Ploop: A Language For Polynomial Time

Peter Møller Neergaard

These notes cover the lectures on our first language for polynomial time which we baptized Ploop. The approach is similar to the one chosen in Bellantoni and Cook’s seminal paper [BC92]. They however worked with a function algebra (more about that later), where we simply continue to use the Loop variant of While.

1 Two Motivating Examples

Before we get all caught up in formalism that us take a look at two examples that motivate the work in the rest of these notes.

First take a look at the following Loop-program $\text{mash}(U, V)$:

\begin{verbatim}
read UV
U:=hd U
V:=tl V
Y:=nil
loop U do
  Z:=V
  loop V do
    Y:=cons (hd Z) Y
    Z:=tl Z
write Y
\end{verbatim}

The program concatenates $|U|$ copies of $V$ reversed. Why you need this is of course a good question, but it is illustrative to compare with the following program $\text{exp}(U)$:

\begin{verbatim}
read U
Y:=1
\end{verbatim}
loop U do
  Z:=Y
loop Y do
  Y:=cons (hd Z) Y
  Z:=tl Z
write Y

While it might not be immediately clear, this program actually computes \(2^{|U|}\). Given that this number is build straightforwardly with \texttt{cons}, this is also the running time of the program.

**Exercise 1.1.** Prove that the program above computes \(2^{|U|}\).

The two programs are very similar in structure, but one notable difference is that in \texttt{exp} we make an assignment to \(Y\) which is also a variable that we loop over. While this does not change the current loop, it affects subsequent loop.

We shall see that by prohibiting assignments to variables that are part of a loop expression, we get a language, \texttt{Ploop}, which captures polynomial time. More precisely we shall see that \texttt{Ploop} is

**Sound:** any \texttt{Ploop}-program can be evaluated in polynomial time.

**Complete:** any polynomial time function can be written as \texttt{Ploop}-program (albeit not necessarily in the way we are used to).
2 Syntax

First let us recall the syntax of Loop:

\[ E,F ::= X \]
\[ \]\[ \mid a \]
\[ \]\[ \mid \text{hd } E \]
\[ \]\[ \mid \text{tl } E \]
\[ \]\[ \mid \text{cons } E F \]
\[ \]\[ \mid =? E F \]
\[ C,D ::= X:=E \]
\[ \]\[ \mid C; C \]
\[ \]\[ \mid \text{loop } E \text{ do } C \]
\[ P ::= \text{read } X; C; \text{write } Y \]

As for semantics this is like While, except for the loop-construction. The number of iterations is fixed as the height \( \|d\| \) of the value \( d \) of the expression \( E \) computed just before the first iteration:

**Definition 2.1.**

1. The height \( \| \cdot \| \) of a value \( d \in \mathbb{D}_A \) is defined inductively as

   \[ \|a\| = 0 \quad \|(d,e)\| = 1 + \max \{ \|d\|, \|e\| \} \]

   where \( a \in A \) is an atom and \( d,e \in \mathbb{D}_A \).

2. As usual we will extend \( \| \cdot \| \) to an environment \( \rho \) by taking \( \|\rho\| = \max \{ \|d\| \mid X \mapsto d \in \rho \} \).

3. We will use \( \| \cdot \|_S \) for the restriction of \( \| \cdot \| \) to a subset \( S \) of the variables in \( \rho \), i.e., \( \|\rho\|_S = \max \{ \|d\| \mid X \mapsto d \in \rho, X \in S \} \).

Notice that for numbers \( n \in \mathbb{N} \) we have \( |n| = \|n\| \).

The next step is to formally define what it means that we do not make assignments to a variable that we loop over. For that we need the following definitions:
Definition 2.2.

1. The set of variables \( \text{vars}(E) \) of an expression \( E \) is defined inductively as

\[
\begin{align*}
\text{vars}(a) &= \emptyset \\
\text{vars}(X) &= \{ X \} \\
\text{vars}(\text{hd } E) &= \text{vars}(\text{tl } E) = \text{vars}(E) \\
\text{vars}(\text{cons } E \ F) &= \text{vars}(\neq? \ E \ F) = \text{vars}(E) \cup \text{vars}(F).
\end{align*}
\]

2. The set of variables \( \text{vars}(C) \) of a command \( C \) is defined inductively as

\[
\begin{align*}
\text{vars}(X := E) &= \{ X \} \cup \text{vars}(E) \\
\text{vars}(C ; D) &= \text{vars}(C) \cup \text{vars}(D) \\
\text{vars}(\text{loop } E \ \text{do } C) &= \text{vars}(E) \cup \text{vars}(C)
\end{align*}
\]

\[\Box\]

Definition 2.3. The loop variables \( \text{loopVars}(C) \) of a command \( C \) is defined inductively as

\[
\begin{align*}
\text{loopVars}(X := E) &= \emptyset \\
\text{loopVars}(C ; D) &= \text{loopVars}(C) \cup \text{loopVars}(D) \\
\text{loopVars}(\text{loop } E \ \text{do } C) &= \text{vars}(E) \cup \text{loopVars}(C)
\end{align*}
\]

\[\Box\]

Definition 2.4. The updated variables \( \text{updVars}(C) \) of a command \( C \) is defined inductively as

\[
\begin{align*}
\text{updVars}(X := E) &= \{ X \} \\
\text{updVars}(C ; D) &= \text{updVars}(C) \cup \text{updVars}(D) \\
\text{updVars}(\text{loop } E \ \text{do } C) &= \text{updVars}(C)
\end{align*}
\]

\[\Box\]

We are then finally ready to define \( \text{Ploop} \):

Definition 2.5. A Loop-program \( p \) belongs to the language \( \text{Ploop} \) if, and only if, all occurrences of \( \text{loop } E \ \text{do } C \) in \( p \) satisfy the condition that \( \text{updVars}(C) \cap \text{loopVars}(\text{loop } E \ \text{do } C) = \emptyset \).  

\[\Box\]
3 Time Semantics

Building on what we saw for While, we use the following time semantics for Ploop:

\[
\begin{align*}
T[x] & = 1 \\
T[a] & = 1 \\
T[\text{hd } E] & = T[\text{tl } E] + 1 + T[E] \\
T[X := E] & = 1 + T[E] \\
T[C; D] & = 1 + T[C] + T[D] \\
T[\text{loop } E \text{ do } C] & = 1 + n + T[E] + \sum_{i=1}^{n} T[C] \\
\end{align*}
\]

where \( n = \|E\| \)

\[ C \vdash \rho_i \rightarrow \rho_{i+1} \text{ for all } 1 \leq i \leq n \]

\[ T[\text{read } X; C; \text{write } Y] d = T[C]^{\text{init}}[\text{read } d] \]

Note that the time to evaluate a fixed expression \( E \) is the same for all environments \( \rho \):

**Lemma 3.1.** For all expressions \( E \) in Ploop there is constant \( c_E \) such that \( T[E] \rho \leq c_E \) for all \( \rho \).

**Proof.** This is by induction over \( E \).

4 Soundness

The next thing to prove is that Ploop is sound with respect to polynomial time, i.e., that all Ploop-programs can be evaluated in polynomial time. More precisely we will prove that \( T[C] \rho \) is polynomial in \( \|\rho\| \) for all \( \rho \) and \( C \).

As it is often (but not always) the case the answer clings on figuring out “what is the biggest number we can compute?” Intuitively knowing the answer to this, will allow us to spin a loop running for that time and thus realize a bound by that number.

We start out examining expressions which are easy:
Lemma 4.1. For every expression $E$ of Ploop there is a constant $c_E$ such that $\|E\rho\| \leq \|\rho\| + c_E$ for every environment $\rho$. 

Proof. This is by induction over $E$. 

Instead of going directly to arbitrary command of Ploop, we first examine what happens inside an outermost loop loop $E$ do $C$. We notice that by the definition of Ploop all variables in $C$ that occur in a loop expression are never updated. We call these variables normal because we can iterate over them like normal Ploop-variables. The remaining variables of $C$ are safe to update (because they do not control a loop). Following this idea through we can prove the following that shows that the growth is only due to normal variables that we loop over:

Lemma 4.2. Let $C$ be a command with $\text{vars}(C)$ divided in to two disjunct sets Safe and Normal such that $\text{loopVars}(C) \subseteq \text{Normal}$ and $\text{updVars}(C) \subseteq \text{Safe}$.

1. There exist a monotone\(^1\) polynomial $q$ such that $\|\rho\|_{\text{Normal}} = \|\rho\|_{\text{Normal}}$ and $\|\rho\|_{\text{Safe}} \leq q(\|\rho\|_{\text{Normal}}) + \|\rho\|_{\text{Safe}}$ for all $\rho, \rho'$ where $C \vdash \rho \rightarrow \rho'$.

2. $T[C]\rho = O(q(\|\rho\|_{\text{Normal}}))$ for all environments $\rho$. 

Proof. We use induction on $C$:

$X := E$: We have $X := E \vdash \rho \rightarrow \rho[X \mapsto [E]\rho]$.

Since $\text{updVars}(C) \subseteq \text{Safe}$, we obviously have $\|\rho\|_{\text{Normal}} = \|\rho\|_{\text{Normal}}$. By Lemma 4.1 there is a $c$ such that $\|[E]\rho\| \leq \|\rho\| + c$. By choosing $q(n) = n + c$ we have $\|\rho\|_{\text{Safe}} \leq \|\rho\| + c \leq \|\rho\|_{\text{Safe}} + \|\rho\|_{\text{Normal}} + c = q(\|\rho\|_{\text{Normal}}) + \|\rho\|_{\text{Safe}}$.

We note that $q$ is monotone. As for the time we have $T[X := E] \rho = 1 + T[E] \rho = k = O(q(\|\rho\|_{\text{Normal}}))$.

where $k$ is some constant.

\(^1\)Recall that a function $f(x)$ is monotone if, and only if, $f(x') > f(x)$ when $x' > x$. 

We have $C; D \vdash \rho \rightarrow \rho''$ where $C \vdash \rho \rightarrow \rho'$ or $D \vdash \rho' \rightarrow \rho''$. It follows immediately from the induction hypothesis that \( \|\rho''\|_{\text{Normal}} = \|\rho'\|_{\text{Normal}} = \|\rho\|_{\text{Normal}} \). From the induction hypothesis, we also have two monotone polynomials \( q_C \) and \( q_D \) such that
\[
\|\rho''\|_{\text{Safe}} \leq q_C(\|\rho\|_{\text{Normal}}) + \|\rho\|_{\text{Safe}} \tag{\star}
\]
and
\[
\|\rho''\|_{\text{Normal}} \leq q_D(\|\rho'\|_{\text{Normal}}) + \|\rho'\|_{\text{Safe}} \tag{\star\star}
\]
Adding two monotone polynomials gives a new monotone polynomial so \( q(n) = q_D(n) + q_C(n) \) is a monotone polynomial. From the equations (\star) and (\star\star) we have
\[
\|\rho''\|_{\text{Normal}} \leq q_D(\|\rho'\|_{\text{Normal}}) + \|\rho'\|_{\text{Safe}}
\leq q_D(\|\rho\|_{\text{Normal}}) + q_C(\|\rho\|_{\text{Normal}}) + \|\rho\|_{\text{Safe}}
= q(\|\rho\|_{\text{Normal}}) + \|\rho\|_{\text{Safe}}.
\]
As for the time we have
\[
1 + T[C]\rho + T[D]\rho = 1 + O(q_C(\|\rho\|_{\text{Normal}})) + O(q_D(\|\rho\|_{\text{Normal}}))
= O(q(\|\rho\|_{\text{Normal}})).
\]

\textbf{loop E do C:} We have \textbf{loop E do C} \vdash \rho_1 \rightarrow \rho_{n+1} \ where \( n = \|[E]\rho\| \) and \( C \vdash \rho_i \rightarrow \rho_{i+1} \) for all \( 1 \leq i \leq n \). By a quick induction on \( n \) it is clear from the general induction hypothesis that \( \|\rho_{n+1}\|_{\text{Normal}} = \|\rho_1\|_{\text{Normal}} \). The induction hypothesis also gives us a monotone polynomial \( q_C \) such that
\[
\|\rho'\|_{\text{Safe}} \leq q_C(\|\rho\|_{\text{Normal}}) + \|\rho\|_{\text{Safe}} \quad \text{when} \ C \vdash \rho \rightarrow \rho'. \tag{1}
\]
We show by induction on \( n \) that \( \|\rho_{n+1}\|_{\text{Safe}} \leq n q_C(\|\rho_1\|_{\text{Normal}}) + \|\rho_1\|_{\text{Safe}} \):
\[
n = 0: \ We \ trivially \ have \|\rho_1\|_{\text{Safe}} \leq 0 q_C(\|\rho_1\|_{\text{Normal}}) + \|\rho_1\|_{\text{Safe}}.
\]
\[
n > 0: \ We \ have \ from \ the \ general \ induction \ hypothesis
\]
\[
\|\rho_{n+1}\|_{\text{Safe}} \leq q_C(\|\rho_n\|_{\text{Normal}}) + \|\rho_n\|_{\text{Safe}}
\leq q_C(\|\rho_1\|_{\text{Normal}}) + (n - 1) q_C(\|\rho_1\|_{\text{Normal}}) + \|\rho_1\|_{\text{Safe}}
\leq n q_C(\|\rho_1\|_{\text{Normal}}) + \|\rho_1\|_{\text{Safe}}.
\]
In conclusion we get a suitable $q$ by choosing $q(n) = nq_C(n)$.

As for the time we have

$$1 + n + T[E]_{\rho_1} + \sum_{i=1}^{n} T[C]_{\rho_i}$$

$$\leq 1 + n + k + \sum_{i=1}^{n} O(q_C(\|\rho_1\|_{\text{Normal}}))$$

$$= O(nq_C(\|\rho_1\|_{\text{Normal}}))$$

$$= O(q(\|\rho_1\|_{\text{Normal}}))$$

This finishes the three cases and concludes the proof. \qed

**Exercise 4.3.** Explain why the polynomial has to be monotone. \qed

This lays the ground work and we can now prove the following theorem:

**Theorem 4.4.** Let $C$ be a command in Ploop. Then there is a monotone polynomial $q$ such that $\|\rho'\| \leq q(\|\rho\|)$ and $T[C]_{\rho} = O(q(\|\rho\|_{\text{Normal}}))$ for all environments $\rho, \rho'$ where $C \vdash \rho \rightarrow \rho'$.

*Proof.* We use induction on $C$:

- $X := E$: This follows immediately from Lemma 4.2.

- `loop E do C`: This follows immediately from Lemma 4.2.

- $C; D$: We have $C; D \vdash \rho \rightarrow \rho''$ where $C \vdash \rho \rightarrow \rho'$ and $D \vdash \rho' \rightarrow \rho''$

  By the induction hypothesis we have polynomials $q_C$ and $q_D$ such that

  $$\|\rho'\| \leq q_C(\|\rho\|) \quad \text{and} \quad T[C]_{\rho} = O(q_C(\|\rho\|))$$

  and

  $$\|\rho''\| \leq q_D(\|\rho'\|) \quad \text{and} \quad T[D]_{\rho} = O(q_D(\|\rho'\|))$$

  By composing these two polynomials as $q(x) = q_D(q_C(x)) + q_C(x)$ we have

  $$\|\rho''\| \leq q_D(\|\rho'\|) \leq q_D(q_C(\|\rho\|)) \leq q(\|\rho\|)$$
and

\[
1 + T[C]\rho + T[D]\rho' = 1 + O(q_C(\|\rho\|)) + O(q_D(\|\rho'\|)) \\
= 1 + O(q_C(\|\rho\|)) + O(q_D(q_C(\|\rho'\|))) \\
= O(q\|\rho\|)
\]

This exhaust the cases and completes the proof. \qed

EXERCISE 4.5. At first sight Theorem 4.4 and Lemma 4.2 look very alike. Actually one might be tempted to consider the former a corollary of latter. Explain what this is not the case. (Hint: It has to do with the handling of composition and iteration: Why can we not just prove Theorem 4.4 directly by induction? Also, look at Lemma 4.2 and find a Ploop program where the full command \( C \) does not meet the conditions of the lemma.) \qed

5 Completeness

Having shown that all Ploop programs can be computed in polynomial time, we will show that all functions computable in polynomial are computable by a Ploop program. We shall see that any polynomial time Turing Machine can be represented as a Ploop program. This consists of two parts:

1. representing the state transition function
2. iterating the transition function long enough.

You have probably met Turing Machines before; if not you can also a more detailed description in [Jon97, 7.3]. There are as many ways to define them as authors, so for here we will just recall the definition used in [Jon97, 7.3]. Turing Machines operate on an infinite tape of symbols \{0, 1, \_\}.\footnote{For convenience we will assume the tape to be infinite in both directions and initialized with the blank space symbol \_.} At each step of the computation the tape head is over one of the symbols. A program consists of a fixed number of lines each with an instruction \( I \) of the following type:

right: move the tape head right
**left**: move the tape head left

**write** $S$: write symbol $S$ on the tape at the current head position

**if** $S$ **goto** $l$ **else** $l'$: if the symbol under the tape is $S$ continue with line $l$
otherwise continue with line $l'$.

In the three first cases the program continues with the following instruction. To make our life easier we will assume that the Turing Machine is never stock, i.e., it can always make a transition; when the computation is finished it simple idles until its time is up.

The state of a Turing Machine is a tuple consisting of the current label, the head position, and the tape. The first question we will answer is how to represent a state. For convenience we will assume that our set of atoms $A$ includes the three tape symbols. We represent the tape and head position in two lists:

- $R$ holds the symbols under and to the right of the tape head
- $L$ holds the symbols to the left of the tape head in reverse order.

The labels are simply represented as a number $n \in \mathbb{N}$; we hold the current label in the variable $P$. With these conventions we have

\[
\text{right} = L := \text{cons} \ (\text{hd} \ R) \ L; \ R := \text{tl} \ R ; \ P := \text{cons} \ \text{nil} \ P
\]
\[
\text{left} = R := \text{cons} \ (\text{hd} \ L) \ R; \ L := \text{tl} \ L; \ P := \text{cons} \ \text{nil} \ P
\]
\[
\text{write} \ S = R := \text{cons} \ S \ (\text{tl} \ R); \ P := \text{cons} \ \text{nil} \ P
\]
\[
\text{if} \ S \ \text{goto} \ l \ \text{else} \ l' = \text{if} \ =? \ (\text{hd} \ R) \ \text{then} \ P := l \ \text{else} \ P := l'
\]

We can then translate a program to the following transition function $T$:

\[
0 : I_0; \cdots ; m : I_m = \text{if} \ =? \ P \ 0 \ \text{then} \ I_0
\]
\[
\text{else if} \ =? \ P \ 1 \ \text{then} \ I_1
\]
\[
\cdot \cdot \cdot
\]
\[
\text{else if} \ =? \ P \ m-1 \ \text{then} \ I_{m-1}
\]
\[
\text{else} \ I_m
\]

At this point, all that is left is to initialize the variables and iterate $T$ the right number of times. As the Turing Machine is in polynomial time we
have a polynomial $q$ such that the number of steps is less than $q(|\sigma|)$ for all
input tapes $\sigma$. Assuming (without loss of generality) that the input is never
empty, we have constants $c$ and $n$ such that $q(|\sigma|) \leq c|\sigma|^n$.\(^3\) Exploiting this
we code the Turing Machine as follows

```plaintext
read R
  (* Do the computation *)
  L:=nil
  I:=0
  Length:=R
  loop Length do
    loop Length do
      ...
    loop Length do
      loop $c$ do $I$
      (* Now append (reverse L) to R *)
    Count:=L
    loop Count do
      R:=cons (hd L) R
      L:=tl L
  write R
```

where there are $n$ nested iterations of `loop Length do`.

One could criticise the program above for the fact that the resulting tape
might contain blank symbols in front and after the real data. Consequently,
the representation of the output is not unique. It is however easy to extend
with code that removes the blank symbols before and after the non-blank
symbols.

**Exercise 5.1.** Write the *Ploop*-code that cleans up the tape as outlined
above.  \(\square\)

**Exercise 5.2.** Some authors prefer Turing Machines that are only infinite
to the right. We will therefore change the Turing Machine so we never move
left of the leftmost symbol. Show how to do the change. \((\text{Hint: The change}
\text{ is one line.})\) \(\square\)

\(^3\)Too see this fact notice that a polynomial generally has the form $\Sigma_{i=0}^n c_i x^i = c_n x^n +
\cdots + c_1 x^n + c_0$. First, we can assume that all the constants are non-negative as $\Sigma_{i=0}^n c_i x^i \leq
\Sigma_{i=0}^n \max\{0, c_i\} x^i$. Then note that $c_i x^i \leq c_i x^n$ when $x > 0$. Finally take $c = c_n + \cdots + c_0$. 

Exercise 5.3. What is the overhead of the simulation, i.e., given a Turing Machine bounded by the polynomial $q$ what is the bound on running time of the Ploop program. (Hint: Your equation might include the length of the program $m$.)

6 The Real Bellantoni-Cook approach

Finally, let us take a quick look at the original approach taken by Bellantoni-Cook. Instead of Loop they use a characterization of the primitive recursive functions based on functions over non-negative integers $\mathbb{N}$. It is convenient to think of the numbers in binary: 0 is the empty string $\epsilon$, and $n \cdot 0$ is a number where the last bit is 0 (an even number), while the last bit is 1 in $n \cdot 1$. We then have the following definition:

**Definition 6.1.** The class of primitive recursive functions is the least set of functions $\mathbb{N}^k \to \mathbb{N}$, $k \in \mathbb{N}$ which

1. contains the following base functions:
   
   $0() = \epsilon$
   
   $s_b(n) = n \cdot b$
   
   $p(n) = \begin{cases} 
   \epsilon & \text{if } n = \epsilon \\
   n' & \text{if } n = n' \cdot b
   \end{cases}$

   $\pi^i_j(x_1, \ldots, x_j) = x_i$

   $c(x, y, z) = \begin{cases} 
   y & \text{if } x = \epsilon \text{ or } x = x' \cdot 0 \\
   z & \text{if } x = x' \cdot 1
   \end{cases}$

2. is closed under function composition: if the function $f : \mathbb{N}^k \to \mathbb{N}$ and $k$ functions $g_1, \ldots, g_k : \mathbb{N}^l \to \mathbb{N}$ are in the set, then the function

   $f(g_1(x_1, \ldots, x_l), \ldots, g_k(x_1, \ldots, x_l)) : \mathbb{N}^l \to \mathbb{N}$

   is also in the set.

3. is closed under primitive recursion: if the function $g : \mathbb{N}^{k-1} \to \mathbb{N}$ and functions $h_0, h_1 : \mathbb{N}^{k+1} \to \mathbb{N}$ are in the set, then the set also contains the function $f : \mathbb{N}^k \to \mathbb{N}$ defined by

   $f(x_1, x_2, \ldots, x_k) = \begin{cases} 
   g(x_2, \ldots, x_k) & \text{if } x_1 = 0 \\
   h_b(x', x_2, \ldots, x_k, f(x', x_2, \ldots, x_k)) & \text{if } x_1 = x' \cdot b
   \end{cases}$
One can prove that this is equivalent to Loop: It is not too difficult to turn any of these functions into a Loop-program when we use the loop-construction to simulate the primitive recursion. The other way is more tricky as we would need to represent $D_A$ as numbers, but it can be done.

Like us, they recognized\textsuperscript{4} that doing recursion on values that are build using recursion is bad. They then split the arguments into normal and safe arguments where recursion is allowed only on normal arguments. Notationally we distinguish the normal from safe with a semicolon: We write functions as $f(x; y) : \mathbb{N}^k \to \mathbb{N}$ where $\mathbb{N}^k = \mathbb{N}^{k+1}$, $x$ is the $k$ normal arguments, and $y$ are the $l$ safe arguments.

They modify the base functions as follow:
\[
0() = 0 \\
s_b(; x) = x \cdot b \\
\pi_j^k(x_1, \ldots, x_k; y_1, \ldots, y_l) = \begin{cases} x_j & \text{if } 1 \leq j \leq k \\ y_{j-k} & \text{if } k + 1 \leq j \leq k + l \end{cases} \\
c(; x, y, z) = \begin{cases} y & \text{if } x = \varepsilon \text{ or } x = x' \cdot 0 \\ z & \text{if } x = x' \cdot 1 \end{cases}
\]

Function composition is replaced by safe composition where the functions computing the normal arguments only get the normal arguments. In other words: given a function $f : \mathbb{N}^k \to \mathbb{N}$, $k$ functions $g_1, \ldots, g_k : \mathbb{N}^{m,0} \to \mathbb{N}$, and $l$ functions $h_1, \ldots, h_l : \mathbb{N}^{m,n} \to \mathbb{N}$ we define the safe composition as
\[
f(g_1(x); \ldots, g_k(x); h_1(x; y), \ldots, h_l(x; y)) : \mathbb{N}^{m,n} \to \mathbb{N}
\]
So safe arguments can depend on both normal and safe arguments, while normal arguments can only depend on normal arguments.

Safe recursion is primitive recursion restricted so we only recurse over normal arguments: given function $g : \mathbb{N}^{k-1,l} \to \mathbb{N}$ and functions $h_0, h_1 : \mathbb{N}^{k+1} \to \mathbb{N}$ the function $f : \mathbb{N}^{k,l} \to \mathbb{N}$ defined by safe recursion is
\[
f(n; x; y) = \begin{cases} g(x; y) & \text{if } n = \varepsilon \\ h_b(n'; x; y; f(n'; x; y)) & \text{if } n = n' \cdot b \end{cases}
\]
\textsuperscript{4}Actually it would be more fair to say that we, like them, recognized \ldots
Notice that the recursive call is a safe argument so we cannot recurse over it.

Their completeness proof is somewhat different as they do not use characterization based on Turing Machines. Their soundness proof is however very similar to our and depends on the following lemma:

**Lemma 6.2 ([BC92, Lemma 4.1]).** Let $f$ be a function defined by safe recursion and composition from the base functions. Then there is a monotone polynomial $q_f$ such that $|f(\bar{x};\bar{y})| \leq q_f(|\bar{x}|) + \max\{y_1, \ldots, y_l\}$ for all $\bar{x}, \bar{y}$.

Notice how similar this lemma is to our Lemma 4.2.

**References**
