Implementing the Goodstein Function in λ-Calculus

Bertram Felgenhauer

March 17, 2025

Abstract

In this formalization, we develop an implementation of the Goodstein function $\mathcal{G}$ in plain λ-calculus, linked to a concise, self-contained specification. The implementation works on a Church-encoded representation of countable ordinals. The initial conversion to hereditary base 2 is not covered, but the material is sufficient to compute the particular value $\mathcal{G}(16)$, and easily extends to other fixed arguments.

Contents

1 Introduction

Given a number n and a base b, we can write n in *hereditary base* b, which results from writing n in base b, and then each exponent in hereditary base b again. For example, 7 in hereditary base 3 is $3^1 \cdot 2 + 1$. Given the hereditary base b representation of n, we can reinterpret it in base $b + 1$ by replacing all occurrences of b by $b + 1$.

The Goodstein sequence starting at n in base 2 is obtained by iteratively taking a number in hereditary base b, reinterpreting it in base $b + 1$, and subtracting 1. The next step is the same with b incremented by 1, and so on. So starting for example at 4, we compute

$$\begin{aligned} 4 = 2^{2^1} &\to 3^{3^1} - 1 = 26 \\ 26 = 3^2 \cdot 2 + 3^1 \cdot 2 + 2 &\to 4^2 \cdot 2 + 4^1 \cdot 2 + 1 \cdot 2 - 1 = 41 \\ 41 = 4^2 \cdot 2 + 4^1 \cdot 2 + 1 &\to 5^2 \cdot 2 + 5^1 \cdot 2 + 1 - 1 = 60 \end{aligned}$$

and so on. We stop when we reach 0. Goodstein's theorem states that this process always terminates [3]. This result is independent of Peano Arithmetic, and is intimately connected to countable ordinals and the slow growing hierarchy (e.g., the Hardy function) [2]. The length of the resulting sequence is the Goodstein function, denoted by $\mathcal{G}(n)$. For example, $\mathcal{G}(3) = 6$.

For this formalization, we are interested in implementing the Goodstein function in λ-calculus. More concretely, we want to define the value $\mathcal{G}(16)$ (which is huge; for example, it exceeds Graham's number), in order to bound its Kolmogorov complexity. Our concrete measure of Kolmogorov complexity is the program length in the Binary Lambda Calculus [4, 5]. It turns out that we can define $\mathcal{G}(16)$ as follows, giving a complexity bound of 195 bits.

$$\begin{aligned} \mathit{exp}\omega &= (\lambda z\, s\, l.\, n\, s\, (\lambda x\, z.\, l\, (\lambda n.\, n\, x\, z))\, (\lambda f\, z.\, l\, (\lambda n.\, f\, n\, z))\, z) \\ \mathit{goodstein} &= (\lambda n\, c.\, n \\ &\qquad (\lambda x.\, x) \\ &\qquad (\lambda n\, m.\, n\, (\lambda f\, x.\, m\, f\, (f\, x))) \\ &\qquad (\lambda f\, m.\, f\, (\lambda f\, x.\, m\, f\, (f\, (f\, x)))\, m) \\ &\qquad c) \\ \mathcal{G}_{16} &= (\lambda e.\, \mathit{goodstein}\, (e\, (e\, (e\, (e\, (\lambda z\, s\, l.\, z)))))\, (\lambda x.\, x))\, \mathit{exp}\omega \end{aligned}$$

We rely on a shallow embedding of the λ-calculus throughout the formalization, so it turns out that we cannot quite prove this claim in Isabelle/HOL;

the expression for $\mathcal{G}_{16}$ cannot be typed. However, we can prove that the building blocks *expω* and *goodstein* work correctly in the sense that

- $exp\omega^4\ (\lambda z\ s\ l.\ z)$ is the hereditary base 2 representation of 16; and
- *goodstein* $c\,n$ computes the length of a Goodstein sequence given that the hereditary base $c{+}1$ representation of the c-th value in the sequence is equal to n.

The remaining steps are easily verified by a human.

Contributions. Our main contributions are a concise specification of the Goodstein function, another proof of Goodstein's theorem, and establishing the connection to λ-calculus as already outlined.

Related work. There is already a formalization of Goodstein's theorem in the AFP entry on nested multisets [1], which comes with a formalization of ordinal arithmetic. Our focus is different, since our goal is to obtain an implementation of the Goodstein function in λ-calculus. Most notably, the intermediate type *Ord* that we use to represent ordinal numbers has far more structure than the ordinals themselves. In particular it can represent arbitrary trees; if we were to compute $\omega + 1$, $1 + \omega$ and ω on this type, we would get three different results. However, we will use the operations such that $1 + \omega$ is never computed, keeping the connection to countable ordinals intact. Proving this is a large, albeit hidden, part of our formalization.

Acknowledgement. John Tromp raised the question of a concise λ-calculus term computing $\mathcal{G}(16)$. He also provided feedback on a draft version of this document.

2 Specification

theory *Goodstein-Lambda*
  imports *Main*
begin

2.1 Hereditary base representation

We define a data type of trees and an evaluation function that sums siblings and exponentiates with respect to the given base on nesting.

datatype *C = C (unC: C list)*

fun *evalC* **where**
  evalC b (C []) = 0
| *evalC b (C (x # xs)) = b^evalC b x + evalC b (C xs)*

value *evalC 2* (*C* []) — 0
value *evalC 2* (*C* [*C* []]) — $2^0 = 1$
value *evalC 2* (*C* [*C* [*C* []]]) — $2^1 = 2$
value *evalC 2* (*C* [*C* [], *C* []]) — $2^0 + 2^0 = 2^0 \cdot 2 = 2$; not in hereditary base 2

The hereditary base representation is characterized as trees (i.e., nested lists) whose lists have monotonically increasing evaluations, with fewer than b repetitions for each value. We will show later that this representation is unique.

inductive-set *hbase* **for** *b* **where**
  $C\ [] \in hbase\ b$
| $i \neq 0 \Longrightarrow i < b \Longrightarrow n \in hbase\ b \Longrightarrow$
   $C\ ms \in hbase\ b \Longrightarrow (\bigwedge m'.\ m' \in set\ ms \Longrightarrow evalC\ b\ n < evalC\ b\ m') \Longrightarrow$
   $C\ (replicate\ i\ n\ @\ ms) \in hbase\ b$

We can convert to and from natural numbers as follows.

definition *H2N* **where**
  $H2N\ b\ n = evalC\ b\ n$

As we will show later, *H2N b* restricted to *hbase n* is bijective if $2 \leq b$, so we can convert from natural numbers by taking the inverse.

definition *N2H* **where**
  $N2H\ b\ n = inv\text{-}into\ (hbase\ b)\ (H2N\ b)\ n$

2.2 The Goodstein function

We define a function that computes the length of the Goodstein sequence whose c-th element is $g_c = n$. Termination will be shown later, thereby establishing Goodstein's theorem.

function (*sequential*) *goodstein* :: *nat* ⇒ *nat* ⇒ *nat* **where**
  $goodstein\ 0\ n = 0$
  — we start counting at 1; also note that the initial base is $c + 1$ and
  — hereditary base 1 makes no sense, so we have to avoid this case
| $goodstein\ c\ 0 = c$
| $goodstein\ c\ n = goodstein\ (c+1)\ (H2N\ (c+2)\ (N2H\ (c+1)\ n) - 1)$
  by *pat-completeness auto*

abbreviation $\mathcal{G}$ **where**
  $\mathcal{G}\ n \equiv goodstein\ (Suc\ 0)\ n$

3 Ordinals

The following type contains countable ordinals, by the usual case distinction into 0, successor ordinal, or limit ordinal; limit ordinals are given by their fundamental sequence. Hereditary base b representations carry over to such ordinals by replacing each occurrence of the base by ω.

datatype $Ord = Z \mid S\ Ord \mid L\ nat \Rightarrow Ord$

Note that the following arithmetic operations are not correct for all ordinals. However, they will only be used in cases where they actually correspond to the ordinal arithmetic operations.

primrec *addO* **where**
  $addO\ n\ Z = n$
$\mid addO\ n\ (S\ m) = S\ (addO\ n\ m)$
$\mid addO\ n\ (L\ f) = L\ (\lambda i.\ addO\ n\ (f\ i))$

primrec *mulO* **where**
  $mulO\ n\ Z = Z$
$\mid mulO\ n\ (S\ m) = addO\ (mulO\ n\ m)\ n$
$\mid mulO\ n\ (L\ f) = L\ (\lambda i.\ mulO\ n\ (f\ i))$

definition ω **where**
  $\omega = L\ (\lambda n.\ (S\ \widehat{}\widehat{}\ n)\ Z)$

primrec *expω* **where**
  $exp\omega\ Z = S\ Z$
$\mid exp\omega\ (S\ n) = mulO\ (exp\omega\ n)\ \omega$
$\mid exp\omega\ (L\ f) = L\ (\lambda i.\ exp\omega\ (f\ i))$

3.1 Evaluation

Evaluating an ordinal number at base b is accomplished by taking the b-th element of all fundamental sequences and interpreting zero and successor over the natural numbers.

primrec *evalO* **where**
  $evalO\ b\ Z = 0$
$\mid evalO\ b\ (S\ n) = Suc\ (evalO\ b\ n)$
$\mid evalO\ b\ (L\ f) = evalO\ b\ (f\ b)$

3.2 Goodstein function and sequence

We can define the Goodstein function very easily, but proving correctness will take a while.

primrec *goodsteinO* **where**
  $goodsteinO\ c\ Z = c$
$\mid goodsteinO\ c\ (S\ n) = goodsteinO\ (c+1)\ n$
$\mid goodsteinO\ c\ (L\ f) = goodsteinO\ c\ (f\ (c+2))$

primrec *stepO* **where**
  $stepO\ c\ Z = Z$
$\mid stepO\ c\ (S\ n) = n$

| *stepO c (L f) = stepO c (f (c+2))*

We can compute a few values of the Goodstein sequence starting at 4.

definition *g4O* **where**
g4O n = fold stepO [1..<Suc n] ((expω ^^ 3) Z)

value *map (λn. evalO (n+2) (g4O n)) [0..<10]*
— [*4*, *26*, *41*, *60*, *83*, *109*, *139*, *173*, *211*, *253*]

3.3 Properties of evaluation

lemma *evalO-addO* [*simp*]:
evalO b (addO n m) = evalO b n + evalO b m
by (*induct m*) *auto*

lemma *evalO-mulO* [*simp*]:
*evalO b (mulO n m) = evalO b n * evalO b m*
by (*induct m*) *auto*

lemma *evalO-n* [*simp*]:
evalO b ((S ^^ n) Z) = n
by (*induct n*) *auto*

lemma *evalO-ω* [*simp*]:
evalO b ω = b
by (*auto simp*: *ω-def*)

lemma *evalO-expω* [*simp*]:
evalO b (expω n) = b^(evalO b n)
by (*induct n*) *auto*

Note that evaluation is useful for proving that *Ord* values are distinct:

notepad begin
have *addO n (expω m) ≠ n* **for** *n m* **by** (*auto dest*: *arg-cong*[*of - - evalO 1*])
end

3.4 Arithmetic properties

lemma *addO-Z* [*simp*]:
addO Z n = n
by (*induct n*) *auto*

lemma *addO-assoc* [*simp*]:
addO n (addO m p) = addO (addO n m) p
by (*induct p*) *auto*

lemma *mul0-distrib* [*simp*]:
mulO n (addO p q) = addO (mulO n p) (mulO n q)
by (*induct q*) *auto*

lemma *mulO-assoc* [*simp*]:
  mulO n (*mulO m p*) = *mulO* (*mulO n m*) *p*
  by (*induct p*) *auto*

lemma *expω-addO* [*simp*]:
  expω (*addO n m*) = *mulO* (*expω n*) (*expω m*)
  by (*induct m*) *auto*

4 Cantor normal form

The previously introduced tree type C can be used to represent Cantor normal forms; they are trees (evaluated at base ω) such that siblings are in non-decreasing order. One can think of this as hereditary base ω. The plan is to mirror selected operations on ordinals in Cantor normal forms.

4.1 Conversion to and from the ordinal type *Ord*

fun *C2O* **where**
  C2O (*C* []) = *Z*
| *C2O* (*C* (*n* # *ns*)) = *addO* (*C2O* (*C ns*)) (*expω* (*C2O n*))

definition *O2C* **where**
  O2C = *inv C2O*

We show that *C2O* is injective, meaning the inverse is unique.

lemma *addO-expω-inj*:
  assumes *addO n* (*expω m*) = *addO n′* (*expω m′*)
  shows *n* = *n′* **and** *m* = *m′*
proof −
  have *addO n* (*expω m*) = *addO n′* (*expω m′*) ⟹ *n* = *n′*
    by (*induct m arbitrary*: *m′*; *case-tac m′*;
      force simp: *ω-def dest*!: *fun-cong*[*of* - - *1*])
  moreover have *addO n* (*expω m*) = *addO n* (*expω m′*) ⟹ *m* = *m′*
    apply (*induct m arbitrary*: *n m′*; *case-tac m′*)
    apply (*auto 0 3 simp*: *ω-def intro*: *rangeI*
      dest: *arg-cong*[*of* - - *evalO 1*] *fun-cong*[*of* - - *0*] *fun-cong*[*of* - - *1*])[*8*]
    by *simp* (*meson ext rangeI*)
  ultimately show *n* = *n′* **and** *m* = *m′* **using** *assms* **by** *simp-all*
qed

lemma *C2O-inj*:
  C2O n = *C2O m* ⟹ *n* = *m*
  by (*induct n arbitrary*: *m rule*: *C2O.induct*; *case-tac m rule*: *C2O.cases*)
    (*auto dest*: *addO-expω-inj arg-cong*[*of* - - *evalO 1*])

lemma *O2C-C2O* [*simp*]:
  O2C (*C2O n*) = *n*

by (*auto intro*!: *inv-f-f simp*: *O2C-def inj-def C2O-inj*)

lemma *O2C-Z* [*simp*]:
O2C Z = *C* []
using *O2C-C2O*[*of C* [], *unfolded C2O.simps*] **.**

lemma *C2O-replicate*:
C2O (*C* (*replicate i n*)) = *mulO* (*expω* (*C2O n*)) ((*S* ^^ *i*) *Z*)
by (*induct i*) *auto*

lemma *C2O-app*:
C2O (*C* (*xs* @ *ys*)) = *addO* (*C2O* (*C ys*)) (*C2O* (*C xs*))
by (*induct xs arbitrary*: *ys*) *auto*

4.2 Evaluation

lemma *evalC-def'*:
evalC b n = *evalO b* (*C2O n*)
by (*induct n rule*: *C2O.induct*) *auto*

lemma *evalC-app* [*simp*]:
evalC b (*C* (*ns* @ *ms*)) = *evalC b* (*C ns*) + *evalC b* (*C ms*)
by (*induct ns*) *auto*

lemma *evalC-replicate* [*simp*]:
evalC b (*C* (*replicate c n*)) = *c* ∗ *evalC b* (*C* [*n*])
by (*induct c*) *auto*

4.3 Transfer of the *Ord* induction principle to *C*

fun *funC* **where** — *funC* computes the fundamental sequence on *C*
funC (*C* []) = (λ*i*. [*C* []])
| *funC* (*C* (*C* [] # *ns*)) = (λ*i*. *replicate i* (*C ns*))
| *funC* (*C* (*n* # *ns*)) = (λ*i*. [*C* (*funC n i* @ *ns*)])

lemma *C2O-cons*:
C2O (*C* (*n* # *ns*)) =
(*if n* = *C* [] *then S* (*C2O* (*C ns*)) *else L* (λ*i*. *C2O* (*C* (*funC n i* @ *ns*))))
by (*induct n arbitrary*: *ns rule*: *funC.induct*)
(*simp-all add*: *ω-def C2O-replicate C2O-app flip*: *expω-addO*)

lemma *C-Ord-induct*:
assumes *P* (*C* [])
and ⋀*ns*. *P* (*C ns*) ⟹ *P* (*C* (*C* [] # *ns*))
and ⋀*n ns ms*. (⋀*i*. *P* (*C* (*funC* (*C* (*n* # *ns*)) *i* @ *ms*))) ⟹
P (*C* (*C* (*n* # *ns*) # *ms*))
shows *P n*
proof −
have ∀ *n*. *C2O n* = *m* ⟶ *P n* **for** *m*
by (*induct m*; *intro allI*; *case-tac n rule*: *funC.cases*)

(*auto simp*: *C2O-cons simp del*: *C2O.simps*(*2*) *intro*: *assms*)
then show *?thesis* **by** *simp*
qed

4.4 Goodstein function and sequence on *C*

function (*domintros*) *goodsteinC* **where**
goodsteinC c (*C* []) = *c*
| *goodsteinC c* (*C* (*C* [] # *ns*)) = *goodsteinC* (*c*+*1*) (*C ns*)
| *goodsteinC c* (*C* (*C* (*n* # *ns*) # *ms*)) =
goodsteinC c (*C* (*funC* (*C* (*n* # *ns*)) (*c*+*2*) @ *ms*))
by *pat-completeness auto*

termination
proof −
have *goodsteinC-dom* (*c*, *n*) **for** *c n*
by (*induct n arbitrary*: *c rule*: *C-Ord-induct*) (*auto intro*: *goodsteinC.domintros*)
then show *?thesis* **by** *simp*
qed

lemma *goodsteinC-def′*:
goodsteinC c n = *goodsteinO c* (*C2O n*)
by (*induct c n rule*: *goodsteinC.induct*) (*simp-all add*: *C2O-cons del*: *C2O.simps*(*2*))

function (*domintros*) *stepC* **where**
stepC c (*C* []) = *C* []
| *stepC c* (*C* (*C* [] # *ns*)) = *C ns*
| *stepC c* (*C* (*C* (*n* # *ns*) # *ms*)) =
stepC c (*C* (*funC* (*C* (*n* # *ns*)) (*Suc* (*Suc c*)) @ *ms*))
by *pat-completeness auto*

termination
proof −
have *stepC-dom* (*c*, *n*) **for** *c n*
by (*induct n arbitrary*: *c rule*: *C-Ord-induct*) (*auto intro*: *stepC.domintros*)
then show *?thesis* **by** *simp*
qed

definition *g4C* **where**
g4C n = *fold stepC* [*1*..<*Suc n*] (*C* [*C* [*C* [*C* []]]])

value *map* (λ*n*. *evalC* (*n*+*2*) (*g4C n*)) [*0*..<*10*]
— [*4*, *26*, *41*, *60*, *83*, *109*, *139*, *173*, *211*, *253*]

4.5 Properties

lemma *stepC-def′*:
stepC c n = *O2C* (*stepO c* (*C2O n*))
by (*induct c n rule*: *stepC.induct*) (*simp-all add*: *C2O-cons del*: *C2O.simps*(*2*))

lemma *funC-ne* [*simp*]:
  funC m (*Suc n*) ≠ []
  by (*cases m rule*: *funC.cases*) *simp-all*

lemma *evalC-funC* [*simp*]:
  evalC b (*C* (*funC n b*)) = *evalC b* (*C* [*n*])
  by (*induct n rule*: *funC.induct*) *simp-all*

lemma *stepC-app* [*simp*]:
  n ≠ *C* [] ⟹ *stepC c* (*C* (*unC n* @ *ns*)) = *C* (*unC* (*stepC c n*) @ *ns*)
  by (*induct n arbitrary*: *ns rule*: *stepC.induct*) *simp-all*

lemma *stepC-cons* [*simp*]:
  ns ≠ [] ⟹ *stepC c* (*C* (*n* # *ns*)) = *C* (*unC* (*stepC c* (*C* [*n*])) @ *ns*)
  using *stepC-app*[*of C*[*n*] *c ns*] **by** *simp*

lemma *stepC-dec*:
  n ≠ *C* [] ⟹ *Suc* (*evalC* (*Suc* (*Suc c*)) (*stepC c n*)) = *evalC* (*Suc* (*Suc c*)) *n*
  by (*induct c n rule*: *stepC.induct*) *simp-all*

lemma *stepC-dec′*:
  n ≠ *C* [] ⟹ *evalC* (*c*+*3*) (*stepC c n*) < *evalC* (*c*+*3*) *n*
proof (*induct c n rule*: *stepC.induct*)
  case (*3 c n ns ms*)
  have *evalC* (*c*+*3*) (*C* (*funC* (*C* (*n* # *ns*)) (*Suc* (*Suc c*)))) ≤
    (*c*+*3*) ^ ((*c*+*3*) ^ *evalC* (*c*+*3*) *n* + *evalC* (*c*+*3*) (*C ns*))
   by (*induct n rule*: *funC.induct*) (*simp-all add*: *distrib-right*)
  then show *?case* **using** *3* **by** *simp*
qed *simp-all*

5 Hereditary base *b* representation

We now turn to properties of the *hbase b* subset of trees.

5.1 Uniqueness

We show uniqueness of the hereditary base representation by showing that *evalC b* restricted to *hbase b* is injective.

lemma *hbaseI2*:
  i < *b* ⟹ *n* ∈ *hbase b* ⟹ *C m* ∈ *hbase b* ⟹
   (⋀*m′*. *m′* ∈ *set m* ⟹ *evalC b n* < *evalC b m′*) ⟹
   C (*replicate i n* @ *m*) ∈ *hbase b*
  by (*cases i*) (*auto intro*: *hbase.intros simp del*: *replicate.simps*(*2*))

lemmas *hbase-singletonI* =
  hbase.intros(*2*)[*of 1 Suc* (*Suc b*) **for** *b*, *OF* - - - *hbase.intros*(*1*), *simplified*]

lemma *hbase-hd*:
  $C\ ns \in hbase\ b \Longrightarrow ns \neq [] \Longrightarrow hd\ ns \in hbase\ b$
  by (*cases rule*: *hbase.cases*) *auto*

lemmas *hbase-hd'* [*dest*] = *hbase-hd*[*of n # ns* **for** *n ns*, *simplified*]

lemma *hbase-tl*:
  $C\ ns \in hbase\ b \Longrightarrow ns \neq [] \Longrightarrow C\ (tl\ ns) \in hbase\ b$
  by (*cases C ns b rule*: *hbase.cases*) (*auto intro*: *hbaseI2*)

lemmas *hbase-tl'* [*dest*] = *hbase-tl*[*of n # ns* **for** *n ns*, *simplified*]

lemma *hbase-elt* [*dest*]:
  $C\ ns \in hbase\ b \Longrightarrow n \in set\ ns \Longrightarrow n \in hbase\ b$
  by (*induct ns*) *auto*

lemma *evalC-sum-list*:
  evalC b (*C ns*) = *sum-list* (*map* (λ*n*. *b*^*evalC b n*) *ns*)
  by (*induct ns*) *auto*

lemma *sum-list-replicate*:
  sum-list (*replicate n x*) = *n* ∗ *x*
  by (*induct n*) *auto*

lemma *base-red*:
  fixes *b* :: *nat*
  assumes *n*: $\bigwedge n'.\ n' \in set\ ns \Longrightarrow n < n'\ i < b\ i \neq 0$
  and *m*: $\bigwedge m'.\ m' \in set\ ms \Longrightarrow m < m'\ j < b\ j \neq 0$
  and *s*: *i* ∗ *b*^*n* + *sum-list* (*map* (λ*n*. *b*^*n*) *ns*) = *j* ∗ *b*^*m* + *sum-list* (*map* (λ*n*. *b*^*n*) *ms*)
  shows $i = j \wedge n = m$
  using *n*(*1*) *m*(*1*) *s*
proof (*induct n arbitrary*: *m ns ms*)
  { **fix** *ns ms* :: *nat list* **and** *i j m* :: *nat*
    assume *n'*: $\bigwedge n'.\ n' \in set\ ns \Longrightarrow 0 < n'\ i < b\ i \neq 0$
    assume *m'*: $\bigwedge m'.\ m' \in set\ ms \Longrightarrow m < m'\ j < b\ j \neq 0$
    assume *s'*: *i* ∗ *b*^*0* + *sum-list* (*map* (λ*n*. *b*^*n*) *ns*) = *j* ∗ *b*^*m* + *sum-list* (*map* (λ*n*. *b*^*n*) *ms*)
    obtain *x* **where** [*simp*]: *sum-list* (*map* ((^) *b*) *ns*) = *x*∗*b*
      using *n'*(*1*)
      by (*intro that*[*of sum-list* (*map* (λ*n*. *b*^(*n*−*1*)) *ns*)])
        (*simp add*: *ac-simps flip*: *sum-list-const-mult power-Suc cong*: *map-cong*)
    obtain *y* **where** [*simp*]: *sum-list* (*map* ((^) *b*) *ms*) = *y*∗*b*
      using *order.strict-trans1*[*OF le0 m'*(*1*)]
      by (*intro that*[*of sum-list* (*map* (λ*n*. *b*^(*n*−*1*)) *ms*)])
        (*simp add*: *ac-simps flip*: *sum-list-const-mult power-Suc cong*: *map-cong*)
    have [*simp*]: *m* = *0*
      using *s' n'*(*2*,*3*)
      by (*cases m*, *simp-all*)

(*metis Groups.mult-ac*(*2*) *Groups.mult-ac*(*3*) *Suc-pred div-less mod-div-mult-eq mod-mult-self2 mod-mult-self2-is-0 mult-zero-right nat.simps*(*3*))
have $i = j \wedge 0 = m$ **using** s' $n'(2,3)$ $m'(2,3)$
by *simp* (*metis div-less mod-div-mult-eq mod-mult-self1*)
} **note** *BASE* = *this*
{
case *0* **show** *?case* **by** (*rule BASE*; *fact*)
next
case (*Suc n m*$'$)
have $j = i \wedge 0 = Suc\ n$ **if** $m' = 0$ **using** *Suc*(*2*−*4*)
by (*intro BASE*[*of ms j ns Suc n i*]) (*simp-all add*: *ac-simps that n*(*2*,*3*) *m*(*2*,*3*))
then obtain *m* **where** m' [*simp*]: $m' = Suc\ m$
by (*cases* m') *auto*
obtain ns' **where** [*simp*]: $ns = map\ Suc\ ns'$ $\bigwedge n'.\ n' \in set\ ns' \Longrightarrow n < n'$
using *Suc*(*2*) *less-trans*[*OF zero-less-Suc Suc*(*2*)]
by (*intro that*[*of map* ($\lambda n.\ n-1$) *ns*]; *force cong*: *map-cong*)
obtain ms' **where** [*simp*]: $ms = map\ Suc\ ms'$ $\bigwedge m'.\ m' \in set\ ms' \Longrightarrow m < m'$
using *Suc*(*3*)[*unfolded m*$'$] *less-trans*[*OF zero-less-Suc Suc*(*3*)[*unfolded m*$'$]]
by (*intro that*[*of map* ($\lambda n.\ n-1$) *ms*]; *force cong*: *map-cong*)
have ∗: $b * x = b * y \Longrightarrow x = y$ **for** *x y* **using** *n*(*2*) **by** *simp*
have $i = j \wedge n = m$
proof (*rule Suc*(*1*)[*of map* ($\lambda n.\ n-1$) *ns map* ($\lambda n.\ n-1$) *ms m*, *OF* - - ∗], *goal-cases*)
case *3* **show** *?case* **using** *Suc*(*4*) **unfolding** *add-mult-distrib2*
by (*simp add*: *comp-def ac-simps flip*: *sum-list-const-mult*)
qed *simp-all*
then show *?case* **by** *simp*
}
qed

lemma *evalC-inj-on-hbase*:
$n \in hbase\ b \Longrightarrow m \in hbase\ b \Longrightarrow evalC\ b\ n = evalC\ b\ m \Longrightarrow n = m$
proof (*induct n arbitrary*: *m rule*: *hbase.induct*)
case *1*
then show *?case* **by** (*cases m rule*: *hbase.cases*) *simp-all*
next
case (*2 i n ns m*$'$)
obtain *j m ms* **where** [*simp*]: $m' = C$ (*replicate j m* @ *ms*) **and**
m: $j \neq 0$ $j < b$ $m \in hbase\ b$ $C\ ms \in hbase\ b$ $\bigwedge m'.\ m' \in set\ ms \Longrightarrow evalC\ b\ m < evalC\ b\ m'$
using *2*(*8*,*1*,*2*,*9*) **by** (*cases* m' *rule*: *hbase.cases*) *simp-all*
have $i = j \wedge evalC\ b\ n = evalC\ b\ m$ **using** *2*(*1*,*2*,*7*,*9*) *m*(*1*,*2*,*5*)
by (*intro base-red*[*of map* (*evalC b*) *ns* - - *b map* (*evalC b*) *ms*])
(*auto simp*: *comp-def evalC-sum-list sum-list-replicate*)
then show *?case*
using *2*(*4*)[*OF m*(*3*)] *2*(*6*)[*OF m*(*4*)] *2*(*9*) **by** *simp*
qed

5.2 Correctness of *stepC*

We show that *stepC c* preserves hereditary base *c* + *2* representations. In order to cover intermediate results produced by *stepC*, we extend the hereditary base representation to allow the least significant digit to be equal to *b*, which essentially means that we may have an extra sibling in front on every level.

```
inductive-set hbase-ext for b where
  n ∈ hbase b ⟹ n ∈ hbase-ext b
| n ∈ hbase-ext b ⟹
   C m ∈ hbase b ⟹ (⋀m'. m' ∈ set m ⟹ evalC b n ≤ evalC b m') ⟹
   C (n # m) ∈ hbase-ext b

lemma hbase-ext-hd' [dest]:
  C (n # ns) ∈ hbase-ext b ⟹ n ∈ hbase-ext b
  by (cases rule: hbase-ext.cases) (auto intro: hbase-ext.intros(1))

lemma hbase-ext-tl:
  C ns ∈ hbase-ext b ⟹ ns ≠ [] ⟹ C (tl ns) ∈ hbase b
  by (cases C ns b rule: hbase-ext.cases; cases ns) (simp-all add: hbase-tl')

lemmas hbase-ext-tl' [dest] = hbase-ext-tl[of n # ns for n ns, simplified]

lemma hbase-funC:
  c ≠ 0 ⟹ C (n # ns) ∈ hbase-ext (Suc c) ⟹
   C (funC n (Suc c) @ ns) ∈ hbase-ext (Suc c)
proof (induct n arbitrary: ns rule: funC.induct)
  case (2 ms)
  have [simp]: evalC (Suc c) (C ms) < evalC (Suc c) m' if m' ∈ set ns for m'
   using 2(2)
  proof (cases rule: hbase-ext.cases)
   case 1 then show ?thesis using that
    by (cases rule: hbase.cases, case-tac i) (auto intro: Suc-lessD)
  qed (auto simp: Suc-le-eq that)
  show ?case using 2
   by (auto 0 4 intro: hbase-ext.intros hbase.intros(2) order.strict-implies-order)
next
  case (3 m ms ms')
  show ?case
   unfolding funC.simps append-Cons append-Nil
  proof (rule hbase-ext.intros(2), goal-cases 31 32 33)
   case (33 m')
   show ?case using 3(3)
   proof (cases rule: hbase-ext.cases)
    case 1 show ?thesis using 1 3(1,2) 33
     by (cases rule: hbase.cases, case-tac i) (auto intro: less-or-eq-imp-le)
   qed (insert 33, simp)
  qed (insert 3, blast+)
```

qed *auto*

lemma *stepC-sound*:
  $n \in$ *hbase-ext* (*Suc* (*Suc* *c*)) $\Longrightarrow$ *stepC c n* $\in$ *hbase* (*Suc* (*Suc* *c*))
proof (*induct c n rule*: *stepC.induct*)
  case (*3 c n ns ms*)
  show *?case* **using** *3*(*2*,*1*)
    by (*cases rule*: *hbase-ext.cases*; *unfold stepC.simps*) (*auto intro*: *hbase-funC*)
qed (*auto intro*: *hbase.intros*)

5.3 Surjectivity of *evalC*

Note that the base must be at least *2*.

lemma *evalC-surjective*:
  $\exists n' \in$ *hbase* (*Suc* (*Suc* *b*)). *evalC* (*Suc* (*Suc* *b*)) $n' = n$
proof (*induct n*)
  case *0* **then show** *?case* **by** (*auto intro*: *bexI*[*of - C* []] *hbase.intros*)
next
  have [*simp*]: *Suc x* $\leq$ *Suc* (*Suc b*)^*x* **for** *x* **by** (*induct x*) *auto*
  case (*Suc n*)
  then obtain n' **where** $n' \in$ *hbase* (*Suc* (*Suc* *b*)) *evalC* (*Suc* (*Suc* *b*)) $n' = n$ **by** *blast*
  then obtain n' *j* **where** n': *Suc n* $\leq$ *j* *j* = *evalC* (*Suc* (*Suc* *b*)) n' $n' \in$ *hbase* (*Suc* (*Suc* *b*))
    by (*intro that*[*of - C* [*n'*]])
      (*auto intro*!: *intro*: *hbase.intros*(*1*) *dest*!: *hbaseI2*[*of 1 b+2* n' [], *simplified*])
  then show *?case*
  proof (*induct rule*: *inc-induct*)
    case (*step m*)
    obtain n' **where** $n' \in$ *hbase* (*Suc* (*Suc* *b*)) *evalC* (*Suc* (*Suc* *b*)) $n' =$ *Suc m*
      using *step*(*3*)[*OF step*(*4*,*5*)] **by** *blast*
    then show *?case* **using** *stepC-dec*[*of* n' *b*]
      by (*cases* n' *rule*: *C2O.cases*) (*auto intro*: *stepC-sound hbase-ext.intros*(*1*))
  qed *blast*
qed

5.4 Monotonicity of *hbase*

Here we show that every hereditary base *b* number is also a valid hereditary base *b + 1* number. This is not immediate because we have to show that monotonicity of siblings is preserved.

lemma *hbase-evalC-mono*:
  assumes $n \in$ *hbase b* $m \in$ *hbase b* *evalC b n* < *evalC b m*
  shows *evalC* (*Suc b*) *n* < *evalC* (*Suc b*) *m*
proof (*cases b* < *2*)
  case *True* **show** *?thesis* **using** *assms*(*2*,*3*) *True* **by** (*cases rule*: *hbase.cases*) *simp-all*
next

```
  case False
  then obtain b' where [simp]: b = Suc (Suc b')
    by (auto simp: numeral-2-eq-2 not-less-eq dest: less-imp-Suc-add)
  show ?thesis using assms(3,1,2)
  proof (induct evalC b n evalC b m arbitrary: n m rule: less-Suc-induct)
    case 1 then show ?case using stepC-sound[of m b', OF hbase-ext.intros(1)]
      stepC-dec[of m b'] stepC-dec'[of m b'] evalC-inj-on-hbase
      by (cases m rule: C2O.cases) (fastforce simp: eval-nat-numeral)+
  next
    case (2 j) then show ?case
      using evalC-surjective[of b' j] less-trans by fastforce
  qed
qed

lemma hbase-mono:
  n ∈ hbase b ⟹ n ∈ hbase (Suc b)
  by (induct n rule: hbase.induct) (auto 0 3 intro: hbase.intros hbase-evalC-mono)
```

5.5 Conversion to and from *nat*

We have previously defined *H2N b* = *evalC b* and *N2H b* as its inverse. So we can use the injectivity and surjectivity of *evalC b* for simplification.

```
lemma N2H-inv:
  n ∈ hbase b ⟹ N2H b (H2N b n) = n
  using evalC-inj-on-hbase
  by (auto simp: N2H-def H2N-def[abs-def] inj-on-def intro!: inv-into-f-f)

lemma H2N-inv:
  H2N (Suc (Suc b)) (N2H (Suc (Suc b)) n) = n
  using evalC-surjective[of b n]
  by (auto simp: N2H-def H2N-def[abs-def] intro: f-inv-into-f)

lemma N2H-eqI:
  n ∈ hbase (Suc (Suc b)) ⟹
   H2N (Suc (Suc b)) n = m ⟹ N2H (Suc (Suc b)) m = n
  using N2H-inv by blast

lemma N2H-neI:
  n ∈ hbase (Suc (Suc b)) ⟹
   H2N (Suc (Suc b)) n ≠ m ⟹ N2H (Suc (Suc b)) m ≠ n
  using H2N-inv by blast

lemma N2H-0 [simp]:
  N2H (Suc (Suc c)) 0 = C []
  using H2N-def N2H-inv hbase.intros(1) by fastforce

lemma N2H-nz [simp]:
  0 < n ⟹ N2H (Suc (Suc c)) n ≠ C []
  by (metis N2H-0 H2N-inv neq0-conv)
```

6 The Goodstein function revisited

We are now ready to prove termination of the Goodstein function *goodstein* as well as its relation to *goodsteinC* and *goodsteinO*.

lemma *goodstein-aux*:
goodsteinC (*Suc c*) (*N2H* (*Suc* (*Suc c*)) (*Suc n*)) =
goodsteinC (*c+2*) (*N2H* (*c+3*) (*H2N* (*c+3*) (*N2H* (*c+2*) (*n+1*)) − *1*))
proof −
have [*simp*]: $n \neq C\ [] \Longrightarrow$ *goodsteinC c n* = *goodsteinC* (*c+1*) (*stepC c n*) **for** *c n*
by (*induct c n rule*: *stepC.induct*) *simp-all*
have [*simp*]: *stepC* (*Suc c*) (*N2H* (*Suc* (*Suc c*)) (*Suc n*)) ∈ *hbase* (*Suc* (*Suc* (*Suc c*)))
by (*metis H2N-def N2H-inv evalC-surjective hbase-ext.intros*(*1*) *hbase-mono stepC-sound*)
show *?thesis*
using *arg-cong*[*OF stepC-dec*[*of N2H* (*c+2*) (*n+1*) *c+1*, *folded H2N-def*], *of* λ*n*. *N2H* (*c+3*) (*n−1*)]
by (*simp add*: *eval-nat-numeral N2H-inv*)
qed

termination *goodstein*
proof (*relation measure* (λ(*c*, *n*). *goodsteinC c* (*N2H* (*c+1*) *n*) − *c*), *goal-cases* - *1*)
case (*1 c n*)
have ∗: *goodsteinC c n* ≥ *c* **for** *c n*
by (*induct c n rule*: *goodsteinC.induct*) *simp-all*
show *?case* **by** (*simp add*: *goodstein-aux eval-nat-numeral*) (*meson Suc-le-eq diff-less-mono2 lessI* ∗)
qed *simp*

lemma *goodstein-def′*:
$c \neq 0 \Longrightarrow$ *goodstein c n* = *goodsteinC c* (*N2H* (*c+1*) *n*)
by (*induct c n rule*: *goodstein.induct*) (*simp-all add*: *goodstein-aux eval-nat-numeral*)

lemma *goodstein-impl*:
$c \neq 0 \Longrightarrow$ *goodstein c n* = *goodsteinO c* (*C2O* (*N2H* (*c+1*) *n*))
— but note that *N2H* is not executable as currently defined
using *goodstein-def′*[*unfolded goodsteinC-def′*] **.**

lemma *goodstein-16*:
$\mathcal{G}$ *16* = *goodsteinO 1* (*exp*ω (*exp*ω (*exp*ω (*exp*ω *Z*))))
proof −
have *N2H* (*Suc* (*Suc 0*)) *16* = *C* [*C* [*C* [*C* [*C* []]]]]
by (*auto simp*: *H2N-def intro*!: *N2H-eqI hbase-singletonI hbase.intros*(*1*))
then show *?thesis* **by** (*simp add*: *goodstein-impl*)
qed

7 Translation to λ-calculus

We define Church encodings for *nat* and *Ord*. Note that we are basically in a Hindley-Milner type system, so we cannot use a proper polymorphic type. We can still express Church encodings as folds over values of the original type.

abbreviation Z_N **where** $Z_N \equiv (\lambda s\ z.\ z)$
abbreviation S_N **where** $S_N \equiv (\lambda n\ s\ z.\ s\ (n\ s\ z))$

primrec *fold-nat* (‹$\langle$-$\rangle_N$›) **where**
  $\langle 0\rangle_N = Z_N$
| $\langle Suc\ n\rangle_N = S_N\ \langle n\rangle_N$

lemma one_N:
  $\langle 1\rangle_N = (\lambda x.\ x)$
  by *simp*

abbreviation Z_O **where** $Z_O \equiv (\lambda z\ s\ l.\ z)$
abbreviation S_O **where** $S_O \equiv (\lambda n\ z\ s\ l.\ s\ (n\ z\ s\ l))$
abbreviation L_O **where** $L_O \equiv (\lambda f\ z\ s\ l.\ l\ (\lambda i.\ f\ i\ z\ s\ l))$

primrec *fold-Ord* (‹$\langle$-$\rangle_O$›) **where**
  $\langle Z\rangle_O = Z_O$
| $\langle S\ n\rangle_O = S_O\ \langle n\rangle_O$
| $\langle L\ f\rangle_O = L_O\ (\lambda i.\ \langle f\ i\rangle_O)$

The following abbreviations and lemmas show how to implement the arithmetic functions and the Goodstein function on a Church-encoded *Ord* in lambda calculus.

abbreviation (*input*) add_O **where**
  $add_O\ n\ m \equiv (\lambda z\ s\ l.\ m\ (n\ z\ s\ l)\ s\ l)$

lemma add_O:
  $\langle addO\ n\ m\rangle_O = add_O\ \langle n\rangle_O\ \langle m\rangle_O$
  by (*induct m*) *simp-all*

abbreviation (*input*) mul_O **where**
  $mul_O\ n\ m \equiv (\lambda z\ s\ l.\ m\ z\ (\lambda m.\ n\ m\ s\ l)\ l)$

lemma mul_O:
  $\langle mulO\ n\ m\rangle_O = mul_O\ \langle n\rangle_O\ \langle m\rangle_O$
  by (*induct m*) (*simp-all add*: add_O)

abbreviation (*input*) ω_O **where**
  $\omega_O \equiv (\lambda z\ s\ l.\ l\ (\lambda n.\ \langle n\rangle_N\ s\ z))$

lemma ω_O:
  $\langle \omega\rangle_O = \omega_O$

proof −
have [*simp*]: $\langle (S \,\widehat{}\widehat{}\, i)\ Z\rangle_O\ z\ s\ l = \langle i\rangle_N\ s\ z$ **for** *i z s l* **by** (*induct i*) *simp-all*
show *?thesis* **by** (*simp add*: *ω-def*)
qed

abbreviation (*input*) $exp\omega_O$ **where**
$exp\omega_O\ n \equiv (\lambda z\ s\ l.\ n\ s\ (\lambda x\ z.\ l\ (\lambda n.\ \langle n\rangle_N\ x\ z))\ (\lambda f\ z.\ l\ (\lambda n.\ f\ n\ z))\ z)$

lemma $exp\omega_O$:
$\langle exp\omega\ n\rangle_O = exp\omega_O\ \langle n\rangle_O$
by (*induct n*) (*simp-all add*: $mul_O\ \omega_O$)

abbreviation (*input*) $goodstein_O$ **where**
$goodstein_O \equiv (\lambda c\ n.\ n\ (\lambda x.\ x)\ (\lambda n\ m.\ n\ (m + 1))\ (\lambda f\ m.\ f\ (m + 2)\ m)\ c)$

lemma $goodstein_O$:
$goodsteinO\ c\ n = goodstein_O\ c\ \langle n\rangle_O$
by (*induct n arbitrary*: *c*) *simp-all*

Note that modeling Church encodings with folds is still limited. For example, the meaningful expression $\langle n\rangle_N\ exp\omega_O\ Z_O$ cannot be typed in Isabelle/HOL, as that would require rank-2 polymorphism.

7.1 Alternative: free theorems

The following is essentially the free theorem for Church-encoded *Ord* values.

lemma *freeOrd*:
assumes $\bigwedge n.\ h\ (s\ n) = s'\ (h\ n)$ **and** $\bigwedge f.\ h\ (l\ f) = l'\ (\lambda i.\ h\ (f\ i))$
shows $h\ (\langle n\rangle_O\ z\ s\ l) = \langle n\rangle_O\ (h\ z)\ s'\ l'$
by (*induct n*) (*simp-all add*: *assms*)

Each of the following proofs first states a naive definition of the corresponding function (which is proved correct by induction), from which we then derive the optimized version using the free theorem, by (conditional) rewriting (without induction).

lemma $add_O{}'$:
$\langle addO\ n\ m\rangle_O = add_O\ \langle n\rangle_O\ \langle m\rangle_O$
proof −
have [*simp*]: $\langle addO\ n\ m\rangle_O = \langle m\rangle_O\ \langle n\rangle_O\ S_O\ L_O$
by (*induct m*) *simp-all*
show *?thesis*
by (*intro ext*) (*simp add*: *freeOrd*[**where** $h = \lambda n.\ n$ - - -])
qed

lemma $mul_O{}'$:
$\langle mulO\ n\ m\rangle_O = mul_O\ \langle n\rangle_O\ \langle m\rangle_O$
proof −
have [*simp*]: $\langle mulO\ n\ m\rangle_O = \langle m\rangle_O\ Z_O\ (\lambda m.\ add_O\ m\ \langle n\rangle_O)\ L_O$

by (*induct m*) (*simp-all add*: add_O)
show *?thesis*
by (*intro ext*) (*simp add*: *freeOrd*[**where** $h = \lambda n.\ n$ - - -])
qed

lemma $exp\omega_O{}'$:
$\langle exp\omega\ n\rangle_O = exp\omega_O\ \langle n\rangle_O$
proof −
have [*simp*]: $\langle exp\omega\ n\rangle_O = \langle n\rangle_O\ (S_O\ Z_O)\ (\lambda m.\ mul_O\ m\ \omega_O)\ L_O$
by (*induct n*) (*simp-all add*: $mul_O\ \omega_O$)
show *?thesis*
by (*intro ext*) (*simp add*: *fun-cong*[*OF freeOrd*[**where** $h = \lambda n\ z.\ n\ z$ - -]])
qed

end

References

[1] J. C. Blanchette, M. Fleury, and D. Traytel. Formalization of nested multisets, hereditary multisets, and syntactic ordinals. *Archive of Formal Proofs*, Nov. 2016. http://isa-afp.org/entries/Nested_Multisets_Ordinals.html, Formal proof development.

[2] E. A. Cichon. A short proof of two recently discovered independence results using recursion theoretic methods. *Proceedings of the American Mathematical Society*, 87:704–706, Apr. 1983. doi:10.2307/2043364.

[3] R. L. Goodstein. On the restricted ordinal theorem. *Journal of Symbolic Logic*, 9:33–41, 1944. doi:10.2307/2268019.

[4] J. Tromp. Binary lambda calculus. https://tromp.github.io/cl/Binary_lambda_calculus.html.

[5] J. Tromp. Binary lambda calculus and combinatory logic. In C. S. Calude, editor, *Randomness And Complexity, from Leibniz To Chaitin*, pages 237–260. World Scientific Publishing Company, Oct. 2008.