On an Interpretation of Safe Recursion in Light Affine Logic [1]

A. S. Murawski [2] C.-H. L. Ong [3]

{*Andrzej.Murawski,Luke.Ong*}@*comlab.ox.ac.uk*

Oxford University Computing Laboratory (OUCL), Wolfson Building, Parks Road, Oxford OX1 3QD, UK.

Contents

Abstract

We introduce a subalgebra BC^- of Bellantoni and Cook's safe-recursion function algebra BC. Functions of the subalgebra have safe arguments that are non-contractible (i.e non-duplicable). We propose a definition of safe and normal variables in Light Affine Logic (LAL), and show that BC^- is the largest subalgebra that is *interpretable* in LAL, relative to that definition. Though BC^- itself is not PF complete, there are extensions of it (by additional schemes for defining functions with safe arguments) that are, and are still interpretable in LAL and so preserve PF closure. We focus on one such which is BC^- augmented by a definition-by-cases construct and a restricted form of definition-by-recursion scheme over *safe* arguments. As a corollary we obtain a new proof of the PF completeness of LAL.

[1] A preliminary version of this paper was presented at the *Implicit Computational Complexity Workshop* in Santa Barbara, USA, June 2000.
[2] On leave from Nicholas Copernicus University, Toruń, Poland
[3] Homepage: `http://www.comlab.ox.ac.uk/oucl/people/luke.ong.html`

1 Introduction

PF (PF_1) is the collection of numeric functions (unary numeric functions) that are computable by a Turing machine whose runtime is bounded by a polynomial in the length of its input. Bellantoni and Cook's function algebra BC [2] is generated from several basic functions by a form of composition scheme and a certain primitive recursion scheme, called safe composition and safe recursion respectively. BC captures PF by partitioning the argument positions of each function $g(\vec{x};\vec{y})$ into those that are *normal* $\vec{x}$ and those that are *safe* $\vec{y}$: the arguments of the basic functions are all safe, and primitive recursion is only allowed over normal arguments. A crucial feature of the safe recursion scheme: for $i = 0, 1$

$$\begin{aligned} f(0,\vec{x};\vec{y}) &= g(\vec{x};\vec{y}) \\ f(\mathsf{S}_i(z),\vec{x};\vec{y}) &= h_i(z,\vec{x};\vec{y},f(z,\vec{x};\vec{y})) \text{ if } \mathsf{S}_i(z) > 0 \end{aligned}$$

is that the recursive call to the function being defined $f(z,\vec{x};\vec{y})$ may only appear in a safe argument position of h_i. This ensures, in particular, that recursion over the result of a function defined by recursion is not possible. For ease of reference, we give the rules that define BC. BC is the smallest class of functions containing the initial functions:

- $0(;)$ (a 0-ary function)
- projections $\pi_j^{m,n}(x_1,\cdots,x_m;x_{m+1},\cdots,x_{m+n}) = x_j$ for $1 \le j \le m+n$
- successors $\mathsf{S}_i(;x) = 2x + i$ for $i = 0, 1$
- predecessor $\mathsf{P}(;x) = \lfloor \frac{x}{2} \rfloor$
- conditional

$$\mathbf{cond}(;a,b,c) = \begin{cases} b \text{ if } a \bmod 2 = 0 \\ c \text{ otherwise} \end{cases}$$

and closed under the rules of ***safe composition***

$$f(\vec{x};\vec{y}) = h(g_1(\vec{x};),\ldots,g_m(\vec{x};);h_1(\vec{x};\vec{y}),\cdots,h_n(\vec{x};\vec{y}))$$

and ***safe recursion***: for $i = 0, 1$

$$\begin{aligned} f(0,\vec{x};\vec{y}) &= g(\vec{x};\vec{y}) \\ f(\mathsf{S}_i(z),\vec{x};\vec{y}) &= h_i(z,\vec{x};\vec{y},f(z,\vec{x};\vec{y})) \text{ if } \mathsf{S}_i(z) > 0 \end{aligned}$$

The main result is:

Theorem 1 (Bellantoni-Cook) *BC-definable functions of the form $f(x;)$ are exactly the PF_1 functions.* $\square$

Girard and Asperti's Light Affine Logic [3,1] is a second-order type theory. It has a polytime cut-elimination procedure and can encode all polytime numeric functions. In Girard's words, it is an "intrinsically polytime system", whose proofs may be regarded as (representations of) polytime algorithms. Both BC and Light Affine Logic (LAL) are resource-free characterizations of PF. This paper is concerned with the connexions between them. In particular, we ask if it is possible to interpret BC in LAL, in a sense that should be made precise.

Our starting point is the observation that in LAL the type constructor § is a mark of iteration. I.e. by iterating an algorithm s (say) of type $\textsc{Bint} \multimap \textsc{Bint}$ over $\textsc{Bint}$ (where $\textsc{Bint}$ is the LAL type that encodes $\mathbb{N}$ as binary words), we obtain another, s' say, that has type $\textsc{Bint} \multimap \S\textsc{Bint}$. Thus we cannot iterate s' any more ($\S\textsc{Bint} \multimap \textsc{Bint}$ is not provable and so s' cannot be converted back to a term of type $\textsc{Bint} \multimap \textsc{Bint}$, or indeed to one of type $A \multimap A$ for any A). Since step functions in recursive definitions must have types of the form $f : (\cdots \otimes A \otimes \cdots) \multimap A$, we propose a notion of safe and normal variables in LAL: Suppose the sequent

$$x_1 : \textsc{Bint}, \cdots, x_m : \textsc{Bint}, y_1 : \S^k\textsc{Bint}, \cdots, y_n : \S^k\textsc{Bint} \vdash t : \S^k\textsc{Bint}$$

is provable. If $k \geq 1$, we say that the variables x_i are *normal* and y_j are *safe* in t; if $k = 0$ we say that all variables are safe.

LAL safe variables are different from those in BC in an important way: they are not contractible (i.e. not duplicable) in the sense of Proposition 8, though normal variables are. In Section 3 we introduce a subalgebra BC^- of the function algebra BC, defined by restricting the safe composition and safe recursion schemes in a way that respects the non-contractibility of safe variables. We show that BC^- is the largest subalgebra that is *interpretable* in LAL in a sense which will be made precise.

Though BC^- itself is not PF complete, there are extensions of it (by additional schemes for defining functions of safe arguments) that are, and are still interpretable in LAL and so preserve PF closure. We focus on one such in Section 4 which is BC^- augmented by a definition-by-cases construct and a restricted form of definition-by-recursion scheme over *safe* arguments. We call the augmented algebra $BC^{\pm}$. In Section 5 we prove that all PF functions $\mathbb{N} \to \mathbb{N}$ are contained in $BC^{\pm}$. As a corollary we obtain a new proof of the PF completeness of LAL. Finally in Section 6 we briefly mention two other ways of completing BC^-.

Convention 2 In this paper, by ***strings*** we mean elements of $\{0,1\}^*$. We call a string ***numeric*** if it is empty or its leftmost symbol is 1. For any numeric string s, we write 's' to mean the number whose binary representation is s; e.g. '100' $= 4$. For ease of reading, we write $\mathsf{S}_i(;n)$ (and from Section 3 onwards also $\mathsf{S}_i(:n)$) simply as $\mathsf{S}_i(n)$, similarly for $\mathsf{P}(;n)$ and $0(;)$. For any string s, we define $\mathsf{S}_s(:n)$ by recursion on s: $\mathsf{S}_{si} = \mathsf{S}_s \circ \mathsf{S}_i$ and S_ϵ is the identity map on $\mathbb{N}$.

2 Safety in Light Affine Logic

Light Affine Logic (LAL) is a sequent calculus that has two modalities ! and § which enable a systematic elimination of cuts from the outermost level to the innermost and a careful management of duplication. In this section we present a term assignment system for LAL; the valid LAL typing sequents are defined by the rules in Figure 1.

The convention for the term language is that variables which appear in the denominator position (i.e. on the right of "-/-") are binders. This is not the place to justify the design and analyse the syntax of the term language (we direct readers who are interested to [7,8]). Our aim here is simply to use the term language as a compact (indeed LaTeX-able) programming notation to express LAL derivations. To make the terms more readable, we introduce some shorthand notation.

Notation 3 We abbreviate $!\langle x'/x \rangle(s)\{t/x'\}$ to $!\langle t/x \rangle(s)$ and analogously for $\S\langle \overline{z'}/\overline{z} \rangle(s)\{t/z'\}$. For instance, $\S\langle z_1'/z_1, t/z_2, z_3'/z_3 \rangle(s)$ stands for

$$\S\langle z_1'/z_1, z_2'/z_2, z_3'/z_3 \rangle(s)\{t/z_2'\}.$$

We write $(\langle z\,,\,t/x \rangle x)\{s/z\}$ and $(\langle z\,,\,T/x \rangle x)\{t/z\}$ as respectively the standard application st and the standard type application $t[T]$. $\boldsymbol{\lambda}(z_1 \otimes z_2).t$ is shorthand for $\boldsymbol{\lambda} z.\langle z/z_1 \otimes z_2 \rangle t$, and T^n stands for $\underbrace{T \otimes \cdots \otimes T}_{n}$.

Natural numbers, *qua* numeric strings, can be represented in LAL as closed terms of type

$$\textsc{Bint} \;=\; \forall\alpha.!(\alpha \multimap \alpha) \multimap !(\alpha \multimap \alpha) \multimap \S(\alpha \multimap \alpha).$$

For instance 0 is represented by $\boldsymbol{\Lambda}\alpha.\boldsymbol{\lambda} f_0 f_1.\S(\boldsymbol{\lambda} x.x)$ and '100' by

$$\boldsymbol{\Lambda}\alpha.\boldsymbol{\lambda} f_0^{!(\alpha\multimap\alpha)} f_1^{!(\alpha\multimap\alpha)}.\S\langle f_0/f_0^0, f_0/f_0^1, f_1/f_1^2 \rangle(\boldsymbol{\lambda} x.f_0^0(f_0^1(f_1^2 x)))).$$

(ax) $x : A \vdash x^A : A$

(exch) $$\frac{\Gamma, x : A, y : B, \Delta \vdash s : C}{\Gamma, y : B, x : A, \Delta \vdash s : C}$$

(⊗-l) $$\frac{x : A, y : B, \Gamma \vdash s : C}{z : A \otimes B, \Gamma \vdash \langle z^{A\otimes B} / x^A \otimes y^B \rangle s : C}$$

(⊗-r) $$\frac{\Gamma \vdash s : A \qquad \Delta \vdash t : B}{\Gamma, \Delta \vdash s \otimes t : A \otimes B}$$

(⊸-l) $$\frac{\Gamma \vdash s : A \qquad y : B, \Delta \vdash t : C}{z : A \multimap B, \Gamma, \Delta \vdash \langle z^{A\multimap B}\,,\ s/y^B \rangle t : C}$$

(⊸-r) $$\frac{\Gamma, x : A \vdash s : B}{\Gamma \vdash \boldsymbol{\lambda} x^A . s : A \multimap B}$$

(contr) $$\frac{x_1 : !A, x_2 : !A, \Gamma \vdash s : B}{x : !A, \Gamma \vdash s[x/x_1, x/x_2] : B}$$

(weak) $$\frac{\Gamma \vdash s : A}{y : B, \Gamma \vdash s : A}$$

(!) $$\frac{x' : A \vdash s : B}{x : !A \vdash !\langle x/x' \rangle (s) : !B}$$

(!$_0$) $$\frac{\vdash s : B}{\vdash !(s) : !B}$$

(§) $$\frac{\overline{x} : \Gamma, \overline{y} : \Delta \vdash s : A}{\overline{x'} : !\Gamma, \overline{y'} : \S\Delta \vdash \S\langle \overline{x'}/\overline{x}, \overline{y'}/\overline{y} \rangle (s) : \S A}$$

(§$_0$) $$\frac{\vdash s : B}{\vdash \S(s) : \S B}$$

(∀-l) $$\frac{y : A[B/\alpha], \Gamma \vdash t : C}{z : \forall\alpha.A, \Gamma \vdash \langle z^{\forall\alpha.A}\,,\ B/y \rangle t : C}$$

(∀-r) $$\frac{\Gamma \vdash s : A}{\Gamma \vdash \boldsymbol{\Lambda}\alpha.s : \forall\alpha.A}$$

(cut) $$\frac{\Gamma, x : A \vdash s : B \qquad \Delta \vdash t : A}{\Gamma, \Delta \vdash s\{t / x^A\} : B}$$

N.B. The rule (∀-**r**) has a side condition: α does not occur free in Γ.

Fig. 1. Rules that define valid LAL sequents

The successor S_1 can be defined as

$$n : \textsc{Bint} \vdash \boldsymbol{\Lambda}\alpha.\boldsymbol{\lambda} f_0 f_1.\S\langle\, n[\alpha] f_0 f_1 / n', f_1/f_1' \,\rangle(\boldsymbol{\lambda} x.f_1'(n'x)) : \textsc{Bint}.$$

The definition of the other successor S_0 is slightly complicated by the fact that $\mathsf{S}_0(0) = 0$, so that we may not simply append 0 to the right of the input numeric string (i.e. to the left of its representation). We will soon deal with the problem once and for all by availing ourselves of a term **strip** : $\textsc{Bint} \multimap \textsc{Bint}$ whose effect on a string (containing at least one occurrence of 1) is to erase the leftmost symbol iteratively as long as it is 0.

Many numeric functions can be represented in LAL with the help of an ***iteration principle***. Suppose we have $f' : A \vdash x : X$, $f_0' : A_0 \vdash g_0 : X \multimap X$, $f_1' : A_1 \vdash g_1 : X \multimap X$ and $\Box \in \{!, \S\}$. The iterative application of the step functions g_0 and g_1 to the seed x, following the pattern given by the binary representation of n, is represented by

$$\begin{aligned} & n : \textsc{Bint}, f : \Box A, f_0 : !A_0, f_1 : !A_1 \vdash \\ & \qquad \S\langle\, n[X]\, !\langle\, f_0/f_0' \,\rangle(g_0)\, !\langle\, f_1/f_1' \,\rangle(g_1)/h, \Box\langle\, f/f' \,\rangle(x)/y \,\rangle(hy) : \S X \end{aligned}$$

Note that each time the iteration principle is invoked, the result type is “lifted” by a §.

For example, we can apply the iteration principle to show that $\textsc{Bint}$ is embeddable in $\S\Box^k\textsc{Bint}$, for any $k \geq 0$, where $\Box$ stands for either § or !. To see that, we use 0 as the seed and $\vdash \Box^k \mathsf{S}_i : \Box^k\textsc{Bint} \multimap \Box^k\textsc{Bint}$ as the step functions to construct a closed term

$$\mathbf{coerc}_{\S\Box^k} \ : \ \textsc{Bint} \multimap \S\Box^k\textsc{Bint}$$

Such terms are called *coercions* in [3].

Lemma 4 *The term* **strip** : $\textsc{Bint} \multimap \textsc{Bint}$ *is definable in LAL.*

PROOF. We use iteration for $X = P \otimes (\alpha \multimap \alpha)$ where $P = \forall\alpha.\alpha \otimes \alpha \multimap \alpha$. Let $\pi_i = \boldsymbol{\Lambda}\alpha.\boldsymbol{\lambda}(x_0 \otimes x_1).x_i$ for $i = 0, 1$. The first component of X will be π_0 as long as the iteration should proceed according to the offending zeros. When the first 1 occurs, it will be set to π_1. From then on, the binary symbols may be appended to the result. Depending on the projection, different actions will be taken. When processing 0's, we will just copy the projection from the previous stage and ignore the current bit. 1's will be fixing the projection at π_1 and adding the current bit to the aggregate which ensures the desired effect. Moreover, the projections will always be used to select the prescribed actions. Note that this means that we have to duplicate the projection when processing 0's. To that end, we can use their polymorphic type and set $\mathbf{dup}\, p = p[P \otimes P]((\pi_0 \otimes \pi_0) \otimes (\pi_1 \otimes \pi_1))$. The step functions are:

$$f_0 : (\alpha \multimap \alpha) \vdash \boldsymbol{\lambda}(p \otimes c).\langle \mathbf{dup}\, p \,/ p_1 \otimes p_2 \,\rangle(p_1 \otimes (p_2[F](F_0 \otimes F_1))(f_0 \otimes c))$$

and

$$f_1 : (\alpha \multimap \alpha) \vdash \boldsymbol{\lambda}(p \otimes c).(\pi_1 \otimes (p[F](F_0 \otimes F_1))(f_0 \otimes c))$$

where $F_i : F$ are functions which manipulate with the data and append the bit when the current projection is π_1 while ignoring it in the other case. Hence $F = (\alpha \multimap \alpha)^2 \multimap (\alpha \multimap \alpha)$,

$$F_0 = \boldsymbol{\lambda}(f^{(\alpha \multimap \alpha)} \otimes c^{(\alpha \multimap \alpha)}).c$$

and

$$F_1 = \boldsymbol{\lambda}(f^{(\alpha \multimap \alpha)} \otimes c^{(\alpha \multimap \alpha)}).\boldsymbol{\lambda} x^{\alpha}.f(cx).$$

The iteration principle with $x = \pi_0 \otimes I_\alpha$ gives a proof of:

$$\textsc{Bint}, !(\alpha \multimap \alpha), !(\alpha \multimap \alpha) \vdash \S(\alpha \multimap \alpha)$$

after we extract the component of type $(\alpha \multimap \alpha)$. By lambda and type abstractions, we arrive at the term **strip**. $\Box$

We can now define S_0 by applying **strip** to the result of

$$n : \textsc{Bint} \vdash \boldsymbol{\Lambda}\alpha.\boldsymbol{\lambda} f_0 f_1.\S\langle\, n[\alpha] f_0 f_1/n', f_0/f_0' \,\rangle(\boldsymbol{\lambda} x.f_0'(n'x)) : \textsc{Bint}$$

The resulting LAL term – call its curried form s – *represents* the successor function $x \in \mathbb{N} \mapsto 2x$ in the obvious sense that for any $n \geq 0$, $s\,\overline{n}$ is equal (in the theory of LAL) to $\overline{2n}$, where $\overline{n}$ is the term of type $\textsc{Bint}$ that represents the numeral n. Generally take a valid LAL term-in-context

$$x_1 : \Xi_1\textsc{Bint}, \cdots, x_n : \Xi_n\textsc{Bint} \vdash t : \Xi\textsc{Bint}$$

where each $\Xi_i\textsc{Bint}$ and $\Xi\textsc{Bint}$ are either $\textsc{Bint}$ or $\S\Box^k\textsc{Bint}$ for some $k \geq 0$; we write the corresponding coercion term as $c_i : \textsc{Bint} \multimap \Xi_i\textsc{Bint}$ and c :

$\textsc{Bint} \multimap \Xi\textsc{Bint}$ (we take the coercion term of type $\textsc{Bint} \multimap \textsc{Bint}$ to be the identity). We say that t ***represents*** the numeric function $f : \mathbb{N}^n \to \mathbb{N}$ just in case for any $a_i \in \mathbb{N}$, $f\,a_1 \cdots a_n = b$ if and only if $(\boldsymbol{\lambda} x_1 \cdots x_n.t)\,(c_1\overline{a_1}) \cdots (c_n\overline{a_n})$ is equal to $c\overline{b}$ in the theory of LAL.

Computation in LAL is exactly cut-elimination. There is a cut-elimination algorithm which can be formulated in terms of reduction rules on congruence classes (defined by commuting conversions) of terms. By the *depth* of a type, we mean the maximum nesting depth of ! and § that occur in it; e.g. $\S(a \multimap !b)$ and $a \multimap b$ have depths 2 and 0 respectively. The *erasure* $er(s)$ of a term s is obtained by erasing all type annotations present in s including the additional term constructs that interpret the $\forall$-rules (one should make sure no variable bindings are broken during this process by suitable renaming of variables).

Theorem 5 (Girard) *(i) For any type T of depth d and any term t of type T, the erasure $er(t)$ of t can be reduced to a cut-free form in time proportional to $|er(t)|^{2^{d+2}}$, where $|s|$ is the size of s.*

(ii) Every PF_1 function $\mathbb{N} \to \mathbb{N}$ is represented by a closed LAL term of type $\textsc{Bint} \multimap \S^k\textsc{Bint}$, for some k. $\square$

Part (i) of the theorem says that the value $f(n)$ of a numeric function f defined by a term of type $\textsc{Bint} \multimap \S^k\textsc{Bint}$ is computable in $O(|n|^{2^{k+3}})$ steps, because cut-free erasures of terms of type $\S^k\textsc{Bint}$ uniquely determine the underlying number.

Safe and normal variables in LAL

The first step in interpreting safe recursion in LAL is to type safe and normal arguments. We have already seen that the type constructor § is a mark of iteration. Since safe arguments are the (only) places where recursive calls may occur, they should have the same type as the result type of the recursive function being defined. Thus we refer to *variables* of type $\textsc{Bint}$ in the antecedent of an LAL sequent (of a certain shape) as normal, and to those of type $\S^k\textsc{Bint}$ where $k \geq 0$ as safe. More precisely, we make the following definition:

Definition 6 Suppose the sequent

$$x_1 : \textsc{Bint}, \cdots, x_m : \textsc{Bint}, y_1 : \S^k\textsc{Bint}, \cdots, y_n : \S^k\textsc{Bint} \vdash t : \S^k\textsc{Bint}$$

is provable. If $k \geq 1$, we say that the variables x_i are ***normal*** and y_j are ***safe*** in t; if $k = 0$ we say that all variables are safe. To save writing, we shall write $t(x_1, \cdots, x_m : y_1, \cdots, y_n \mid k)$ as a shorthand for the sequent.

For example, the antecedants of the typing sequents of **strip**, S_0 and S_1 consist of only safe variables.

Remark 7 The definition of **strip** follows a general pattern which we will often use to obtain derivations with safe variables: one starts with a type T such that there exists a "projection" $p : T \multimap (\alpha \multimap \alpha)$. Then the iteration principle applied to T and functions $T \multimap T$ produces a derivation of

$$\textsc{Bint}, !(\alpha \multimap \alpha), !(\alpha \multimap \alpha) \vdash \S T,$$

which can easily be converted to one of $\textsc{Bint} \vdash \textsc{Bint}$ with the help of p. Examples of such use will include the following terms: σ-**rec**, **case** and **even-shift**.

Safe variables can always be made normal: if $t(\vec{x} : \vec{u}, y, \vec{v} \mid k)$ is provable then there exists some t' such that $t'(\vec{x}, y : \vec{u}, \vec{v} \mid k)$, which represents the same numeric function, is provable – just precompose with $\mathbf{coerc}_{\S^k}$ appropriately. We state two further useful principles:

Proposition 8 *(i)* **(Lifting)** *If $t(\vec{x} : \vec{y} \mid k)$ is provable, then for each $l > k$ there exists t' such that $t'(\vec{x} : \vec{y} \mid l)$, which represents the same numeric function, is provable.*

(ii) **(Normal contractibility)** *If $t(x_1, \cdots, x_{i-1}, x_i, x_{i+1}, x_{i+2}, \cdots, x_m : \vec{y} \mid k)$ represents $f : \mathbb{N}^m \times \mathbb{N}^n \to \mathbb{N}$ then there exists*

$$t'(x_1, \cdots, x_{i-1}, x_i, x_{i+2}, \cdots, x_m : \vec{y} \mid l)$$

representing $f' : \mathbb{N}^{m-1} \times \mathbb{N}^n \to \mathbb{N}$ such that

$$f'(\cdots, x_{i-1}, x_i, x_{i+2}, \cdots) = f(\cdots, x_{i-1}, x_i, x_i, x_{i+2}, \cdots).$$

I.e. normal variables in LAL may be contracted.

PROOF. For (i), by assumption, we have a derivation of

$$(\textsc{Bint})^m, (\S^k \textsc{Bint})^n \vdash \S^k \textsc{Bint}$$

to which we apply the $\S$-rule $(l-k)$ times to get a derivation of

$$(\S^{l-k} \textsc{Bint})^m, (\S^{l-k}(\S^k \textsc{Bint}))^n \vdash \S^{l-k}(\S^k \textsc{Bint}).$$

Finally we pre-compose with $(\mathbf{coerc}_{\S^{l-k}})^m \otimes (I_{\S^l \textsc{Bint}})^n$ to end up with the expected proof of

$$(\textsc{Bint})^m, (\S^l \textsc{Bint})^n \vdash \S^l \textsc{Bint}.$$

For (ii), we start from the given proof of $(\textsc{Bint})^m, (\S^k\textsc{Bint})^n \vdash \S^k\textsc{Bint}$ and use the §-rule with !'s for normal arguments to get $(!\textsc{Bint})^m, (\S\S^k\textsc{Bint})^n \vdash \S\S^k\textsc{Bint}$. Next we perform contraction for appropriate copies of $!\textsc{Bint}$ which results in

$$(!\textsc{Bint})^{m-1}, (\S\S^k\textsc{Bint})^n \vdash \S\S^k\textsc{Bint},$$

and apply the §-rule, this time using only §'s on the lhs, to have

$$(\S!\textsc{Bint})^{m-1}, (\S^{k+2}\textsc{Bint})^n \vdash \S^{k+2}\textsc{Bint}.$$

Finally a cut with $(\mathbf{coerc}_{\S!})^{m-1}$ gives: $(\textsc{Bint})^{m-1}, (\S^{k+2}\textsc{Bint})^n \vdash \S^{k+2}\textsc{Bint}$. In this way we obtained a requisite term for which $l = k + 2$, but it is also possible to do a little better and without the use of contraction. When one employs the diagonal map $\textsc{Bint} \multimap \S(\textsc{Bint} \otimes \textsc{Bint})$ (obtained by iteration for $X = \textsc{Bint} \otimes \textsc{Bint}$, $g_i = \mathsf{S}_i \otimes \mathsf{S}_i$ and $x = 0 \otimes 0$), the desired term will have $l = k + 1$. □

In the following section we show that safe variables in LAL cannot be similarly contracted. However, it is possible to normalize safe variables first, and then contract them as normal variables.

3 BC^-: safe recursion with non-contractible safe variables

We introduce the function algebra BC^- as the fragment of BC defined by all of BC's rules, except that safe variables that occur in safe recursion and safe composition are constrained to be (affine) *linear* in the following sense:

$$\begin{aligned} f(0, \vec{x} : \vec{y}) &= g(\vec{x} : \vec{y}) \\ f(\mathsf{S}_i(z), \vec{x} : \vec{y}) &= h_i(z, \vec{x} : f(z, \vec{x} : \vec{y})) \text{ if } \mathsf{S}_i(z) > 0 \end{aligned}$$

and

$$f(\vec{x} : \vec{y}) = h(g_1(\vec{x} :), \ldots, g_a(\vec{x} :) : h_1(\vec{x} : \vec{y_1}), \cdots, h_b(\vec{x} : \vec{y_b}))$$

where $\vec{y_1}, \cdots, \vec{y_b}$ are a partition of $\vec{y}$.

Henceforth we write $f(\vec{x} : \vec{y})$ to mean functions definable in the function algebra BC^-.

Example 9 The function $concat : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ defined by

$$\begin{aligned} concat(0, \text{`}t\text{'}) &= \text{`}t\text{'} \\ concat(\text{`}s\text{'}, \text{`}t\text{'}) &= \text{`}ts\text{'} \; s \neq \epsilon \end{aligned}$$

where s and t range over numeric strings, is definable in BC^- (and hence also in BC with one normal and one safe argument) by

$$\begin{aligned} \mathbf{concat}(0 : y) &= \pi_1^{0,1}(: y) \\ \mathbf{concat}(\mathsf{S}_i(x) : y) &= \mathsf{S}_i(\pi_2^{1,1}(x : \mathbf{concat}(x : y))) \text{ if } \mathsf{S}_i(x) > 0. \end{aligned}$$

Our first result is that relative to our notion of safety and normality in LAL, BC^- can be interpreted in LAL. Precisely, this means that for each $f(\vec{x} : \vec{y})$ in BC^- there exist a $k \geq 0$ and an LAL term-in-context $t_f(\vec{x} : \vec{y} \mid k)$ (in the sense of Definition 6) that represents f. Further the mapping $f(\vec{x} : \vec{y}) \mapsto t_f(\vec{x} : \vec{y} \mid k)$ is *compositional* in the (standard) sense that it respects the two rules of formation of BC^-. I.e. in the case of linear safe recursion (say), the interpretation of the BC^--function $f(z, \vec{x} : \vec{y})$, defined by linear safe recursion using $g(\vec{x} : \vec{y})$ as the seed and $h_i(z, \vec{x} : y)$ (where $i = 0, 1$) as step functions, is given in terms of an operation on the interpretations t_g, t_{h_0} and t_{h_1}.

Theorem 10 *Relative to our notion of safe and normal variables in LAL, there is a compositional interpretation of BC^- in LAL.*

PROOF. The successors have already been dealt with, so here we show the derivation for the predecessor. It uses iteration for $X = (\alpha \multimap \alpha) \otimes (\alpha \multimap \alpha)$. In the first component we store the outermost symbol of the binary representation, in the second - the rest (i.e. the tail). Finally all we need to do is to use 2nd projection. The whole term is:

$$\begin{aligned} n : \textsc{Bint} \vdash \boldsymbol{\Lambda}\alpha.\boldsymbol{\lambda} f_0 f_1.\S\langle\, n[X]\,!\langle\, f_0/f_0' \,\rangle(h_0)\,!\langle\, f_1/f_1' \,\rangle(h_1)/n' \,\rangle& \\ (\langle n'(I_\alpha \otimes I_\alpha)\,/n_1 \otimes n_2 \,\rangle n_2)& \end{aligned}$$

where

$$h_i = \boldsymbol{\lambda}(z_1 \otimes z_2).(f_i' \otimes \boldsymbol{\lambda} x.z_1(z_2 x)) : X \multimap X$$

For the conditional, we use iteration for $X = (\alpha \multimap \alpha) \otimes (\alpha \multimap \alpha) \multimap (\alpha \multimap \alpha)$ and $x = \boldsymbol{\lambda} x_0 \otimes x_1.x_0 : X$ with the value meaning the appropriate projection that should be applied to the two inputs. The step functions p_i are constant and return the respective projections:

$$p_i = \boldsymbol{\lambda} x^X.(\boldsymbol{\lambda}(x_0 \otimes x_1).x_i) : X \multimap X.$$

At the end, only the one corresponding to the least significant bit is used:

$$n, n_0, n_1 : \textsc{Bint} \vdash \mathbf{\Lambda}\alpha.\boldsymbol{\lambda} f_0 f_1.$$
$$\S\langle\, n[X]!(p_0)!(p_1)/n',\ n_0[\alpha]f_0 f_1/n'_0,\ n_1[\alpha]f_0 f_1/n'_1\,\rangle(n'x(n'_0 \otimes n'_1)) : \textsc{Bint}$$

Projections for safe arguments are simply the canonical projections, whereas for normal arguments they have to be composed with a suitable coercion map.

The restricted version of safe composition is interpreted as follows. By the Lifting Principle it is possible to find terms representing the functions in question with the following typings:

$$\textsc{Bint}^a, (\S^k\textsc{Bint})^b \vdash h : \S^k\textsc{Bint}$$

$$\textsc{Bint}^m \vdash g_i : \S^l\textsc{Bint} \quad i = 1, \cdots, a$$

$$\textsc{Bint}^m, (\S^{k+l}\textsc{Bint})^{n_j} \vdash h_j : \S^{k+l}\textsc{Bint} \quad j = 1, \cdots, b.$$

Then $(\bigotimes_{i=1}^a g_i) \otimes (\bigotimes_{j=1}^b h_j)$ after a few applications of the (**exch**) rule yields a derivation of:

$$(\textsc{Bint}^m)^a, (\textsc{Bint}^m)^{n_1+\cdots+n_b}, (\S^{k+l}\textsc{Bint})^{n_1+\cdots+n_b} \vdash$$
$$(\S^l\textsc{Bint})^a \otimes (\S^{k+l}\textsc{Bint})^b$$

which we cut with:

$$(\S^l\textsc{Bint})^a, (\S^{k+l}\textsc{Bint})^b \vdash \S^l h : \S^{k+l}\textsc{Bint}$$

to arrive at:

$$(\textsc{Bint}^m)^a, (\textsc{Bint}^m)^{n_1+\cdots+n_b}, (\S^{k+l}\textsc{Bint})^{n_1+\cdots+n_b} \vdash \S^{k+l}\textsc{Bint}.$$

Now contractions on appropriate normal variables result in:

$$\textsc{Bint}^m, (\S^{k+l+2}\textsc{Bint})^{n_1+\cdots+n_b} \vdash \S^{k+l+2}\textsc{Bint},$$

which corresponds to the composite function.

For safe recursion with linear safe variables we can assume (by the Lifting principle) that g, h_0, h_1 are interpreted as:

$$\textsc{Bint}^m, (\S^k\textsc{Bint})^n \vdash g : \S^k\textsc{Bint},$$

and

$$\textsc{Bint}, \textsc{Bint}^m, \S^k\textsc{Bint} \vdash h_i : \S^k\textsc{Bint}.$$

We apply the § rule appropriately to get:

$$(!\textsc{Bint})^m, (\S^{k+1}\textsc{Bint})^n \vdash g' : \S^{k+1}\textsc{Bint}$$

$$!\textsc{Bint}, (!\textsc{Bint})^m, \S^{k+1}\textsc{Bint} \vdash h_i' : \S^{k+1}\textsc{Bint}.$$

In what follows, we write g', h_i' for the curried version of these proofs. Now we use iteration for $X = (!\textsc{Bint})^m \otimes (\S^{k+1}\textsc{Bint})^n \multimap !\textsc{Bint} \otimes \S^{k+1}\textsc{Bint}$. The value is to represent $\boldsymbol{\lambda}\vec{x}\vec{y}.(z \otimes f(z, \vec{x}; \vec{y}))$ during the iteration following z. The step functions are:

$$g_i = \boldsymbol{\lambda} f\vec{x}\vec{y}.\langle f\vec{x}\vec{y}\,/z \otimes r\,\rangle((!\langle\, z/z'\,\rangle(\mathsf{S}_i z')) \otimes h_i' z\vec{x}r) : X \multimap X$$

with the initial value $x = \boldsymbol{\lambda}\vec{x}\vec{y}.((!0) \otimes g'\vec{x}\vec{y})$. A derivation of:

$$\textsc{Bint} \vdash \S X$$

results from this. Now observe that:

$$\S(A^m \otimes B^n \multimap A \otimes B) \vdash (\S A)^m \otimes (\S B)^n \multimap \S B$$

is provable and we use cut with the previous derivation to get:

$$\textsc{Bint} \vdash (\S!\textsc{Bint})^m \otimes (\S^{k+2}\textsc{Bint})^n \multimap \S^{k+2}\textsc{Bint}.$$

Finally we uncurry and apply $\mathbf{coerc}_{\S!}$ to get:

$$\textsc{Bint}^{m+1}, (\S^{k+2}\textsc{Bint})^n \vdash \S^{k+2}\textsc{Bint}.$$

□

How does our notion of safety (and normality) in LAL relate to the original notion of safety due to Bellantoni and Cook? We would argue that our notion is weaker in the sense that more numeric functions can be represented in LAL. More precisely:

Remark 11 There are numeric functions, representable as terms of solely safe variables in LAL, which cannot be defined as BC-functions with solely safe arguments. The function *concat* of Example 9 is one such. The following term-in-context, which we shall refer to as $\mathbf{concat}(: x, y \mid 0)$, represents the numeric function *concat* (as defined in Example 9):

$$n_1 : \mathrm{BINT}, n_2 : \mathrm{BINT} \vdash \mathbf{\Lambda}\alpha.\boldsymbol{\lambda} f_0 f_1.$$
$$\S\langle\, n_1[\alpha] f_0 f_1 / n_1', n_2[\alpha] f_0 f_1 / n_2' \,\rangle (\boldsymbol{\lambda} x. n_1'(n_2' x)) : \mathrm{BINT}$$

That *concat* with two safe arguments is not definable in *BC* is a consequence of the following invariance of *BC* (see [2]): for each $f(\vec{x}; \vec{y})$ in *BC* there exists a polynomial p_f such that

$$|f(\vec{x}; \vec{y})| \;\leq\; p_f(|x_1|, \cdots, |x_m|) + \max\{\, |y_j| : 1 \leq j \leq n \,\}$$

where $|x| = \lceil \log_2(x+1) \rceil$.

For the rest of the section, we examine the status of BC^- as a subalgebra of *BC*. We say that a numeric function $f(\vec{x}; \vec{y})$ together with a designation of normal arguments $\vec{x}$ and safe arguments $\vec{y}$ (e.g. a *BC*-function) is ***interpretable in LAL*** just in case there are $k \geq 0$ and LAL sequent $t_f(\vec{x} : \vec{y} \mid k)$ (in the sense of Definition 6) that represents f. Theorem 10 states that every function from BC^- is interpretable in LAL. On the other hand, Remark 11 says that there are functions that are not *BC*-definable which are interpretable in LAL. It is then natural to ask: Is BC^- the *largest* subalgebra of *BC* that is interpretable in LAL? We would argue that the answer is yes, relative to our notion of safe and normality in LAL.

First we consider the basic question: Is the definition of BC^- reasonable? Take the restrictions on the safe recursion and safe composition schemes of BC^-. Are they really necessary for our notion of (compositional) interpretation in LAL? It seems clear that they are, if we require safe variables to be (affine) linear. So the key question is whether safe variables *in LAL* are contractible i.e. whether the analogous version of Proposition 8(ii) is valid for safe variables. The answer is no; for if they were then the numeric function $\mathbf{dup}(x) = x + 2^{|x|} \cdot x$ (which "duplicates" the input e.g. $\mathbf{dup}(\text{‘10’}) = \text{‘1010’}$) would be representable by a term with a safe variable (by contracting the safe variables in $\mathbf{concat}(: x, y \mid 0)$). It then follows that the following function would be representable in LAL:

$$\begin{aligned} f(0) &= 2 \\ f(\mathsf{S}_i(x)) &= \mathbf{dup}(f(x)) \text{ if } \mathsf{S}_i(x) > 0 \end{aligned}$$

but it is easy to check that f grows exponentially.

Consider another notable difference between BC and BC^-: the step functions of the safe recursion scheme in BC^- must not have more than one safe argument. Is this inevitable? It turns out that this restriction is also necessary, for were the full safe recursion scheme interpretable in LAL (in a way that is consistent with our definition of safe and normal variables in the Logic), the definitions below:

$$\begin{aligned} f(0, y) &= y \\ f(\mathsf{S}_i(x), y) &= concat(y, f(x, y)) \text{ if } \mathsf{S}_i(x) > 0 \end{aligned}$$

which keeps y safe in LAL and then:

$$\begin{aligned} g(0, y) &= 1 \\ g(\mathsf{S}_i(x), y) &= f(y, g(x, y)) \text{ if } \mathsf{S}_i(x) > 0 \end{aligned}$$

would be translatable into LAL (we interpret *concat* as $\mathbf{concat}(: x, y \mid 0)$ in Remark 11). Now $f(x, y)$ has the effect of copying y in binary $|x|$ times and g produces (a number whose binary is) a string of $|y|^{|x|}$ 1's. (A similar restriction has appeared in a different context in [5, Remark 3.2.1].)

4 Completing BC^-

We have seen in the preceding section that augmenting BC by a version of *concat* that has two safe arguments (which we know is not BC-definable) breaks the property of PF closure . By contrast, there are functions which must be recursively defined in BC (and so have at least one normal argument), but which can be added to BC^- as functions with solely safe arguments (and interpretable in LAL), without breaking its PF closure. In this section, we enrich BC^- with new constructs that are not definable in it, though they are interpretable in LAL.

A definition-by-cases construct

As safe variables are not contractible in BC^-, it is impossible to define branching constructs of solely safe arguments such that the selector expression (x say) can still be manipulated after the choice has been made e.g. $\mathbf{cond}(: x, \mathsf{S}_0(x), \mathsf{P}(x))$. Fortunately it turns out that some such functions are interpretable in LAL, though the range of actions that may be defined on the selector expression is rather limited. Thus we introduce a pattern-matching case construct: for $K, m \geq 0$, and for strings $p_1, \cdots, p_m$ such that $|p_i| \leq K$

$$\mathbf{case}_K(: u)[p_1 : f_1 \mid \cdots \mid p_m : f_m \mid \mathsf{else} : f_{m+1}]$$

is defined to be

$$\begin{cases} f_i(u) & \text{if the least sig. } K \text{ bits} \\ & \text{of } u \text{ match } p_i \\ f_{m+1}(u) & \text{otherwise} \end{cases}$$

We stipulate that any pattern p_i shorter than K can only be matched by $u =$ 'p_i'; any non-empty string p_i of length less than K must have leftmost symbol 1; and the empty pattern be matchable only by $u = 0$. Further the actions f_j's are required to be of a certain form:

$$f_j(n) \;=\; concat(\mathsf{S}_{s_j}(\mathsf{P}^{k_j}(n)) : N_j)$$

where $N_j > 0, k_j \geq 0$ and s_j is some string, i.e. we may delete a fixed number of the least significant bits and then append some bits at that end and also some bits at the other end.

A construct related to our **case** (based on remainders modulo 2^K) is definable in BC (see [4]); in that case construct, any BC-definable function may be chosen as the action after the selection. This does not seem possible in LAL. For this reason, we believe that functions definable in the function algebra $BC^- + \mathbf{case}_K$ are a proper subalgebra of BC.

Recursion on safe *argument:* σ-**rec**

Surprisingly BC^- can be extended by a form of definition-by-recursion scheme on *safe* arguments which is interpretable in LAL, and so, the extension preserves PF closure. Certainly not all functions can be used as the corresponding step functions of the recursion scheme. Here we extract the first such scheme from LAL.

The functions that are permissible as step functions are instances of a branching construct $\mathbf{perm}_L(: u)$ that permutes the last L bits of the input u: for $L > 0$, and for strings $p_1, \cdots, p_k$ all of length L where $k \geq 0$

$$\mathbf{perm}_L(: u)[p_1 : f_1 \mid \cdots \mid p_k : f_k]$$

is equal to

$$\begin{cases} f_i(: u) & \text{if the } L \text{ least sig. bits of } u \text{ are } p_i \\ u & \text{otherwise} \end{cases}$$

Each action $f_i = \mathsf{S}_{r_i} \circ \mathsf{P}^L$ where the string r_i is a permutation of p_i. The construct returns u if $|u| < L$ or if the L least significant bits of u do not match any of the patterns. For example, let $p(: u)$ be

$$\mathbf{perm}_3(: u)[101 : \mathsf{S}_{110} \circ \mathsf{P}^3 \mid 001 : \mathsf{S}_{001} \circ \mathsf{P}^3 \mid 010 : \mathsf{S}_{100} \circ \mathsf{P}^3].$$

For $u =$ '1010', '1001', '10' we have $p(: u) =$ '1001', '1100' and '10' respectively. Note that the above construct may decrease the length of the input (e.g. $p(:$ '101') = '11'). Though it is non-size-increasing in the sense of Hofmann [6], we have not explored whether there is any real connexion yet.

The new definition-by-recursion scheme over safe arguments, which we call σ-**rec**, has the form:

$$\begin{aligned} f(\vec{x} : 0, \vec{y}) &= h(\vec{x} : \vec{y}) \\ f(\vec{x} : \mathsf{S}_i(z), \vec{y}) &= \mathbf{step}_i(: f(\vec{x} : z, \vec{y})) \text{ if } \mathsf{S}_i(z) > 0 \end{aligned}$$

the step functions $\mathbf{step}_i(: u) = p(: \mathsf{S}_{j_i}(u))$ where $j_i \in \{0, 1\}$ and $p(: x)$ is some fixed instance of the **perm** construct. Note the special case of $\mathbf{step}_i = \mathsf{S}_{j_i}$.

Example 12 We define

$$\begin{aligned} f(: 0) &= 0 \\ f(: \mathsf{S}_i(z)) &= \mathbf{step}_i(: f(: z)) \text{ if } \mathsf{S}_i(z) > 0 \end{aligned}$$

using the **perm** construct in the preceding example as $p(: n)$ and taking j_i as i. We compute $f($'110101'$)$. Since f('110') = '110', we have $f($'1101'$) = p(: \mathsf{S}_1 f($'110'$)) = p(:$ '1101'$) =$ '1011'. And so $f($'11010'$) = p(: \mathsf{S}_0 f($'1101'$)) = p(:$ '10110'$) =$ '10110'. Finally $f($'110101'$) = p(: \mathsf{S}_1 f($'11010'$)) = p(:$ '101101'$) =$ '101011'.

We define a new function algebra $BC^{\pm}$ by augmenting BC^{-} by **case** and σ-**rec**. Observe that $\mathbf{concat}(: x, y)$ is definable in $BC^{\pm}$:

$$\begin{aligned} \mathbf{concat}(: 0, y) &= \pi_1^{0,1}(: y) \\ \mathbf{concat}(: \mathsf{S}_i(x), y) &= \mathsf{S}_i(\mathbf{concat}(: x, y)) \text{ if } \mathsf{S}_i(x) > 0 \end{aligned}$$

In the following section we prove the following result:

Theorem 13 *$BC^{\pm}$ is PF sound and complete: $BC^{\pm}$-functions of the form $f(x :)$ are exactly the PF_1 functions $\mathbb{N} \to \mathbb{N}$.* $\square$

5 Proof of the PF completeness of $BC^{\pm}$

Next we show that the two new constructs **case** and σ-**rec** are interpretable in LAL. Thus we can infer that $BC^{\pm}$-definable functions are in PF. The rest of the section is devoted to a proof of the completeness direction.

$BC^{\pm}$ is PF closed

Interpreting $\mathbf{case}_{K,m}$ *in LAL*

Recall that $f_j(n) = \mathit{concat}(\mathsf{S}_{x_j}(\mathsf{P}^{k_j}(n)), N_j)$ where $N_j \neq 0$. Suppose that $x_j = y_j 0^{z_j}$ where the rightmost symbol in y_j is 1 or y_j is empty. Let $k = \max(k_1, \cdots, k_m, k_{m+1})$. We shall use iteration for $X = (P_3)^K \otimes (P_2)^{k+1} \otimes (\alpha \multimap \alpha)^k \otimes (\alpha \multimap \alpha)$ where $P_3 = \forall\alpha.\alpha \otimes \alpha \otimes \alpha \multimap \alpha$, $P_2 = \forall\alpha.\alpha \otimes \alpha \multimap \alpha$ and the three (respectively two) associated projections are π_0, π_1, π_2 (π_0 and π_1). The K-tuple is used to store K projections corresponding to the K least significant bits (π_0 or π_1) or their absence (signaled by π_2). The ith element (counting from 0) of the following $(k+1)$-tuple is π_0 if the representation of the number is shorter than $i+1$ (we assume that 0 has length 0). Otherwise it is π_1. The next k-tuple shall contain the k least significant bits of the selector value and the last component gives $\mathsf{P}^k(n)$. To preserve this meaning of the tuple the step functions must be:

$$\begin{aligned} &f_i : (\alpha \multimap \alpha) \vdash \\ &\quad \boldsymbol{\lambda}((p_0 \otimes \cdots \otimes p_{K-1}) \otimes (r_0 \otimes \cdots \otimes r_k) \otimes (b_0 \otimes \cdots \otimes b_{k-1}) \otimes p). \\ &\qquad\qquad ((\pi_i \otimes p_0 \otimes \cdots \otimes p_{K-2}) \otimes (\pi_1 \otimes r_0 \otimes \cdots \otimes r_{k-1}) \otimes \\ &\qquad\qquad\qquad (f_i \otimes b_0 \otimes \cdots \otimes b_{k-2}) \otimes (\boldsymbol{\lambda} x^{\alpha}.b_{k-1}(px))) \end{aligned}$$

with the initial value $(\pi_2)^K \otimes (\pi_0)^{k+1} \otimes (I_\alpha)^k \otimes I_\alpha$. The iteration gives a derivation of:

$$\textsc{Bint}, !(\alpha \multimap \alpha), !(\alpha \multimap \alpha) \vdash \S X.$$

To complete the translation we define a proof of

$$!(\alpha \multimap \alpha), !(\alpha \multimap \alpha), \S X \vdash \S(\alpha \multimap \alpha)$$

whose task will be to select the appropriate functions to be applied to $\mathsf{P}^k(n)$ and the k least significant bits. The derivation will be of shape

$$\begin{aligned} &\S\langle \{f_0\}/f_0', \{f_1\}/f_1', x/x' \rangle \\ &\langle x' / (p_0 \otimes \cdots \otimes p_{K-1}) \otimes (r_0 \otimes \cdots \otimes r_k) \otimes (b_0 \otimes \cdots \otimes b_{k-1}) \otimes p \rangle \mathbf{handle}. \end{aligned}$$

$\{f_i\}/f'_i$ is shorthand for $f_i/f'_{i1}, \cdots, f_i/f'_{ij_i}$ for some j_i. For simplicity each of f'_{ik}'s will be referred to as f'_i. It will become clear later how to choose sufficiently large j_i after examination of N_j's and x_j's. Suppose we are going to implement $f(n) = \mathit{concat}(\mathsf{S}_x(\mathsf{P}^h(n)), N)$ where $N \neq 0$, $x = y0^z$ and y's rightmost symbol is 0 or y is empty. Given the k least significant bits of n and the kth predecessor, the value $f(n)$ can be computed by applying

$$\begin{aligned}&\mathbf{act}^f_1 = \boldsymbol{\lambda}(b_0 \otimes \cdots \otimes b_{k-1} \otimes p).\boldsymbol{\lambda} x.\\ &f'_{x_{|x|-1}}(\cdots f'_{x_0}(b_h \cdots (b_{k-1}(p(f'_{N_0}(\cdots (f'_{N_{|N|-1}} x) \cdots)))) \cdots) \cdots) : \textsc{Action}\end{aligned}$$

to them, if $\mathsf{P}^h(n) \neq 0$ (we have $r_h = \pi_1$ then). Note that

$$\textsc{Action} = (\alpha \multimap \alpha)^{k+1} \multimap (\alpha \multimap \alpha)\,.$$

If $\mathsf{P}^h(n) = 0$, i.e. $r_h = \pi_0$, one should use

$$\mathbf{act}^f_0 = \boldsymbol{\lambda}(b_0 \otimes \cdots \otimes b_{k-1} \otimes p).\boldsymbol{\lambda} x.f'_{y_{|y|-1}}(\cdots f'_{y_0}(f'_{N_0}(\cdots (f'_{N_{|N|-1}} x) \cdots)))$$

instead. Thus the appropriate action can be selected by applying

$$\mathbf{sel}^f = \boldsymbol{\lambda}(q_0 \otimes \cdots \otimes q_k).q_h[\textsc{Action}]\,(\mathbf{act}^f_0 \otimes \mathbf{act}^f_1) : (P_2)^{k+1} \multimap \textsc{Action}$$

to $r_0 \otimes \cdots \otimes r_k$. The terms selecting the right actions should subsequently be arranged in a table **table** of type $\textsc{row}_K$, where $\textsc{row}_{i+1} = (\textsc{row}_i)^3$ for $i = 0, \cdots, K-1$ and $\textsc{row}_0 = (P_2)^{k+1} \multimap \textsc{Action}$. As many as 3^K such pairs of terms can be stored inside the table, which exhausts all combinations of the last K bits $(0, 1$ or non-existent). Obviously not all combinations make sense, because if the second bit is not available, neither is the third. In any case the K projections stored in the tuple can be used to take suitable action as follows:

$$\begin{aligned}\mathbf{handle} = p_0[\textsc{row}_0](\cdots (p_{K-2}[\textsc{row}_{K-2}](p_{K-1}[\textsc{row}_{K-1}]\,\mathbf{table}\,)) \cdots)\\ (r_0 \otimes \cdots \otimes r_k)(b_0 \otimes \cdots \otimes b_{k-1} \otimes p) : (\alpha \multimap \alpha).\end{aligned}$$

This completes the encoding of **case** as a proof of $\textsc{Bint} \vdash \textsc{Bint}$. Below we show the results for a concrete instance of **case**.

Example 14 Consider:

$$\mathbf{case}_1(: u)[1 : g_1 \mid \mathsf{else} : g_2]$$

where

$$\begin{aligned} g_1(n) &= concat(\mathsf{S}_0(\mathsf{P}(n)), 1) \\ g_2(n) &= concat(\mathsf{S}_1(\mathsf{P}^2(n)), 1) \,. \end{aligned}$$

The auxiliary terms needed for the translation are:

$$\begin{aligned} \mathbf{act}_1^{g_1} &= \boldsymbol{\lambda}(b_0 \otimes b_1 \otimes p).\boldsymbol{\lambda}x.f_0(b_1(p(f_1 x))) \\ \mathbf{act}_0^{g_1} &= \boldsymbol{\lambda}(b_0 \otimes b_1 \otimes p).\boldsymbol{\lambda}x.f_1 x \\ \mathbf{sel}^{g_1} &= \boldsymbol{\lambda}(q_0 \otimes q_1 \otimes q_2).q_1[\text{ACTION}](\mathbf{act}_0^{g_1} \otimes \mathbf{act}_1^{g_1}) \\ \\ \mathbf{act}_1^{g_2} &= \boldsymbol{\lambda}(b_0 \otimes b_1 \otimes p).\boldsymbol{\lambda}x.f_1(p(f_1 x)) \\ \mathbf{act}_1^{g_2} &= \boldsymbol{\lambda}(b_0 \otimes b_1 \otimes p).\boldsymbol{\lambda}x.f_1(f_1 x) \\ \mathbf{sel}^{g_2} &= \boldsymbol{\lambda}(q_0 \otimes q_1 \otimes q_2).q_2[\text{ACTION}](\mathbf{act}_0^{g_2} \otimes \mathbf{act}_1^{g_2}) \\ \\ \mathbf{table} &= \mathbf{sel}^{g_2} \otimes \mathbf{sel}^{g_1} \otimes \mathbf{sel}^{g_2} \\ \mathbf{handle} &= (p_0[\text{ROW}_0]\ \mathbf{table})(r_0 \otimes r_1 \otimes r_2)\ (b_0 \otimes b_1 \otimes p) \,. \end{aligned}$$

*Interpreting σ-**rec** in LAL*

To express σ-**rec** in LAL we use iteration for

$$X = (P_3)^{L-1} \otimes P_2 \otimes (\alpha \multimap \alpha)^{L-1} \otimes (\alpha \multimap \alpha).$$

The first $L-1$ elements of the tuple are projections $p_1, \cdots, p_{L-1}$ of type $P_3 = \forall \alpha.\alpha \otimes \alpha \otimes \alpha \multimap \alpha$ which correspond to the $L-1$ least significant bits of the interim iteration result. The projections are: π_0, π_1, π_2, where π_2 means that the corresponding bit is not available. The following projection r of type P_2 indicates whether $\mathsf{P}^{L-1}(u)$ is zero. The next $L-1$ elements $b_1, \cdots, b_{L-1}$ are the last $L-1$ bits of u in the form of imported f_0's and f_1's 'filtered' through the interface. The last component p contains $\mathsf{P}^{L-1}(u)$ as a term of type $(\alpha \multimap \alpha)$.

During iteration the step functions will add f_i and π_i to the current tuple as b_0 and p_0 respectively. The projections will be used to select the right action to be taken with respect to $b_0, \cdots, b_{L-1}, p_0, \cdots, p_{L-1}$ and p. The bits and projections should be permuted and the most significant bit appended to p. Projections are needed twice so have to be copied by:

$$\mathbf{copy}\, p = p[P_3 \otimes P_3]((\pi_0 \otimes \pi_0) \otimes (\pi_1 \otimes \pi_1) \otimes (\pi_2 \otimes \pi_2)).$$

The actions will have type $\textsc{Action}$:

$$(P_3)^L \otimes (\alpha \multimap \alpha)^L \otimes (\alpha \multimap \alpha) \multimap \\ (P_3)^{L-1} \otimes P_2 \otimes (\alpha \multimap \alpha)^{L-1} \otimes (\alpha \multimap \alpha).$$

For instance, if 100 is to be converted into 010 ($L = 3$) and the projections and bits are stored in $i = ((p_0 \otimes p_1 \otimes p_2) \otimes (b_0 \otimes b_1 \otimes b_2) \otimes p)$,

$$\mathbf{res}_1 = (p_0 \otimes p_2) \otimes \pi_1 \otimes (b_0 \otimes b_2) \otimes \boldsymbol{\lambda} x.b_1(px)$$

will be the desired outcome, when $\mathsf{P}^{L-1}(u) \neq 0$ ($r = \pi_1$). Otherwise it should be

$$\mathbf{res}_0 = (p_0 \otimes p_2) \otimes \pi_0 \otimes (b_0 \otimes b_2) \otimes p,$$

i.e. the zero bit is not appended. In general, r can be used to select the corresponding operation as follows

$$\boldsymbol{\lambda} r.r[\textsc{Action}]((\boldsymbol{\lambda} i.\mathbf{res}_0) \otimes (\boldsymbol{\lambda} i.\mathbf{res}_1)) : P_2 \multimap \textsc{Action}$$

Actions depending on r will be arranged in **table** : $\textsc{row}_L$, where $\textsc{row}_{i+1} = (\textsc{row}_i)^3$ for $i = 0, \cdots, L-1$ and $\textsc{row}_0 = P_2 \multimap \textsc{Action}$. Assuming the copied projections are $p'_0, \cdots, p'_{L-1}$ the desired action can be chosen by:

$$\mathbf{select} = p'_0[\textsc{row}_0](p'_1[\textsc{row}_1] \cdots (p'_{L-1}[\textsc{row}_{L-1}]\, \mathbf{table}) \cdots)$$

which is of type $P_2 \multimap \textsc{Action}$. Finally, the step function can be defined as

$$\boldsymbol{\lambda}((\bigotimes p_k) \otimes r \otimes (\bigotimes b_l) \otimes p).\langle \bigotimes(\mathbf{copy}\, p_k) \,/ \bigotimes(p'_k \otimes p''_k) \rangle \\ \mathbf{select}(r)(\pi_i \otimes (\bigotimes p''_k)) \otimes (f'_i \otimes (\bigotimes b_l)) \otimes p) : X \multimap X$$

and the iteration principle for $\Box = \S$ and the initial value $x : X \vdash x : X$ gives a derivation of:

$$\textsc{Bint}, \S X, !(\alpha \multimap \alpha), !(\alpha \multimap \alpha) \vdash \S X.$$

To complete the definition we have to extract the result from X with

$$\boldsymbol{\lambda}((\bigotimes p_k) \otimes r \otimes (b_1 \otimes \cdots \otimes b_{L-1}) \otimes p).\boldsymbol{\lambda} x^\alpha.b_1(\cdots(b_{L-1}(px))\cdots)$$

of type $X \multimap (\alpha \multimap \alpha)$. Then after applying λ-abstraction twice, we get a derivation of

$$\textsc{Bint}, \S X \vdash !(\alpha \multimap \alpha) \multimap !(\alpha \multimap \alpha) \multimap \S(\alpha \multimap \alpha).$$

As the initial value of the recursion is $g(\vec{x} : \vec{y})$, we have to cut the corresponding derivation t_g of

$$\textsc{Bint}^m, (\S^k\textsc{Bint})^n \vdash \S^k\textsc{Bint}$$

with a proof of $\textsc{Bint} \vdash \S X$ which initializes X. For the latter we use iteration for X with $x = (\pi_3)^{L-1} \otimes \pi_0 \otimes (I_\alpha)^{L-1} \otimes I_\alpha$ and step functions:

$$\begin{aligned} f_i : (\alpha \multimap \alpha) \vdash \boldsymbol{\lambda}((p_0 \otimes \cdots \otimes p_{L-2}) \otimes r \otimes (b_0 \otimes \cdots \otimes b_{L-2}) \otimes p). \\ (\pi_i \otimes p_0 \otimes \cdots \otimes p_{L-3}) \otimes p_{L-2}[P_2](\pi_1 \otimes \pi_1 \otimes \pi_0) \otimes \\ (f_i \otimes b_0 \otimes \cdots \otimes b_{L-3} \otimes (\boldsymbol{\lambda}x.b_{L-2}(px))) \end{aligned}$$

to define the required derivation of $\textsc{Bint} \vdash \S X$. After cutting with

$$\textsc{Bint}, \S X \vdash !(\alpha \multimap \alpha) \multimap !(\alpha \multimap \alpha) \multimap \S(\alpha \multimap \alpha).$$

we get a proof of :

$$\textsc{Bint}, \textsc{Bint} \vdash !(\alpha \multimap \alpha) \multimap !(\alpha \multimap \alpha) \multimap \S(\alpha \multimap \alpha),$$

which defines a proof of

$$\S^k\textsc{Bint}, \S^k\textsc{Bint} \vdash \S^k\textsc{Bint}$$

in the obvious way. Now we use the cut rule for the second copy of $\textsc{Bint}$ on the left with t_g, to get:

$$(\textsc{Bint})^m, (\S^k\textsc{Bint})^{n+1} \vdash \S^k\textsc{Bint}.$$

All PF_1 functions are representable in $BC^\pm$

Roversi's completeness proof [10] for LAL cannot be repeated in our setting, because it uses a carefully selected type for coding machine configurations which is related to but different from $\textsc{Bint}$. Handley's proof [4] for *BC* does use $\mathbb{N}$ to encode configurations, but it relies critically on the contractibility of safe arguments. So does Bellantoni and Cook's which demonstrates how Cobham-style definitions can be translated into *BC*. Here we propose a new way of encoding polytime Turing machines using the constructs of $BC^\pm$.

Polynomials

First we need a way of representing polynomials for defining the polytime clock. For each polynomial p we define $f_p(n :) = \underbrace{\text{‘}1 \cdots 1\text{’}}_{p(|n|)}$ by induction on (representations of) polynomials with one indeterminate x:

- $f_1(n :) = \mathsf{S}_1(0)$
- $f_{p_1+p_2}(n :) = \mathbf{concat}(f_{p_1}(n :) : f_{p_2}(n :))$
- For $f_{x \cdot p}$ we first define an auxiliary function $f'(n_1, n_2 :) = \underbrace{\text{‘}1 \cdots 1\text{’}}_{|n_1| \cdot p(|n_2|)}$; the desired function $f_{x \cdot p}$ is then obtained by contracting the two normal arguments of f'. f' is defined as follows:

$$\begin{aligned} f'(0, x :) &= 0 \\ f'(\mathsf{S}_i(z), x :) &= \mathbf{concat}(f_p(x :) : f'(z, x :)) \text{ if } \mathsf{S}_i(z) > 0 \end{aligned}$$

Configurations

Fix a Turing machine. Suppose its symbols include $0, 1$ and the blank symbol $\sqcup$. Call the initial state q_0. We require the input to be placed to the right of the head with the most significant bit below the head. The same convention applies to the output, and we further require that the rest of the tape be blank at the end of the computation.

We pick an odd number L which is sufficiently large, and decree that symbols and states are strings of L bits beginning with 1 with the rest of the string containing $\frac{L-1}{2}$ zeros and $\frac{L-1}{2}$ ones. Clearly the code of each symbol can be converted into that of another by permutation. The same is true of the states. We write $\ulcorner\theta\urcorner$ for the code of θ where θ ranges over symbols and states, and we require the function $\ulcorner-\urcorner$ to be injective.

We code the configuration of the machine by placing the code of the current state followed by L zeros and L ones between (codes for) the left and right part of the tape in order to distinguish the position of the head. If the current state has code $s_0 \cdots s_{L-1}$ and the symbol under the head has code $r_0 \cdots r_{L-1}$, the configuration is coded by the string

$$\cdots l_0 \cdots l_{L-1} s_0 \cdots s_{L-1} \underbrace{0 \cdots 0}_{L} \underbrace{1 \cdots 1}_{L} \overline{r_0 \cdots r_{L-1}} r_L \cdots r_{2L-1} \cdots$$

where $\cdots l_0 \cdots l_{L-1}$ refers to the string that codes the part of the tape which is to the left of the head; the string which consists of the overlined bits code the symbol under the head.

Transition

To see that each possible transition corresponds to a permutation of the above $6L$ bits, we give the code of the new configuration if the next state is coded by $s'_0 \cdots s'_{L-1}$, $r'_0 \cdots r'_{L-1}$ is written to the tape and the head moves to the right:

$$\cdots l_0 \cdots l_{L-1} r'_0 \cdots r'_{L-1} s'_0 \cdots s'_{L-1} \underbrace{0 \cdots 0}_{L} \underbrace{1 \cdots 1}_{L} \overline{r_L \cdots r_{2L-1}} \cdots .$$

or to the left:

$$\cdots s'_0 \cdots s'_{L-1} \underbrace{0 \cdots 0}_{L} \underbrace{1 \cdots 1}_{L} \overline{l_0 \cdots l_{L-1}} r'_0 \cdots r'_{L-1} r_L \cdots r_{2L-1} \cdots$$

To mark the completion of a transition, we change the $2L$ bits $\underbrace{0 \cdots 0}_{L} \underbrace{1 \cdots 1}_{L}$ to $\underbrace{1 \cdots 1}_{L} \underbrace{0 \cdots 0}_{L}$. Next we can use σ-**rec** with $\mathbf{perm}_{2L}$ just to change $\underbrace{1 \cdots 1}_{L} \underbrace{0 \cdots 0}_{L}$ back to $\underbrace{0 \cdots 0}_{L} \underbrace{1 \cdots 1}_{L}$ to enable the next step.

Therefore the transition can be performed by a function $\mathbf{transition}(: n)$ using σ-**rec** with $\mathbf{perm}_{6L}$ first and then $\mathbf{perm}_{2L}$. That way the type of the encoded configurations is safe, and so, eligible for iteration from an initial configuration $\mathbf{init}(n :)$ by $\mathbf{iterate}(f_p(n :) : \mathbf{init}(n :))$, where

$$\begin{aligned} \mathbf{iterate}(0 : y) &= y \\ \mathbf{iterate}(\mathsf{S}_i(n) : y) &= \mathbf{transition}(: \mathbf{iterate}(n : y)) \text{ if } \mathsf{S}_i(n) > 0 \end{aligned}$$

Initial configuration

Because **perm** cannot increase the size of the tape, the initial tape has to be long enough for the whole computation, the duration of which is controlled by a fixed polynomial clock. We simply supply that very number of cells to the left and right of the tape which will suffice for the computation.

The code of a number is easily given by

$$\begin{aligned} \mathbf{rep}(0 :) &= 0 \\ \mathbf{rep}(\mathsf{S}_i(x) :) &= \mathsf{S}_{\ulcorner i \urcorner}(\mathbf{rep}(x :)) \text{ if } \mathsf{S}_i(x) > 0 \end{aligned}$$

For $x \geq 0$, $|x|$ blank cells are encoded by

$$\begin{aligned} \mathbf{bl}(0 :) &= 0 \\ \mathbf{bl}(\mathsf{S}_i(x) :) &= \mathsf{S}_{\ulcorner \sqcup \urcorner}(\mathbf{bl}(x :)) \text{ if } \mathsf{S}_i(x) > 0 \end{aligned}$$

The initial configuration for input n is thus

$$\begin{aligned}\mathbf{init}(n:) = \mathbf{concat}(&\mathbf{bl}(f_p(n:):): \\ &\mathbf{concat}(\mathbf{concat}(\mathbf{rep}(n:):\mathsf{S}_{1^L0^L}(\ulcorner q_0\urcorner)):\mathbf{bl}(f_p(n:):))).\end{aligned}$$

Extraction

To convert the final configuration to the numeric output, we use an auxiliary function $\mathbf{aux}(n:)$. It attaches L flag bits first which it later uses to recognize the moments in which representations of states and characterized are being processed. All these representations are subsequently erased except for encodings of 0 and 1 which are replaced with 0 and 1 respectively:

$$\begin{aligned}\mathbf{aux}(0:) &= 10^{L-1}\\ \mathbf{aux}(\mathsf{S}_i(x):) &= \mathbf{case}_{2L}(\mathsf{S}_i(\mathbf{aux}(x:))[Q] \text{ if } \mathsf{S}_i(x) > 0\end{aligned}$$

where $Q =$

$$\begin{array}{lllllll} [\ 10^{L-1}\ulcorner\sqcup\urcorner & : & \mathsf{P}^L & \mid 10^{L-1}\ulcorner 0\urcorner & : & \mathsf{S}_{0^{L-1}10}\mathsf{P}^{2L} & \mid \\ \ \ 10^{L-1}\ulcorner 1\urcorner & : & \mathsf{S}_{0^{L-1}11}\mathsf{P}^{2L} & \mid 10^{L-1}\ulcorner q\urcorner & : & \mathsf{P}^L & \mid \\ \ \ 10^{L-1}0^L & : & \mathsf{P}^L & \mid 10^{L-1}1^L & : & \mathsf{P}^L & \mid \\ \ \ \mathsf{else} & : & I_{\mathbb{N}} &] & & & \end{array}$$

and q stands for any state. Finally we define

$$\mathbf{extract}(n:) = \mathsf{P}^L(\mathbf{aux}(n:))$$

by erasing the flag bits.

The whole computation

Every numeric function $g(n)$ computable by a Turing machine running in time $p(|n|)$ can be simulated by:

$$\mathbf{extract}(\mathbf{iterate}(f_p(n:):\mathbf{init}(n:)):)$$

which wraps up our proof of $BC^{\pm}$'s PF completeness. Note that this also gives a new proof of LAL's PF completeness.

6 Another PF completion of BC^-

We have seen that $BC^{\pm}$ is PF complete. There are other ways to complete BC^-.

Another function we can safely add to BC^- is **e-shift**$(: n)$ which “shifts even bits to the left”:

$$\begin{aligned} \textbf{e-shift}(:\, `s_{2n+1}\cdots s_1 s_0\text{'}) &= `s_{2n}s_{2n+1}\cdots s_4 s_5 s_2 s_3 s_0 s_1 0\text{'} \ \text{ if } s_{2n} \neq 0 \\ \textbf{e-shift}(:\, `s_{2n+1}\cdots s_1 s_0\text{'}) &= `s_{2n+1}\cdots s_4 s_5 s_2 s_3 s_0 s_1 0\text{'} \ \text{ if } s_{2n} = 0 \\ \textbf{e-shift}(:\, `s_{2n}\cdots s_1 s_0\text{'}) &= `s_{2n}0 s_{2n-2}s_{2n-1}\cdots s_2 s_3 s_0 s_1 0\text{'} \end{aligned}$$

For instance **e-shift** : ‘$1\underline{1}$’ $\mapsto$ ‘$\underline{1}10$’, ‘$\underline{1}1\underline{1}$’ $\mapsto$ ‘$\underline{1}0\underline{1}10$’ and ‘$\underline{1}0\underline{1}1\underline{1}$’ $\mapsto$ ‘$\underline{1}0\underline{1}0\underline{1}10$’. (We underline the even bits to indicate their movements.) Note that **e-shift** does not violate the invariance in Remark 11, but is undefinable in BC. Next we show that the function algebra $BC^- + \textbf{case} + \textbf{e-shift}$ is also PF complete.

Representation of **e-shift** *in LAL*

Here we begin our definition of **e-shift** : $\textsc{Bint} \multimap \textsc{Bint}$. First we define a function which reverses a binary list in order to use it later. This is not a numeric function in the strict sense as we do not care about the leading zeros. A list $[1, 0, 1, 0]$ is denoted by the term:

$$\boldsymbol{\Lambda}\alpha.\boldsymbol{\lambda} f_0 f_1.\S\langle\, f_0/f_{00}, f_0/f_{01}, f_1/f_{10}, f_1/f_{11}\,\rangle(\boldsymbol{\lambda} x.f_{11}(f_{01}(f_{10}(f_{00}x)))))$$

To reverse a list we apply iteration for $X = (\alpha \multimap \alpha)$ starting from I with step functions:

$$f_i : !(\alpha \multimap \alpha) \vdash \boldsymbol{\lambda} f^{(\alpha\multimap\alpha)}.\boldsymbol{\lambda} x.f(f_i x)) : (\alpha \multimap \alpha) \multimap (\alpha \multimap \alpha).$$

We name the resultant term of type $\textsc{Bint} \multimap \textsc{Bint}$ **rev**.

Shift

The function to be defined in this section will take a list and shift every other element to the left starting from the last:

$$\begin{aligned} b_{2k}\cdots b_2 b_1 b_0 &\mapsto b_{2k}0 b_{2k-2}\cdots b_0 b_1 0 \\ b_{2k+1}b_{2k}\cdots b_2 b_1 b_0 &\mapsto b_{2k}b_{2k+1}b_{2k-2}\cdots b_0 b_1 0 \end{aligned}$$

We take advantage of the iteration principle for $P_2 \otimes (\alpha \multimap \alpha) \otimes (\alpha \multimap \alpha)$ with intended meaning:

$$\pi_0 \otimes I \otimes \boldsymbol{\lambda} x.b_{2k}(b_{2k+1}(\cdots b_0(b_1(f_0 x))))$$

after processing step functions for $b_0, \cdots, b_{2k+1}$ and

$$\pi_1 \otimes b_{2k} \otimes \boldsymbol{\lambda} x.b_{2k-1}(b_{2k-2}(\cdots b_0(b_1(f_0 x))))$$

if the pattern processed so far is $b_0, \cdots, b_{2k}$. The first component of the triple is a projection indicating the case we have to deal with. The initial value is thus $\pi_0 \otimes I_\alpha \otimes f'_0 : (\alpha \multimap \alpha) \otimes (\alpha \multimap \alpha)$ as the next step should following the pattern b_0. In that case the current bit f and the second and third components of the triple must be processed by:

$$F_1 = \boldsymbol{\lambda}(f \otimes c_1 \otimes c_2).f \otimes c_2 : (\alpha \multimap \alpha)^3 \multimap (\alpha \multimap \alpha)^2.$$

In the other case

$$F_2 = \boldsymbol{\lambda}(f \otimes c_1 \otimes c_2).I_\alpha \otimes \boldsymbol{\lambda} x.c_1(f(c_2 x)) : (\alpha \multimap \alpha)^3 \multimap (\alpha \multimap \alpha)^2$$

should be used. The projection from the triple is used to pick one of the two above and the other projection must be put in place of the old one. We can get copies of the current projection p and the opposite one as $p' = p[P_2 \otimes P_2](\pi_0 \otimes \pi_1)(\pi_1 \otimes \pi_0)$. Finally we can define step functions:

$$\begin{aligned} f_i : (\alpha \multimap \alpha) \vdash \boldsymbol{\lambda} x.\langle x\,/p \otimes c_1 \otimes c_2\,\rangle\langle p'\,/p_1 \otimes p_2\,\rangle \\ (p_2 \otimes (p_1[F](F_0 \otimes F_1))(f_i \otimes c_1 \otimes c_2)) : X \multimap X \end{aligned}$$

and employ the iteration principle to obtain a derivation of:

$$\textsc{Bint}, !(\alpha \multimap \alpha), !(\alpha \multimap \alpha) \vdash \S X.$$

To complete the definition, we use the second and third elements from X. If the list was of even length, all we need is the third component so

$$H_1 = \boldsymbol{\lambda}(c_1 \otimes c_2).c_2 : H = (\alpha \multimap \alpha)^2 \multimap (\alpha \multimap \alpha)$$

applied to them yields the correct result. Otherwise, the two components $c_1, c_2 : (\alpha \multimap \alpha)$ need to be transformed by

$$H_2 = \boldsymbol{\lambda}(c_1 \otimes c_2).\boldsymbol{\lambda} x.c_1(f_0(c_2 x)) : (\alpha \multimap \alpha)^2 \multimap (\alpha \multimap \alpha).$$

Which is the case, tells us the first component - an appropriate projection. To end the definition 'cut' the previous derivation with:

$$f_0 : !(\alpha \multimap \alpha), x : \S X \vdash \\ \S\langle\, f_0/f_0', x/p \otimes c_1 \otimes c_2\,\rangle((p[H](H_0 \otimes H_1))(c_1 \otimes c_2)) : \S(\alpha \multimap \alpha)\,,$$

contract the two copies corresponding to f_0 and use lambda and type abstractions to get a term **shift** of type $\textsc{Bint} \multimap \textsc{Bint}$.

Even-bits-shift

The term **shift** shifts the bits starting from the most significant one, but now we would like the procedure to begin with the least significant bit. Therefore we reverse the representation first, apply **shift**, reverse it again and delete the leading zeros that may have arisen:

$$n : \textsc{Bint} \vdash \textbf{e-shift} = \textbf{strip}(\textbf{rev}(\textbf{shift}(\textbf{rev}\, n))) : \textsc{Bint}.$$

Encoding polytime Turing Machines

Turing Machines

W.l.o.g. we assume that both states and symbols can be represented by strings of 0's and 1's of length L.

For technical reasons, we assume that $L > 2$, L is even and 0, 1 and the blank symbol are represented by $1\underbrace{0\cdots 0}_{L-2}1$, $1\underbrace{1\cdots 1}_{L-2}1$ and L zeros respectively. Besides, all other symbols and representations of states are strings of form $1\underbrace{\cdots}_{L-2}1$ (the two surrounding 1's are important).

Tape and input

We encode the contents of the tape and the state as a single natural number. The L least significant bits of the number correspond to the state. Suppose the head is above a symbol $r_0 \cdots r_{L-1}$ and r_i's start the right part of tape. We represent (say)

$$l_5 l_4 l_3 l_2 l_1 l_0 r_0 r_1 r_2 r_3 r_4$$

as the binary list:

$$r_0 l_0 r_1 l_1 r_2 l_2 r_3 l_3 r_4 l_4 0 l_5.$$

If one part of the tape is longer than the other, we use 0's to make up for the shortfall. This is the reason why we represent the blank symbol as L zeros - the padding is consistent with the fact that the tape is potentially infinite. Moreover, due to the requirement that blank symbols be strings of zeros, the convention that natural numbers must begin with 1 as terms of type BINT will not cause any loss of information on the tape.

When the state is appended at the front we get:

$$s_0 \cdots s_{L-1} r_0 l_0 r_1 l_1 r_2 l_2 r_3 l_3 r_4 l_4 0 l_5 \cdots$$

We stipulate that the input and output strings be placed to the right of the head, and the head of the list we use to represent the tape is to be the leftmost symbol of the string corresponding to the symbol being scanned by the head of the Turing machine. Moreover, we want the rest of the tape to be blank. Clearly, these conventions do not matter for polynomial time computability.

Transition

In order for a move to be made, the state and the head symbol $r_0 \cdots r_{L-1}$ have to be examined. This means that the first $3L$ least significant bits of our representation are enough to decide the next step. First we use $\mathbf{case}_{3L}$ to replace the bits corresponding to the current state with the bits representing the new state $s'_0 \cdots s'_{L-1}$. If the head is to move to the right we append one additional 0 to the representation. The tape after this step is either:

$$\mathbf{tempL} = s'_0 \cdots s'_{L-1} r_0 l_0 r_1 l_1 r_2 l_2 r_3 l_3 r_4 l_4 0 l_5 \cdots$$

or

$$\mathbf{tempR} = 0 s'_0 \cdots s'_{L-1} r_0 l_0 r_1 l_1 r_2 l_2 r_3 l_3 r_4 l_4 0 l_5 \cdots.$$

What modifications must be made to the tape becomes clear when we examine how the representation of the tape changes when the head moves to the left or right.

First we consider moves by one field containing 0 or 1, note however that to simulate what the Turing machine does, we will have to repeat it L times so that the head scans the first bit of the representation of the new true cell.

The table shows how the representation should change, if the head changes its position.

	tape	representation + state
L	$l_5l_4l_3l_2l_1\overline{l_0}r_0r_1r_2r_3r_4$	$s'_0\cdots s'_{L-1}l_0l_1r_0l_2r_1l_3r_2l_4r_3l_5r_4$
R	$l_5l_4l_3l_2l_1l_0r_0\overline{r_1}r_2r_3r_4$	$0s'_0\cdots s'_{L-1}r_1r_0r_2l_0r_3l_1r_4l_20l_30l_40l_5$

Observe that the change to the tape resembles the effect of $\mathbf{e\text{-}shift}^2$. For R this is due to the addition of the additional 0 bit. Note that we can tell the current direction by looking at the 4th bit of the representation, because $s'_0 = 1$ by convention.

$$\begin{aligned}
&\mathbf{e\text{-}shift}^2(:\mathbf{tempL}) = 0s'_10s'_3s'_0\cdots s'_{L-1}s'_{L-4}l_0s'_{L-2}l_1r_0l_2r_1l_3r_2l_4r_3l_5r_4\\
&\mathbf{e\text{-}shift}^2(:\mathbf{tempR}) =\\
&\qquad 0s'_00s'_20s'_4s'_1\cdots s_{L-2}s'_{L-5}r_0s_{L-3}r_1s'_{L-1}r_2l_0r_3l_1r_4l_20l_30l_40l_5
\end{aligned}$$

Note that the above differs from what should be modelled in the first $L+5$ bits. Hence the correction can be made using $\mathbf{case}_{L+5}$. Let us call this correcting term $\mathbf{correct}(:u)$. The move of the head is then mimicked by applying $\mathbf{correct}(:\mathbf{e\text{-}shift}^2(:u))$ to the representation L times. We must not forget to erase the extra bit added for the move to the right. That case can be recognized by looking at the first bit. If it is 0, we should remove it (just use $\mathbf{case}_1$). What results is a term $\mathbf{transition}(:u)$ which can be iterated $|n|$ times from some initial value $\mathbf{init}(:\mathbf{n})$ by $\mathbf{iterate}(n:\mathbf{init}(n:))$.

Input conversion

What is the value of $\mathbf{init}(n:)$? The function $\mathbf{bint2tape}(n:)$ is defined by safe recursion:

$$\begin{aligned}
\mathbf{bint2tape}(0:) &= 0\\
\mathbf{bint2tape}(\mathsf{S}_0(x):) &= \mathsf{S}_{1\underbrace{0\cdots0}_{2L-3}10}(:\mathbf{bint2tape}(x:)) \text{ if } \mathsf{S}_i(x) > 0\\
\mathbf{bint2tape}(\mathsf{S}_1(x):) &= \mathsf{S}_{\underbrace{10\cdots10}_{2L}}(:\mathbf{bint2tape}(x:))
\end{aligned}$$

Assuming $s_{L-1}\cdots s_0$ represents the initial state we set:

$$\mathbf{init}(:n) = \mathbf{concat}(‘s_0\cdots s_{L-1}’:\mathbf{bint2tape}(n:)).$$

Extraction of the result

First, to strip the first L bits describing the state, we can use P^L. After that, the bits corresponding to the right tape should be extracted i.e. every other bit ought to be ignored. The following function does that by attaching a flag bit to the intermediate result. When the flag is 1 we append the currently processed bit and change the flag to 0. On flag 0, we only replace it with 1 thus ignoring the corresponding bit. Using this idea we define the function $\mathbf{aux}(n:)$ first:

$$\begin{aligned} \mathbf{aux}(0:) &= 1 \\ \mathbf{aux}(\mathsf{S}_i(x):) &= \mathbf{case}_1(:\mathbf{aux}(x:)) \\ & \quad 0 \qquad\qquad \mathsf{S}_1(\mathsf{P}(\mathbf{aux}(x:))) \\ & \quad 1 \qquad\qquad \mathsf{S}_0(\mathsf{S}_i(\mathsf{P}(\mathbf{aux}(x:)))) \end{aligned}$$

and erase the flag to have the relevant bits:

$$\mathbf{extractR}(u:) = \mathsf{P}(\mathbf{aux}(u:)).$$

Now recall that 0 and 1 are represented by $1\underbrace{0\cdots0}_{L-2}1$ and $1\underbrace{1\cdots1}_{L-2}1$ respectively. That is when scanning the tape, after reading the initial 1 starting the representation of 0 or 1, the next bit will already reveal the represented bit, and $L-2$ further symbols can be read without action. As previously we can use flags to simulate state. This time we have L states depending on the symbol currently processed, so the flag will consist of several (say B) bits. The intended behaviour can be programmed by safe recursion with appropriate $\mathbf{case}_B$ as the step function. At the end we must not forget to erase the bits using P^B to get $\mathbf{extract}(u:)$.

7 Further directions

A different approach to completing BC^- is to embed it in a type theory and add suitable higher-type operators. Hofmann [5] has shown how BC can be embedded in the typed system SLR. The subsystem of SLR that corresponds to BC^- is what we call SLR^-, which is SLR less the axiom (S-AX), so that N is non-duplicable. We show in [9] that SLR^- too is PF complete: closed SLR^--terms of type $\Box\mathsf{N} \to \mathsf{N}$ define exactly the PF functions $\mathbb{N} \to \mathbb{N}$.

$$\begin{tikzcd}[arrows=dash]
BC \arrow[dr] & SLR^{-} \arrow[d] & BC^{\pm} \arrow[dl] \\
& BC^{-} &
\end{tikzcd}$$

Is safe recursion interpretable in LAL? Our answer is yes, but up to a point, as defined by BC^{-}. In exploring PF completions of BC^{-}, we have gone to LAL in search of constructs (admittedly so low-level as to be good only for programming Turing machines) interpretable by LAL terms of safe arguments, which can then be used as step functions. The supply of such functions seems unlimited, and as the encoding of σ-**rec** and of **e-shift** shows, there are many LAL coding tricks one can exploit. For future work, we would like to understand how our notion of safe and normal variables in LAL may be extended to higher types. If successful, we can then investigate higher-order extensions of BC^{-}, using our methodology of interpretability in LAL to obtain PF-sound systems. (One specific direction is to find a new proof of the PF closure of SLR^{-} by interpreting it in LAL.)

Acknowledgements

Murawski is supported by a University of Oxford Scatcherd European Scholarship and a British Overseas Research Students Award 98032135.

References

[1] A. Asperti. Light affine logic. In *Proceedings of 13th IEEE Annual Symposium on Logic in Computer Science 1998*. IEEE Computer Society, 1998.

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

[3] J.-Y. Girard. Light linear logic. *Information and Computation*, 143:175–204, 1998.

[4] W. G. Handley. Bellantoni and Cook's characterization of polynomial time with some related results. In S. Wainer's Marktoberdorf'97 Lecture Notes, 1997.

[5] M. Hofmann. Type systems for polynomial-time computation. Technische Universität Darmstadt, Habilitationsschrift, 1998.

[6] M. Hofmann. Linear types and non-size-increasing polynomial time computation. In *Proceedings of 14th IEEE Annual Symposium on Logic in Computer Science*. IEEE Computer Society, 1999.

[7] T. W. Koh and C.-H. L. Ong. Explicit substitution internal languages for autonomous and $*$-autonomous categories. *Electronic Notes in Theoretical Computer Science*, 29, 1999. Proceedings of the 8th Conf. on Category Theory and Computer Science 1999, 30 pp.

[8] A. S. Murawski. On semantic and type-theoretic aspects of polynomial-time computability. D.Phil. thesis, Oxford University Computing Laboratory, 2001.

[9] A. S. Murawski and C.-H. L. Ong. SLR^{-} is PTIME complete. Unpublished note. Ftp-able from Ong's home page, 2000.

[10] L. Roversi. A PTIME completeness proof for light logics. In *Proceedings of CSL'99*, volume 1683 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.