BC^-_ε: A Recursion-Theoretic Characterization of LOGSPACE

Peter Møller Neergaard*

March 7, 2004

Abstract

We present BC^-_ε which is a function algebra that is sound and complete for LOGSPACE. It is based on the novel recursion-theoretic principle of generalized recursion where the step length of primitive recursion can be varied at each step. This allows elegant representations of functions like logarithm and division. Unlike characterizations found in the literature, it is noticeable that there does not appear to be a way to represent pairs in the algebra.

The soundness proof uses a simulation based on "computational amnesia" where, analogously to tail recursion optimization, a recursive call replaces its own activation record. Even though the call is not necessarily tail, we recover the full recursion by repeatedly restarting the computation.

1 Introduction

In this report we present a recursion theoretic characterization of LOGSPACE. It is based on a boiled down version of primitive recursion which we (for lack of a shorter name) call *safe generalized affine recursion*. The basic idea is as follows:

- in the recursive step, recursion over the recursive value is prohibited (*safe recursion*),
- the recursive value can be used only once in the recursive step function (*affinity*), but
- we can skip steps in the recursive chain (*generalized recursion* rather than *primitive recursion*).

The characterization is an alternative to the standard way of characterizing complexity classes through Turing machines: any Turing machine can be used as long as it explicitly can be shown that it adhere to the complexity class. Besides the mathematical simplicity of Turing machines the advantage of this approach is that all standard programming tricks can be used. It does however fail to make clear the implicit limitations of the complexity classes. That is where an alternative characterization like our come in: it is a cumbersome for programming, but it tells us something profound about what programming constructions can make us escape the limited world of LOGSPACE.

This work started as an attempt to understand BC^- which is presented by Murawski and Ong [14] as a first step to connect light affine logic [5, 1] and BC, a function algebra for polynomial time due to Bellantoni and Cook [3]. Mairson established in unpublished

*Supported by the Danish Research Agency grants 1999-114-0027 and 642-00-0062 and the NSF grant CCR-9806718.

work [13] that BC$^-$ can be evaluated in LOGSPACE. The recursion in BC$^-$ has the following features:

- safe recursion,
- affinity, and
- primitive recursion, i.e., the recursion iterates through every step in the chain.

In standard recursion theory the computable functions are described in terms of recursive functions defined from a small set of basic functions and combined through function composition and various recursion principles. A standard exposition is [15] which establishes how more advanced principles like course-of-value recursion over multiple variables can be encoded using primitive recursion over one variable. (Technically this is achieved through a Gödel numbering, but a computer scientist would prefer thinking about it terms of functions working over lists and pairs.) For the theoretician it is therefore not a loss restricting ourselves to primitive recursion.

A similar pattern plays out in Bellantoni and Cook's characterization of polynomial time through safe recursion and composition. The system still admits encoding course-of-value recursion and recursion over multiple variable. And similarly, the system allows one to encode pairs.

However, in BC$^-$, the affinity appears to be a true party-killer. Effectively, it enforces a quantum effect on the recursive result: you can ask for its value, but in doing so you loose the value. In effect: during the course of a recursion, only one bit of information can determine the control in the next step of the recursion. This appears to prevent the encoding of more advanced principles like course-of-value recursion or recursion over multiple variables. This is why we here propose the principle of *generalized recursion* which (as we shall see) allows us to encode recursion over multiple variables.

In more details, BC$^-_\varepsilon$ is a function algebra with a very limited course-of-values: a different step length is allowed at each step of the recursion, e.g., you can recurse through $f(10001111:), f(1000:), f(10:), f(1:), f(0:)$ where the step length is $4, 2, 1, 1$. This is accomplished by having a separate function computing the step length from the normal arguments. Recursion now has the syntactic form $\texttt{rec}(g,h_0,d_0,h_1,d_1)$ and is defined through

$$f(n,\vec{x}:\vec{y}) = \begin{cases} g(\vec{x}:\vec{y}) & \text{if } n = 0 \\ h_b(n',\vec{x}:f(n' \gg d_b(n',\vec{x}:),\vec{x}:\vec{y})) & \text{if } n = 2x'+b \end{cases}$$

where $n \gg i$ right shifts n by i, i.e., drop the bottom i bits of n. Standard primitive recursion arises by simply choosing $d_b(n',\vec{x}) = 0$.

2 BC^-

Notation 2.1. We will represent natural numbers in two forms:

1. As *unary*: a string of 1s with the length being n. We use ε to explicitly denote the empty string. We will use $\mathbb{N}_1$ to make it explicit that we are using natural numbers $\mathbb{N}$ in unary representation.
2. As *binary*: we use $\cdot$ for the concatenation of two binary numbers. For single digits of a binary string we write $n1$ or nb. We do not write 0s to the left of the most significant bit; consequently $0 = \varepsilon$. The set $\mathbb{N}_2$ is the set of natural numbers in binary representation.

In cases of confusion we subscript 1 and 2 to distinguish, i.e., $5 = 101_2 = 11111_1$. □

Definition 2.2. Let $\mathbb{N}^{k,l}$ stand for $\mathbb{N}^k \times \mathbb{N}^l$.

1. We define the set of base functions to be the functions:
   (a) $\mathtt{0} : \mathbb{N}^{0,0} \to \mathbb{N}$ where $\mathtt{0}(:) = 0$,
   (b) $\pi_j^{m,n} : \mathbb{N}^{m,n} \to \mathbb{N}$ with $1 \leq j \leq m+n$

$$\pi_j^{m,n}(x_1, \ldots, x_m : y_1, \ldots, y_n) = \begin{cases} x_j & \text{when } j \leq m \\ y_{j-m} & \text{when } j > m \ , \end{cases}$$

   (c) $\mathtt{p} : \mathbb{N}^{0,1} \to \mathbb{N}$ where

$$\mathtt{p}(: y) = \begin{cases} \varepsilon & \text{when } y = \varepsilon \\ y' & \text{when } y = y'b \ . \end{cases}$$

   (d) $\mathtt{s}_b : \mathbb{N}^{0,1} \to \mathbb{N}$ where $b = \mathtt{0}$ or $b = \mathtt{1}$ and $\mathtt{s}_b(: y) = yb$, and
   (e) $\mathtt{c} : \mathbb{N}^{0,3} \to \mathbb{N}$ where

$$\mathtt{c}(: y_1, y_2, y_3) = \begin{cases} y_2 & \text{when } y_1 = y_1'\mathtt{1} \\ y_3 & \text{otherwise} \ . \end{cases}$$

2. Let the following functions be given: $f : \mathbb{N}^{m,n} \to \mathbb{N}$, $g_1, \ldots, g_m : \mathbb{N}^{m,0} \to \mathbb{N}$, and $h_1 : \mathbb{N}^{m,n_1} \to \mathbb{N}, \ldots, h_n : \mathbb{N}^{m,n_n} \to \mathbb{N}$. Let $n \geq n_1 + \cdots + n_n$ and define safe affine composition of the functions as the following function in $\mathbb{N}^{m,n} \to \mathbb{N}$:

$$(f \circ \langle g_1, \ldots, g_m : h_1, \ldots, h_n \rangle)(x_1, \ldots, x_m : y_1, \ldots, y_n) = \\ f(g_1(\vec{x} :), \ldots, g_m(\vec{x} :) : h_1(\vec{x} : \vec{y_1}), \ldots, h_n(\vec{x} : \vec{y_n}))$$

   where $\vec{x} = x_1, \ldots, x_m$ and $\vec{y_1}, \ldots, \vec{y_n}$ is a division of the variables $y_1, \ldots, y_n$ such that each y_i occurs at most once in any of the vectors $\vec{y_1}, \ldots, \vec{y_n}$.

3. Given functions $g : \mathbb{N}^{m-1,n} \to \mathbb{N}$, $h_0, h_1 : \mathbb{N}^{m,1} \to \mathbb{N}$, $d_0, d_1 : \mathbb{N}^{m,0} \to \mathbb{N}$ we define the *safe affine generalized recursion* of the functions to be the function $f : \mathbb{N}^{m,n} \to \mathbb{N}$ defined as follows:

$$f(n, \vec{x} : \vec{y}) = \begin{cases} g(\vec{x} : \vec{y}) & \text{when } n = \varepsilon \\ h_{b_1}(b_k \cdots b_2, \vec{x} : f(b_k \cdots b_{2+\delta}, \vec{x} : \vec{y})) & \text{when } n = b_k \cdots b_1 \text{ for } k \geq 1 \\ & \text{and } \delta = |d_{b_1}(b_k \cdots b_2, \vec{x} :)| \end{cases}$$

4. The function algebra $\mathrm{BC}_\varepsilon^-$ is the least set of functions over the integers $\mathbb{N}$ containing the base functions and closed under safe affine composition and safe affine generalized recursion.

□

Notation 2.3. Notice that if f is a nullary function (i.e., a constant), e.g., $\mathtt{0}$, then $f \circ \langle : \rangle$ belongs to $\mathbb{N}^m \times \mathbb{N}^n \to \mathbb{N}$ for any m and n. We will therefore write f rather than $f \circ \langle : \rangle$ whereever we use a nullary function.

When the two branches of the recursion use the same step and distance functions, i.e., $\mathtt{rec}(g,h,\delta,h,\delta)$ we will write abbreviate it as $\mathtt{rec}(g,h,\delta)$. □

In [3] the following restriction on the length of the output is proved for Bellantoni and Cook's function algebra.

$$|f(\vec{x};\vec{y})| \leq q_f(|\vec{x}|) + \max_i |y_i| \tag{1}$$

Since BC$^-$ is a subalgebra of Bellantoni and Cook's system the inequality carries over to BC$^-$. Even though BC$^-_\varepsilon$ appears to be more expressive than BC$^-$, the inequality holds for BC$^-_\varepsilon$ as well: in any recursion we will do at most the same number of recursions as with BC$^-$; the proof of (1) in [3] can therefore trivially be extended to BC$^-_\varepsilon$.

3 LOGSPACE completeness

We will now show that BC$^-_\varepsilon$ is complete for LOGSPACE-computations. We will do so by showing given any LOGSPACE Turing machine we can define the following function in BC$^-_\varepsilon$:

> $T(t,s,w,i,j) =$ "the output tape in reverse and preceded by a 1 after t iterations starting in state s with the work tape being w, i the position of the head on the read-only input tape, and j the position of the tape head on the work tape."

Before giving the definition of the function T it seems appropriate to state exactly what we mean by LOGSPACE Turing machine in this paper; after all every author has her own slightly different definition of Turing Machine.

Notation 3.1. A space-bounded Turing machine is a function f and a Turing machine with three tapes: a read-only input tape (denoted I), a work tape (denoted W) with $f(n)$ symbols where n is the length of the input tape, and a write-only output tape (denoted O) infinite to the right. The tapes use only the binary alphabet, i.e., the symbols 0 and 1. The machine's program consists of labeled instructions $l : I$ consecutively numbered as $0, \ldots, S-1$ where I is one of the following commands:

$\texttt{left}_T$ **with** $T = \texttt{I}$ **or** $T = \texttt{W}$**:** move the tape head left on tape T. We assume that the program will never try to move to the left of the leftmost position of the tape. The machine continues with the following instruction.

$\texttt{right}_T$ **with** $T = \texttt{I}$ **or** $T = \texttt{W}$**:** move the tape head right on tape T. We assume that the program will never try to move to the right of the rightmost position of the tape. The machine continues with the following instruction.

$\texttt{if}_T$ $\texttt{then}\ l'\ \texttt{else}\ l''$ **with** $T = \texttt{I}$ **or** $T = \texttt{W}$**:** look at the symbol under the head of tape T: if it is 1 goto program line l' and otherwise line l''.

$\texttt{write}_T\ b$ **with** $T = \texttt{W}$ **or** $T = \texttt{O}$ **and** $b = \texttt{0}$ **or** $b = \texttt{1}$**:** write the symbol b under the tape head of tape T. If $T = \texttt{O}$ move the output tape one to the right. Continue with the following instruction.

We assume that the program idles on a single state, i.e., executes $l : \texttt{if}_\texttt{W}\ \texttt{then}\ l\ \texttt{else}\ l$, when there is no more output to be produced.

A LOGSPACE Turing machine is a space-bounded Turing machine with $f(n) = c\log_2 n$ for some c. □

It should (hopefully) be clear that these conventions do not differ essentially from the reader's favorite definition of a Turing machine.

$$
\begin{aligned}
\mathrm{T}(-1, s, w, i, j, x :) &= 1 \\
\mathrm{T}(t, s, w, i, j, x :) &= \text{let } b_{\mathtt{I}} = \mathrm{bit}(i : x) \text{ in} \\
&\qquad \text{let } b_{\mathtt{W}} = \mathrm{bit}(j : \mathrm{unary2bin}(w :)) \text{ in} \\
&\qquad\quad \mathrm{T}(t-1, \mathrm{S}(b_{\mathtt{W}}, b_{\mathtt{I}}, s :), \mathrm{W}(w, j, s :), \mathrm{I}(i, s :), \mathrm{J}(j, s :), x :) \\
\mathrm{S}(b_{\mathtt{W}}, b_{\mathtt{I}}, s :) &= \begin{cases} l' & \text{if } I_T = \mathtt{if}_T \ \mathtt{then}\ l' \ \mathtt{else}\ l'' \ \mathtt{and}\ b_T = \mathtt{1} \\ l'' & \text{if } I_s = \mathtt{if}_T \ \mathtt{then}\ l' \ \mathtt{else}\ l'' \ \mathtt{and}\ b_T = \mathtt{0} \\ s+1 & \text{otherwise} \end{cases} \\
\mathrm{W}(w, j, s :) &= \begin{cases} \mathrm{bin2unary}(\mathrm{SET}_b(j, \mathrm{unary2bin}(w :) :) & \text{if } I_s = \mathtt{write}_{\mathtt{W}}\ b \\ w & \text{otherwise} \end{cases} \\
\mathrm{I}(i, s :) &= \begin{cases} i+1 & \text{if } I_s = \mathtt{right}_{\mathtt{I}} \\ i-1 & \text{if } I_s = \mathtt{left}_{\mathtt{I}} \\ i \end{cases} \\
\mathrm{J}(j, s :) &= \begin{cases} j+1 & \text{if } I_s = \mathtt{right}_{\mathtt{W}} \\ j-1 & \text{if } I_s = \mathtt{left}_{\mathtt{W}} \\ j \end{cases} \\
\mathrm{bit}(i : x) &= \mathtt{c}(: \mathrm{shift}^{\mathrm{R}}(i : x), 1, 0) \\
\mathrm{shift}^{\mathrm{R}}(i : x) &= \begin{cases} x & \text{when } i = \varepsilon \\ p(: \mathrm{shift}^{\mathrm{R}}(i' : x)) & \text{when } i = ni' \end{cases} \\
\mathrm{SET}_b(n, b_k \cdots b_0 :) &= \begin{cases} b_k \cdots b_{|m|+1} b b_{|m|-1} \cdots b_0 & \text{when } 0 \le k \le |m| \\ b b_{k-1} \cdots b_0 & \text{otherwise} \end{cases}
\end{aligned}
$$

Figure 1: The function T simulating a LOGSPACE Turing machine.

We can now turn to the function T; the function is presented in pseudo-code in Fig. 1. The reminder of this section will be devoted to showing that the function can be encoded in $\mathrm{BC}^-_\varepsilon$. That is done by showing that the functions involved can be coded in $\mathrm{BC}^-_\varepsilon$ and that the recursion over multiple variables can be turned into a recursion over a single variable.

Proposition 3.2. *The following functions are definable in $\mathrm{BC}^-_\varepsilon$:*

1. $\mathrm{shift}^{\mathrm{R}} : \mathbb{N}^{1,1} \to \mathbb{N}$ *where*

$$\mathrm{shift}^{\mathrm{R}}(m : b_k \cdots b_1 b_0) = \begin{cases} b_k \cdots b_{|m|} & \textit{when } k \geq |m| \\ \mathtt{0} & \textit{otherwise} \end{cases}$$

2. $\mathrm{bit} : \mathbb{N}^{1,1} \to \{0,1\}$ *where*

$$\mathrm{bit}(m : b_k \cdots b_1 b_0) = \begin{cases} b_{|m|} & \textit{when } k \geq |m| \\ \mathtt{0} & \textit{otherwise} \end{cases}$$

□

Proof. The function $\mathrm{shift}^{\mathrm{R}}$ is as outlined in Fig. 1. It is represented by the following $\mathrm{BC}^-_\varepsilon$ definition

$$\mathtt{rec}(\pi_1^{0,1}, \mathtt{p} \circ \langle : \pi_2^{1,1} \rangle, \mathtt{0}) \ .$$

It follows by induction on m that it has the required properties.

Given $\mathrm{shift}^{\mathrm{R}}$, we find bit as given in Fig. 1, i.e.,

$$\mathtt{c} \circ \langle \mathrm{shift}^{\mathrm{R}}, \mathtt{1}, \mathtt{0} \rangle \ .$$

The correctness follows from the correctness of $\mathrm{shift}^{\mathrm{R}}$. □

Proposition 3.3. *Let b be either* 0 *or* 1*. The following class of functions is representable in $\mathrm{BC}^-_\varepsilon$:*

$$\mathrm{SET}_b(m, b_k \cdots b_0 :) = \begin{cases} b_k \cdots b_{|m|+1} b b_{|m|-1} \cdots b_1 & \textit{when } k \geq |m|+1 \\ b b_{k-1} \cdots b_0 & \textit{otherwise} \end{cases} \ .$$

□

Notation 3.4. We use all capital letters for COND since it is macro over $\mathrm{BC}^-_\varepsilon$-functions rather than a function in itself. □

Lemma 3.3. We represent the function as follows:

$$\begin{aligned}
\mathrm{SET}_b(m, b_k \cdots b_0 :) &= \mathrm{SET}'(b_k \cdots b_0, b_k \cdots b_{|m|+1} b :) \\
&= \mathrm{SET}'(b_k \cdots b_0, \mathtt{s}_b(: \mathtt{p}(: \mathrm{shift}^{\mathrm{R}}(m : b_k \cdots b_0))) :) \\
&= \mathrm{SET}' \circ \langle \pi_2^{2,0}, \mathtt{s}_b \circ \langle : \mathtt{p} \circ \langle : \mathrm{shift}^{\mathrm{R}} \circ \langle \pi_1^{2,0} : \pi_2^{2,0} \rangle \rangle \rangle \rangle \\
\mathrm{SET}'(b_k \cdots b_0, t :) &= \begin{cases} t & \text{when } k+1 \leq |t| \\ \mathrm{SET}'(b_k \cdots b_1, t :) b_0 & \text{otherwise} \end{cases} \\
&= \begin{cases} t & \text{when } k < 0 \\ \mathtt{c}(: \mathrm{zero?}(\mathrm{shift}^{\mathrm{R}}(\mathtt{p}(: t) : b_k \cdots b_1) :), & \text{when } k \geq 0 \\ \quad t, \\ \quad \mathtt{s}_{b_0}(: \mathrm{SET}'(b_k \cdots b_1, t :))) \end{cases} \\
&= \mathtt{rec}(\pi_1^{1,0}, \\
&\qquad \mathtt{c} \circ \langle \mathrm{zero?} \circ \langle \mathrm{shift}^{\mathrm{R}} \circ \langle \mathtt{p} \circ \langle : \pi_2^{2,0} \rangle : \pi_1^{2,0} \rangle \rangle, \pi_2^{2,0}, \mathtt{s}_0 \circ \langle : \pi_3^{2,1} \rangle \rangle, \\
&\qquad \mathtt{0}, \\
&\qquad \mathtt{c} \circ \langle \mathrm{zero?} \circ \langle \mathrm{shift}^{\mathrm{R}} \circ \langle \mathtt{p} \circ \langle : \pi_2^{2,0} \rangle : \pi_1^{2,0} \rangle \rangle, \pi_2^{2,0}, \mathtt{s}_1 \circ \langle : \pi_3^{2,1} \rangle \rangle, \\
&\qquad \mathtt{0}) \\
\end{aligned}$$

For correctness, we prove that $\mathrm{SET}'(b_k \cdots b_0, t :) = t b_{k-|t|} \cdots b_0$ using induction over k with the base case being $k < |t|$. From this it follows that

$$\begin{aligned}
\mathrm{SET}_b(m, b_k \cdots b_0 :) &= \mathrm{SET}'(b_k \cdots b_1, b_k \cdots b_{|m|+1} b :) \\
&= b_k \cdots b_{|m|+1} b b_{k-(k-|m|+1)} \\
&= b_k \cdots b_{|m|+1} b b_{|m|-1} \ .
\end{aligned}$$

□

We then continue with the arithmetic:

Proposition 3.5. *Let m and n by integers in unary notation. Then the following functions are representable BC_{ε}^{-}:*

1. $\mathrm{plus}(m : n) = m + n$,
2. $\mathrm{minus}(n : m) = \max(m - n, 0)$,
3. $\mathrm{mult}(m, n :) = m \cdot n$,
4. $\mathrm{div}(m, n :) = \lfloor m/n \rfloor$ *where* $n \neq 0$,
5. $\mathrm{mod}(m, n :) = m \bmod n$ *where* $n \neq 0$,
6. $\mathrm{zero?}(m :) = \mathtt{1}$ *when* $m = 0$, $\mathrm{zero?}(m :) = \mathtt{0}$ *when* $m > 0$, *and*
7. $<?(m, n :) = \mathtt{1}$ *when* $m < n$ *and* $<?(m, n :) = \mathtt{0}$ *when* $m \geq n$.

□

Proof. First, Case 1 where we obtain addition through the following program:

$$\mathrm{plus}(m : n) = \begin{cases} n & \text{when } m = \varepsilon \\ 1 + \mathrm{plus}(m' : n) & \text{when } m = m' + 1 \end{cases} = \mathtt{rec}(\pi_1^{0,1}, \mathtt{0}, \mathtt{0}, \mathtt{s}_1 \circ \langle : \pi_2^{1,1} \rangle, \mathtt{0}) \ . \quad (2)$$

By induction on n it computes addition.

As for subtraction (Case 2) we do as for addition, but subtract one instead of adding one:

$$\mathrm{minus}(n : m) = \begin{cases} m & \text{when } n = \varepsilon \\ \mathrm{minus}(n' : m) - 1 & \text{when } n = n' + 1 \end{cases} = \mathtt{rec}(\pi_1^{0,1},0,0,\mathtt{p}\circ\langle : \pi_2^{1,1}\rangle,0) \ .$$

The correctness follows by induction on n exploiting that $\mathtt{p}$ never counts below 0.

For Case 3, we use repeated addition:

$$\begin{aligned} \mathrm{mult}(m, n :) &= \begin{cases} 0 & \text{when } m = \varepsilon \\ \mathrm{plus}(n : \mathrm{mult}(m', n :)) & \text{when } m = m' + 1 \end{cases} \\ &= \mathtt{rec}(0,0,0,\mathrm{plus}\circ\langle\pi_2^{2,0} : \pi_3^{2,1}\rangle,0) \ . \end{aligned}$$

In this definition plus serves as an abbreviation for the program in (2). Correctness is proved by induction on m.

As for Case 4, the generalized recursion principle of $\mathrm{BC}_\varepsilon^-$ yields a very simple definition:

$$\begin{aligned} \mathrm{div}(m, n :) &= \mathrm{div}'(m + 1, n :) - 1 \\ \mathrm{div}'(m, n :) &= \begin{cases} 0 & \text{when } m \leq 0 \\ 1 + \mathrm{div}'(m - n, n :) & \text{when } m \geq n \end{cases} \end{aligned}$$

This results in the following $\mathrm{BC}_\varepsilon^-$ encoding of div

$$\mathrm{div}(m, n :) = \mathtt{p}\circ\langle : \mathtt{rec}(0,0,0,\mathtt{s}_1\circ\langle : \pi_3^{2,1}\rangle,\pi_2^{2,0})\circ\langle\mathtt{s}_1\circ\langle : \pi_1^{2,0}\rangle, \mathtt{p}\circ\langle : \pi_2^{2,0}\rangle : \rangle\rangle \ .$$

where we have to subtract one from n since rec adds one to the displacement. Correctness follows by induction over $m \div n$.

For Case 5 we simply combine mult, div, and minus:

$$\begin{aligned} \mathrm{mod}(m, n :) &= m - \lfloor m/n \rfloor \cdot n \\ &= \mathrm{minus}(\mathrm{mult}(\mathrm{div}(m, n :), n :) : m) \\ &= \mathrm{minus}\circ\langle\mathrm{mult}\circ\langle\mathrm{div}\circ\langle\pi_1^{2,0}, \pi_2^{2,0} : \rangle, \pi_2^{2,0} : \rangle : \pi_1^{2,0}\rangle \end{aligned}$$

Finally, we get the zero-test of Case 6 through the following piece of code:

$$\mathrm{zero?}(m :) = \begin{cases} 1 & \text{when } m = \varepsilon \\ 0 & \text{otherwise} \end{cases} = \mathtt{rec}(1,0,0) \ .$$

From this we derive $<?$ as

$$<?(m, n :) = \mathrm{zero?}(\mathrm{shift}^{\mathrm{R}}(n : m) :) = \mathrm{shift}^{\mathrm{R}}\circ\langle\pi_2^{2,0} : \pi_1^{2,0}\rangle \ .$$

□

REMARK 3.6. Though we use the generalized recursion principle of $\mathrm{BC}_\varepsilon^-$ division (and thus modulus), it should be noted that it can be encoded BC^- by searching through $i = 0, \ldots, m$ finding the largest i such that $i \cdot n \leq m$. □

A limitation on the conditional in $\mathrm{BC}_\varepsilon^-$ is that even though only one of the branches is actually taken the two branches cannot share any safe arguments. In a recursion this prevents using a conditional to choose between two different actions to the recursive element. Using recursion, we can regain a conditional that shares the variables of the two branches; it comes with the cost of being controlled by a normal variable though:

Lemma 3.7. *Let $f, g : \mathbb{N}^{0,1} \to \mathbb{N}$ be two functions representable in BC^-_ε. Then the following function is representable in BC^-_ε:*

$$\mathrm{COND}_{f,g}(m : n) = \begin{cases} f(: n) & \textit{when } m \bmod 2 = 1 \\ g(: n) & \textit{when } m \bmod 2 = 0 \end{cases}$$

□

Proof. We give the encoding when m is a unary number:

$$\begin{aligned} \mathrm{COND}_{f,g}(m : n) &= \mathrm{IF}_f(\mathrm{mod}(m, 2 :) : \mathrm{IF}_g(\mathrm{not}(: \mathrm{mod}(m, 2 :)) :)) \\ &= \mathrm{IF}_f \circ \langle \mathrm{mod} \circ \langle \pi_1^{1,0}, 2_1 : \rangle : \mathrm{IF}_g \circ \langle \mathrm{not} \circ \langle : \mathrm{mod} \circ \langle \pi_1^{1,0}, 2_1 : \rangle \rangle : \pi_2^{1,1} \rangle \rangle \\ \mathrm{IF}_f(m : n) &= \begin{cases} n & \text{when } m = \varepsilon \\ f(: \mathrm{IF}_f(m' : n)) & \text{when } m = m' + 1 \end{cases} \\ &= \mathtt{rec}(\pi_1^{0,1}, 0, 0, f \circ \langle : \pi_2^{1,1} \rangle, 0) \\ \mathrm{not}(: m) &= \mathtt{c} \circ \langle : \pi_1^{0,1}, 0, 1 \rangle \end{aligned}$$

If m is in binary, we replace $\mathrm{mod}(m, 2 :)$ by $\mathtt{c}(: m, 1, 0)$. By case-analysis of $m \bmod 2 = 0$, we find that the function has the desired property. □

Another useful function is reversing a bit string. Since we are working with numbers, not bit vectors, one should notice we do not always have $\mathrm{rev}(\mathrm{rev}(x :) :) = x$.

Lemma 3.8. *There is a BC^-_ε function $\mathrm{rev} : \mathbb{N}_2 \times \mathbb{N}_1 \to \mathbb{N}$ that reverses a bit string, i.e., given a number n in unary $\mathrm{rev}(b_k \cdots b_0, n :) = b_0 b_1 \cdots b_{k-n}$.* □

Proof. We use the following definition:

$$\begin{aligned} \mathrm{rev}(m, n :) &= \mathrm{rev}'(\mathrm{shift}^{\mathrm{R}}(n : m), m, n :) = \mathrm{rev}' \circ \langle \mathrm{shift}^{\mathrm{R}} \circ \langle \pi_2^{2,0} : \pi_1^{2,0} \rangle, \pi_1^{2,0}, \pi_2^{2,0} : \rangle \\ \mathrm{rev}'(m, \overline{m}, n :) &= \begin{cases} 0 & \text{when } m = \varepsilon \\ \mathtt{s}_{b_{k-i}}(: \mathrm{rev}'(b_i \cdots b_1, \overline{m}, n :)) & \text{when } m = b_i \cdots b_1 b_0 \text{ for } i \geq 0 \\ & \text{and } \overline{m} = b_k \cdots b_1 b_0 \end{cases} \\ &= \begin{cases} 0 & \text{when } m = \varepsilon \\ \mathrm{COND}_{\mathtt{s}_1, \mathtt{s}_0}(\mathrm{shift}^{\mathrm{R}}(m' : \overline{m}) : \mathrm{rev}'(m', \overline{m}, n :)) & \text{when } m = m' \cdot b \end{cases} \\ &= \mathtt{rec}(0, \mathrm{COND}_{\mathtt{s}_1, \mathtt{s}_0} \circ \langle \mathrm{shift}^{\mathrm{R}} \circ \langle \pi_1^{3,0} : \pi_2^{3,0} \rangle \rangle : \pi_4^{3,1}, 0) \end{aligned}$$

The correctness follows as follows: As n is in unary we have $|n| = n$. We therefore recurse through the values $i = |m| - n, |m| - n - 1, \ldots, 0$. We observe that $|m'| = i$ and consequently $\mathrm{shift}^{\mathrm{R}}(m' : \overline{m}) = b_k \cdots b_i$. It follows by induction on i that the result is

$$\mathtt{s}_{b_{|m|-n}}(: \mathtt{s}_{b_{|m|-n-1}}(: \cdots : \mathtt{s}_{b_0}(: 0)) \cdots)) = b_0 b_1 \cdots b_{k-n-1} b_{k-n} \ .$$

□

As the next step, we will show how to convert between binary and unary notation. This involves computing the logarithm which the author has failed to represent in BC^-.

Lemma 3.9. *Let m by an integer in unary notation. Then there is a BC^-_ε-function $\log(m :) = \lfloor \log_2 m \rfloor + 1$ when $m \geq 0$ and $\log(0 :) = 0$.* □

Proof. We define $\log$ as follows:

$$\log(m :) = \begin{cases} 0 & \text{when } m = 0 \\ 1 + \log(m \div 2 :) & \text{when } m > 0 \end{cases}$$

which is implemented by the following $\mathrm{BC}^-_\varepsilon$ program:

$$\log(m :) = \mathtt{rec}(0,0,0,\mathtt{s}_1 \circ \langle : \pi_2^{1,1} \rangle, \mathrm{div} \circ \langle \pi_1^{1,0}, 2_1 : \rangle) \ .$$

The correctness for $m = 0$ is immediate. For $m > 0$ we use induction over i where $m = 2^i + j$ for $0 \leq j \leq 2^i - 1$:

- $i = 0$: we can only have $j = 0$ and thus $m = 1$. We immediately have

$$\log(1 :) = \mathtt{s}_1 \circ \langle : \log(0 :) \rangle \qquad = 1 \ .$$

- $i > 0$: By case analysis, we notice that $m \div 2 = (m-1) - (m-1) \div 2$ as $m > 0$. We find that for all $m = 2^i + 0, \ldots, 2^i + (2^i - 1)$ we have

$$\begin{aligned} (m-1) - (m-1) \div 2 = (2^i + j - 1) - \lfloor (2^i + j - 1)/2 \rfloor = \\ 2^i + (j-1) - 2^{i-1} - \lfloor (j-1)/2 \rfloor = \\ 2^{i-1} + \lceil (j-1)/2 \rceil = \\ 2^{i-1} + j' \end{aligned}$$

  for some j' with $0 \leq j' \leq 2^{i-1} - 1$. From the induction hypothesis we know that $\log(2^{i-1} + j' :)$ is correct for all j'. As $(m-1) \div 2 = \mathrm{div} \circ \langle \pi_1^{1,0}, 2_1 : \rangle$, the function's recursive call goes to $\log(m \div 2 :)$. It follows that $\log(m :) = \lfloor \log_2 m \rfloor + 1$.

□

In converting from binary to unary we need the powers of 2. It is generally not possible to do exponentation in $\mathrm{BC}^-_\varepsilon$ as it would break the bound in (1). We can however test whether a number is a power and two. By searching all the numbers between $0, \ldots, m$ we find the largest power of 2 smaller than m. This is captured in the following lemma:

Lemma 3.10. *Let $m, n \in \mathbb{N}$ be integers coded in unary representation. The following functions are representable in $\mathrm{BC}^-_\varepsilon$:*

1. $\mathrm{power?}(m, n :) = 1$ *if, and only if, m is a power of n, i.e.*

$$\mathrm{power?}(m, n :) = \begin{cases} 1 & \textit{when } \exists i \in \mathbb{N}. m = n^i \\ 0 & \textit{otherwise} \end{cases}$$

2. $\mathrm{largest\_power}(m, n :) = \max(\{n^i \leq m \mid i \in \mathbb{N}\} \cup \{0\})$

□

Proof. We start with Case 1 which we code by noting that repeated division by n should not produce a remainder until we reach 1:

$$\text{power?}(m, n :) = \begin{cases} \mathtt{0} & \text{when } m = \varepsilon \\ \mathtt{1} & \text{when } m = \mathtt{1} \\ \text{power?}(\lfloor m/n \rfloor, n :) & \text{when } m = m' + 1 \text{ and } m \bmod n = 0 \\ \mathtt{0} & \text{otherwise} \end{cases}$$

$$= \begin{cases} \mathtt{0} & \text{when } m = \varepsilon \\ \mathtt{c}(: \text{ zero?}(m' :), & \text{when } m = m' + 1 \\ \quad \mathtt{1}, & \\ \quad \mathtt{c}(: \text{ zero?}(\text{mod}(\mathtt{s}_1(: m'), n :) :), & \\ \qquad \text{power?}(m' - (m' - \text{div}(m' + 1, n :)), n :), & \\ \qquad \mathtt{0})) & \end{cases}$$

$$= \mathtt{rec}(\, \mathtt{0},$$
$$\mathtt{0},$$
$$\mathtt{0},$$
$$\mathtt{c} \circ \langle \text{zero?} \circ \langle \pi_1^{2,0} \rangle, \mathtt{1}, \mathtt{c} \circ \langle \text{zero?} \circ \langle \text{mod} \circ \langle \mathtt{s}_1 \circ \langle : \pi_1^{2,0} \rangle, \pi_2^{2,0} \rangle : \rangle, \pi_3^{2,1}, \mathtt{0} \rangle \rangle,$$
$$\text{minus} \circ \langle \text{div} \circ \langle \mathtt{s}_1 \circ \langle : \pi_1^{2,0} \rangle, \pi_2^{2,0} \rangle : \pi_1^{2,0} \rangle)$$

The correctness proof comes in two steps:

1. The "if"-part, i.e., that when $m = n^i$ for some i the function return 1. We use induction over i:

   - $i = 0$: We have $m = 1$ and the result follows immediately since $m' = \varepsilon$ in the first recursive call. Consequently, $\text{zero?}(m' :)$ returns $\mathtt{1}$ and the conditional chose the first branch.
   - $i > 0$: As $m^i = m \cdot m^{i-1}$ we know that $m \bmod n = 0$ and thus

     $$(\text{zero?} \circ \langle \text{mod} \circ \langle \mathtt{s}_1 \circ \langle : \pi_1^{2,0} \rangle, \pi_2^{2,0} \rangle : \rangle)(m', n :) = \mathtt{1}$$

     so we are taking the recursive call. We find that

     $$\begin{aligned} m' - (m' - \text{div}(m' + 1, n :)) &= n^i - 1 - (n^i - 1 - \text{div}(n^i - 1 + 1, n :)) \\ &= \text{div}(n^i, n :) \\ &= n^{i-1} \end{aligned}$$

     so the recursive call returns $\text{power?}(n^{i-1}, n :)$ which is 1 by the induction hypothesis.

2. For then "only if"-part we assume that $\text{power?}(m, n :) = 0$. By induction on the number of recursive calls k, we find that either $m = 0$ or $m = n^k \cdot p$ where n does not divide p and $p \geq 2$:

   - $k = 0$: this is immediate from the definition.
   - $k > 0$: Since we are taking the recursive call, we must have $m = n \cdot q$ for some q. We find that:

     $$\begin{aligned} m' - (m' - \text{div}(m' + 1, n :)) &= n \cdot q - 1 - (n \cdot q - 1 - \text{div}(n \cdot q - 1 + 1, n :)) \\ &= \text{div}(n \cdot q, n :) \\ &= q \ . \end{aligned}$$

We have $\text{power?}(q, n :) = 0$ using $k-1$ recursive calls and thus by induction hypothesis, $q = n^{k-1} \cdot p$ for some p where n does not divided p and $p \geq 2$. We immediately find $m = n^k \cdot q$.

It follows that m is not a power of n.

It is now straightforward to find the largest power of n that is less than or equal to m:

$$\begin{aligned}
\text{largest_power}(m, n :) &= \begin{cases} \mathtt{0} & \text{when } m = \varepsilon \\ m & \text{when } m \neq \varepsilon \\ & \text{and } \text{power?}(m, n :) = \mathtt{1} \\ \text{largest_power}(m-1, n :) & \text{otherwise} \end{cases} \\
&= \begin{cases} \mathtt{0} & \text{when } m = \varepsilon \\ \mathtt{c}(:\text{power?}(m'+1, n :), & \text{when } m = m'+1 \\ \quad m'+1, & \\ \quad \text{largest_power}(m', n :)) & \end{cases} \\
&= \mathtt{rec}(0,0,0,\mathtt{c} \circ \langle \text{power?} \circ \langle \mathtt{s}_1 \circ \langle \pi_1^{2,0} \rangle, \pi_2^{2,0} \rangle, \mathtt{s}_1 \circ \langle : \pi_1^{2,0} \rangle, \pi_3^{2,1} \rangle, 0) \ .
\end{aligned}$$

The correctness follows by induction over m. □

We now have the tools to code the functions converting between the two representations of integers. When converting from binary to unary, the output might grow exponentially. This obviously violates the bound in (1) and we therefore has to provide an extra argument limiting the size.

Proposition 3.11.

1. *The function* $\text{unary2bin} : \mathbb{N}_1 \to \mathbb{N}_2$ *is representable in* BC_ε^-.
2. *The following function is representable in* BC_ε^-*:* $\text{bin2unary} : \mathbb{N}_2 \times \mathbb{N}_1 \to \mathbb{N}_1$ *where* $\text{bin2unary}(m, n :)$ *is the unary representation of* m *provided that* $m \leq n$.

□

Proof. As for turning a number in unary into binary we simply use the standard algorithm for finding the representation of number in a given radix. The only problem is that the straightforward implementation provides the bits in the wrong order. Consequently, we first convert the number and then reverse it.

$$\begin{aligned}
\text{unary2bin}(m :) &= \text{rev}(\text{unary2bin}'(m :), \mathtt{1} :) \\
&= \text{rev} \circ \langle \text{unary2bin}' \circ \langle \pi_1^{1,0} : \rangle, \mathtt{1} : \rangle \\
\text{unary2bin}'(m :) &= \begin{cases} \mathtt{1} & \text{when } m = \varepsilon \\ \mathtt{s}_0(\text{unary2bin}'(\lfloor m/2 \rfloor :)) & \text{when } m = m'+1 \text{ and } \text{mod}(m, 2 :) = \mathtt{0} \\ \mathtt{s}_1(\text{unary2bin}'(\lfloor m/2 \rfloor :)) & \text{when } m = m'+1 \text{ and } \text{mod}(m, 2 :) = \mathtt{1} \end{cases} \\
&= \mathtt{rec}(0,0,0,\text{COND}_{\mathtt{s}_0,\mathtt{s}_1} \circ \langle \pi_1^{1,0} : \pi_2^{1,1} \rangle, \text{div} \circ \langle \pi_1^{1,0}, 2_1 : \rangle) \ .
\end{aligned}$$

Since $\text{unary2bin}'$ provides the bits in reverse order, we prefix the string with 1 as we would otherwise loose bits; consequently we use the second argument of reverse to specify that we should ignore the leftmost bit of the string.

For correctness, we first prove the correctness of $\text{unary2bin}'$ using induction on m. The claim is that when m is a unary number with a binary representation of $b_k \cdots b_1$ then $\text{unary2bin}'(m :) = \mathtt{1} b_1 \cdots b_k$:

- $m = 0$: It is immediate from the definition that $\text{unary2bin}'(\mathtt{0}:) = \mathtt{1}$.
- $m = 2m' + i > 0$ with $i = 0$ or $i = 1$. As m is unary we have $|m| = m$. The distance function is applied to $m - 1$ so we get

  $$\text{div}\circ\langle \pi_1^{1,0}, \mathtt{2}_1 : \rangle(m - 1 :) = \text{div}(m - 1, \mathtt{2}_1 :) \ .$$

  and thus recursively call $\text{unary2bin}(m_{\text{rec}} :)$ with $m_{\text{rec}} = m - 1 - \text{div}(m - 1, \mathtt{2}_1 :)$. We split into cases based on i:

  - $i = 0$: We have $m - 1 = 2(m' - 1) + 1$ and thus find $m_{\text{rec}} = m - 1 - (m' - 1) = m'$.
  - $i = 1$: We have $\text{div}\circ\langle m, \mathtt{2}_1 : \rangle = m'$ and therefore $m_{\text{rec}} = m - 1 - m' = m'$.

  We thus recurse to m' in both cases. We have from the induction hypothesis that $\text{unary2bin}'(m' :) = \mathtt{1}b_1 \cdots b_{k-1}$. It thus follows that $\text{unary2bin}'(m :) = \mathtt{1}b_1 \cdots b_{k-1}i = \mathtt{1}b_1 \cdots b_k$.

This establishes that $\text{unary2bin}'$ finds the binary digits in reverse. Using the correctness of rev we have the correctness of unary2bin.

As for bin2unary we use largest_power to find the highest power of 2 so we can find the most significant bit to query for:

$$\text{bin2unary}(m, n :) = \text{bin2unary}'(\text{largest\_power}(n, \mathtt{2} :), m :)$$

$$\text{bin2unary}'(p, m :) = \begin{cases} \mathtt{0} & \text{when } p = \mathtt{0} \\ p + \text{bin2unary}'(p \div \mathtt{2}, m :) & \text{when } p = p' + 1 \\ & \text{and } \text{bit}(\log(p :) : m) = \mathtt{1} \\ \text{bin2unary}'(p \div \mathtt{2}, m :) & \text{otherwise} \end{cases}$$

$$= \begin{cases} \mathtt{0} & \text{when } p = \mathtt{0} \\ \text{plus}(\mathtt{c}(: \text{bit}(\log(p' + 1 :) - 1 : m), p, \mathtt{0}) : & \text{when } p = p' + 1 \\ \quad \text{bin2unary}'((p' + 1) \div \mathtt{2}, m :) & \end{cases}$$

$$= \mathtt{rec}(\mathtt{0},$$
$$\mathtt{0},$$
$$\mathtt{0},$$
$$\text{plus}\circ\langle \mathtt{c}\circ\langle :\text{bit}\circ\langle \mathtt{p}\circ\langle : \log\circ\langle \mathtt{s}_1\circ\langle : \pi_1^{2,0}\rangle : \rangle\rangle : \pi_2^{2,0}\rangle,$$
$$\mathtt{s}_1\circ\langle : \pi_1^{2,0}\rangle,$$
$$\mathtt{0}\rangle, \pi_3^{2,1}\rangle,$$
$$\text{div}\circ\langle \pi_1^{2,0}, \mathtt{2} : \rangle)$$

It is immediate that the function is correct when $m = 0$ where $\text{largest\_power}(\mathtt{0}, \mathtt{2} :) = \mathtt{0}$. For $m \geq 1$, we have $\text{largest\_power}(\mathtt{0}, \mathtt{2} :) = 2^k$ for some $k \geq 0$. We use long induction on k to show that $\text{bin2unary}'(2^k, m :) = (b_k \cdots b_0)_{\mathtt{1}}$ when $m_{\mathtt{2}} = b_l \cdots b_0$. We notice that the recursive call goes to $2^k - 1 - (2^k - 1) \div 2 = 2^k - 1 - (2^{k-1} - 1) = 2^{k-1}$ and split into two cases:

- $b^k = 0$: We have $\log(2^k :) = k + 1$ and $\text{bit}(\mathtt{p}(: \log(p' + 1 :)) : m) = \mathtt{0}$. Consequently, $\text{bin2unary}'(2^k, m :) = \text{bin2unary}'(2^{k-1}, m :)$ and correctness follows from the induction hypothesis.

- $b^k \geq 1$: We have $\mathrm{bit}(\mathtt{p}(:\log(p'+1:)):m) = \mathtt{1}$ and thus

$$\begin{aligned}\mathrm{bin2unary}'(2^k, m:) &= (2^k)_1 +_1 \mathrm{bin2unary}'(2^{k-1}, m:)\\ &= (2^k)_1 +_1 (b_{k-1}\cdots b_0)_1\\ &= (b_k \cdots b_0)_1\ .\end{aligned}$$

This concludes the proof of correctness for bin2unary. □

We have now shown that all parts of the Turing machine encoding in Fig. 1 are in $\mathrm{BC}^-_\varepsilon$. We can therefore show that the function is in $\mathrm{BC}^-_\varepsilon$.

Proposition 3.12. *The function* T *given in Fig. 1 is representable in* BC^-_ε. □

Proof. The functions used in Fig. 1 are representable in $\mathrm{BC}^-_\varepsilon$. There are thus only two remaining issues: how to turn the recursion over several variables into a recursion over one argument, and how to represent the step functions S, W, I, and J.

We first deal with the step functions. The conditions, e.g., "$I_s = \mathtt{right}_\mathrm{I}$" can all statically be turned into comparisons on the numerical value of s. Given the program, $0 : I_0, \ldots, S-1 : I_{S-1}$, the function I becomes

$$\begin{aligned}\mathrm{I}(i, s:) = \ & \mathtt{c}(:<?(s,\mathtt{1}:), f_0(:i),\\ & \qquad \mathtt{c}(:<?(s,\mathtt{2}:), f_1(:i),\\ & \qquad\qquad \cdots,\\ & \qquad\qquad \mathtt{c}(:<?(s,S_1:), f_{S-1}(:i), f_S(:i))\cdots))\\ = \ & \mathtt{c}\circ\langle :<?\circ\langle\pi_2^{2,0}, \mathtt{1}:\rangle,\\ & \quad f_0\circ\langle:\pi_1^{2,0}\rangle,\\ & \quad \mathtt{c}\circ\langle :<?\circ\langle\pi_2^{2,0}, \mathtt{2}:\rangle,\\ & \qquad f_1\circ\langle:\pi_1^{2,0}\rangle,\\ & \qquad \cdots\\ & \qquad \mathtt{c}\circ\langle :<?\circ\langle\pi_2^{2,0}, (S-1)_1:\rangle, f_{S-2}\circ\langle:\pi_1^{2,0}\rangle, f_{S-1}\circ\langle:\pi_1^{2,0}\rangle\rangle\rangle\rangle\end{aligned}$$

where

$$f_k = \begin{cases}\mathtt{s}_1 & \text{if } I_k = \mathtt{right}_\mathrm{I}\\ \mathtt{p} & \text{if } I_k = \mathtt{left}_\mathrm{I}\\ \pi_1^{0,1}\end{cases}$$

The constructions for J, W, and S are similar. In W we exploit that the number of configurations of the work tape is $2^{c\cdot\log_2 n} = n^c$. We augment W with a fourth argument, the input tape x. Taking w' to be the updated work tape $\mathrm{SET}_b(j, \mathrm{unary2bin}(w:):)$, we can therefore compute its unary representation as

$$\begin{aligned}\mathrm{bin2unary}(w', n^c:) &= \mathrm{bin2unary}(w', \overbrace{\mathrm{mult}(x, \mathrm{mult}(x, \cdots \mathrm{mult}(x, x:):):)}^{c-1 \text{ mults}}:)\\ &= \mathrm{bin2unary}\circ\langle \mathrm{SET}_b(:)\circ\langle\pi_2^{4,0}, \mathrm{unary2bin}\circ\langle\pi_1^{4,0}:\rangle\rangle,\\ &\qquad\qquad \mathrm{mult}\circ\langle\pi_4^{4,0}, \cdots \mathrm{mult}\circ\langle\pi_4^{4,0}, \pi_4^{4,0}\rangle\rangle\rangle\end{aligned}$$

For the mutual recursion, we observe that there are the following bounds: $0 \leq s < S$, $0 \leq i < n$, $0 \leq j < \log_2 n \leq n$. Since the Turing machine cannot twice be in the same state without doing an infinite loop, the maximum number of steps the machine can take before idling infinitely is $N = S \cdot n \cdot n \cdot n^c = S \cdot n^{c+2}$. We can encode the tuple (t, s, w, i, j) uniquely as the number:

$$t \cdot N + (((w \cdot n) + i) \cdot n + j) \cdot S) + s \ . \tag{3}$$

By representing this number in unary, we can use the $\mathrm{BC}^-_\varepsilon$ recursion scheme to recurse over the tuple:

$$\begin{aligned}
\mathrm{t}(m, n, x :) &= \begin{cases} \mathtt{1} & \text{when } m = \varepsilon \\ \mathtt{s}_b(: \mathrm{t}(m' - \delta(m', n, x :), n, x :)) & \text{when } m = m' + 1 \text{ and } I_j = \mathtt{write}_0\ b \\ \mathrm{t}(m' - \delta(m', n, x :), n, x :) & \text{when } m = m' + 1 \text{ and } I_j \neq \mathtt{write}_0\ b \end{cases} \\
&= \begin{cases} \mathtt{1} & \text{when } m = \varepsilon \\ \text{let } j = m + 1 \bmod n \text{ in} & \text{when } m = m' + 1 \\ \quad \mathrm{IF}_{\mathtt{s}_b}(I_j = \mathtt{write}_0\ b : \mathrm{t}(m - \delta(m, n, x :), n, x :))) & \end{cases} \\
&= \mathtt{rec}(\mathtt{1},\mathtt{0},\mathtt{0},\mathrm{IF}_{\mathtt{s}_b} \circ \langle I_{\mathrm{mod} \circ \langle \mathtt{s}_1 \circ \langle \pi^{2,0}_1 \rangle, \pi^{2,0}_2 \rangle} = \mathtt{write}_0\ b : \pi^{2,1}_3 \rangle, \delta)
\end{aligned}$$

where

$$
\begin{aligned}
\delta(m,n,x:) = {} & \text{let } N = \mathrm{mult}(S_1, \overbrace{\mathrm{mult}(n, \cdots \mathrm{mult}(n,n:):)}^{c+1 \text{ mult}}:) \text{ in} \\
& \text{let } t = (m+1) \div N \text{ in} \\
& \text{let } w = (m+1-t\cdot N) \div (S\cdot n^2) \text{ in} \\
& \text{let } i = (m+1-t\cdot N - w\cdot S\cdot n^2) \div (S\cdot n) \text{ in} \\
& \text{let } j = (m+1-t\cdot N - w\cdot S\cdot n^2 - i\cdot S\cdot n) \div S \text{ in} \\
& \text{let } s = (m+1-t\cdot N - w\cdot S\cdot n^2 - i\cdot S\cdot n - j\cdot S) \text{ in} \\
& \text{let } b_{\mathtt{I}} = \mathrm{bit}(i:x) \text{ in} \\
& \text{let } b_{\mathtt{W}} = \mathrm{bit}(j : \mathrm{unary2bin}(w:)) \text{ in} \\
& \quad m - (((((t-1)\cdot n + \mathrm{W}(w,j,s:))\cdot n + \mathrm{I}(i,s:))\cdot n + \\
& \quad \mathrm{J}(j,s:))\cdot S) + \mathrm{S}(b_{\mathtt{I}}, b_{\mathtt{W}}, s:) \\
= {} & (\mathrm{minus}\circ\langle \mathrm{pluso}\circ\langle \pi_6^{6,0} : \mathrm{mult}\circ\langle S_1, \mathrm{pluso}\circ\langle \pi_5^{6,0} : \mathrm{mult}\circ\langle \pi_2^{6,0}, \\
& \qquad \mathrm{pluso}\circ\langle \pi_4^{6,0} : \mathrm{mult}\circ\langle \pi_2^{6,0}, \pi_3^{6,0} : \rangle : \rangle : \rangle : \rangle : \rangle : \pi_1^{6,0} \rangle) \\
& \circ\langle \pi_2^{10,0}, \\
& \quad \pi_3^{10,0}, \\
& \quad \mathrm{W}\circ\langle \pi_6^{10,0}, \pi_8^{10,0}, \pi_1^{10,0}, \pi_5^{10,0} : \rangle, \\
& \quad \mathrm{I}\circ\langle \pi_7^{10,0}, \pi_1^{10,0} : \rangle, \\
& \quad \mathrm{J}\circ\langle \pi_8^{10,0}, \pi_1^{10,0} : \rangle, \\
& \quad \mathrm{S}\circ\langle \pi_{10}^{10,0}, \pi_9^{10,0}, \pi_1^{10,0} : \rangle\rangle \\
& \circ\langle \pi_1^{8,0}, \ldots, \pi_8^{8,0}, \mathrm{bit}\circ\langle \pi_7^{8,0} : \pi_4^{8,0}\rangle, \mathrm{bit}\circ\langle \pi_8^{8,0} : \mathrm{unary2bin}\circ\langle \pi_6^{8,0} : \rangle\rangle\rangle \\
& \circ\langle \mathrm{minus}\circ\langle \mathrm{mult}\circ\langle \pi_8^{8,0}, S_1\rangle : \pi_1^{8,0}\rangle, \pi_2^{8,0}, \ldots, \pi_8^{8,0} : \rangle \\
& \circ\langle \pi_1^{7,0}, \ldots, \pi_7^{7,0}, \mathrm{div}\circ\langle \pi_1^{7,0}, S_1\rangle : \rangle \\
& \circ\langle \mathrm{minus}\circ\langle \mathrm{mult}\circ\langle \pi_8^{8,0}, \pi_2^{8,0}\rangle : \pi_1^{8,0}\rangle, \pi_3^{8,0}, \ldots, \pi_8^{8,0} : \rangle \\
& \circ\langle \pi_1^{7,0}, \ldots, \pi_7^{7,0}, \mathrm{div}\circ\langle \pi_1^{7,0}, \pi_2^{7,0}\rangle : \rangle \\
& \circ\langle \mathrm{minus}\circ\langle \mathrm{mult}\circ\langle \pi_7^{7,0}, \pi_2^{7,0}\rangle : \pi_1^{7,0}\rangle, \mathrm{mult}\circ\langle S_1, \pi_4^{7,0} : \rangle, \pi_3^{7,0}, \ldots, \pi_7^{7,0} : \rangle \\
& \circ\langle \pi_1^{6,0}, \ldots, \pi_6^{6,0}, \mathrm{div}\circ\langle \pi_1^{6,0}, \pi_2^{6,0}\rangle : \rangle \\
& \circ\langle \mathrm{minus}\circ\langle \mathrm{mult}\circ\langle \pi_6^{6,0}, \pi_2^{6,0} : \rangle : \pi_1^{6,0}\rangle, \mathrm{mult}\circ\langle S_1, \mathrm{mult}\circ\langle \pi_4^{6,0}, \pi_4^{6,0} : \rangle\rangle, \\
& \qquad \pi_3^{6,0}, \ldots, \pi_6^{6,0} : \rangle \\
& \circ\langle \pi_1^{5,0}, \ldots, \pi_5^{5,0}, \mathrm{div}\circ\langle \pi_1^{5,0}, \pi_2^{5,0}\rangle : \rangle \\
& \circ\langle \mathtt{s}_1\circ\langle : \pi_1^{4,0}\rangle, \pi_1^{4,0}, \ldots, \pi_4^{4,0} : \rangle \\
& \circ\langle \mathrm{mult}\circ\langle S_1, \underbrace{\mathrm{mult}\circ\langle \pi_2^{3,0}, \mathrm{mult}\circ\langle \pi_2^{3,0}, \cdots \mathrm{mult}\circ\langle \pi_2^{3,0}, \pi_2^{3,0}\rangle : \rangle : \rangle}_{c+1 \text{ mult}} :\rangle, \\
& \qquad \pi_1^{3,0}, \pi_2^{3,0}, \pi_3^{3,0}\rangle
\end{aligned}
$$

That it correctly simulates the Turing Machine is seen by inspection of Fig.1. □

We conclude with the following corrolary:

Corollary 3.13. *For any* LOGSPACE *Turing Machine T, there is BC^-_ε function $f_T : \mathbb{N}^{1,0} \to \mathbb{N}$ computing the same function: for all x, if T on input x produces the output tape y we have $f_T(x :) = 1y$.*

The function f_T is constructible in time $O(S^2)$ and space $O(\log S)$ whereas the input can be transduced in time $O(|x|)$ and space $O(1)$. □

We stress that it is not an issue that we use logarithmic space to compile between the Turing machine and BC^-_ε as it is the logarithmic in the *size of the program*.

Proof. The function f_T is constructed immediately as

$$\mathrm{T}(S \cdot |x|^{c+2}, 0, 0, 0, 0, x :) = \mathrm{T}(\mathrm{mult}(S_1, \mathrm{mult}(x, \ldots \mathrm{mult}(x, x :) :) :), 0, 0, 0, 0, x :) \ .$$

We reuse the encoding of the tuple into a unary number used in the proof above.

Precise time and space bounds of course depend on how the BC^-_ε are represented; we consider only outputting their textual representation. An inspection of the construction in the proof shows that S, W, I, and J can be output with one scan over the program. For each instruction we need to output the unary representation of the instruction number. (In the case of S we might also need to output the unary representation of the label.) This can be done in time $O(S^2)$. The $O(\log S)$ space uses arises from keeping track of the instruction. □

4 LOGSPACE soundness

Having shown that BC^-_ε is LOGSPACE complete, we continue to show that it is also LOGSPACE sound. We reuse a construction by Mairson in the unpublished work [13]. The construction depends on two facts:

1. The output and all intermediate values are bound by (1); this is established in [3].[1] We therefore need only a counter of size $O(\log |\vec{x}; \vec{y}|)$ to iterate over the bits of the output.

2. One bit of the output can be produced storing only one bit of the safe arguments and a constant number of bits from the normal arguments. This is straight forward except for the case of safe affine recursion.

   As for safe recursion consider as an example the parity function P

$$\mathrm{P}(x :) = \begin{cases} 0 & \text{when } x = \varepsilon \\ 0 & \text{when } x = x'0 \text{ and } \mathrm{P}(x' :) = 0 \\ 1 & \text{when } x = x'0 \text{ and } \mathrm{P}(x' :) = 1 \\ 1 & \text{when } x = x'1 \text{ and } \mathrm{P}(x' :) = 0 \\ 0 & \text{when } x = x'1 \text{ and } \mathrm{P}(x' :) = 1 \end{cases}$$

   which corresponds to

$$\begin{aligned} g(:) &= 0 \\ h_0(x' : r) &= c\circ\langle : r, 1, 0\rangle \\ h_1(x' : r) &= c\circ\langle : r, 0, 1\rangle \ . \end{aligned}$$

[1] In [3, Lemma 4.1] this inequality is only stated as a bound on the output. It is however a corollary, due to the compositional flavor of the proof, that it also bounds the intermediate values.

A standard evaluation of $\mathrm{P}(\mathtt{10111}:)$ would result in the following unwinding

$$\mathrm{P}(\mathtt{10111}:) = h_1(\mathtt{1011} : h_1(\mathtt{101} : h_1(\mathtt{10} : h_0(\mathtt{1} : h_1(\varepsilon : 0))))) \ .$$

It is not possible to keep this in LOGSPACE as the size of the call stack grows to the length of the recursion parameter.

Instead, we use a "computational amnesia" where the recursive call forces the computation of the caller to be *replaced* with the computation of the callee, e.g., computing $\mathrm{P}(\mathtt{10111}:) = h_1(\mathtt{1011} : \mathrm{P}(\mathtt{1011}:))$ results in the call to $\mathrm{P}(\mathtt{1011}:)$ replacing the call to $\mathrm{P}(\mathtt{10111}:)$. In the example at hand, we thus go through

$$\mathrm{P}(\mathtt{10111}:), \mathrm{P}(\mathtt{1011}:), \mathrm{P}(\mathtt{101}:), \mathrm{P}(\mathtt{10}:), \mathrm{P}(\mathtt{1}:), \mathrm{P}(\varepsilon:) = 0 \ .$$

This is similar to optimizing tail calls except that we need to return to the original caller. Without a call stack we cannot do this directly. Instead we *remember* the result (and its recursion depth, 5) and *restart* the computation of $\mathrm{P}(\mathtt{10111}:)$. We now only need to go through the calls

$$\mathrm{P}(\mathtt{10111}:), \mathrm{P}(\mathtt{1011}:), \mathrm{P}(\mathtt{101}:), \mathrm{P}(\mathtt{10}:), \mathrm{P}(\mathtt{1}:) = h_1(\varepsilon : \mathrm{P}(\varepsilon:)) = h_1(\varepsilon : 0) = 1$$

as we can lookup the stored value of $\mathrm{P}(\varepsilon:)$. We remember that the value 1 was found at recursion depth 4. We restart the computation and go through

$$\mathrm{P}(\mathtt{10111}:), \mathrm{P}(\mathtt{1011}:), \mathrm{P}(\mathtt{101}:), \mathrm{P}(\mathtt{10}:) = h_0(\mathtt{1} : 1) = 1 \ .$$

Continuing like this, restarting and remembering, we can eventually work our way up and find $\mathrm{P}(\mathtt{10111}:) = 0$.

Our goal in this section is to formalize the idea outlined above and prove that it stays within LOGSPACE. The core is to show that we in LOGSPACE can find any given bit of the output by only storing a fixed number of bits of the inputs to each subexpression. We do this by translating each $\mathrm{BC}^-_\varepsilon$-expression into a SML-expression that finds any given bit of the output by querying individual bits of the input. The technical part of this section consists of establishing the correctness of the functions and that only logarithmic space is needed to evaluate them.

The translation of the base constructors is given in Fig. 2. A function with m normal arguments and n safe arguments becomes a SML-function of type `program-`m`-`n. The base constructors take the normal and safe arguments and the index of the bit requested. They either return the bit or marks that the bit is non-existing; this is caught in the type `bit`. The arguments are given as second-order functions that returns a bit of the input upon request. The translation of safe affine composition is given in Fig. 3 and is also straightforward.

Finally, the translation of safe affine generalized recursion is given in Fig. 4. The construction embodies the idea outlined above: we start out computing the requested bit at depth 0. If there is a request for a bit from the recursive call, the function `recursiveCall` takes care of updating our goal and restarting the computation by raising the exception `Restart`n (where n is a unique identifier of the recursion). When we succeed in computing the goal bit—either because we reached the base case or because the recursive call was cached—we update the cache kept in `result`. We keep starting the search from depth 0 until the restarting we have succeed in computing the bit at depth 0. The last remaining part, is to notice that we can find the length of the displacement function by simply querying for the bits 0, 1, 2, etc. until we reach the index where the bit is not-existing.

We capture the above discussion in the following definition of the ML function $\lceil f \rceil$ which computes a requested bit of the $\mathrm{BC}^-_\varepsilon$function f:

```
type bit = int option
type input = int -> bit
type program-m-n = input -> ··· -> input -> int -> bit
                   (m + n times)
fun zero (bit : int) = NONE
fun succ0 (y1 : input) (bit : int) =
    if bit = 0 then SOME 0 else y1 (bit - 1))
fun succ1 (y1 : input) (bit : int) =
    if bit = 0 then SOME 1 else y1 (bit - 1))
fun pred (y1 : input) (bit : int) = y1 (bit + 1)
fun cond (y1 : input) (y2 : input) (y3 : input) (bit : int) =
    let fun boolFromBit NONE = false
          | boolFromBit (SOME b) = (b = 1)
    in
       if boolFromBit (y1 0) then y2 bit else y3 bit
    end
(* Translation of π_j^{m,n}; first when j≤m and then when m > j *)
fun proj-m-n-j (x1 : input) ... (xm : input)
               (y1 : input) ... (yn : input) (bit : int) = xj bit
fun proj-m-n-j (x1 : input) ... (xm : input)
               (y1 : input) ... (yn : input) (bit : int) = y(j − m) bit
```

Figure 2: Translations of each of the base constructors into a SML function.

```
fun comp (f : program-M-N)
         (g1 : program-m-0) ··· (gM : program-m-0)
         (h1 : program-m-n_1) ··· (hN : program-m-n_N)
         (x1 : input) ··· (xm : input)
         (y1 : input) ··· (yn : input) (bit : int) =
    let fun X1 bit = g1 x1 ... xm bit
        ⋮
        fun XM bit = gM x1 ... xm bit
        fun Y1 bit = h1 x1 ... xm yi_{1,1} ... yi_{1,l_1} bit
        ⋮
        fun YN bit = hN x1 ... xm yi_{N,1} ... yi_{N,l_N} bit
    in
        f X1 ... XM Y1 ... YN bit
    end
```

Figure 3: Translation of safe affine composition into an SML function. The division of the variables y1, ..., yn between the functions h1, ..., hN computing safe arguments is given by the vectors of indices $\vec{i_1}, \ldots, \vec{i_N}$. They satisfy the relation $\{i_{1,1}, \ldots, i_{1,l_1}, i_{2,1}, \ldots, i_{N-1,l_{N-1}}, i_{N,1}, \ldots, i_{L,l_N}\} \subseteq \{1, \ldots, n\}$ and that there is at most one $i_{j,j'}$ for any given number in $\{1, \ldots, n\}$.

```
exception Restartn
val NORESULT = { depth = ~1, res = NONE, bit=~1 }

fun saferec (g : program-(m − 1)-n)
            (h0 : program-m-1) (d0 : program-m-0)
            (h1 : program-m-1) (d1 : program-m-0)
            (x1 : input) ... (xm : input)
            (y1 : input) ... (yn : input) (bit : int) =
    let val result = ref NORESULT
        val goal = ref ({ bit=bit, depth=0 })
        fun loop1 body = if body () then () else loop1 body
        fun loop2 body = if body () then () else loop2 body
        fun findLength (z : input) =
            let fun search i = if z i <> NONE then search (i + 1) else i
            in
               search 0
            end
        fun x' (bit : int) = x1 (1 + bit + #depth (!goal))
        fun recursiveCall (d : program-m-0) (bit : int) =
            let val delta = 1 + findLength (d x' x2 ... xm)
            in
                if #depth (!goal) + delta = #depth (!result)
                   andalso #bit (!result) = bit
                then #res (!result)
                else
                    goal := { bit=bit, depth = #depth (!goal) + delta };
                    raise Restartn
            end
    in
    ( loop1 (fn () =>   (* Loops until we have the bit at depth 0 *)
      ( goal := { bit=bit, depth=0 };
        loop2 (fn () => (* Loops while the computation is restarted *)
          let val res =
            case x1 (#depth (!goal)) of
              NONE => g x2 ... xm y1 ... yn (#bit (!goal))
            | SOME b =>
              let val (h, d) = if b=0 then (h0,d0) else (h1,d1)
              in
                  h x' x2 ... xm (recursiveCall d) (#bit (!goal))
              end
          in ( result := { depth = #depth (!goal),
                           res = res,
                           bit = #bit (!goal) };
               true )
          end handle Restartn => false
        0 = #depth (!result) ));
      #res (!result))
    end
```

Figure 4: Translations of safe affine recursion

Definition 4.1. Let f be a function of $\mathrm{BC}^-_\varepsilon$ with m normal and n safe arguments. The ML function $\lceil f \rceil$ of type `program-`m`-`n is defined inductively based on Figs. 2–4:

$$\begin{aligned}
\lceil \mathtt{0} \rceil &= \mathtt{zero} \\
\lceil \mathtt{s}_b \rceil &= \mathtt{succ}b \\
\lceil \mathtt{p} \rceil &= \mathtt{pred} \\
\lceil \mathtt{c} \rceil &= \mathtt{cond} \\
\lceil \pi_j^{m,n} \rceil &= \mathtt{proj}\text{-}m\text{-}n\text{-}j \\
\lceil f \circ \langle g_1, \ldots, g_M : h_1, \ldots, h_N \rangle \rceil &= \mathtt{comp}\ \lceil g_1 \rceil\ \cdots\ \lceil g_m \rceil\ \lceil h_1 \rceil\ \cdots\ \lceil h_n \rceil \\
\lceil \mathtt{rec}(g,h_0,\delta_0,h_1,\delta_1) \rceil &= \mathtt{saferec}\ \lceil g \rceil\ \lceil h_0 \rceil\ \lceil \delta_0 \rceil\ \lceil h_1 \rceil\ \lceil \delta_1 \rceil
\end{aligned}$$

□

In the following we will establish correctness through the following facts:

- The functions are tail recursive as defined, e.g., by Jones [8]. Consequently there is a fixed maximal depth of the stack for any program.
- Each stack entry is representable in LOGSPACE of the inputs.
- The function succeeds and returns the correct bit provided that the input functions are correct and succeed without raising an exception `Restart`n.

From the first two facts we get that there is a Turing Machine which implements the program. From the last fact we get the correctness of the algorithm.

Notation 4.2.

- We recall according to the SML standard "`fun f x1` $\cdots$ `x`m `= e`" is just syntactical sugar for "`val f = fn x1 =>` $\cdots$ `fn x`m `=> e`". The programs are built with definitions on this form; for instance the definition of `succ0` is really

```
val succ0 = fn
       y1 => fn bit => if bit = 0 then SOME 0 else y1 (bit - 1)
```

  This is in accordance with the rewriting of derived forms mentioned in [12, App. A].

- We adapt the convention of Barendregt [2] and use $\equiv$ for the syntactic equivalence of functions, i.e., $e \equiv e'$ when the expressions e and e' are equivalent up to α-equivalence.
- We assign an *abstraction label* a to each function abstraction; we write `fn`^{a} `x => e`. We will refer to the abstraction labels as pairs of the function name and the parameter, e.g., `succ0` above has two abstraction labels: $\langle \mathtt{succ0}, \mathtt{y1} \rangle$ and $\langle \mathtt{succ0}, \mathtt{bit} \rangle$.

  We recall that when evaluating the program every function applied—even when it is the result of evaluating an expression—is constructed by one of the abstraction occurrences `fn`^{a} `x => e` in the program.

□

We want to establish that the program is tail recursive. Notions of tail recursion for higher-order functional programs are discussed by Kfoury [10, 11]. His work presents different reasonable, but incompatible notions. Clinger [4] uses an operational approach to characterize tail recursion optimization and other space optimizations for (core) Scheme [9]; in his approach all calls are tail call and the language constructs explicitly saves the store when evaluating subexpressions. Jones [8] has a characterization for whole programs; he uses a static ordering of the functions in combination with a dynamic criteria. We follow the latter and first recall the notion of partial order:

Notation 4.3. A partial order $\succeq$ on a set A is a subset of $A \times A$ that is transitive and reflexive. If $x \succeq y$ and $y \preceq x$, we consider the two elements equivalent (written $x \approx y$). We use $x \succ y$ as an abbreviation for $x \succeq$ and $x \not\approx y$. If $x \succ y$ we call x a successor of y and y a predecessor of x. □

We then define tail recursion. We will adapt a notion that is slightly more liberal than the usual notion: we allow the call stack to grow as long as there is never two stack entries for the same abstraction.

Definition 4.4.

1. An occurrence of expression e in e_0 is *in tail position in* e_0
   (a) $e_0 \equiv e$,
   (b) $e_0 \equiv$ `if` e_1 `then` e_2 `else` e_3 and e is in tail position of e_2 or e_3,
   (c) $e_0 \equiv$ `case` e_0' `of` p_1 `=>` e_1 $\cdots$ p_l `=>` e_l and e is in tail position of e_i for some $i = 1, \ldots, l$.
   (d) $e_0 = e_1;\ldots;e_l$ and e is in tail position of e_l.
   (e) $e_0 =$ `let val` v_1`=`e_1 $\cdots$ `val` v_l`=`e_l `in` e_{l+1} `end` and e is in tail position of e_{l+1}.
2. An expression e is *tail recursive* if there is a partial order $\succeq$ on e's abstraction labels such that for every abstraction `fn`^{a_0} `x => e`$_0$ in e and every abstraction label a_1 of an abstraction that can result from evaluating the function part `e`$_1$ of any application `e`$_1$ `e`$_2$ occurring in `e`$_0$, we have either (a) $a_0 \succ a_1$ or (b) $a_0 = a_1$ and the application is in tail position.

□

As the definition involves the possible results of evaluating an expression, it is obviously undecidable in general. It can however be safe approximated using abstract interpretation.

The partial order in the above definition ensures that the program can be evaluated without having more than one entry per abstraction on the call stack. A formal proof would have to rely on the formal definition of ML [12];[2] this is beyond the scope of this report. An alternative approach would be to translate the ML program into the functional language in [8] which would be straightforward. We will however just rely on an informal understanding of SML evaluation. We will furthermore make some idealized assumptions about SML:

1. Arithmetic can be done with arbitrary precision. In particular none of the functions, `+`, `-`, `*`, `=`, and `<>`, do not raise any exceptions.

[2] ML is formally defined through an operational semantics. There is therefore not a call stack; instead the depth of the recursion is reflected through the number of times we have extended the environment evaluating a function call (this is inference (113) in [12]). Formally, we would therefore establish that the number of extensions depends only on the program, not the input.

2. The memory is arbitrarily large. In particular, the function `ref` will never raise an exception.

3. The basic functions are implemented tail recursively, i.e, `+`, `-`, `*`, `=`, `<>`,`#`,`!`, and `ref` are all tail recursive.

Proposition 4.5. *Let $F \in \mathbb{N}^{m,n}$ be a function of BC^{-}_{ε}. Its encoding into an SML expression is $\lceil F \rceil \equiv$ `fn`^{a_1} `inp1 => ` $\cdots$ `=> fn`^{a_n} `inp`n `=> fn`$^{a_{n=1}}$ `bit => e` of type `program-`m`-`n. We consider an application* `P` $= \lceil F \rceil$ `exp1` $\cdots$ `exp`l *where $l = m + n$. Each of the argument expressions* `exp`i *can evaluate to abstractions with abstraction labels in the set A_i. Furthermore, none of the abstraction labels in $A_1 \cup \cdots \cup A_l$ occurs in* `P`*. There exists a partial order $\succeq$ on the abstraction labels with the following properties:*

1. *Consider any abstraction* `fn`$^{a'}$ `x =>` e' *in* `P` *and any application e'_1 e'_2 occurring in e'. For any abstraction* `fn`$^{a'_1}$ x `=>` e''_1 *that e'_1 can evaluate to, we have either (a) $a' \succ a'_1$ or (b) $a' = a'_1$ and the application is in tail position.*

2. *There is an abstraction label $\top_{\succeq}$ that is the successor of all other abstraction labels that occur in $\lceil F \rceil$ and $A_1 \cup \cdots \cup A_l$.*

3. *It holds that $a_1 \succ \cdots \succ a_{l+1}$.*

4. *Non of the elements $a'_i \in A_i$ for $1 \leq i \leq l$ has a predecessor in $\succeq$.*

□

Proof. We prove the proposition using induction on the structure of F.

1. $F =$ `0`: We have `P` $\equiv \lceil$`0`$\rceil$. We choose $\succeq$ as the partial order containing the single element $\langle$`zero`$,$`bit`$\rangle \succeq \langle$`zero`$,$`bit`$\rangle$ and $\top_{\succ} = \langle$`zero`$,$`bit`$\rangle$. As there is no application in `zero`, Point 1 is vacuously true, Point 2 and Point 3 are vacuously true as there is only one abstraction label, and Point 4 is trivially true as $l = 0$.

2. $F =$ `s`$_b$: We have `P` $\equiv \lceil$`s`$_b\rceil$ `exp1`. We choose $\top_{\succeq} = \langle$`succ`$b,$`y1`$\rangle$ and $\succeq$ to be the smallest partial order containing $\langle$`succ`$b,$`y1`$\rangle \succ \langle$`succ`$b,$`bit`$\rangle$ and $\langle$`succ`$b,$`bit`$\rangle \succ a'$ for $a' \in A_1$. As $\lceil$`s`$_b\rceil$ and `exp1` do not share abstraction labels, the previous relations are strict in $\succeq$. This also makes Point 4 true. Point 1 holds as the only relevant application is `y1 (bit - 1)` and $\langle$`succ`$b,$`y1`$\rangle \succ a'$ for all a' that `y1` can evaluate to. Point 2 holds as $\langle$`succ`$b,$`y1`$\rangle \succ \langle$`succ`$b,$`bit`$\rangle$ which are the only abstraction labels in $\lceil F \rceil$. Point 3 is obviously true.

3. $F =$ `p`: We have `P` $\equiv \lceil$`p`$\rceil$ `exp1`. We choose $\top_{\succeq} = \langle$`pred`$,$`y1`$\rangle$ We choose $\succeq$ to be the smallest partial order containing $\langle$`pred`$,$`y1`$\rangle \succ \langle$`pred`$,$`bit`$\rangle$ and $\langle$`succ`$b,$`bit`$\rangle \succ a'$ for $a' \in A_1$. As $\lceil$`p`$\rceil$ and `exp1` do not share abstraction labels, the previous relations are strict in $\succeq$. This also makes Point 4 true. Point 1 holds as the only relevant application is `y1 (bit + 1)` and $\langle$`pred`$,$`y1`$\rangle \succ a'$ for all a' that `y1` can evaluate to. Point 2 holds as $\langle$`pred`$,$`y1`$\rangle \succ \langle$`pred`$,$`bit`$\rangle$ which are the only abstraction labels in $\lceil F \rceil$. Point 3 is obviously true.

4. $F =$ `c`: We have `P` $\equiv \lceil$`c`$\rceil$ `exp1 exp2 exp3`. We choose $\top_{\succeq} = \langle$`cond`$,$`y1`$\rangle$ and $\succeq$ to be the smallest partial order such that

$$\langle \texttt{cond}, \texttt{y1} \rangle \succ \langle \texttt{cond}, \texttt{y2} \rangle \succ \langle \texttt{cond}, \texttt{y3} \rangle \succ \langle \texttt{cond}, \texttt{bit} \rangle \succ \langle \texttt{boolFromBit}, \texttt{bit} \rangle$$

and $\langle \texttt{cond}, \texttt{bit} \rangle \succ a'$ for any $a' \in A_1 \cup A_2 \cup A_3$. As there is no overlap between the abstraction labels in $\lceil \texttt{c} \rceil$ and $A_1 \cup A_2 \cup A_3$, the preceding relations are strict. Furthermore, it makes Point 4 valid. As for Point 1 the relevant applications are `y1 0`, `boolFromBit (y1 0)`, `y2 bit`, and `y3 bit` which all occur inside the abstraction $\langle \texttt{cond}, \texttt{bit} \rangle$ and thus true as $\langle \texttt{cond}, \texttt{bit} \rangle \succ \langle \texttt{boolFromBit}, \texttt{bit} \rangle$ and $\langle \texttt{cond}, \texttt{bit} \rangle \succ a'$ for any a' that `y1`, `y2`, and `y3` can evaluate to. Point 2 and Point 3 are obviously true.

5. $F = \pi_j^{m,n}$: We have $\texttt{P} \equiv \lceil \pi_j^{m,n} \rceil$ $\texttt{exp1} \ \cdots \ \texttt{exp}l$. We choose $\top_{\succeq} = \langle \texttt{prj-}m\texttt{-}n\texttt{-}j, \texttt{x1} \rangle$ and $\succeq$ to be the smallest partial order containing

$$\langle \texttt{prj-}m\texttt{-}n\texttt{-}j, \texttt{x1} \rangle \succ \cdots \succ \langle \texttt{prj-}m\texttt{-}n\texttt{-}j, \texttt{x}m \rangle \succ \langle \texttt{prj-}m\texttt{-}n\texttt{-}j, \texttt{y1} \rangle \tag{4}$$

$$\langle \texttt{prj-}m\texttt{-}n\texttt{-}j, \texttt{y1} \rangle \succ \cdots \succ \langle \texttt{prj-}m\texttt{-}n\texttt{-}j, \texttt{y}n \rangle \succ \langle \texttt{prj-}m\texttt{-}n\texttt{-}j, \texttt{bit} \rangle \tag{5}$$

and $\langle \texttt{prj-}m\texttt{-}n\texttt{-}j, \texttt{bit} \rangle \succ a'$ for any $a' \in A_1 \cup \cdots \cup A_l$.

As $\lceil \pi_j^{m,n} \rceil$ and $\texttt{exp1}, \ldots, \texttt{exp}n$ do not share abstraction labels, the previous relations are strict in $\succeq$. This also makes Point 4 true. Point 1 holds as the only relevant application is $\texttt{x}j$ `bit` or $\texttt{y}(j-m)$ and $\langle \texttt{prj-}m\texttt{-}n\texttt{-}j, \texttt{bit} \rangle \succ a'$ for any abstraction labels that any of the arguments $\texttt{exp1}, \ldots, \texttt{exp}n$ can evaluate to. Point 2 and Point 3 are obviously true.

6. $F = f \circ \langle g_1, \ldots, g_M : h_1, \ldots, h_N \rangle$: We have

$$\lceil F \rceil = \texttt{comp} \ \lceil f \rceil \ \lceil \texttt{g}_1 \rceil \ \cdots \ \lceil \texttt{g}_M \rceil \ \lceil \texttt{h}_1 \rceil \ \cdots \ \lceil \texttt{h}_N \rceil$$

and $\texttt{P} \equiv \lceil F \rceil$ $\texttt{exp1} \ \cdots \ \texttt{exp}l$. We apply the induction hypothesis on $\texttt{f} = \lceil f \rceil$ with the sets

$$\begin{aligned} A_{\texttt{f},1} &= \{\langle \texttt{X1}, \texttt{bit} \rangle\} & \cdots \quad & A_{\texttt{f},M} = \{\langle \texttt{X}M, \texttt{bit} \rangle\} \\ A_{\texttt{f},M+1} &= \{\langle \texttt{Y1}, \texttt{bit} \rangle\} & \cdots \quad & A_{\texttt{f},M+N} = \{\langle \texttt{Y}N, \texttt{bit} \rangle\} \end{aligned}$$

and obtain $\succeq_f$. We obtain $\succeq_{g_1}, \ldots, \succeq_{g_M}, \succeq_{h_1}, \ldots, \succeq_{h_N}$ corresponding to the programs $\texttt{f} = \lceil f \rceil, \texttt{g1} = \lceil g_1 \rceil, \ldots, \texttt{g}M = \lceil g_M \rceil, \texttt{h1} = \lceil h_1 \rceil, \ldots, \texttt{h}N = \lceil h_N \rceil$, resp., by applying the induction hypothesis on $g_1, \ldots, g_M$ with $A_1, \ldots, A_M$ and on h_j for $1 \leq j \leq N$ with $A_1, \ldots, A_M, A_{M+i_{j,1}}, \ldots, A_{M+i_{j,l_j}}$. These partial orders satisfy the properties in Point 1–4 above.

We choose $\top_{\succeq} = \langle \texttt{comp}, \texttt{f} \rangle$ and obtain the desired order as the smallest partial order containing the union

$$\succ_f \cup \succ_{g_1} \cup \ldots \cup \succ_{g_M} \cup \succ_{h_1} \cup \ldots \cup \succ_{h_N} \tag{6}$$

and the following pairs:

$$\begin{aligned} &\langle \texttt{comp}, \texttt{f} \rangle \succ \langle \texttt{comp}, \texttt{g1} \rangle \succ \cdots \succ \langle \texttt{comp}, \texttt{g}M \rangle \succ \langle \texttt{comp}, \texttt{h1} \rangle \succ \cdots \succ \langle \texttt{comp}, \texttt{h}N \rangle \succ \\ &\langle \texttt{comp}, \texttt{x1} \rangle \succ \cdots \succ \langle \texttt{comp}, \texttt{x}m \rangle \succ \langle \texttt{comp}, \texttt{y1} \rangle \succ \cdots \succ \langle \texttt{comp}, \texttt{y}n \rangle \succ \langle \texttt{comp}, \texttt{bit} \rangle \end{aligned} \tag{7}$$

$$\begin{aligned} &\langle \texttt{comp}, \texttt{bit} \rangle \succ \top_{\texttt{f}}, \langle \texttt{X1}, \texttt{bit} \rangle \succ \top_{\texttt{g1}}, \cdots, \langle \texttt{X}M, \texttt{bit} \rangle \succ \top_{\texttt{g}M} \\ &\qquad \langle \texttt{Y1}, \texttt{bit} \rangle \succ \top_{\texttt{h1}}, \cdots, \langle \texttt{Y}N, \texttt{bit} \rangle \succ \top_{\texttt{h}N} \ . \end{aligned} \tag{8}$$

As the abstraction labels in $A_1, \ldots, A_{M+N}$ are independent from the abstraction labels in $\lceil F \rceil$, the relations in (7) and (8) holds in $\succeq$.

We start with Point 1 and will first consider all the applications in the body: `f X1`, `F X1 X2`, $\ldots$, `F X1` $\cdots$ `X`M `Y1` $\cdots$ `Y`N. They all appear under the abstraction labels $\langle \texttt{comp}, \texttt{f}\rangle, \ldots, \langle \texttt{comp}, \texttt{bit}\rangle$. From the induction hypothesis, we know that $\top_f$ is the successor of any abstraction labels in $\top_{\succeq_f}$ and $A_{\texttt{f},1} \cup \cdots \cup A_{\texttt{f},M+N}$. Consequently, any abstraction we get from these applications will be majorized by $\top_{\succeq_f}$. It follows from $\langle \texttt{comp}, \texttt{bit}\rangle \succ \top_{\succeq_f}$ that all the applications satisfy option a) in Point 1.

We then consider the applications `g1 x1`, `g1 x1 x2`, $\ldots$, `g1 x1` $\cdots$ `x`m `bit` that appear under the abstractions $\langle \texttt{comp}, \texttt{f}\rangle, \ldots, \langle \texttt{comp}, \texttt{y}n\rangle, \langle \texttt{comp}, \texttt{bit}\rangle, \langle \texttt{X1}, \texttt{bit}\rangle$. By induction hypothesis, we have $\top_{\succeq_{g_1}}$ as an successor of all the abstractions that emerge from these applications. As $\langle \texttt{X1}, \texttt{bit}\rangle \succ \top_{\succeq_{g_1}}$ and, by the induction hypothesis on f, $\langle \texttt{comp}, \texttt{bit}\rangle \succ \top_{\succeq_f} \succ \langle \texttt{X1}, \texttt{bit}\rangle$ we have option a) of Point 1. The applications inside `X2`, $\ldots$, `X`M, `Y1`, $\ldots$, `Y`N are handled in a similar way.

By (7) we have $\top_\succeq$ as a successor of all the abstractions in the definition of `comp`. Using the induction hypothesis on f and $\langle \texttt{comp}, \texttt{bit}\rangle \succ \top_{\succeq_f}$ we have

$$\top_\succeq \succ \langle \texttt{X1}, \texttt{bit}\rangle, \ldots, \langle \texttt{X}M, \texttt{bit}\rangle, \langle \texttt{Y1}, \texttt{bit}\rangle, \ldots, \langle \texttt{Y}N, \texttt{bit}\rangle \ .$$

According to the relations in (8) we have $\langle \texttt{X1}, \texttt{bit}\rangle \succ \top_{\succeq_{g_1}}; \cdots; \langle \texttt{X}M, \texttt{bit}\rangle \succ \top_{\succeq_{g_M}}$; $\langle \texttt{Y1}, \texttt{bit}\rangle \succ \top_{\succeq_{h_1}}; \cdots; \langle \texttt{Y}N, \texttt{bit}\rangle \succ \top_{\succeq_{h_N}}$ and therefore $\top_\succeq \succ a$ for any $a \in A_1 \cup \cdots \cup A_{M+N}$. This establishes Point 2.

Point 3 holds by inspection of (7).

Using the induction hypothesis on $g_1, \ldots, g_M, y_1, \ldots, y_N$ we know that none of the elements in $A_1 \cup \cdots \cup A_{M+N}$ have a predecessor in $\succeq_{g_1} \cup \cdots \cup \succeq_{g_M} \cup \succeq_{h_1} \cup \cdots \cup \succeq_{h_N}$. Furthermore, they are not mentioned on any left-hand side in of (7) and (8). As there is no overlap between the labels in $A_1 \cup \cdots \cup A_{M+N}$ and `comp`, we find that none of the elements in $A_1 \cup \cdots \cup A_{M+N}$ have a predecessor in $\succeq$.

7. $F = \texttt{rec}(g,h_0,d_0,h_1,d_1)$: We have $\lceil F \rceil = \texttt{saferec}\ \lceil \texttt{g} \rceil\ \lceil \texttt{h0} \rceil\ \lceil \texttt{h1} \rceil\ \lceil \texttt{d0} \rceil\ \lceil \texttt{d1} \rceil$ and $\texttt{P} \equiv \lceil F \rceil\ \texttt{exp1} \cdots \texttt{exp}l$. We will use `body1` as the abstraction label for the anonymous function in `loop1` and `body2` for the anonymous function in `loop2`.

We let $A' = \{\langle \texttt{x'}, \texttt{bit}\rangle\}$ and $A_{\text{rec}} = \{\langle \texttt{recursiveCall}, \texttt{bit}\rangle\}$ and apply the induction hypothesis to h_0 and h_1 with $A', A_2, \ldots, A_M, A_{\text{rec}}$ to obtain the partial orders $\succ_{h_0}$ and $\succ_{h_1}$. We apply the induction hypothesis to `g` with $A_2, \ldots, A_{M+N}$ and on `d0` and `d1` with with $A_1, \ldots, A_M$ to obtain $\succ_g$, $\succ_{d_0}$, and $\succ_{d_1}$.

We choose $\top_\succeq = \langle \texttt{saferec}, \texttt{g}\rangle$ and obtain $\succeq$ as the smallest partial order containing

$$\succ_g \cup \succ_{h_0} \cup \succ_{h_1} \cup \succ_{d_0} \cup \succ_{d_1} \tag{9}$$

and the following pairs

$$\begin{gathered}\langle \texttt{saferec}, \texttt{x1}\rangle \succ \cdots \succ \langle \texttt{saferec}, \texttt{x}m\rangle \succ \langle \texttt{saferec}, \texttt{y1}\rangle \\ \langle \texttt{saferec}, \texttt{y1}\rangle \succ \cdots \succ \langle \texttt{saferec}, \texttt{y}n\rangle \succ \langle \texttt{saferec}, \texttt{bit}\rangle\end{gathered} \tag{10}$$

$$\begin{gathered}\langle \texttt{saferec}, \texttt{bit}\rangle \succ \langle \texttt{loop1}, \texttt{body}\rangle \succ \texttt{body1} \succ \langle \texttt{loop2}, \texttt{body}\rangle \succ \texttt{body2} \\ \texttt{body2} \succ a_1 \quad \text{where } a_1 \in A_1 \\ \texttt{body2} \succ \top_{\texttt{g}} \\ \texttt{body2} \succ \langle \texttt{recursiveCall}, \texttt{d}\rangle, \top_{\texttt{h0}}, \top_{\texttt{h1}} \\ \langle \texttt{x'}, \texttt{bit}\rangle \succ a_1 \quad \text{where } a_1 \in A_1 \\ \langle \texttt{recursiveCall}, \texttt{bit}\rangle \succ \langle \texttt{findLength}, \texttt{x}\rangle \\ \langle \texttt{findLength}, \texttt{x}\rangle \succ \langle \texttt{search}, \texttt{i}\rangle \succ \top_{\texttt{d0}}, \top_{\texttt{d1}} \ .\end{gathered} \tag{11}$$

As $A_1 \cup \cdots \cup A_{M+N}$ does not have any abstraction labels in common with `saferec` all the relations in (10) and (11) are strict.

To prove Point 1 we consider the various applications:

(a) `loop1 body` in `loop1`: We meet option b) as $\langle \texttt{loop1}, \texttt{body} \rangle \succeq \langle \texttt{loop1}, \texttt{body} \rangle$ by the reflexivity of $\succeq$ and the call is in tail position. As `loop1` occurs inside `saferec` we should also establish Point 1 wrt. the abstractions of `saferec`: this follows from (10) and $\langle \texttt{saferec}, \texttt{bit} \rangle \succ \langle \texttt{loop1}, \texttt{body} \rangle$.

The argument for `loop2 body` in `loop2` is similar.

(b) `body ()` in `loop1`: The variable `body` can only evaluate to the abstraction label `body1`. We have $\langle \texttt{loop1}, \texttt{body} \rangle \succ \texttt{body1}$ by definition. Using (10) and the fact $\langle \texttt{saferec}, \texttt{bit} \rangle \succ \langle \texttt{loop1}, \texttt{body} \rangle$, this also establishes Point 1 wrtthe abstractions of `saferec`.

The argument for `body ()` in `loop1` is similar.

(c) `search 0` inside `findLength`: We have $\langle \texttt{findLength}, \texttt{x} \rangle \succ \langle \texttt{search}, \texttt{i} \rangle$ by definition. Furthermore, from the induction hypothesis on h_0 and h_1 and (11), we have $\langle \texttt{saferec}, \texttt{bit} \rangle \succ \langle \texttt{findLength}, \texttt{x} \rangle$ so we are covered wrt. the abstractions of `saferec`.

(d) `z i` inside `search`: `z` is the result of evaluating `d x' x2` $\cdots$ `x`m where `d` $=$ `d0` or `d` $=$ `d1`. By definition $\langle \texttt{search}, \texttt{i} \rangle \succ \top_{\succ_{d_0}}, \top_{\succ_{d_1}}$ which by induction hypothesis is a successor of any closure that `d0` or `d1` can evaluate to. Furthermore, as $\langle \texttt{saferec}, \texttt{bit} \rangle \succ \langle \texttt{findLength}, \texttt{x} \rangle$ we are covered wrt. the abstractions of `saferec`.

(e) `search (i + 1)` inside `search`: the call is in tail position and $\langle \texttt{search}, \texttt{i} \rangle \succeq \langle \texttt{search}, \texttt{i} \rangle$ by the reflexivity in partial orders. We are furthermore covered wrt. the abstractions of `saferec` as seen in the previous point.

(f) `x1 (1 + bit + #depth (!goal))` in `x'`: we have $\langle \texttt{x1}, \texttt{bit} \rangle \succ a_1$ for all abstraction labels a_1 that `x1` can evaluate to. From the induction hypothesis on h_0 and h_1 and (11), we find that $\langle \texttt{saferec}, \texttt{bit} \rangle \succ a_1$.

(g) `findLength (`$\cdots$`)` inside `recursiveCall`: Using the relations in (11) we have $\langle \texttt{findLength}, \texttt{x} \rangle \prec \langle \texttt{recursiveCall}, \texttt{bit} \rangle$. From the induction hypothesis on h_0 and h_1 and (11) we find that $\langle \texttt{saferec}, \texttt{bit} \rangle \succ \langle \texttt{recursiveCall}, \texttt{bit} \rangle$; the application is thus also covered wrt. the abstractions in `saferec`.

(h) `d x'`, `d x' x2`, $\ldots$, `d x' x2` $\cdots$ `x`m inside `recursiveCall`: From the relations in (11) we have $\langle \texttt{recursiveCall}, \texttt{bit} \rangle \succ \top_{\succ_{d_0}}, \top_{\succ_{d_1}}$; it follows from the induction hypothesis that any abstraction label a which can emerge from evaluating `d` is a predecessor of $\langle \texttt{recursiveCall}, \texttt{bit} \rangle$. By the same argument as in the previous point, we conclude that the applications are covered wrt. the abstraction in `saferec`.

(i) `loop1 (fn () =>` $\cdots$`)`: we have $\langle \texttt{saferec}, \texttt{bit} \rangle \succ \texttt{loop1}$ which covers the fact that it appears under the abstractions of `saferec`.

(j) `loop2 (fn () =>` $\cdots$`)`: we have $\langle \texttt{saferec}, \texttt{bit} \rangle \succ \texttt{loop2}$.

(k) `x1 (#depth (!goal))`: we have $\texttt{body2} \succ a_1$ for all abstraction labels a_1 which `x1` can evaluate to.

(l) `g x2`, `g x2 x3`, $\ldots$, `g x2` $\cdots$ `y`n `(#bit` $\cdots$`)`. By the induction hypothesis on g, we have $\top_{\succeq_g}$ as the successor of any abstraction label in `g`. As both $\texttt{body2}, \succ \top_{\texttt{g}}$ and $\langle \texttt{saferec}, \texttt{bit} \rangle \succ \top_{\texttt{g}}$ it follows that option a) of Point 1.

(m) `h x'`, `h x' x2`, ..., `h x' x2` $\cdots$ `x`m (`recursiveCall d`) (`#bit` ($\cdots$)). We use the induction hypothesis on h_0 and h_1. We thus know that $\top_{\succeq_{h_0}}$ and $\top_{\succeq_{h_1}}$ are successors of any abstraction label that the applications can evaluate to. It therefore follows from $\langle \texttt{saferec}, \texttt{bit}\rangle \succ \texttt{body2} \succ \top_{\succeq_{h_0}}, \top_{\succeq_{h_1}}$.

As $\top_{\succeq} \succ \top_{\succeq_{h_0}}, \top_{\succeq_{h_0}}$ by (10) and the induction hypothesis on h_0 and h_1, we have $\top_{\succeq} \succ \langle \texttt{recursiveCall}, \texttt{bit}\rangle$. It therefore follows from (10) that $\top_{\succeq}$ is a successor of all the abstractions in the definition of `saferec`.

From the induction hypothesis on g, (10) and (11) we find that $\top_{\succeq} \succ \texttt{body2} \succ \top_{\succeq_g} \succeq a$ where $a \in A_2 \cup \cdots \cup A_{M+N}$. It follows directly from (10) and (11) that $\top_{\succeq} \succ \texttt{body2} \succ a_1$ where $a_1 \in A_1$. We have therefore established Point 2.

Point 3 holds by inspection of (10).

Using the induction hypothesis on g,h_0, h_1, d_0, and d_1 we know that none of the elements in $A_2 \cup \cdots \cup A_{M+N}$ have a predecessor in $\succeq_g \cup \succeq_{h_0} \cup \succeq_{h_1} \cup \succeq_{d_0} \cup \succeq_{d_1}$. Furthermore, they are not mentioned on any left-hand side in of (10) and (11). Similarly, the abstraction labels in A_1 are only the right hand sides of $\succeq$ in (10) and (11). As there is no overlap between the labels in $A_1 \cup \cdots \cup A_{M+N}$ and `comp`, we find that none of the elements in $A_1 \cup \cdots \cup A_{M+N}$ have a predecessor in $\succeq$.

□

Proposition 4.6. *Given any BC^-_ε-function $F \in \mathbb{N}^{M,N}$, its SML-encoding $\lceil F \rceil$ can be evaluated in logarithmic space, i.e., for any $x_1, \ldots, x_m, y_1, \ldots, y_N \in \mathbb{N}$ we can compute $F(x_1, \ldots, x_M : y_1, \ldots, y_N)$ using only logarithmic workspace.* □

Proof. We use the driver loop given in Fig. 5 to evaluate the function using the expression `eval` $\lceil F \rceil$. We choose $A_1, \ldots, A_{M+N} = \{\langle \texttt{bit\_nth}, \texttt{bit}\rangle\}$ and use Prop. 4.5 to get a partial order $\succeq_F$ covering all the abstraction labels in $\lceil F \rceil$. We then create a partial order $\succeq$ as the smallest partial order containing $\succeq_F$ and the following pairs

$$\langle \texttt{eval}, \texttt{F}\rangle \succ \langle \texttt{eval}, \texttt{x1}\rangle \succ \cdots \succ \langle \texttt{eval}, \texttt{x}M\rangle \succ \langle \texttt{eval}, \texttt{y1}\rangle \succ \cdots \succ \langle \texttt{eval}, \texttt{y}N\rangle \tag{12}$$

$$\langle \texttt{eval}, \texttt{y}N\rangle \succ \langle \texttt{bit\_nth}, \texttt{x}\rangle, \top_{\succeq_F} \ . \tag{13}$$

We can now verify that the expression is tail recursive:

1. The applications `bit_nth x1`, ..., `bit_nth x`M, `bit_nth y1`, ..., `bit_nth y`N all meet option (a) in Def. 4.4 because $\langle \texttt{eval}, \texttt{y}N\rangle \succ \langle \texttt{bit\_nth}, \texttt{x}\rangle$.
2. The applications `F X1`, `F X1 X2`, ..., `F X1` $\cdots$ `X`M `Y1` $\cdots$ `Y`N `bit` meet option (a) because $\langle \texttt{eval}, \texttt{y}N\rangle \succ \top_{\succeq_F}$.
3. The two applications `iter_eval_bit (bit + 1)` are in tail position and by the reflexivity of partial orders, $\langle \texttt{iter\_eval\_bit}, \texttt{bit}\rangle = \langle \texttt{iter\_eval\_bit}, \texttt{bit}\rangle$.

As the expression is tail recursive, it can be evaluated with an input-independent call stack (the call stack will be bounded by the size of the function expression F). All the first-order function arguments are either of constant size or an index into an input. An index into an input can be represented in logarithmic space. Consequently, any closure generated to pass a function as argument is logarithmic in the size of the input. It follows that the activation records, and therefore the full call stack, can be represented in logarithmic space. We conclude that the expression can be evaluated in space $O(\log \max\{x_1, \ldots, x_m, y_1, \ldots, y_N\})$.
□

```
fun bit_nth n bit =
    let val exponent = exponentiation 2 bit
    in
        if exponent > n then NONE
        else SOME ( n div exponent mod 2 )
    end
fun eval F
         (x1 : int) ··· (xm : int)
         (y1 : int) ··· (yn : int)  =
    (* Iteratively evaluate the bit until there are no more bits. *)
    let val X1 = bit_nth x1
        ⋮
        val XM = bit_nth xM
        val Y1 = bit_nth y1
        ⋮
        val YN = bit_nth yN
        fun iter_eval_bit bit =
            case F X1 ··· XM Y1 ··· YN bit of
                NONE => ()
              | SOME 0 => ( print "0" ; iter_eval_bit (bit + 1))
              | SOME 1 => ( print "1" ; iter_eval_bit (bit + 1))
    in
        iter_eval_bit 0
    end
end
```

Figure 5: Driver loop for evaluating a $\mathrm{BC}^{-}_{\varepsilon}$-function. The function exponentiation is any tail recursive implementation of the exponentiation function on integers. Note that the output will be printed in reverse.

The only tricky case will turn out to be the recursion. For that end we start with a technical lemma that simply establishes the notation for the evaluation of safe affine recursion.

Lemma 4.7. *Let a BC_ε^--function $F = \mathtt{rec}(g,h_0,h_1,d_0,d_1)$ be given. For any normal and safe inputs $x_1 = b_k \cdots b_1, x_2, \ldots, x_M, y_1, \ldots, y_N$ there exists $r \geq 0$ and $D_0, \ldots, D_r \geq 1$ such that*

1. $D_r = 0$
2. $D_{i-1} = D_i + d_{b_{D_i}}(z_i, x_2, \ldots, x_M :) + 1$ *when* $0 \leq i \leq r-1$.
3. $D_0 \geq m$
4. $D_i < m$ *when* $i \geq 1$
5. $F(x_1, \ldots, x_M : y_1, \ldots, y_N) =$
   $h_{b_{D_r+1}}(z_r, x_2, \ldots, x_M :$
   $h_{b_{D_{r-1}+1}}(z_{r-1}, x_2, \ldots, x_M : \cdots$
   $h_{b_{D_1+1}}(z_1, x_2, \ldots, x_M : g(x_2, \ldots, x_M : y_1, \ldots, y_N)) \cdots))$

where $z_r = b_k \cdots b_{D_i+2}$. □

Proof. We use induction over x_1:

- $x_1 = \varepsilon$: We choose $r = 0$ and $D_0 = 0$ which satisfy Points 1, 3, and 5. The Points 2 and 4 are vacousely true.
- $x_1 = b_k \cdots b_1$ for some $k \geq 1$. From the definition of safe affine recursion we have:

$$\begin{aligned} &F(x_1, \ldots, x_M : y_1, \ldots, y_N) = \\ &\quad h_{b_1}(b_k \cdots b_2, x_2, \ldots, x_M : F(b_k \cdots b_{2+\delta}, x_2, \ldots, x_M : y_1, \ldots, y_N)) \end{aligned} \tag{$*$}$$

  where $\delta = d_{b_1}(b_k \cdots b_2, x_2, \ldots, x_M :)$. Using the induction hypothesis, there is an $r' \geq 0$ and $D'_0, \ldots, D'_{r'}$ satisfying the Points 1–5 for $F(b_k \cdots b_{2+\delta}, x_2, \ldots, x_M : y_1, \ldots, y_N)$ where the length k' of $b_k \cdots b_{2+\delta}$ is $k - 1 - \delta$. We choose $r = 1 + r'$, $D_r = 0$, and $D_i = D'_i + \delta + 1$ for $0 \leq i \leq r'$. We find that:

  - Point 1 holds by definition,
  - Point 2 holds by the induction hypothesis when $0 \leq i \leq r' - 1 = r - 2$ while we have

    $$D_{r-1} = D_{r'} = D'_{r'} + \delta + 1 = D_r + \delta + 1 = D_r + d_{b_1}(b_k \cdots b_2, x_2, \ldots, x_M :) + 1$$

    when $i = r'$,
  - Point 3 holds because we by the induction hypothesis have $D_0 = D'_0 + 1 + \delta \geq k' + 1 + \delta = k$,
  - Point 4 holds because $D_r = 0$ and $D_i = D'_i + 1 + \delta < k' + 1 + \delta = k$ when $1 \leq i \leq r' = r - 1$, and
  - Point 5 is true due to $(*)$.

□

Notation 4.8. We will write $\mathtt{e} \rightsquigarrow v$ when the SML-expression $\mathtt{e}$ evaluates to the value v without raising an exception. □

Definition 4.9. Let $v = b_k \cdots b_1 \in \mathbb{N}$ be a number.

1. The *representation* of bit i of v is
   (a) `NONE` when $i \geq k$
   (b) `SOME` b_{i+1} when $0 \leq i \leq k-1$.
2. The SML-expression e of type `int -> bool option` *bitwise represents* v if, and only if, for any expression e' of type `int` where $e' \rightsquigarrow i$ either
   (a) $e\ \ e' \rightsquigarrow R$ where R is the representation of bit i of v, or
   (b) the evaluation of the application $e\ \ e'$ raises an exception.

□

Lemma 4.10. *Let e be an expression that bitwise represents v. Evaluating* `findLength` e *returns* $|v|$. □

Proof. We initially note that `search` $e \rightsquigarrow i$ when $e \rightsquigarrow i$ and $i \geq |v|$. We prove by induction on $|v| - i$ that `search e` $\rightsquigarrow |v|$ when $e \rightsquigarrow i$:

1. $|v| - i = 0$, i.e., $i = |v|$. As e bitwise represents v, we have `z i` $\rightsquigarrow$ `NONE`. We therefore have `search` $e \rightsquigarrow |v|$.
2. $|v| - i > 0$. As $i < |v|$ we have `z i` $\rightsquigarrow$ `SOME` b. We have `i + 1` $\rightsquigarrow i+1$ so by the induction hypothesis, `search (i + 1)` $\rightsquigarrow |v|$. Consequently, `search` $e \rightsquigarrow |v|$.

□

Proposition 4.11. *Let F be a BC^-_ε-function and let $x_1, \ldots, x_M, y_1, \ldots, y_N$ be the normal and safe inputs, resp. Let $x_1, \ldots, x_M, y_1, \ldots, y_N$ be bitwise represented by the SML-expressions* $\mathtt{x1}, \ldots, \mathtt{x}M, \mathtt{y1}, \ldots, \mathtt{y}N$. *Furthermore, assume that* $\mathtt{x1}, \ldots, \mathtt{x}M$ *cannot raise exceptions,*

1. $\lceil F \rceil$ `x1` $\cdots$ $\mathtt{x}M$ `y1` $\cdots$ $\mathtt{y}N$ *bitwise evaluates $F(x_1, \ldots, x_M : y_1, \ldots, y_N)$ raising only exceptions raised by* $\mathtt{y1}, \ldots, \mathtt{y}N$.
2. *In evaluating* $\lceil F \rceil$ `x1` $\cdots$ $\mathtt{x}M$ `y1` $\cdots$ $\mathtt{y}N$ *each of the functions* $\mathtt{y1}, \ldots, \mathtt{y}N$ *are called at most once.*

□

Before we can prove this lemma we need some facts about how the computation of recursion proceeds. This is captured in the following lemmata. The first establishes that `x'` is implemented correct:

Lemma 4.12. *Let $F =$* `rec`(g,h_0,h_1,d_0,d_1) *be a BC^-_ε-function and consider any normal and safe arguments $x_1 = b_k \cdots b_1, x_2, \ldots, x_M, y_1, \ldots, y_N$. Consider the evaluation of* `saferec` *with inputs* $\mathtt{g}, \mathtt{h0}, \mathtt{h1}, \mathtt{d0}, \mathtt{d1}, \mathtt{x1}, \ldots, \mathtt{x}M, \mathtt{y1}, \ldots, \mathtt{y}N$ *that bitwise implements $\lceil g \rceil, \lceil h_0 \rceil, \lceil h_1 \rceil, \lceil d_0 \rceil, \lceil d_1 \rceil, x_1, \ldots, x_M, y_1, \ldots, y_N$, resp, and* `bit` *being a bit index. Assume that $\lceil g \rceil$, $\lceil h_0 \rceil$, and $\lceil h_1 \rceil$ call their safe arguments at most once and that only* $\mathtt{y1}, \ldots, \mathtt{y}N$ *raise exceptions. Assume moreover that* `#depth (!goal)` $\rightsquigarrow D_i$ *for some i where $0 \leq i \leq r$. Then* `x'` *is a bitwise correct implementation of z_i.* □

The second lemma establishes that it is various traces of the same computation that is repeated in `loop2`; it hinges on determinism and the fact that the linearity enforces the safe argument to be used only once.

Lemma 4.13. *Let $F = \texttt{rec}(g,h_0,h_1,d_0,d_1)$ be a BC^-_ε-function and consider any normal and safe arguments $x_1 = b_k \cdots b_1, x_2, \ldots, x_M, y_1, \ldots, y_N$. Consider the evaluation of* `saferec` *with inputs* $\texttt{g}, \texttt{h0}, \texttt{h1}, \texttt{d0}, \texttt{d1}, \texttt{x1}, \ldots, \texttt{x}M, \texttt{y1}, \ldots, \texttt{y}N$ *that bitwise implements $\lceil g \rceil, \lceil h_0 \rceil, \lceil h_1 \rceil, \lceil d_0 \rceil, \lceil d_1 \rceil, x_1, \ldots, x_M, y_1, \ldots, y_N$, resp, and* `bit` *being a bit index. Assume that $\lceil g \rceil$, $\lceil h_0 \rceil$, and $\lceil h_1 \rceil$ call their safe arguments at most once and that only* $\texttt{y1}, \ldots, \texttt{y}N$ *raise exceptions. Then the chain of recursive calls is fixed: There is a unique r' and numbers $j_{r-1'}, \ldots, j_{r-1}$ such that*

1. *$1 \leq r' \leq r+1$ and*
2. *if the function* `recursiveCall` *is called with* `#depth (!goal)` $\rightsquigarrow D_{i'}$ *for some i' then $r' \leq i' \leq r$ and* `bit` $\rightsquigarrow j_{i'-1}$ *inside* `recursiveCall`.

□

Lemma 4.14. *Let $F = \texttt{rec}(g,h_0,h_1,d_0,d_1)$ be a BC^-_ε-function and consider any normal and safe arguments $x_1 = b_k \cdots b_1, x_2, \ldots, x_M, y_1, \ldots, y_N$. Consider the evaluation of* `saferec` *with inputs* $\texttt{g}, \texttt{h0}, \texttt{h1}, \texttt{d0}, \texttt{d1}, \texttt{x1}, \ldots, \texttt{x}M, \texttt{y1}, \ldots, \texttt{y}N$ *that bitwise implements $\lceil g \rceil, \lceil h_0 \rceil, \lceil h_1 \rceil, \lceil d_0 \rceil, \lceil d_1 \rceil, x_1, \ldots, x_M, y_1, \ldots, y_N$, resp, and* `bit` *being a bit index. Assume that $\lceil g \rceil$, $\lceil h_0 \rceil$, and $\lceil h_1 \rceil$ call their safe arguments at most once and that only* $\texttt{y1}, \ldots, \texttt{y}N$ *raise exceptions. Assume moreover that* `#depth (!goal)` $\rightsquigarrow D_i$ *for some i where $0 \leq i \leq r$. Then*

1. *If* `#depth(!result)` $\rightsquigarrow D_{i'}$ *for some i' where $0 \leq i' \leq r$,* `#bit(!result)` $\rightsquigarrow j$ *and* `#res(!result)` $\rightsquigarrow R$ *then the jth bit of $F(z_{i'} b_{D_{i'}+1}, x_2, \ldots, x_M : y_1, \ldots, y_N)$ is represented by R.*
2. *Let* `d` *be a bitwise implementation of $d_{b_{D_i+1}}$. Then* `(recursiveCall d)` *bitwise represents $F(z_{i-1} b_{D_{i-1}+1}, x_2, \ldots, x_M : y_1, \ldots, y_N)$.*
3. *If the computation reaches the body of* `loop2` *with* `#bit (!goal)` $\rightsquigarrow j_i$*, then either an exception is raised from* $\texttt{y1}, \ldots, \texttt{y}N$ *or the computation will eventually reach the end of* `loop1` *with* `#depth (!result)` $\rightsquigarrow D_i$ *and* `#res(!result)` $\rightsquigarrow B_i$ *where B_i is the representation of bit j_i of $F(z_i b_{D_i+1}, x_2, \ldots, x_M : y_1, \ldots, y_N)$.*

□

The first point and Point 2 of this lemma obviously expresses qualities about the invariants of various parts of the code. Part 1 captures that we are keeping track of the computation. The last part in all its technicality expresses the fact that `loop2` succeeds in computing the requested bit.

Corollary 4.15. *If* `recursiveCall` *raises* `Restart`n *we have* `#depth(!goal)` $\rightsquigarrow D_{i-1}$ *and* `#bit(!goal)` $\rightsquigarrow j_{i-1}$. □

Corollary 4.16. *Function* `recursiveCall` *succeeds if, and only if,* `#depth(!goal)` $\rightsquigarrow D_i$*,* `#depth(!result)` $\rightsquigarrow D_{i-1}$*, and* `#bit(!goal)` $\rightsquigarrow j_{i-1}$. □

The lemmata will be proved after the main proof when the reader (hopefully) can appreciate its necessity.

Proposition 4.11. We prove the two claims simultaneous using induction on F:

1. $F = \texttt{0}$: We have $\lceil F \rceil =$ `zero`. As there are no safe arguments, Point 2 is trivially true. As the length of 0 is 0 and `zero` returns `NONE` for all inputs `zero` bitwise represent $\lceil F \rceil$.

2. $F = \mathtt{s}_b$: We have $\lceil F \rceil = \mathtt{succ}b$. It is immediate that the safe argument y1 will be called at most once. Furthermore, recall that $\mathtt{s}_b(: b_k \cdots b_1) = b_k \cdots b_1 b$ and consider the application succ$b$ e where e $\rightsquigarrow v$:

   - When $v = 0$, we have succ$b$ e $\rightsquigarrow$ SOME b.
   - When $1 \leq v \leq k$, we have succ$b$ e $\rightsquigarrow$ SOME b_v or an exception is raised from y1 as y1 bitwise represents y_1.
   - When $v \geq k+1$, we have succ$b$ e $\rightsquigarrow$ NONE or get an exception raised from y1 as y1 bitwise represents y_1.

3. $F = \mathtt{p}$: We have $\lceil F \rceil = \mathtt{pred}$. It is immediate that the safe argument y1 will be called at most once. Furthermore, recall that $\mathtt{p}(: b_k \cdots b_1) = b_k \cdots b_2$ and consider the application pred e where $e \rightsquigarrow v$:

   - When $0 \leq v \leq k-1$, we get pred $e \rightsquigarrow$ SOME b_{v+2} or an exception raised from y1 as y1 bitwise represents y_1.
   - When $v \geq k$, we get pred $e \rightsquigarrow$ NONE or an exception raised from y1 as y1 bitwise represents y_1.

4. $F = \mathtt{c}$: We have $\lceil F \rceil = \mathtt{cond}$. It is trivially seen that the safe argument y1 will be called once and that the arguments y2 and y3 will be called at most once. As y1 bitwise represents y_1, evaluating y1 0 will raise an exception from y1, evaluate to SOME b where b is the lowermost bit of y_1 or NONE when $y_1 = \varepsilon$. Consequently, y2 bit is evaluated when the lowermost is 1 and y3 bit otherwise. As y2 and y3 bitwise represents y_2 and y_3 we get the desired bit or an exception raised by y2 or y3.

5. $F = \pi_j^{m,n}$: We have $\lceil F \rceil = \mathtt{proj}\text{-}m\text{-}n\text{-}j$. Clearly any y$i$ will be called at most once. Furthermore, an exception is raised by $\mathtt{y1}, \ldots, \mathtt{y}N$ or the relevant bit is returned as the inputs $\mathtt{x1}, \ldots, \mathtt{x}M, \mathtt{y1}, \ldots, \mathtt{y}N$ bitwise represents the inputs $x_1, \ldots, x_M, y_1, \ldots, y_N$, resp.

6. $F = f \circ \langle g_1, \ldots, g_M : h_1, \ldots, h_N \rangle$: We have

   $$\lceil F \rceil = \mathtt{scomp}\ \lceil f \rceil\ \lceil g_1 \rceil\ \cdots\ \lceil g_M \rceil\ \lceil h_1 \rceil\ \cdots\ \lceil h_N \rceil\ .$$

   Due to the linearity condition on safe linear composition the arguments $\mathtt{y1}, \ldots, \mathtt{y}n$ is argument of at most one of the functions $\mathtt{h1}, \ldots, \mathtt{h}N$. By the induction hypothesis, f calls each of the functions $\mathtt{Y1}, \ldots, \mathtt{Y}N$ at most once. If Yj is evaluated, hj is evaluated and, by the induction hypothesis, each of the arguments $\mathtt{y}i_{j,1}, \ldots, \mathtt{y}i_{j,l_j}$ is evaluated at most once. We conclude that each of the arguments $\mathtt{y1}, \ldots, \mathtt{y}n$ is called at most once.

   By the induction hypothesis, the SML-functions $\lceil g_1 \rceil, \ldots, \lceil g_M \rceil, \lceil h_1 \rceil, \ldots, \lceil h_N \rceil$ bitwise evaluates $g_1, \ldots, g_M, h_1, \ldots, h_N$, resp. Thus $\mathtt{X1}, \ldots, \mathtt{X}M, \mathtt{Y1}, \ldots, \mathtt{Y}N$ bitwise evaluates

   $$g_1(x_1, \ldots, x_m :), \ldots, g_M(x_1, \ldots, x_m :),$$
   $$h_1(x_1, \ldots, x_m : y_{1,1}, \ldots, y_{1,l_1}), \ldots, h_N(x_1, \ldots, x_m : y_{1,1}, \ldots, y_{N,l_N})\ .$$

   As non of the functions $\mathtt{x1}, \ldots, \mathtt{x}M$ can raise exceptions, the functions $\mathtt{X1}, \ldots, \mathtt{X}M$ raise exceptions. By the induction hypothesis, we know f X1 $\cdots$ XM Y1 $\cdots$ yN bitwise evaluates

   $$f \circ \langle g_1, \ldots, g_M : h_1, \ldots, h_N \rangle =$$
   $$f(\, g_1(x_1, \ldots, x_m :), \ldots, g_M(x_1, \ldots, x_m :) :$$
   $$h_1(x_1, \ldots, x_m : y_{1,1}, \ldots, y_{1,l_1}), \ldots, h_L(x_1, \ldots, x_m : y_{1,1}, \ldots, y_{N,l_N}))\ .$$

7. F = rec(g,h_0,h_1,d_0,d_1): We have $\lceil F \rceil$ = saferec $\lceil g \rceil$ $\lceil h_0 \rceil$ $\lceil d_0 \rceil$ $\lceil h_1 \rceil$ $\lceil d_1 \rceil$. The ground work of this case has really been done by Point 3 of Lemma 4.14: loop2 is initialized with #depth (!goal) $\rightsquigarrow 0 = D_r$ and #bit (!goal) $\rightsquigarrow j$ where bit $\rightsquigarrow j$. From Point 3 of Lemma 4.14 it follows that either an exception is raised from the safe arguments or the computation reaches the end of loop1 with #depth (!result) $\rightsquigarrow 0$ and bit j of $F(x1, \ldots, x_M : y_1, \ldots, y_N)$. The latter means that loop2 terminates and saferec returns the requested bit.

□

Corollary 4.17. *Using the driver loop in Fig. 5 the computation of the BC^-_ε-function F succeeds.* □

Proof. None of the functions provided by the driver loop to compute the safe arguments can raise an exception. It follows from the previous proposition that the computation of the individual bits succeed. It is immediate to see that the driver loop will proceed to compute each of the bits in F. □

Theorem 4.18. *For any function definable in the function algebra BC^-_ε, there is a Turing Machine evaluating the function in* LOGSPACE. *Moreover, the Turing Machine can be constructed from the function expression in logarithmic space in the constant work space.* □

Before we raise our arms in victory let us establish the correctness of the points claimed in Lemma 4.12–4.14.

Lemma 4.12. We want to establish that x' bitwise implements $z_i = b_k \cdots b_{D_i+2}$. The length of z_i is $k - D_i - 1$. We consider x' e where $e \rightsquigarrow v$. We have two cases:

1. When $v \geq k - D_i - 1$ we have 1 + bit + #depth(!goal) $\rightsquigarrow 1 + v + D_i - 1 \geq k$ so x' returns NONE.

2. When $v < k - D_i$ we have 1 + bit + #depth(!goal) $\rightsquigarrow 1 + v + D_i$. Consequently, SOME b_{v+2+D_i} is returned since $0 \leq 1 + v + D_i \leq k$. This is the representation of bit v of z_i.

□

Lemma 4.13. we first consider the case where !result has been initialized to NORESULT. We assume that #depth (!goal) $\rightsquigarrow D_i$ for some i where $0 \leq i \leq r$. We furthermore assume that there are unique $j_i, \ldots, j_{r-1}$ such that recursiveCall has been called with bit $\rightsquigarrow j_{i'-1}$ and #depth(!goal) for all i' where $i < i' \leq r$. We proceed to prove that the statement holds in loop2 under these assumptions by induction on i.

1. When $i = 0$ we have $D_0 \geq k$. We have x1 (#depth (!goal)) $\rightsquigarrow$ NONE as x1 bitwise implements x_1 so $\lceil g \rceil$ is called with x2, ..., xM, y1, $\cdots$, yN. There is therefore not made any calls to recursiveCall and using $r' = 1$ satisfy the claim.

2. When $i > 0$ we have $D_i < k$. We have x1 (#depth (!goal)) $\rightsquigarrow$ SOME b_{D_i+1} as x1 bitwise implements x_1 . We therefore have h = hb_{D_i+1} and d = db_{D_i+1}. There are two possible outcomes of evaluating h x' $\cdots$ (#bit (!goal)):

   (a) There is not made a call to recursiveCall. Since no exception is raised, loop2 succeeds and result is updated. Using $r' = i + 1$ satisfy the claim.

(b) `recursiveCall` is called. Exploiting Lemma 4.12 it follows `findLength` returns $|d_{b_{D_i+1}}(z_i, x_2, \ldots, x_M)|$, i.e., `delta` $\rightsquigarrow 1 + |d_{b_{D_i+1}}(z_i, x_2, \ldots, x_M)|$. Let `bit` $\rightsquigarrow j$. Since the conditional fails, exception `Restart`n is raised after `goal` has been changed such that `#depth(!goal)` $\rightsquigarrow D_{i-1}$ and `#bit(!goal)` $\rightsquigarrow j$. Consequently, `loop2` is restarted and by choosing $j_{i-1} = j$ we can apply the induction hypothesis to obtain the unique r'.

This establishes the lemma in the special case where `!result` has been initialized to `NORESULT` and unique $j_i, \ldots, j_{r-1}$ are given. We notice that when $i = r$ the condition on unique $j_i, \ldots, j_{r-1}$ is trivially satisfied. It therefore follows that the lemma holds for the first iteration of `loop1`.

We now observe that the only variables governing the control flow is `!goal` (for the choice of bit from `x1`) and `!result` (whether an exception is raised in `recursiveCall`). We notice that `goal` is initialized to the same values at the start of every iteration of `loop1`. It follows from the determinism of SML that the computation will go through the same calls to `recursiveCall`; however, `loop2` might terminate at a higher D_i depending on the contents of `!result`. This concludes the proof that the chain of recursive call is fixed.

□

Lemma 4.14. We recognize that Point 1 is trivially true initially. It is therefore sufficient to check that it is valid after updating `result` at the end of `loop1`. We prove the three statements using simultaneous induction on i as they depend on each other

We exploit Lemma 4.12 to prove Point 2: `findLength` returns $|d_{b_{D_i+1}}(z_i, x_2, \ldots, x_M)|$ when `recursiveCall d` is called. We therefore have `delta` $\rightsquigarrow 1 + |d_{b_{D_i+1}}(z_i, x_2, \ldots, x_M)|$. We split into two cases:

1. When `#depth (!result)` $\rightsquigarrow D_{i-1}$ and `#bit (!result)` $\rightsquigarrow$ `bit`. Using the characterization in Lemma 4.7 it follows that

   `#depth (!goal) + delta = #depth (!result)` $\rightsquigarrow$ `true` .

   From the induction hypothesis on Point 1 and Lemma 4.13 we conclude that the relevant bit is returned.

2. Otherwise, `goal` is changed so `#depth(!goal)` $\rightsquigarrow D_{i-1}$ and `#bit(!goal)` $\rightsquigarrow j_{i-1}$ and the exception `Restart`n is raised.

For Points 3 and 1, we consider two cases:

1. When $i = 0$ we have $D_0 \geq k$. We have `x1 (#depth (!goal))` $\rightsquigarrow$ `NONE` as `x1` bitwise implements x_1 and $\lceil g \rceil$ is called with $\mathtt{x2}, \ldots, \mathtt{x}M, \mathtt{y1}, \cdots, \mathtt{y}N$. By the induction hypothesis for the proposition, $\lceil g \rceil$ bitwise represents g calling $\mathtt{y1}, \ldots, \mathtt{y}N$ at most once. So either an exception is raised from $\mathtt{y1}, \ldots, \mathtt{y}N$, or the requested bit is stored in `result`. In the latter case, `result` is updated and it is immediate that Point 1 is satisfied.

2. When $i > 0$ we have $D_i < k$. We have `x1 (#depth (!goal))` $\rightsquigarrow$ `SOME` b_{D_i+1} as `x1` bitwise implements x_1. We therefore have $\mathtt{h} = \mathtt{h}b_{D_i+1}$ and $\mathtt{d} = \mathtt{d}b_{D_i+1}$.

   It follows from the induction hypothesis for the proposition that $\mathtt{h}b_{D_i}$ bitwise implements $h_{b_{D_i}}$. Furthermore, using Lemma 4.12 and 2 we have one of the following:

(a) `h x' ⋯ (#bit (!goal))` $\rightsquigarrow R$ where R is the represenation of bit `#bit (!goal)` of $h(x_1, \ldots, x_M : F(z_{i-1}b_{D_{i-1}+1}, x_2, \ldots, x_M : y_1, \ldots, y_N))$.

(b) During the evaluation of `h x' x2` $\cdots$ `x`m `(#bit (!goal))` an exception is raised from `recursiveCall d`. By Corollary 4.15, we have `#depth (!goal)` $\rightsquigarrow D_{i-1}$ and `#bit (!goal)` $\rightsquigarrow j_{i-1}$.

The exception is caught by the handler and another iteration of `loop2` is started. With the updated `!goal`, we can apply the induction hypothesis. Consequently, we have one of the following:

i. an exception is raised from `y1`, $\ldots$, `y`M, or

ii. the computation reaches the end of `loop1` with `#depth (!result)` $\rightsquigarrow D_{i-1}$ and `#res(!result)` $\rightsquigarrow R_{i-1}$ where R_{i-1} is the representation of bit j_{i-1} of $F(z_{i-1}b_{D_{i-1}+1}, x_2, \ldots, x_M : y_1, \ldots, y_N)$.
Since $D_{i-1} > 0$ by Lemma 4.7 another iteration of `loop1` is started and `#depth (!goal)` is reset to 0. At this point of the computation, let i' denote the number such that `#depth (!goal)` $\rightsquigarrow D_{i'}$. Initially, we have $i' = r$ when `#depth(!goal)` $\rightsquigarrow 0$. Using induction on $i - i'$ we prove that the computation reaches the end of `loop1` with `#depth (!result)` $\rightsquigarrow D_i$ and `#res(!result)` $\rightsquigarrow R_i$ where R_i is the bit needed from the previous step, i.e., it is the representation of bit j_i of $F(z_i b_{D_i+1}, x_2, \ldots, x_M : y_1, \ldots, y_N) = h_{b_{D_i+1}}(z_i, x_2, \ldots, x_M : y_1, \ldots, y_N)$:

- When $i - i' = 0$: The call of $\lceil h_{b_{D_i+1}} \rceil$ will immediately succeed because the wanted result is cached. Consequently, `result` is updated. We notice that the update is in accordance with Point 1.
- The computation of $h_{b_{D_i+1}}$ raises `Restart`n inside `recursiveCall`. Using Corrollary 4.15, it follows that `goal` is updated. We then apply the induction hypothesis to get the right result.

□

5 Notes

In the literature there are other characterizations of LOGSPACE. Goerdt [6] characterizes LOGSPACE (and other complexity classes) in finite model theory. Jones [8] recasts the results in terms of read-only (`cons`-less) functional program using only `fold` to do iteration. Unlike $\mathrm{BC}^-_\varepsilon$ they cannot produce big output (being read-only). On the other hand they allow pairs of values, which is a data type that does not seem encodable in $\mathrm{BC}^-_\varepsilon$.

Another interesting characterization of LOGSPACE is due to Jones [7]. He establish that the problem $\mathrm{DAG} \cap \mathrm{DGAP}$ is LOGSPACE-complete under so-called rudimentary reductions. The problem can be stated as follows: given a directed acyclic graph where each vertex has at most one outgoing edge; is there a path from the first to the last vertex. Notice that this is easily checkable in LOGSPACE: initialize the first vertex as the current vertex; now repeat the following step: if there is an edge from the current vertex update the current vertex to be the one the edge points to; if we terminate at the last vertex there is a path. It is also easy to represent this algorithm in $\mathrm{BC}^-_\varepsilon$. The author do, however, not believe that it can be done in BC^- due to the fact that only one bit determining control can be transfered in the recursive step (and he is willing to give a good bottle of Whisky to anyone who succeeds).

Acknowledgments

I appreciate the support and motivation provided in mysterious ways by my advisor Harry Mairson.

References

[1] A. Asperti. Light affine logic. In *Proc. 13th Ann. IEEE Symp. Logic in Comput. Sci.*, pages 300–308, 1998.

[2] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, revised edition, 1984.

[3] S. Bellantoni and S. Cook. A new recursion-theoretic characterization of the polytime functions. *Computational Complexity*, 2:97–110, 1992.

[4] W. D. Clinger. Proper tail recursion and space efficiency. In *Proc. ACM SIGPLAN '98 Conf. Prog. Lang. Design & Impl.*, pages 174–185, 1998.

[5] J.-Y. Girard. Light linear logic. *Inform. & Comput.*, 143:175–204, 1998.

[6] A. Goerdt. Characterizing complexity classes by higher type primitive recursive definitions. *Theoret. Comput. Sci.*, 100(1):45–66, June 1992.

[7] N. D. Jones. Space-bounded reducibility among combinatorial problems. *J. Comput. System Sci.*, 11(1):68–85, Aug. 1975.

[8] N. D. Jones. The expressive power of higher-order types, or life without CONS. *J. Funct. Programming*, 11(1):55–94, Jan. 2001.

[9] R. Kelsey, W. Clinger, and J. R. (Editors). Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, Sept. 1998.

[10] A. J. Kfoury. Recursion versus iteration at higher-orders. In *Foundations of Software Technology and Theoret. Comput. Sci., 17th Conf.*, volume 1346 of *LNCS*, pages 57–73. Springer-Verlag, Dec. 1997.

[11] A. J. Kfoury. Recursion, tail-recursion and iteration at higher-orders. Notes for a paper, 1999.

[12] R. Milner, M. Tofte, R. Harper, and D. B. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[13] P. Møller Neergaard and H. G. Mairson. How light is safe recursion? translations between logics of polynomial time. Submitted, 2003.

[14] A. S. Murawski and C.-H. L. Ong. Can safe recursion be interpreted in light logic? In *2nd International Workshop on Implicit Computational Complexity*, June 2000.

[15] R. Péter. *Recursive Functions*. Academic Press, New York, 1967.