Implementing the Goodstein Function in $\lambda$-Calculus

Bertram Felgenhauer

February 23, 2021

Abstract

In this formalization, we develop an implementation of the Goodstein function $G$ in plain $\lambda$-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 $G(16)$, and easily extends to other fixed arguments.

Contents

1 Introduction 2

2 Specification 3
   2.1 Hereditary base representation 3
   2.2 The Goodstein function 4

3 Ordinals 4
   3.1 Evaluation 5
   3.2 Goodstein function and sequence 5
   3.3 Properties of evaluation 6
   3.4 Arithmetic properties 6

4 Cantor normal form 7
   4.1 Conversion to and from the ordinal type $\text{Ord}$ 7
   4.2 Evaluation 8
   4.3 Transfer of the $\text{Ord}$ induction principle to $C$ 8
   4.4 Goodstein function and sequence on $C$ 8
   4.5 Properties 9

5 Hereditary base $b$ representation 10
   5.1 Uniqueness 10
   5.2 Correctness of $\text{stepC}$ 11
   5.3 Surjectivity of $\text{evalC}$ 11
   5.4 Monotonicity of $hbase$ 12
   5.5 Conversion to and from $\text{nat}$ 12
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 \( b \), reinterpretating 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

\[
4 = 2^{2^1} \rightarrow 3^3 - 1 = 26 \\
26 = 3^2 \cdot 2 + 3^1 \cdot 2 + 2 \rightarrow 4^2 \cdot 2 + 4^1 \cdot 2 + 1 \cdot 2 - 1 = 41 \\
41 = 4^2 \cdot 2 + 4^1 \cdot 2 + 1 \rightarrow 5^2 \cdot 2 + 5^1 \cdot 2 + 1 - 1 = 60 \\
\]

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 \( \lambda \)-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.

\[
\begin{align*}
\expomega &= (\lambda z \ s \ l \ n \ s \ (\lambda x \ z \ l \ (\lambda n \ n \ x \ z)) \ (\lambda f \ z \ l \ (\lambda n \ f \ n \ z)) \ z) \\
goodstein &= (\lambda n \ c \ n \\
&\quad (\lambda x \ x) \\
&\quad (\lambda n \ m \ n \ (\lambda f \ x \ m \ f \ (f \ x))) \\
&\quad (\lambda f \ m \ f \ (\lambda f \ x \ m \ f \ (f \ x))) \ m) \\
c) \\
\mathcal{G}_{16} &= (\lambda e. \ \goodstein \ (e \ (e \ (e \ (\lambda z \ s \ l \ z)))) \ (\lambda x \ x)) \ \expomega
\end{align*}
\]

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

the expression for $G_{16}$ cannot be typed. However, we can prove that the building blocks $exp\omega$ and $goodstein$ work correctly in the sense that

- $exp\omega^4 (\lambda z. l. z)$ 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 $\lambda$-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 $\lambda$-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 $\omega$ 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 $\lambda$-calculus term computing $G(16)$. 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 · 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 [] ∈ hbase b
  | i ≠ 0 → i < b → n ∈ hbase b →
  C ms ∈ hbase b → (∀ m'. m' ∈ set ms → evalC b n < evalC b m') →
  C (replicate i n @ ms) ∈ 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 ≤ b, so we can convert from natural numbers by taking the inverse.

definition N2H where
  N2H b n = inv-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 gc = 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)

(proof)

abbreviation G where
  G n ≡ 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 $\omega$.

**datatype** Ord = Z | S Ord | L nat ⇒ 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

\[
\begin{align*}
addO n Z &= n \\
addO n (S m) &= S (addO n m) \\
addO n (L f) &= L (\lambda i. addO n (f i))
\end{align*}
\]

**primrec** $mulO$ where

\[
\begin{align*}
mulO n Z &= Z \\
mulO n (S m) &= addO (mulO n m) n \\
mulO n (L f) &= L (\lambda i. mulO n (f i))
\end{align*}
\]

**definition** $\omega$ where

\[
\omega = L (\lambda n. (S \sim n) Z)
\]

**primrec** $exp\omega$ where

\[
\begin{align*}
exp\omega Z &= S Z \\
exp\omega (S n) &= mulO (exp\omega n) \omega \\
exp\omega (L f) &= L (\lambda i. exp\omega (f i))
\end{align*}
\]

### 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

\[
\begin{align*}
evalO b Z &= 0 \\
evalO b (S n) &= Suc (evalO b n) \\
evalO b (L f) &= evalO b (f b)
\end{align*}
\]

### 3.2 Goodstein function and sequence

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

**primrec** $goodsteinO$ where

\[
\begin{align*}
goodsteinO c Z &= c \\
goodsteinO c (S n) &= goodsteinO (c+1) n \\
goodsteinO c (L f) &= goodsteinO c (f (c+2))
\end{align*}
\]

**primrec** $stepO$ where

\[
\begin{align*}
stepO c Z &= Z \\
stepO c (S n) &= n
\end{align*}
\]
We can compute a few values of the Goodstein sequence starting at 4.

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

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

3.3 Properties of evaluation

```haskell
lemma evalO-addO [simp]:
  evalO b (addO n m) = evalO b n + evalO b m
  ⟨proof⟩

lemma evalO-mulO [simp]:
  evalO b (mulO n m) = evalO b n * evalO b m
  ⟨proof⟩

lemma evalO-n [simp]:
  evalO b ((S ^^ n) Z) = n
  ⟨proof⟩

lemma evalO-ω [simp]:
  evalO b ω = b
  ⟨proof⟩

lemma evalO-expω [simp]:
  evalO b (expω n) = b^(evalO b n)
  ⟨proof⟩
```

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

```haskell
notepad begin
  ⟨proof⟩
end
```

3.4 Arithmetic properties

```haskell
lemma addO-Z [simp]:
  addO Z n = n
  ⟨proof⟩

lemma addO-assoc [simp]:
  addO n (addO m p) = addO (addO n m) p
  ⟨proof⟩

lemma mulO-distrib [simp]:
  mulO n (addO p q) = addO (mulO n p) (mulO n q)
  ⟨proof⟩
```
lemma mulO-assoc [simp]:
\[\text{mulO } n \ (\text{mulO } m \ p) = \text{mulO } (\text{mulO } n \ m) \ p\]
⟨proof⟩

lemma expω-addO [simp]:
\[\text{exp}\omega \ (\text{addO } n \ m) = \text{mulO } (\text{exp}\omega \ n) \ (\text{exp}\omega \ m)\]
⟨proof⟩

4 Cantor normal form

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

4.1 Conversion to and from the ordinal type \( \text{Ord} \)

fun \( \text{C2O} \) where
\[\text{C2O } (C \ [] ) = Z\]
\[\text{C2O } (C \ (n \ # \ ns)) = \text{addO } (\text{C2O } (C \ ns)) \ (\text{exp}\omega \ (\text{C2O } n))\]

definition \( \text{O2C} \) where
\[\text{O2C} = \text{inv } \text{C2O}\]

We show that \( \text{C2O} \) is injective, meaning the inverse is unique.

lemma addO-expω-inj:
assumes \( \text{addO } n \ (\text{exp}\omega \ m) = \text{addO } n' \ (\text{exp}\omega \ m')\)
shows \( n = n' \) and \( m = m'\)
⟨proof⟩

lemma C2O-inj:
\(\text{C2O } n = \text{C2O } m \implies n = m\)
⟨proof⟩

lemma O2C-C2O [simp]:
\(\text{O2C } (\text{C2O } n) = n\)
⟨proof⟩

lemma O2C-Z [simp]:
\(\text{O2C } Z = C \ []\)
⟨proof⟩

lemma C2O-replicate:
\(\text{C2O } (C \ (\text{replicate } i \ n)) = \text{mulO } (\text{exp}\omega \ (\text{C2O } n)) \ ((S \ ^\sim i) \ Z)\)
⟨proof⟩

lemma C2O-app:
\[ C2O \left( C \left( xs \oplus ys \right) \right) = \text{addO} \left( C2O \left( C \left( ys \right) \right) \right) \left( C2O \left( C \left( xs \right) \right) \right) \]

\[ \langle \text{proof} \rangle \]

### 4.2 Evaluation

**Lemma** `evalC-def`:
\[ \text{evalC} \ b \ n = \text{evalO} \ b \ \left( C2O \ n \right) \]
\[ \langle \text{proof} \rangle \]

**Lemma** `evalC-app [simp]`:
\[ \text{evalC} \ b \ \left( C \left( ns \oplus ms \right) \right) = \text{evalC} \ b \ \left( C \left( ns \right) \right) + \text{evalC} \ b \ \left( C \left( ms \right) \right) \]
\[ \langle \text{proof} \rangle \]

**Lemma** `evalC-replicate [simp]`:
\[ \text{evalC} \ b \ \left( C \left( \text{replicate} \ c \ n \right) \right) = c \ast \text{evalC} \ b \ \left( C \left[ n \right] \right) \]
\[ \langle \text{proof} \rangle \]

### 4.3 Transfer of the Ord induction principle to \( C \)

**Fun** `funC` — `funC` computes the fundamental sequence on \( C \)
\[ \text{funC} \left( C \left[ n \right] \right) = \left( \lambda \ i. \ C \left[ n \right] \right) \]
\[ \text{funC} \left( C \left( C \left[ n \right] \# ns \right) \right) = \left( \lambda \ i. \ \text{replicate} \ i \left( C \left( ns \right) \right) \right) \]
\[ \text{funC} \left( C \left( n \# ns \right) \right) = \left( \lambda \ i. \ C \left( \text{funC} \ n \ i \@ ns \right) \right) \]

**Lemma** `C2O-cons`:
\[ C2O \left( C \left( n \# ns \right) \right) = \]
\[ \left( \text{if} \ n = C \left[ \right] \left( \text{then} \ S \left( C2O \left( C \left( ns \right) \right) \right) \text{else} \ L \left( \lambda \ i. \ C2O \left( C \left( \text{funC} \ n \ i \@ ns \right) \right) \right) \right) \right) \]
\[ \langle \text{proof} \rangle \]

**Lemma** `C-Ord-induct`:
\[ \text{assumes} \ P \left( C \left[ \right] \right) \text{ and } \forall \ ns. \ P \left( C \left( ns \right) \right) \implies P \left( C \left( C \left[ \right] \# ns \right) \right) \]
\[ \text{and } \forall \ n \ ms. \ \left( \lambda \ i. \ P \left( C \left( \text{funC} \left( C \left( n \# ns \right) \right) \ i \@ ms \right) \right) \right) \implies P \left( C \left( n \# ns \right) \# ms \right) \]
\[ \text{shows} \ P \ n \]
\[ \langle \text{proof} \rangle \]

### 4.4 Goodstein function and sequence on \( C \)

**Function** (domintros) `goodsteinC` — `goodsteinC` computes the fundamental sequence on \( C \)
\[ \text{goodsteinC} \ c \left( C \left[ \right] \right) = c \]
\[ \text{goodsteinC} \ c \left( C \left( C \left[ \right] \# ns \right) \right) = \text{goodsteinC} \ \left( c+1 \right) \left( C \left( ns \right) \right) \]
\[ \text{goodsteinC} \ c \left( C \left( C \left( n \# ns \right) \# ms \right) \right) = \]
\[ \text{goodsteinC} \ c \left( C \left( \text{funC} \left( C \left( n \# ns \right) \right) \left( c+2 \right) \@ ms \right) \right) \]
\[ \langle \text{proof} \rangle \]

**Termination**
\[ \langle \text{proof} \rangle \]

**Lemma** `goodsteinC-def`:
\[ \langle \text{proof} \rangle \]

8
goodstein\(C\) \(c\ n = \text{goodsteinO}\ (C20\ n)\)

\[\text{proof}\]

\textbf{function (domintros) stepC where}

\begin{align*}
\text{stepC} \ C \ [[]] &= C [] \\
\mid \text{stepC} \ C \ [[\[\] \# ns]] &= C \ ns \\
\mid \text{stepC} \ C \ [[\[\] \# ms]] &= \\
\in \\
\text{stepC} \ C \ (\text{funC} \ (C \ (n \# ns)) \ (\text{Suc} \ (\text{Suc} \ c) \ @ ms))
\end{align*}

\[\text{proof}\]

\textbf{termination}

\[\text{proof}\]

\textbf{definition g4C where}

\[g4C \ n = \text{fold stepC} \ [1..<\text{Suc} \ n] \ (C \ [C \ [[]]])\]

\textbf{value}

\begin{align*}
\text{map} \ (\lambda n. \ \text{evalC} \ (n + 2) \ (g4C \ n)) \ [0..<10] \\
\rightarrow [4, 26, 41, 60, 83, 109, 139, 173, 211, 253]
\end{align*}

\textbf{4.5 Properties}

\textbf{lemma stepC-def':}

\[\text{stepC} \ c \ n = O2C \ (\text{stepO} \ c \ (C20 \ n))\]

\[\text{proof}\]

\textbf{lemma funC-ne [simp]:}

\[\text{funC} \ m \ (\text{Suc} \ n) \neq []\]

\[\text{proof}\]

\textbf{lemma evalC-funC [simp]:}

\[\text{evalC} \ b \ (C \ (\text{funC} \ n \ b)) = \text{evalC} \ b \ (C \ [n])\]

\[\text{proof}\]

\textbf{lemma stepC-app [simp]:}

\[n \neq C [] \implies \text{stepC} \ c \ (C \ (\text{unC} \ n \ @ ns)) = C \ (\text{unC} \ (\text{stepC} \ c \ n) \ @ ns)\]

\[\text{proof}\]

\textbf{lemma stepC-cons [simp]:}

\[ns \neq [] \implies \text{stepC} \ c \ (C \ (n \# ns)) = C \ (\text{unC} \ (\text{stepC} \ c \ (C \ [n])) \ @ ns)\]

\[\text{proof}\]

\textbf{lemma stepC-dec:}

\[n \neq C [] \implies \text{Suc} \ (\text{evalC} \ (\text{Suc} \ (\text{Suc} \ c)) \ (\text{stepC} \ c \ n)) = \text{evalC} \ (\text{Suc} \ (\text{Suc} \ c)) \ n\]

\[\text{proof}\]

\textbf{lemma stepC-dec':}

\[n \neq C [] \implies \text{evalC} \ (c + 3) \ (\text{stepC} \ c \ n) < \text{evalC} \ (c + 3) \ n\]

\[\text{proof}\]
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 \( \text{evalC}_b \) restricted to \( hbase_b \) is injective.

**Lemma hbaseI2:**
\[
i < b \implies n \in hbase_b \implies C \in hbase_b \implies (\forall m'. m' \in \text{set} m \implies \text{evalC}_b C n < \text{evalC}_b m') \implies C \text{ (replicate n m) } \in hbase_b
\]
<br>

**Proof:**

\[
\text{lemmas hbase-singletonI = hbase.intro(2)[of I Suc (Suc b) for b, OF - - hbase.intro(1), simplified]}
\]

**Lemma hbase-hd:**
\[
C ns \in hbase_b \implies ns \neq [] \implies \text{hd ns} \in hbase_b
\]

**Proof:**

\[
\text{lemmas hbase-hd'[dest] = hbase-hd[of n # ns for n ns, simplified]}
\]

**Lemma hbase-tl:**
\[
C ns \in hbase_b \implies ns \neq [] \implies C (\text{tl ns}) \in hbase_b
\]

**Proof:**

\[
\text{lemmas hbase-tl'[dest] = hbase-tl[of n # ns for n ns, simplified]}
\]

**Lemma hbase-elt [dest]:**
\[
C ns \in hbase_b \implies n \in \text{set ns} \implies n \in hbase_b
\]

**Proof:**

**Lemma evalC-sum-list:**
\[
\text{evalC}_b C \text{ (ns)} = \text{sum-list (map (λn. b^\text{evalC}_b n) ns)}
\]

**Proof:**

**Lemma sum-list-replicate:**
\[
\text{sum-list (replicate n x)} = n * x
\]

**Proof:**

**Lemma base-red:**

**Fixes** \( b :: \text{nat} \)

**Assumes** \( n: \forall n'. n' \in \text{set ns} \implies n < n' i < b i \neq 0 \)

**And** \( m: \forall m'. m' \in \text{set ms} \implies m < m' j < b j \neq 0 \)

**And** \( s: i * b^\text{n} + \text{sum-list (map (λn. b^\text{n}) ns)} = j * b^\text{m} + \text{sum-list (map (λn. b^\text{m}) ms)} \)

**Shows** \( i = j \land n = m \)
lemma evalC-inj-on-hbase:
\( n \in \text{hbase } b \Rightarrow m \in \text{hbase } b \Rightarrow \text{evalC } b \ n = \text{evalC } b \ m \Rightarrow n = m \)

\( \langle \text{proof} \rangle \)

5.2 Correctness of \( \text{stepC} \)

We show that \( \text{stepC } c \) preserves hereditary base \( c + 2 \) representations. In order to cover intermediate results produced by \( \text{stepC} \), 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.

\textbf{inductive-set} hbase-ext \textbf{for} \( b \) \textbf{where}
\[
\begin{align*}
  n \in \text{hbase } b & \Rightarrow n \in \text{hbase-ext } b \\
  n \in \text{hbase-ext } b & \Rightarrow C \ m \in \text{hbase } b \Rightarrow (\forall m'. m' \in \text{set } m \Rightarrow \text{evalC } b \ n \leq \text{evalC } b \ m') \Rightarrow C \ (n \# m) \in \text{hbase-ext } b
\end{align*}
\]

\textbf{lemma} hbase-ext-hd' [dest]:
\[
C \ (n \# ns) \in \text{hbase-ext } b \Rightarrow n \in \text{hbase-ext } b
\]
\( \langle \text{proof} \rangle \)

\textbf{lemma} hbase-ext-tl:
\[
C \ ns \in \text{hbase-ext } b \Rightarrow ns \neq [] \Rightarrow C \ (tl ns) \in \text{hbase } b
\]
\( \langle \text{proof} \rangle \)

\textbf{lemmas} hbase-ext-tl' [dest] = hbase-ext-tl[of \( n \# ns \) for \( n \) ns, simplified]

\textbf{lemma} hbase-funC:
\[
c \neq 0 \Rightarrow C \ (n \# ns) \in \text{hbase-ext } (Suc \ c) \Rightarrow C \ (funC \ n \ (Suc \ c) \at ns) \in \text{hbase-ext } (Suc \ c)
\]
\( \langle \text{proof} \rangle \)

\textbf{lemma} stepC-sound:
\[
n \in \text{hbase-ext } (Suc \ (Suc \ c)) \Rightarrow \text{stepC } c \ n \in \text{hbase } (Suc \ (Suc \ c))
\]
\( \langle \text{proof} \rangle \)

5.3 Surjectivity of \( \text{evalC} \)

Note that the base must be at least 2.

\textbf{lemma} evalC-surjective:
\[
\exists n' \in \text{hbase } (Suc \ (Suc \ b)). \text{evalC } (Suc \ (Suc \ b)) \ n' = n
\]
\( \langle \text{proof} \rangle \)

11
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** $n \in hbase \land m \in hbase \land evalC b \ n < evalC b \ m$
- **shows** $evalC (Suc b) n < evalC (Suc b) m$

(proof)

**lemma** $hbase$-mono:

$n \in hbase \rightarrow n \in hbase (Suc b)$

(proof)

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:

$n \in hbase \mapsto N2H b (H2N b n) = n$

(proof)

**lemma** $H2N$-inv:

$H2N (Suc (Suc b)) (N2H (Suc (Suc b)) n) = n$

(proof)

**lemma** $N2H$-eqI:

$n \in hbase (Suc (Suc b)) \mapsto H2N (Suc (Suc b)) n = m \mapsto N2H (Suc (Suc b)) m = n$

(proof)

**lemma** $N2H$-neI:

$n \in hbase (Suc (Suc b)) \mapsto H2N (Suc (Suc b)) n \neq m \mapsto N2H (Suc (Suc b)) m \neq n$

(proof)

**lemma** $N2H$-0 [simp]:

$N2H (Suc (Suc c)) 0 = C []$

(proof)

**lemma** $N2H$-nz [simp]:

$\theta < n \mapsto N2H (Suc (Suc c)) n \neq C []$

(proof)
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:**
\[
\text{goodsteinC (Suc c) (N2H (Suc (Suc c)) (Suc n))} = \\
\text{goodsteinC (c+2) (N2H (c+3) (H2N (c+3) (N2H (c+2) (n+1)) - 1))}
\]

\( \langle \text{proof} \rangle \)

**Termination goodstein**

\( \langle \text{proof} \rangle \)

**Lemma goodstein-def':**
\[
c \neq 0 \implies \text{goodstein c n} = \text{goodsteinC c (N2H (c+1) n)}
\]

\( \langle \text{proof} \rangle \)

**Lemma goodstein-impl:**
\[
c \neq 0 \implies \text{goodstein c n} = \text{goodsteinO c (C2O (N2H (c+1) n))}
\]
— but note that \( N2H \) is not executable as currently defined

\( \langle \text{proof} \rangle \)

**Lemma goodstein-16:**
\[
G_{16} = \text{goodsteinO 1 (expω (expω (expω (expω Z)))))}
\]

\( \langle \text{proof} \rangle \)

7 Translation to \( \lambda \)-calculus

We define Church encodings for \( \text{nat} \) and \( \text{Ord} \). 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 \( Z_N \equiv (\lambda s. z) \)

**Abbreviation** \( S_N \) where \( S_N \equiv (\lambda n s. s (n s z)) \)

**Primrec** \( \text{fold-nat} \ (\langle \cdot \rangle_N) \) where
\[
\langle 0 \rangle_N = Z_N \\
\langle \text{Suc n} \rangle_N = S_N \langle n \rangle_N
\]

**Lemma** \( \text{one}_N \):
\[
\langle 1 \rangle_N = (\lambda x. x)
\]

\( \langle \text{proof} \rangle \)

**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 \( L_O \equiv (\lambda f z s l. (\lambda i. f i z s l)) \)

**Primrec** \( \text{fold-Ord} \ (\langle \cdot \rangle_O) \) 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 \textit{Ord} in lambda calculus.

\textbf{abbreviation (input) \textit{add}\textsubscript{\textit{O}} where}
\[ \textit{add}\textsubscript{\textit{O}} n m \equiv (\lambda z s l. \ m (n z s l) \ s l) \]

\textbf{lemma \textit{add}\textsubscript{\textit{O}}}:
\[ \langle \textit{add}\textsubscript{\textit{O}} \ n \ m \rangle_O = \textit{add}\textsubscript{\textit{O}} \langle \langle n \rangle_O \rangle_O \langle \langle m \rangle_O \rangle_O \]

\textbf{abbreviation (input) \textit{mul}\textsubscript{\textit{O}} where}
\[ \textit{mul}\textsubscript{\textit{O}} n m \equiv (\lambda z s l. \ m z (\lambda m. \ n m s l) \ l) \]

\textbf{lemma \textit{mul}\textsubscript{\textit{O}}}:
\[ \langle \textit{mul}\textsubscript{\textit{O}} \ n \ m \rangle_O = \textit{mul}\textsubscript{\textit{O}} \langle \langle n \rangle_O \rangle_O \langle \langle m \rangle_O \rangle_O \]

\textbf{abbreviation (input) \textit{ω}\textsubscript{\textit{O}} where}
\[ \textit{ω}\textsubscript{\textit{O}} \equiv (\lambda z s l. l (\lambda n. \langle n \rangle_N s z)) \]

\textbf{lemma \textit{ω}\textsubscript{\textit{O}}}:
\[ \langle \textit{ω} \rangle_O = \textit{ω}_O \]

\textbf{abbreviation (input) \textit{exp}\textsubscript{\textit{ω}}\textsubscript{\textit{O}} where}
\[ \textit{exp}\textsubscript{\textit{ω}}\textsubscript{\textit{O}} n \equiv (\lambda z s l. n s (\lambda x z l. (\lambda n. \langle n \rangle_N x z)) (\lambda f z l. (\lambda n. f n z)) \ z) \]

\textbf{lemma \textit{exp}\textsubscript{\textit{ω}}\textsubscript{\textit{O}}}:
\[ \langle \textit{exp}\textsubscript{\textit{ω}} \ n \rangle_O = \textit{exp}\textsubscript{\textit{ω}}\textsubscript{\textit{O}} \langle \langle n \rangle_O \rangle_O \]

\textbf{abbreviation (input) \textit{goodstein}\textsubscript{\textit{O}} where}
\[ \textit{goodstein}\textsubscript{\textit{O}} \equiv (\lambda c n. \ n (\lambda x. x) (\lambda n m. n (m + 1)) (\lambda f m. f (m + 2) m) \ c) \]

\textbf{lemma \textit{goodstein}\textsubscript{\textit{O}}}:
\[ \textit{goodstein}\textsubscript{\textit{O}} \ c \ n = \textit{goodstein}\textsubscript{\textit{O}} \ c \langle \langle n \rangle_O \rangle_O \]

Note that modeling Church encodings with folds is still limited. For example, the meaningful expression \( \langle n \rangle_N \textit{exp}\textsubscript{\textit{ω}}\textsubscript{\textit{O}} Z_O \) cannot be typed in Isabelle/HOL, as that would require rank-2 polymorphism.

\subsection{Alternative: free theorems}

The following is essentially the free theorem for Church-encoded \textit{Ord} values.
lemma freeOrd:
assumes \( \forall n. h (s n) = s' (h n) \) and \( \forall f. h (l f) = l' (\lambda i. h (f i)) \)
shows \( h ((n)O z s l) = (n)O (h z) s' l' \)
(proof)

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 addO':
\( (addO n m)O = addO (n)O (m)O \)
(proof)

lemma mulO':
\( (mulO n m)O = mulO (n)O (m)O \)
(proof)

lemma expO':
\( (expO n)O = expO (n)O \)
(proof)

end

References


