ≠ 0
        hence ∃k. i=2∗k+1 by arith
        then obtain k where i = 2∗k+1 ..
        hence 0<k and k+1<i using ‹~i≤1› by arith+
        have 2∗k < f(2∗k+1)
        proof −
          have 2∗k = 2∗f(k) using less(1) ‹i=2∗k+1› by simp
          also have . . . = f(2∗k) by(simp add:f-times 2)
          also have . . . < f(2∗k+1) using f-mono[of 2∗k 2∗k+1] by simp
          finally show ?thesis .
        qed
```

```
      moreover
      have f(2*k+1) < 2*(k+1)
      proof -
        have f(2*k+1) < f(2*k+2) using f-mono[of 2*k+1 2*k+2] by simp
        also have ... = f(2*(k+1)) by simp
        also have ... = 2*f(k+1) by(simp only:f-times 2)
        also have f(k+1) = k+1 using less(1) ‹i=2*k+1› ‹~i≤1› by simp
        finally show ?thesis .
      qed
      ultimately show f(i) = i using ‹i = 2*k+1› by arith
    qed
  qed
 qed
qed
```

One more from Tao's booklet. If f is also assumed to be continuous, $f\ x = x + 1$ holds for all reals, not only rationals. Extend the proof!

```
theorem plus1:
fixes f :: real ⇒ real
assumes 0: f 0 = 1 and f-add: ⋀x y. f(x+y+1) = f x + f y

assumes r : ℚ shows f(r) = r + 1
proof -
 { fix i have f(of-int i) = of-int i + 1
   proof (induct i rule: int-induct [where k=0])
     case base show ?case using 0 by simp
   next
     case (step1 i)
     have f(of-int (i+1)) = f(of-int i + 0 + 1) by simp
     also have ... = f(of-int i) + f 0 by(rule f-add)
     also have ... = of-int (i+1) + 1 using step1 0 by simp
     finally show ?case .
   next
     case (step2 i)
     have f(of-int i) = f(of-int (i - 1) + 0 + 1) by simp
     also have ... = f(of-int (i - 1)) + f 0 by(rule f-add)
     also have ... = f(of-int (i - 1)) + 1  using 0 by simp
     finally show ?case using step2 by simp
   qed }
 note f-int = this
 { fix n r have f(of-int (Suc n)*r + of-int n) = of-int (Suc n) * f r
   proof(induct n)
     case 0 show ?case by simp
   next
     case (Suc n)
     have of-int (Suc(Suc n))*r + of-int (Suc n) =
          r + (of-int (Suc n)*r + of-int n) + 1 (is ?a = ?b)
       by(simp add: field-simps)
     hence f ?a = f ?b
```

```
      by presburger
    also have ... = f r + f(of-int (Suc n)∗r + of-int n) by(rule f-add)
    also have ... = f r + of-int (Suc n) ∗ f r by(simp only:Suc)
    finally show ?case by(simp add: field-simps)
  qed }
  note 1 = this
  { fix n::nat and r assume n≠0
    have f(of-int (n)∗r + of-int (n − 1)) = of-int (n) ∗ f r
    proof(cases n)
      case 0 thus ?thesis using ‹n≠0› by simp
    next
      case Suc thus ?thesis using ‹n≠0› using 1 by auto
    qed }
  note f-mult = this
  from ‹r:ℚ› obtain i::int and n::nat where r: r = of-int i/of-int n and n≠0
    by(fastforce simp:Rats-eq-int-div-nat)
  have of-int (n) ∗ f(of-int i / of-int n) = f(of-int i + of-int (n − 1))
    using ‹n≠0›
     by (metis (no-types, opaque-lifting) f-mult mult.commute nonzero-divide-eq-eq
of-int-of-nat-eq of-nat-0-eq-iff)
  also have ... = f(of-int (i + int n − 1)) using ‹n≠0›[simplified]
    by (metis One-nat-def Suc-leI of-nat-1 add-diff-eq of-int-add of-nat-diff)
  also have ... = of-int (i + int n − 1) + 1 by(rule f-int)
  also have ... = of-int i + of-int n by arith
  finally show ?thesis using ‹n≠0› unfolding r by (simp add:field-simps)
qed
```

The only total model of a naive recursion equation of factorial on integers is 0 for all negative arguments. Probably folklore.

```
theorem ifac-neg0: fixes ifac :: int ⇒ int
assumes ifac-rec: ⋀i. ifac i = (if i=0 then 1 else i∗ifac(i − 1))
shows i<0 ⟹ ifac i = 0
proof(rule ccontr)
  assume 0: i<0 ifac i ≠ 0
  { fix j assume j ≤ i
    have ifac j ≠ 0
      apply(rule int-le-induct[OF ‹j≤i›])
       apply(rule ‹ifac i ≠ 0›)
      apply (metis ‹i<0› ifac-rec linorder-not-le mult-eq-0-iff)
      done
  } note below0 = this
  { fix j assume j<i
    have 1 < −j using ‹j<i› ‹i<0› by arith
    have ifac(j − 1) ≠ 0 using ‹j<i› by(simp add: below0)
    then have |ifac (j − 1)| < (−j) ∗ |ifac (j − 1)| using ‹j<i›
      mult-le-less-imp-less[OF order-refl[of abs(ifac(j − 1))] ‹1 < −j›]
      by(simp add:mult.commute)
    hence abs(ifac(j − 1)) < abs(ifac j)
      using ‹1 < −j› by(simp add: ifac-rec[of j] abs-mult)
```

```
  } note not-wf = this
  let ?f = %j. nat(abs(ifac(i − int(j+1))))
  obtain k where ¬ ?f (Suc k) < ?f k
    using wf-no-infinite-down-chainE[OF wf-less, of ?f] by blast
  moreover have i − int (k + 1) − 1 = i − int (Suc k + 1) by arith
  ultimately show False using not-wf[of i − int(k+1)]
    by (simp only:) arith
qed

end
```

References

[1] Terence Tao. *Solving Mathematical Problems.* Oxford University Press, 2006.