

# How Light Is Safe Recursion?

## Compositional Translations Between Languages of Polynomial Time

Peter Møller Neergaard<sup>\*</sup>  
Computer Science Department  
Brandeis University  
Waltham, MA 02454, USA  
turtle@achilles.linearity.org

Harry G. Mairson<sup>†</sup>  
Computer Science Department Brandeis  
University Waltham, MA 02454, USA  
mairson@cs.brandeis.edu

### ABSTRACT

We investigate the relationship between two polynomial time languages: Bellantoni and Cook’s functional combinator language BC, and Girard’s Light Linear Logic LAL (as amended by Asperti). It is known that both languages characterize exactly the polynomial time functions. Despite this similarity, an open question has been to provide a *compositional* translation between the two languages: given a PTIME function  $f$  described in a BC program  $p_f$ , how can we translate  $p_f$  into an LAL-program? Compositionality means that the translation of a program is a function of the translation of its constituent parts. The construction of such a compiler is fundamentally an exercise in functional programming.

The best known such translation, due to Murawski and Ong, compiles only the *linear* fragment BC<sup>-</sup>, where some variables can only be referenced once. We show that this fragment can be evaluated in LOGSPACE. Next, we prove that extensions of their approach to the full BC language *cannot* provide a correct translation into LAL. In fact, any translation that interprets integers and primitive recursion via inductive type codings fails. Finally, we provide a correct, compositional translation between the two languages. For this translation we generalize Turing machine simulations into an SECD machine, realized in LAL.

### Categories and Subject Descriptors

D.3.1 [Programming Languages]: Formal Definitions and Theory; F.3.3 [Logics and Meanings of Programs]: Studies of Program Constructs—*Program and recursion schemes*; F.4.1 [Computation by Abstract Devices]: Mathematical Logic—*Computability theory, Computational logic, Lambda*

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

<sup>†</sup>Supported by NSF Grants CCR-9619638, CDA-9806718, and the Tyson Foundation.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright 2001 ACM X-XXXXX-XX-X/XX/XX ...\$5.00.

*calculus and related systems*

### General Terms

Languages, Theory, Performance

### Keywords

Polynomial Time, Bellantoni-Cook, Light Affine Logic, Implicit Computational Complexity

## 1. INTRODUCTION

*History repeats itself, first as tragedy, second as farce.*

—Karl Marx

Paraphrasing not Marx, but Landin, we are now constructing the next 700 languages of polynomial time. We are thereby repeating what Church, Turing, and Post first did for recursive enumerability (r.e.), only at the level of polynomial-time complexity. On the theoretical side, such languages provide precise insights into how programming constructs affect complexity. This issue is investigated, for example by Jones [13], who characterizes a large number of complexity classes by restricting various features of programming languages without constructors. On the practical side, this work illustrates how compilers using evaluation techniques like laziness and tail recursion can control complexity, e.g., [5]. Moreover, it holds the key to a Shangri-La where compilers are capable of reasoning about the complexity of the programs they compile.

A seminal example of such a polynomial time language is the combinator language BC by Bellantoni and Cook [4]. BC introduces a restricted form of primitive recursion and composition, in order to avoid primitive recursion over values computed by a subrecursion. This restriction bounds the length of outputs by polynomials. Hofmann [10, 11] generalizes the approach to higher-order functions by introducing a *modal type-system* restricting the growth of functions. Types can be used to constrain many intensional program behaviors: when investigating type systems that ensure polynomial-time computation, we need to amend Milner’s “well typed programs do not go wrong” to “well typed programs do not run too long.”

In a logical analogue of this work, Girard introduces Light Linear Logic [8], a version of Linear Logic [7] with limited copying and stratified computation. Light Linear Logic

is greatly simplified by Asperti and Roversi in their Light Affine Logic (LAL) [2].<sup>1</sup> Through the Curry-Howard isomorphism [6, 12] this logic implicitly defines a functional programming language [2, 19]. *Proofnets* for linear logics embody the logician’s graph reduction technology for implementing functional languages. The key question in using proofnets is to understand what structures are *shared*, and how. Different such sharing strategies for sharing can characterize evaluation orders, distinguishing (for example) call-by-value and call-by-name. Along exactly these lines, Murawski and Ong [18] construct another operator algebra,  $\text{BC}^\pm$ , in an attempt to encode a variation of BC into LAL. Leivant [15, 16] proposes a variety of systems based on the *tiering* of recursion in the spirit of BC. And finally, the authors investigate intensional limitations to the concept of PTIME languages in [17], where it is shown that slight variations in the input-output conventions yield exponential time bounds rather than PTIME.

The Church-Turing thesis is supported by the fact that the formalisms for describing r.e. simulate each other. Restricting the complexity classes themselves to PTIME, can the formalisms described above be said to simulate each other, giving rise to perhaps a PTIME Church-Turing thesis? To begin answering this question, we continue the investigation of simulating BC in LAL. Given a PTIME function  $f$  described in BC program  $\mathbf{p}_f$ , how can we translate  $\mathbf{p}_f$  to its equivalent in LAL?

**Technical contributions:** The best known translation of BC into LAL, due to Murawski and Ong, compiles only the *linear* fragment  $\text{BC}^-$ , where so called *safe variables* cannot be shared. We show that this fragment can be evaluated in LOGSPACE. The key idea is that a nonstandard procedure calling protocol can trade more time for less space—it goes one better than tail recursion by throwing away the current call context, grabbing instead the context for the recursive call, and later *restarting* the computation once a recursive call produces an answer.

Next, we prove that extensions of their approach to the full BC algebra cannot provide a correct translation into LAL. This is proved through a general theorem of the impossibility of any translation that interprets integers and primitive recursion via inductive type codings. This result shows why the Murawski/Ong result is the best possible result derivable with their technique. Finally, we provide a correct, compositional translation of BC into LAL, where the target code simulates a version of an SECD-machine, and discuss how satisfactory this solution actually is.

## 2. PRELIMINARIES

We begin by briefly recalling the concepts essential to this paper. This section is not intended to be self-contained so readers are encouraged to consult the references for more details.

### 2.1 The Function Algebras BC and $\text{BC}^-$

Bellantoni and Cook introduced in [4] a function algebra that characterizes exactly PTIME without any references to external clocks. The function algebra is an offspring of the

<sup>1</sup>The addition of *affinity* (where inputs to procedures can be discarded) makes programming easier, but was computationally unnecessary: all polynomial time computations live quite happily in the non-affine, multiplicative fragment of light linear logic (i.e.,  $\lambda$ -calculus without  $K$ -redexes).

class of primitive recursive versions. The lower complexity is achieved by dividing the function arguments into *normal* and *safe* arguments (shown with a ; in the argument list and denoted by  $\vec{x}$  and  $\vec{y}$ ) and using this division to limit composition and recursion. The syntax of BC (with program labels  $l$ )<sup>2</sup> is as follows

$$\begin{aligned} b^l ::= & 0^l \mid (\pi_j^{m,m'})^l \mid \mathbf{p}^l \mid \mathbf{s}_b^l \mid \mathbf{c}^l \\ & \mid (f^{l,0} \circ (g_1^{l,1}, \dots, g_k^{l,k}))^l \\ & \mid \mathbf{rec}(g^{l,1}, h_0^{l,2}, h_1^{l,3})^l \end{aligned} \quad (1)$$

where the constructors are the constant function zero, the projection functions, the binary predecessor (right shift), binary successors, conditional, safe composition, and safe recursion, resp. In the case of safe composition the number of arguments of the functions  $f$  and  $g_1, \dots, g_k$  are as dictated by the semantics:

$$\begin{aligned} (f \circ (g_1, \dots, g_m, g_{m+1}, \dots, g_{m+m'}))(\vec{x}; \vec{y}) = \\ f(g_1(\vec{x}); \dots, g_m(\vec{x}); g_{m+1}(\vec{x}; \vec{y}), \dots, g_{m+m'}(\vec{x}; \vec{y})) \end{aligned}$$

Similarly, the functions  $g$ ,  $h_0$ , and  $h_1$  of safe recursion are restricted by  $\mathbf{rec}(g, h_0, h_1) = f$  where

$$f(n, \vec{x}; \vec{y}) = \begin{cases} g(\vec{x}; \vec{y}) & \text{if } n = 0 \\ h_b(n', \vec{x}; \vec{y}, f(n', \vec{x}; \vec{y})) & \text{if } n = n' \cdot b \end{cases}$$

Here  $\cdot$  is the concatenation of two binary numbers and  $b$  is a binary digit.

Murawski and Ong introduced a linear version of BC,  $\text{BC}^-$ , in [18]. Its fundamental syntax is (1). The difference is highlighted by the semantics where *safe linear composition* reads

$$\begin{aligned} (f \circ (g_1, \dots, g_m, g_{m+1}, \dots, g_{m+m'}))(\vec{x} : \vec{y}_1, \dots, \vec{y}_{m'}) = \\ f(g_1(\vec{x} : \cdot), \dots, g_m(\vec{x} : \cdot) : \\ g_{m+1}(\vec{x} : \vec{y}_1), \dots, g_{m+m'}(\vec{x} : \vec{y}_{m'})) \end{aligned}$$

Where the variables  $\vec{y}_i$  are treated linearly and not duplicated, and *safe linear recursion* is defined by  $\mathbf{rec}(g, h_0, h_1) = f$  with

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

To distinguish  $\text{BC}^-$  from BC we have used  $:$  to separate the arguments.

When discussing the syntax in (1) we refer to each of the right hand sides as a syntactic constructor. The arity of a syntactic constructor is the number of subtrees in the syntax tree. In other words,  $0$ ,  $\pi_j^{m,m'}$ ,  $\mathbf{p}$ ,  $\mathbf{s}_b$ , and  $\mathbf{c}$  have arity 0,  $\mathbf{rec}$  has arity 3, and  $\circ$  has arity  $k + 1$ . We denote a syntactic constructor by  $\mathbf{c}$  and its arity is  $k_{\mathbf{c}}$ .

### 2.2 Light Affine Logic

Girard introduces linear logic [7] as a means to make the resource usage in proof normalization explicit. This is done by focusing on the way resources are duplicated. The complexity of proof normalization in linear logic is far beyond

<sup>2</sup>The difference in the labelling of the subterms of safe computation and safe recursion is a notational convenience that will become clear later.

$$\begin{array}{c}
\text{ax} \frac{}{x : A \vdash x : A} \quad \text{weak} \frac{\Gamma \vdash A}{\Gamma, x : B \vdash A} \\
\text{cut} \frac{\Gamma \vdash M : A \quad x : A, \Delta \vdash N : B}{\Gamma, \Delta \vdash N[M/x] : B} \\
\text{contr} \frac{\Gamma, x : !A, y : !A \vdash M : C}{\Gamma, x : !A \vdash M[x/y] : C} \\
\text{-}\circ\text{l} \frac{\Gamma \vdash M : A \quad x : B, \Delta \vdash N : C}{\Gamma, y : A \multimap B \vdash N[(yM)/x] : C} \\
\text{-}\circ\text{r} \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x. A.M : A \multimap B} \\
\forall\text{l} \frac{\Gamma, x : A[B/X] \vdash M : C}{\Gamma, y : \forall X. A \vdash M[(yB)/x] : C} \quad \forall\text{r} \frac{\Gamma \vdash M : A}{\Gamma \vdash M : \forall X. A} \\
!\text{o} \frac{\vdash M : A}{\vdash !M : !A} \quad !\text{l} \frac{x : B \vdash M : A}{y : !B \vdash !M : !A} \\
\text{\$} \frac{\vec{x} : \vec{A}, \vec{y} : \vec{B} \vdash C}{\vec{x}' : \vec{!A}, \vec{y}' : \text{\$}\vec{B} \vdash \text{\$}(M) : \text{\$}C}
\end{array}$$

**Figure 1: Typing rules for ILAL.** The notation  $[\cdot/\cdot]$  is used for capture-free substitution. In the  $\forall$ -rule it is a condition that  $X$  is not free in  $\Gamma$ .

any notion of feasible function. For this reason he introduces a restricted version, *light linear logic* (LLL) [8]. Using LLL as a type system, all programs of a fixed type evaluate in polynomial time. Asperti and Roversi amended LLL to *light affine logic* (LAL) by adding general weakening [1, 2]. Affinity simplified and facilitated programming while restricting attention to the multiplicative fragment without affecting expressiveness, but in fact that expressiveness exists in the non-affine, multiplicative part of LLL. Nonetheless, we employ LAL due to its notational ease for the working programmer.

Through the Curry-Howard isomorphism the intuitionistic fragment, ILAL, of LAL can be seen as a typed  $\lambda$ -calculus. We will restrict us to ILAL since it gives us a programming language where duplication of values is made explicit and limited. The limited duplication is achieved through modal types where only values of type modality  $!$  can be copied. Furthermore, evaluation is stratified into tiers. For this reason (I)LAL has an extra neutral modality  $\text{\$}$  to align expressions in a given tier. Let us first recall the grammar of ILAL-types:

$$A, B ::= V \mid A \multimap B \mid !A \mid \text{\$}A \mid \forall V. A \ .$$

Here  $V$  ranges over propositional variables,  $\forall$  is second-order quantification,  $\multimap$  is linear implication, and  $!$  and  $\text{\$}$  are the modalities.

With these types we can introduce the typing rules in Fig. 2.2. Implicitly this also specifies our syntax; the only non standard feature is the **let**-construct which introduces the modalities and thus serves to split the computation into tiers. Since one can derive the following typing rule:

$$\text{\textcircled{a}} \frac{\Gamma \vdash M : A \multimap B \quad \Delta \vdash N : A}{\Gamma, \Delta \vdash MN}$$

we will write function application by juxtaposition. By defining  $A \otimes B \multimap \multimap \forall X. (A \multimap B \multimap X) \multimap X$  and using the usual  $\lambda$ -calculus encoding of pairs  $M \otimes N$  as  $\lambda x. x M N$ ,

one can derive the following rules:

$$\begin{array}{c}
\otimes\text{l} \frac{\Gamma, x : A, y : B \vdash M : C}{\Gamma, z : A \otimes B \vdash \text{let } x \otimes y = z \text{ in } M : C} \\
\otimes\text{r} \frac{\Gamma \vdash M : A \quad \Delta \vdash N : B}{\Gamma, \Delta \vdash M \otimes N : A \otimes B} \ .
\end{array}$$

By using for instance  $\mathbf{1} = \forall X. X$  we get a unit type. And furthermore, as Asperti points out in [1] we can even define additive types  $A \& B$  which gives us a case construction

$$\begin{array}{c}
&\&\text{l} \frac{\Gamma \vdash M : A}{\Gamma \vdash \text{inl}(M) : A \oplus B} \quad \&\text{l} \frac{\Gamma \vdash M : B}{\Gamma \vdash \text{inr}(M) : A \oplus B} \\
&\&\text{r} \frac{\Gamma, x : A \vdash M : C \quad \Gamma, x : B \vdash N : C}{\Gamma, y : A \oplus B \vdash y?M|N : C} \ .
\end{array}$$

It is standard to extend these constructs to general tuples and branching over inductive types. We thus have a full-fledged functional programming language in ILAL.

As usual for typed  $\lambda$ -calculus one can discuss the issue of *normalization*. In particular one can prove the possibility of proof normalization which removes all uses of the cut-rule. This corresponds to evaluating the program. A word of caution is warranted here: As both Asperti [1] and Terui [21] discuss, there are various problems with doing the evaluation in the term calculus one gets from Fig. 2.2. It is however completely understood how to avoid these problems using the technology of proofnets. We will not divulge into the details since we in this paper are not concerned about the intrinsics of how to evaluate the programs.

It is important to notice that there are no functions of type  $!A \multimap !A$  (*dereliction*) or  $A \multimap !A$  (*promotion*). It is thus not possible to change the number of modalities around an expression. One can therefore do evaluation level by level: first evaluate expressions with 0 modalities, then with 1 modality, then 2 modalities, etc. We refer to this as *stratification*. It is a defining feature of LAL evaluation that the result has size  $O(n^{2^d})$  (see [17] for a simple proof) where  $n$  is the size of the original term and  $d - 1$  is the highest number of modalities occurring in a formula (also referred to as the number of *levels*). It is worth mentioning that the pair encoding above only increases the term size linearly, while the encoding of additives squares it in worst case (practice will be well below though). So using these encodings do not break the polynomial bound.

In the following we will refer to normal form found by evaluation as  $\mathbf{nf}(P)$ . Essentially, the normal form will be a term for which the type derivation does not use the cut-rule. It is obvious that a cut-free type derivation of a term of type  $\diamond_1 \cdots \diamond_k A$  ( $\diamond_i \in \{!, \text{\$}\}$ ) starts with applications of the  $!$  and  $\text{\$}$ -rules. We will use  $\underline{P}$  to denotes the term  $P$  stripped of these outermost applications of the modal rules.

### 3. BC IS CONTAINED IN LOGSPACE

**THEOREM 1.** *For any function definable in the function algebra  $BC$ , there is a Turing Machine evaluating the function in LOGSPACE. Moreover, the Turing Machine can be constructed from the function expression in constant work space.*

The proof depends on two facts. The first, taken from [4], is that for any fixed BC program, the size of the output and all intermediate values are a polynomial in the size of the

inputs:<sup>3</sup>

$$|f(\vec{x}; \vec{y})| \leq q_f(|\vec{x}|) + \max_i |y_i| \quad (2)$$

where  $f$  is the function being computed,  $q_f$  is a polynomial, and  $|\cdot|$  is the usual length function. Thus we can iterate over the bits of the output, using a counter of size  $O(\log |\vec{x}; \vec{y}|)$  to identify the bits. Second, in the linear fragment  $BC^+$ , every bit of output is a function of only a *constant* number of bits of the input; again, in logspace, we can track and determine which bits these are.

This second fact is shown by induction, and is trivial except in the case of linear primitive recursion—recall that

$$f(n \cdot b, \vec{x} : \vec{y}) = h_b(n, \vec{x} : f(n, b, \vec{x} : \vec{y})).$$

If a bit is needed in the recursive call to  $f$ , the naive induction collapses. Instead, the key idea is basically the following: when determining the  $i$ th bit of  $f(n \cdot b, \vec{x} : \vec{y})$  requires the  $j$ th bit of  $f(n, b, \vec{x} : \vec{y})$ , check whether that bit has already been computed. If not, forget about  $i$  and compute only the subtask. This may incur similar amnesia until a bit is finally produced: remember the bit, the depth of the recursion at which it was produced, and *restart* the *original* computation. By memorizing a single computed bit in the call chain, we can slowly emerge from the recursion with the required bit.

In more detail, the evaluation of a function  $f$  uses (2) and  $q_f$  to find an upper bound on the number of bits in  $f(\vec{x}, \vec{y})$ . It then proceeds from this number down to zero to query for the bits from the rightmost to the leftmost. The querying recursively spins of subqueries until a bit can be found either in a constant or in the input. The evaluator thus keeps a stack of requests for bits. When a bit has been evaluated it is stored in fixed-length queue called the *computation window*. Since the size of the entries are logarithmic in the input, the proof consists in showing that it is sufficient to keep one entry per program point on the stack and in the computation window. One could thus get an alternative implementation by augmenting the syntax tree with the needed fields and update the fields as needed.

We introduce the following notation for the data structures used in the evaluator:

*Definition 1.*

1. a *program label* is a description of the paths from the root of the syntax tree to a node in the syntax tree; we write  $l.n$  for the augmentation of the path  $l$  to obtain the  $n$ th subchild of the node  $l$ . We use  $b^l$  to denote the subexpression labelled by  $l$ . We use  $\prec$  for the lexicographic ordering relation on labels.
2. An *execution label* is a triple  $(l, n, r)$  of a program label  $l$ , a bit index  $n$ , and a recursion depth  $r$ . This describes the request for a bit of the output.
3. A *source description* is a number  $s$  describing the right shifting accompanied by one of the following 3 forms: **input**( $n$ ), **comp**( $l$ ), or **rec**( $l, r$ ). This describes whether a function argument comes from program input  $n$ , the subcomputation at program point  $l$ , or the recursion defined at program point  $l$  depth  $r$ , resp.

<sup>3</sup>In [4, 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.

4. An *execution stack entry* is a pair consisting of an execution label  $(l, n, r)$  and an *environment*  $\vec{e} = [e_1, \dots, e_n]$ ; each  $e_i$  is a source description  $(f_i, s_i)$  for each of the arguments of the function at  $l$ .
5. An *computation window entry* is a pair consisting of an execution label  $(l, n, r)$  and a bit value  $b$ . The bit value can be 0, 1, and **fail**; the latter describes a non-existing bit.

For a fixed program, the number of program points is fixed and thus the size of a program label is fixed. The only variable parts of the above definition are thus the bit indexes, the right shift  $s$ , and the recursion depth  $r$ . These are all bounded by the maximum output size of (2) and are thus representable in  $O(\log |\vec{x}; \vec{y}|)$ . So with at most one computation window and execution stack entry per program label, the computation window and execution stack is represented in  $O(\log |\vec{x}; \vec{y}|)$  for a fixed program.

The main part of evaluator is a loop that examines the top of the stack and finds the bit described by the stack entry. To this it uses a function **lookup** which given a source description check whether the bit is available. If the bit is not available, the stack is modified such that the request for the bit is on top and the main loop is restarted. Otherwise, the bit value is returned.

The main trick is handling recursion without storing the call stack: Suppose we have an  $r$ -deep recursion: this is done with the computational amnesia described above. The details of the handling of an expression  $b^l$  at program point  $l$  is in Fig. 3. The function **lookup** is described in Fig. 3.

To prove that the interpreter has the desired properties, we need to formalize the fact that the interpreter traces through the syntax tree from left to right. This is done using the lexicographic order. We first note the following lemma follows from inspecting the definition of the main loop and **lookup**.

LEMMA 1. *Any entry  $(l.j, n, r), \vec{e}$  pushed during execution has the following property: for all  $(\mathbf{comp}(l'), s') \in \vec{e}$  the following holds:*

$j = 0$ :  $l' = l.j_2$  for some  $j_2 > j$ .

$j \leq 1$ : there exists  $l_1, l_2, j_1$ , and  $j_2$  such that  $l.j = l_1.j_1.l_2$  and  $l' = l_1.j_2$  with  $j_2 > j_1$  and the tail label  $l_2$  non-empty. The label  $l_1$  and index  $j_1$  is the same for all the entries in  $\vec{e}$ .

In both cases the indices  $j_2$  are different.

COROLLARY 1. *Any entry  $(l.j, n, r), \vec{e}$  pushed during execution has the following property: for all  $(\mathbf{comp}(l'), s') \in \vec{e}$  it is the case that  $l \prec l'$ .*

COROLLARY 2. *During execution any given label occurs at most once on the stack.*

PROOF. Induction on the length of the computation where the five cases involving **push–restart**( $\cdot, \cdot$ ) are checked in the inductive step.  $\square$

PROPOSITION 1. *The main loop succeeds: the bit corresponding to an entry  $(l, n, r)$  on top of the execution stack will eventually be stored in the computation window.*

$$\left\{ \begin{array}{ll}
\text{fail} & \text{when } b^l = 0^l \\
\text{lookup}(e_j, n) & \text{when } b^l = \pi_j^{m, m'} \\
\text{lookup}(e_1, n+1) & \text{when } b^l = \mathbf{p}^l \\
\left\{ \begin{array}{ll}
b & \text{when } n = 0 \\
\text{lookup}(e_1, n-1) & \text{o.w.}
\end{array} \right. & \text{when } b^l = \mathbf{s}_b^l \\
\left\{ \begin{array}{ll}
\text{lookup}(e_3, n) & \text{when } \text{lookup}(e_1, 0) = 1 \\
\text{lookup}(e_2, n) & \text{o.w.}
\end{array} \right. & \text{when } b^l = \mathbf{c}^l \\
\text{lookup}(\text{comp}(l.0), n) & \text{when } b^l = (f \circ \langle g_1, \dots, g_k \rangle)^l \\
\left\{ \begin{array}{ll}
\text{lookup}(\text{comp}(l.1), n) & \text{when } \text{lookup}(e_1, 0) = \text{fail} \\
\text{lookup}(\text{comp}(l.(d+2)), n) & \text{when } \text{lookup}(e_1, 0) = d
\end{array} \right. & \text{when } b^l = \text{rec}(g, h_0, h_1)^l
\end{array} \right.$$

**Figure 2: The branching in the main loop.**

Split on the form of the source description:

$w = \text{input}(j)$ : bit  $n + s$  of the  $j$ th program input; **fail** if non-existing

$w = \text{comp}(l.j)$ : if  $(l.j, 0, n + s)$  is in the computation window return the value. Otherwise, use the environment  $e$  and the recursion depth  $r$  from the execution label for  $l$  to do the following:

$$\left\{ \begin{array}{ll}
\text{push-restart}((l.0, 0, n + s), [\text{comp}(l.1), \dots, \text{comp}(l.k)]) & b^l = (f \circ \langle g_1, \dots, g_k \rangle)^l, j = 0 \\
\text{push-restart}((l.j, 0, n + s), [\vec{e}_x, \vec{e}_{y_j}]) & b^l = (f \circ \langle g_1, \dots, g_k \rangle)^l, j > 0 \\
\text{push-restart}((l.1, 0, n + s), [e_2, \dots, e_k]) & b^l = \text{rec}(g, h_0, h_1)^l, j = 1 \\
\text{push-restart}((l.j, 0, n + s), [\text{shift}(e_1), e_2, \dots, e_k, \text{rec}(l, r + 1)]) & b^l = \text{rec}(g, h_0, h_1)^l, j > 1
\end{array} \right.$$

The function  $\text{shift}_k(w, s) = (w, s + k)$  right shifts a source description  $k$  bits (with  $k = 1$  when  $k$  is omitted). For the safe composition,  $\vec{e}_x$  and  $\vec{e}_{y_j}$  denotes the elements in  $l$ 's environment for the normal and relevant safe arguments, resp. The newly created source descriptions in the first and the last case have zero right shift.

$w = \text{rec}(l, r)$ : Let  $(l, r', n')$ ,  $e'$  be the current execution stack element for label  $l$ . Pop the stack up until and including this element. Remove all computation window entries related to subtrees of  $l$ . Then

$$\text{push-restart}((l, r, n + s), [\text{shift}_{r-r'}(e_1), e_2, \dots, e_k])$$

The function  $\text{push-restart}(\cdot, \cdot)$  pushes an element on the execution stack and restarts the main loop

**Figure 3: The function  $\text{lookup}((w, s), n)$ .**

**COROLLARY 3.** *The main loop terminates.*

**PROOF.** We first check the fragment of BC without recursion. We use induction on the structure of the syntax. The only non-trivial case is the conditional where we should check that we can find the zeroth bit of  $e_1$  without overwriting an entry stored for  $e_2$  or  $e_3$ . We consider the case of  $e_1$  and  $e_2$ . Suppose they both depend on subcomputations, say  $\text{comp}(l_1, n_1)$  and  $\text{comp}(l_2, n_2)$ . Due to Lemma 1,  $l_1$  and  $l_2$  are distinct siblings of a the same node  $l'$  with indices  $j_1, j_2 > 0$ . Using the lexicographic order and the non-emptiness of the tail label in Case 1 of Lemma 1, it follows that neither  $l_1$  nor  $l_2$  will depend on the other.

With recursion, the unrolling of the stack to replace  $(l', n', r')$  by  $(l', n'', r' + 1)$  can result in the entry  $(l, n, r)$  disappearing from the stack. However, upon the success of the computation of  $(l', n'', r' + 1)$ , the entry  $(l', n', r')$  will reappear on the stack and thus also  $(l, n, r)$ . Due to the affinity of BC, the subcomputation of  $(l', n', r')$  has only one recursive call to  $l'$  so the lookup of  $(l', n'', r' + 1)$  will be last computed value of for  $l'$  and thus in the computation window. Consequently, the stack will not be unrolled and the computation of  $(l, n, r)$

will succeed. Since every recursion right shifts the recursion variable, the recursion will eventually request a non-existing bit and thus choose the base case of the recursion.  $\square$

It is worth noting how the affinity is crucial in the above proof and thus why it fails for BC and PTIME in general: Without affinity a function could have more than one, say  $k$ , recursive calls in the recursive step. We would thus need to store up to  $k - 1$  bits for each of the recursive levels. With the number of recursive levels being polynomial in the input, this is obviously not storable in LOGSPACE. The affinity thus make the situation comparable to tail recursion optimizations used in compiler. For similar reasons the interpreter cannot be extended to work for the PTIME-language  $\text{BC}^\pm$  of [18]; the  $\text{case}_K$ -construct can need up to  $K + 1$  different bits of a recursive value.

An implementation of the evaluator is available from the authors' web page.

## 4. COMPOSITIONAL ENCODINGS OF BC

Terui proves in [21] that ILAL evaluation is polynomial time under all evaluation strategies. On the other hand,

Beckmann and Weiermann give a rewriting system for BC in [3] and establish that it is only polynomial time when evaluated call-by-value. An example illustrating the latter is the following function  $f$ :

$$\begin{aligned} g(\cdot) &= 3 \\ h(x; y) &= c(\cdot; y, y) \\ f(n) &= \text{rec}(g, h, h)(n) \end{aligned}$$

which is represented by the BC-term

$$\text{rec}(c \circ \langle \pi_2^{1,1}, \pi_2^{1,1}, \pi_2^{1,1} \rangle, c \circ \langle \pi_2^{1,1}, \pi_1^{1,1}, \pi_2^{1,1} \rangle, \mathbf{s}_1 \circ \mathbf{s}_1 \circ 0) \quad (3)$$

Though the function evaluates to 3, the running time will be  $O(2^n)$  under call-by-name. This example suggests that it is only possible to encode BC into LAL if one also fixes the evaluation strategy to call-by-value. This restriction is incompatible with the natural, inductive-type encodings in the style of System F, as for instance in [18, 20], where integers are coded as binary Church numerals, and each BC construct translates to a fixed family of LAL-terms that are composed in a syntax-directed manner.

A closer analysis exposes the conditional  $c$  as the culprit among the primitive functions: it requires evaluation of two of the arguments and together with safe composition it thus allows copying of work. Suppose in an encoding of the above BC program into LAL, we code  $c$  as a  $\lambda$ -term in LAL of type  $\text{Int} \multimap \text{Int} \multimap \text{Int} \multimap \text{Int}$ . While we cannot add safe arguments in BC as a primitive operation (this would break the polynomial bound), we *can* replace the purported encoding by an LAL function of that type that *appends* its three arguments.<sup>4</sup> Regarded as a  $\lambda$ -term and ignoring LAL type information, the term would then compute a numeric function with outputs of exponential length. This output size contradicts the  $O(n^{2^d})$  normalization bound on LAL. We now formalize this notion of inductive encoding. Given the formalization we prove that if an inductive encoding existed, we could reinterpret the conditional to derive an LAL term that grows exponentially during evaluation.

*Definition 2.* A single-typed *compositional encoding* of BC into LAL consists of the following:

1. A type  $\text{Int}$  representing binary integers. Associated with this there is an embedding  $\lceil \cdot \rceil$  of the natural numbers into LAL-proof-nets of type  $\text{Int}$  and a mapping  $\lfloor \cdot \rfloor$  from normal forms of type  $\text{Int}$  to integers.
2. A set  $\mathcal{P}_c$  of terms for each syntactic constructor  $c$  of BC. Each term  $P \in \mathcal{P}_c$  fulfills the following:
  - the number of variables is  $k_c$ .
  - the type of  $P$  and each of its free variables have a type of the form  $\tau_1 \multimap \dots \multimap \tau_k \multimap \tau_0$  where  $k$  is arity of the function being encoded. Each  $\tau_i$  has the form  $\diamond_1 \dots \diamond_{n_i} \text{Int}$  for some  $n_i \geq 0$  with each  $\diamond_j \in \{!, \}\}$ .
  - $P$  is a *correct interpretation* of  $c$ , i.e., the normal form of  $P$  cut against all type correct input is the value computed by  $c$ .
3. A mapping  $\lceil \cdot \rceil$  that when given any BC-function

$$b(x_1, \dots, x_m; y_1, \dots, y_{m'})$$

<sup>4</sup>Addition is not good enough, since we are using a binary representation.

takes it into a LAL-term. The mapping is constrained as follows: the term  $\lceil b \rceil$  can be constructed from the syntax tree of  $b$  as follows: for each node  $c(b_1, \dots, b_{k_c})$  in the tree choose a  $P_c \in \mathcal{P}_c$  and cut it against the successive  $P_{b_1}, \dots, P_{b_{k_c}}$  constructed for the sub-trees.

Some comments about the definition:

- We have labeled the encoding single-typed as it fundamentally uses only one type to encode integers. We shall see in the next section that relaxing this allows a compositional encoding.
- Since  $\multimap$ -introduction and elimination are both reversible, it is only a notational convenience, not a further restriction, that we require each term to be of a function type.

It is easily checked that the encodings in [18, 20] are compositional as defined above. Furthermore it follows from the definition that the mapping  $\lceil \cdot \rceil$  on integers is injective. Generally  $\lceil n \rceil$  gets bigger when  $n$  grows, though  $\lceil \cdot \rceil$  is not necessarily non-size-decreasing. Furthermore, we are not guaranteed that finding a number with a bigger representation is actually computable in LAL. We therefore need the following additional constraint.

*Definition 3.* A single-typed compositional encoding has *concatenation* if there exists an LAL-term  $\text{concat} : \text{Int} \multimap \text{Int} \multimap \text{Int}$  such that  $|\mathbf{nf}(\text{concat } N_1 N_2)| \geq |\mathbf{nf}(N_1)| + |\mathbf{nf}(N_2)|$ .

Though there might exist compositional encodings without concatenation, it is a property shared by for instance the LAL-encoding of the standard way of doing inductive data types in System F (see [9] for a compendium of such encodings).

With the notion of compositional encoding fixed, we can now prove that the existence of a single-typed compositional encoding with concatenation would violate the exponential bounds on proof normalization in LAL. This leads to the following theorem:

*THEOREM 2.* *There does not exist a single-typed compositional encoding of BC into LAL which also has concatenation.*

*PROOF.* We use *reductio ad absurdum* and assume the existence of a compositional encoding of BC into LAL having concatenation. We consider the encoding of the function in (3) and use the basic definition of safe recursion to unfold  $f$ . We have

$$\begin{aligned} \text{rec}(g, h, h)(b_l \dots b_0) &= \\ &h(b_l \dots b_1; h(b_l \dots b_2; \dots h(b_l; g()) \dots)) \end{aligned} \quad (4)$$

for all choices of  $b_l \dots b_0$ . We now consider the term  $P_{\text{rec}}$  used to encode the safe recursion. Since it is required to correctly interpret all possible functions, it can only find the value of  $h(x)$  by applying  $\lceil h \rceil$  to the representation of  $x$ . In [17] it is shown that the normalization of a level can only square the size of the proofnet corresponding to the term. Since  $P_{\text{rec}}$  has a fixed type it is limited to compute  $|n|^k$  values of  $h$  for some  $k$ . It can therefore only interpret (4) correctly, by threading the output of one  $\lceil h \rceil$  into the next  $\lceil h \rceil$ . We thus have  $\sigma_2 = \sigma_0$  for  $\lceil h \rceil$  with type  $\sigma_1 \multimap \sigma_2 \multimap \sigma_0$ ; otherwise each iteration would add modalities and there would

be some  $l$  where the number of added modalities could not fit in with the type of **rec**.

We now focus attention on  $P_c$ . This is a term of type  $\tau_1 \multimap \tau_2 \multimap \tau_3 \multimap \tau_0$  with no free variables. Due to the stratification of LAL-computation, the requirement about correct interpretation implies that  $\tau_0$  must have at least the same modality as  $\tau_1$ ,  $\tau_2$ , and  $\tau_3$ . Furthermore,  $\tau_3$  must have neutral modalities on its outermost levels since it depends on several inputs. Combining this with the considerations about  $[h]$  we conclude that  $P_c$  is a term of type  $\tau \multimap \tau \multimap \tau \multimap \tau$  where  $\tau = \S^l \text{Int}$  for some  $l$ . We can thus replace  $P_c$  by **concat** embedded in  $l$  boxes. This results in a term with an exponential size normal form in contradiction with the polynomial bound on LAL-normalization of proof-nets with a fixed depth.  $\square$

## 5. A COMPOSITIONAL TRANSLATION OF BC INTO LAL

### 5.1 A TM simulation retrospective

Given that the inductive-type approach to an interpretation is doomed to failure, we resort to a simulation inspired by the SECD machine [14], which is really an elaborated Turing machine. As a first step, it is useful to recollect how the latter may be represented as an LAL proof-net. For brevity we describe these nets using  $\lambda$ -terms elaborated with modalities  $!$ ,  $\S$ , and explicit substitutions marking inputs to boxes. Represent a TM ID as the proof-net  $\mathbf{ID} = \Lambda\beta.\lambda c.!(\mathbf{B} \multimap \beta \multimap \beta).\lambda n_L : \beta.\lambda n_R : \beta.(cl_1 \cdots (cl_p n_L)) \otimes (cr_1 \cdots (cr_p n_L)) \otimes \mathbf{State}$ , where  $\mathbf{B} = \forall\tau.\tau \multimap \tau \multimap \tau$  with constants 0 and 1, and  $\mathbf{State} : \Pi_k = \forall\alpha.\alpha \multimap \cdots \alpha \multimap \alpha$  is a  $k$ -ary projection function, representing the finite state control. Then to implement a transition function, note that  $\psi = \mathbf{ID} [\mathbf{B} \otimes \alpha]!(\lambda x : \mathbf{B}.\lambda b \otimes a : \mathbf{B} \otimes \alpha.x \otimes (c'ba))[\mathbf{cons}/c']$  has type  $\S(\mathbf{B} \otimes \alpha \multimap \mathbf{B} \otimes \alpha \multimap (\mathbf{B} \otimes \alpha) \otimes (\mathbf{B} \otimes \alpha) \otimes \Pi_k)$ ; when this value is input to the auxiliary port of a  $\S$ -box and applied to  $0 \otimes n_L$  and  $0 \otimes n_R$  (thus appending a 0 to each end of the tape), we get an output of type  $(\mathbf{B} \otimes \alpha) \otimes (\mathbf{B} \otimes \alpha) \otimes \Pi_k$ . This value presents the symbols on the left- and right-hand side of the TM tape incident on the read/write head; the state is then used to choose the list operations which construct a machine configuration.

In order to construct a PTIME TM simulation, duplicate the input, coded as a list of type  $\forall\alpha.!(\mathbf{B} \multimap \alpha \multimap \alpha) \multimap \S(\alpha \multimap \alpha)$ , to create both an initial ID, and a polynomial iterator; the latter must reduce the list to a (Church) tally integer of type  $\forall\alpha.!(\alpha \multimap \alpha) \multimap \S(\alpha \multimap \alpha)$ , then raised to a fixed power (the degree of the polynomial runtime), introducing  $\S$ -modalities dependent on the degree; the latter can be bounded by a logarithm of the power by iterated squaring. The derived tally integer can then be used as a clock to iteratively apply the transition function to the initial ID.

### 5.2 An SECD implementation

Intuitively speaking, what qualifies as a compositional translation? Let  $\Psi$  be a translator from BC programs to equivalent LAL proofnets. Ideally, we would like the following to hold: for some fixed functions  $\mathcal{C}, \mathcal{R}$  on proofnets,

$$\Psi f \circ (g_1, \dots, g_k) = \mathcal{C}(\Psi f, \Psi g_1, \dots, \Psi g_k) \quad (5)$$

$$\Psi \mathbf{rec}(g, h_0, h_1) = \mathcal{R}(\Psi g, \Psi h_0, \Psi h_1) \quad (6)$$

where  $\Psi$  is also defined on the basis combinators of BC.

The essence of these equations is that the translation of a BC program is given by a function of the translation of its constituent parts. Here, we preserve this essential idea by instead defining translation  $\Phi P = \mathcal{A}(\Phi_0 P, \Phi_1 P, \Phi_2 P)$  where for each  $\Phi_i$ , there are also functions  $\mathcal{C}_i$  and  $\mathcal{R}_i$  such that the above equations hold for  $(\Psi, \mathcal{C}, \mathcal{R}) = (\Phi_i, \mathcal{C}_i, \mathcal{R}_i)$ .

The  $\Phi_i$  compute—compositionally—components of the desired proofnets, and the function  $\mathcal{A}$  assembles them together; specifically,  $\Phi_0$  constructs a *core transition function* at the heart of the LAL simulation,  $\Phi_1$  constructs an interface for the core transition function to derive an LAL term of type  $\text{ID} \multimap \text{ID}$  on representations of BC computation states, and  $\Phi_2$  constructs a (polynomial) iterator on the latter function. The functions  $\mathcal{A}, \mathcal{C}_i, \mathcal{R}_i$  are LAL type-dependent, but in a very uniform, natural way—they just plug together LAL terms, while the types of those wires vary as a function of the translated BC program.

**Machine overview:** We interpret every BC program as an LAL proofnet simulating an SECD-like machine. The machine has a *store* stack, an *environment* stack, a *command* stack, and two *dump* stacks that implement copying and restoration of arguments. Each stack is an LAL “tally” list of type  $\forall X.!(T \multimap X \multimap X) \multimap \S(X \multimap X)$ ; the  $!(T \multimap X \multimap X)$  represents a sharable *cons*, and the next  $X$  represents an initial *nil*; by instantiating  $X$  with an “output type” and supplying a suitable function and initial value for *cons, nil*, we implement list iteration. Observe that  $T$  is a *fixed* type  $T$  (representing 0, 1 and delimiters) for storage, environment, and dumps, and  $T$  *varies* for commands—it is a type  $\mathcal{P}$  depending on the BC program being simulated. Every “machine cycle” can consume the top symbol of each stack, and push a finite number of symbols on each stack. The machine cycles are iterated or driven by a polynomial “tally” integer computed from the BC program. The term we construct for the SECD machine is LAL-typable because there is really only *one* loop: the clock that updates the representation of SECD machine states.

Suppose we knew how to code LAL “machine state update” terms of type  $\mathbf{ID}_p \multimap \mathbf{ID}_p$ , where  $\mathbf{ID}_p$  is the type of a machine for BC program  $p \in \{g, h_0, h_1\}$ . We have no idea how to combine these terms to construct a similar term for  $p = \mathbf{rec}(g, h_0, h_1)$ . But we *do* know how to construct something similar if we could separate these terms into a finite number of well-defined *components*. By instead maintaining these pieces *initially*, from the beginning of the construction, we can derive a compositional translation.

**SECD machine IDs:** A machine instantaneous description (ID) has form

$$\begin{aligned} & \Lambda\alpha.\Lambda\beta.\lambda c' :!(T \multimap \alpha \multimap \alpha).\lambda c'' :!(\mathcal{P} \multimap \beta \multimap \beta). \\ & \S(\lambda n_s : \alpha.\lambda n_e : \alpha.\lambda n_{d_1} : \alpha.\lambda n_{d_2} : \alpha.\lambda n_c : \beta. \\ & \quad (c' t_{1,1}(\cdots (c' t_{1,s} n_s) \cdots)) \\ & \quad \otimes (c' t_{2,1}(\cdots (c' t_{2,e} n_e) \cdots)) \\ & \quad \otimes (c' t_{3,1}(\cdots (c' t_{3,d_1} n_{d_1}) \cdots)) \\ & \quad \otimes (c' t_{4,1}(\cdots (c' t_{4,d_2} n_{d_2}) \cdots)) \\ & \quad \otimes (c'' \text{com}_1(\cdots (c'' \text{com}_k n_c) \cdots))) \\ & : \forall\alpha.\forall\beta.!(T \multimap \alpha \multimap \alpha) \multimap !(\mathcal{P} \multimap \beta \multimap \beta) \multimap \\ & \quad \S(\alpha \multimap \alpha \multimap \alpha \multimap \alpha \multimap \beta \multimap \alpha^{(4)} \otimes \beta) \end{aligned}$$

We write MID : ID for a machine ID of the defined type.

**Commands:** The basis of the BC combinators are implemented by *basic commands* of type  $\mathcal{B}$ , a finite set of com-

mands that move stack symbols. The set of commands for  $\mathbf{rec}(g, h_0, h_1)$  must include  $\mathcal{B}$  and all the commands for  $g, h_0, h_1$ , plus a command to *call* the code for  $\mathbf{rec}(g, h_0, h_1)$ . We thus take their disjoint sum using the construction

$$\begin{aligned} \Lambda X. \lambda \mathit{basic} : \mathcal{B} \multimap X. \lambda g : \mathcal{G} \multimap X. \\ \lambda h_0 : \mathcal{H}_0 \multimap X. \lambda h_1 : \mathcal{H}_1 \multimap X. \\ \lambda \mathit{choose} : X. \lambda \mathit{branch} : X. \lambda \mathit{call} : X. E \end{aligned}$$

to represent commands, where  $\mathcal{G}, \mathcal{H}_0, \mathcal{H}_1$  are the types of the commands for BC programs  $g, h_0, h_1$ , and  $E$  is either of the form *basic c* (for a basic command), *g c*, *h<sub>i</sub> c*, *choose* or *branch* (two commands needed to implement primitive recursion), or *call*.<sup>5</sup> (An alternative construction could use additive types.) The structure of the commands and types is clearly compositional. A disjoint sum construction of the command set for function composition  $f \circ \langle g_1, \dots, g_k \rangle$  is worked out similarly.

**Basic commands:** This is not a paper on microprogramming—suffice it to say that these commands cover conventional data movement between stacks. Basic commands can cause other basic commands to be pushed onto the command stack, in order to implement primitive subroutines for saving, copying, and removing data.

**Input conventions:** When BC program  $P$  is run on integer inputs  $n_i$ , the coded  $n_i$  (separated by delimiters) are in the environment (stack). Every program is invariant on the contents of the dumps. When  $P$  terminates, the output is in the store, which may hold multiple answers (separated by delimiters).

**Commands for primitive recursion:** We discuss modifications of the command stack to implement  $\mathbf{rec}(g, h_0, h_1)(n, \vec{x}; \vec{y})$ , where  $|\vec{x}| = p$  and  $|\vec{y}| = q$ , which uses the following basic commands. Recall that the environment is the stack representing  $(n, \vec{x}; \vec{y})$ .

ZERO? if the first integer in the environment is even (odd), write 1 (0) in the store.

ECOPY<sub>2</sub> pops an integer argument from the environment, copying it to both dumps.

ERESTORE<sub>*i*</sub> pops an integer argument in dump *i*, pushed to the environment.

ELOAD pops integer argument in stack, pushed to environment.

EPOP remove top integer from the environment.

Write  $c^{(r)}$  for  $r$  successive instances of command  $c$ . Primitive recursion begins with *call* on the top of the command stack. When popped and interpreted, it is replaced with ZERO?; *branch* in the commands; *branch* tests the result of ZERO? (in the store), pushing the commands EPOP; (*g call*) for  $g$ , or a more complex sequence for the recursive call:

```

ECOPY2;
pop lowest bit in top integer of dump 1;
pop top integer in dump 2 to stack;
ECOPY2(p+q); ERESTORE2(p+q); call;
ELOAD; ERESTORE2(p+q+1); choose;

```

<sup>5</sup>Recall for types  $A_1, A_2$  the coding of disjoint sum as  $A_1 + A_2 = \forall X. (A_1 \multimap X) \multimap (A_2 \multimap X) \multimap X$  with injections  $\mathit{in}_i a = \Lambda X. \lambda u_1 : A_1 \multimap X. \lambda u_2 : A_2 \multimap X. u_i a$ . Then a case dispatch on  $v : A_1 + A_2$  with methods  $g_i : A_i \multimap B$  is just  $v[B]g_1 g_2$ .

where *choose* pops the store, and that bit determines whether the symbol ( $H_0$  call) or ( $H_1$  call) is pushed on the command stack. All this is tedious in detail but important in its high-level form. A similar construction is carried out for commands for function composition.

**Interpreter for primitive recursion:** This LAL term  $\hat{\chi}_{\mathbf{rec}(g, h_0, h_1)}$  must interpret each command for primitive recursion appropriately, pushing and popping stack symbols. The interpreter is *compositional* since it depends on the construction of  $\hat{\chi}_g, \hat{\chi}_{h_0}, \hat{\chi}_{h_1}$ , but an awkward technical difficulty is that  $\hat{\chi}_g$  emits commands for the coding of BC program  $g$  into LAL; we need to *inject* these commands, using a function  $I : \mathcal{P} \multimap Y$  ( $Y$  quantified), into the LAL command set for  $\mathbf{rec}(g, h_0, h_1)$ . Recall  $\mathcal{P}$ , the type of commands for  $\mathbf{rec}(g, h_0, h_1)$ ; we define

$$\begin{aligned} \hat{\chi}_{\mathbf{rec}(g, h_0, h_1)} = \\ \Lambda Y. \lambda I^{(n)} : (\mathcal{P} \multimap Y)^{(n)}. \lambda C : \mathcal{P}. \lambda B : T^{(4)}. \\ \text{copy } T^{(4)} \text{ as } (T_b^{(4)}, T_g^{(4)}, T_{h_0}^{(4)}, T_{h_1}^{(4)}, T^{(4)}) \text{ in} \\ C [(\delta \multimap \delta) \otimes (\gamma \multimap \gamma)^{(4)}] \\ (\lambda \mathit{com} : \mathcal{B}. \hat{\chi}_{\mathbf{basic}}[Y](I \circ B)^{(a)} \text{ com } T_b^{(4)}) \\ (\lambda \mathit{com} : \mathcal{G}. \hat{\chi}_g[Y](I \circ G)^{(b)} \text{ com } T_g^{(4)}) \\ (\lambda \mathit{com} : \mathcal{H}_0. \hat{\chi}_{h_0}[Y](I \circ H_0)^{(c)} \text{ com } T_{h_0}^{(4)}) \\ (\lambda \mathit{com} : \mathcal{H}_1. \hat{\chi}_{h_1}[Y](I \circ H_1)^{(c)} \text{ com } T_{h_1}^{(4)}) \\ (\mathit{cons}'' (I \circ c_{1,1}) \circ \dots \circ (\mathit{cons}'' (I \circ c_{1,n_1})) \otimes \\ (\mathit{cons}' T_1^{(4)} \otimes \dots \otimes (\mathit{cons}' T_4^{(4)})) \\ (\mathit{cons}'' (I \circ c_{2,1}) \circ \dots \circ (\mathit{cons}'' (I \circ c_{2,n_2})) \otimes \\ (\mathit{cons}' T_1^{(4)} \otimes \dots \otimes (\mathit{cons}' T_4^{(4)})) \\ (\mathit{cons}'' (I \circ c_{3,1}) \circ \dots \circ (\mathit{cons}'' (I \circ c_{2,n_3})) \otimes \\ (\mathit{cons}' T_1^{(4)} \otimes \dots \otimes (\mathit{cons}' T_4^{(4)})) \end{aligned}$$

where  $\gamma, \delta$  are defined later,  $\mathit{cons}' : Y \multimap \delta \multimap \delta$ , and  $\mathit{cons}'' : T \multimap \gamma \multimap \gamma$ .

In this thorny bit of code, the command  $C$  is used to dispatch to the possible different cases; for example, consider when  $C$  is a  $g$ -command, namely, an LAL term with body ( $g c$ ) appearing after all the  $\lambda$ -abstractions in term  $C$ . In this case,  $c$  is a  $g$ -command, and  $(\lambda \mathit{com} : \mathcal{G}. \hat{\chi}_g[Y](I \circ G)^{(b)} \text{ com } T_g^{(4)})$  is substituted for  $g$ , and the  $g$ -command is substituted for  $\mathit{com}$ , deriving  $\hat{\chi}_g[Y](I \circ G)^{(b)} c T_g^{(4)}$ . The meaning of this latter term is the interpretation of command  $c$  by  $\hat{\chi}_g$ , supplied with enough copies of the injection function  $G : \mathcal{G} \multimap \mathcal{P}$  so that emitted commands are coerced into those for  $\mathbf{rec}(g, h_0, h_1)$ , not those for  $g$ . The injections are composed with  $I$ , so that  $\hat{\chi}_{\mathbf{rec}(g, h_0, h_1)}$  can later be used, as we have just used  $\hat{\chi}_g$ , to build an LAL term for a larger BC program.

Observe that the *number*  $n$  of needed injection functions can be statically, *compositionally* determined by the sum of those needed for  $\hat{\chi}_{\mathbf{basic}}, \hat{\chi}_g, \hat{\chi}_{h_i}$ , plus the number of commands needed to implement *choose*, *branch*, and *call* (at the end of  $\hat{\chi}_{\mathbf{rec}(g, h_0, h_1)}$ ). Elements of a constant type  $T$  can be copied without modalities: for example booleans can be duplicated using  $\mathit{Copybool} = \lambda b : \mathbf{B}. b[\mathbf{B} \otimes \mathbf{B}](0 \otimes 0)(1 \otimes 1)$ . Note also the injection functions  $B : \mathcal{B} \multimap \mathcal{P}$ ,  $G : \mathcal{G} \multimap \mathcal{P}$ ,  $H_i : \mathcal{H}_i \multimap \mathcal{P}$ , and that  $\mathit{cons}'$  and  $\mathit{cons}''$  have different types—one for commands, one for other stacks. The in-

terpreter is then

$$\begin{aligned} \chi_{\text{rec}(g, h_0, h_1)} &= \hat{\chi}_{\text{rec}(g, h_0, h_1)}[\mathcal{P}](\lambda \text{ com} : \mathcal{P}. \text{com}) \\ &: \mathcal{P} \multimap T^{(4)} \multimap (\delta \multimap \delta) \otimes (\gamma \multimap \gamma)^{(4)} \end{aligned}$$

using the *identity* coercion—these coercions were only introduced to compositionally plug the  $\hat{\chi}$  together. In other words, the term  $\chi_{\text{rec}(g, h_0, h_1)}$  tells: given the top symbols of each stack, what gets pushed onto them? For brevity, we omit the similar coding of function composition; our construction summarizes the compositional definition of  $\Phi_0$ , the LAL *core transition function* defined by a BC program.

**Constructing a transition function of type**  $\text{ID} \multimap \text{ID}$ : Recall  $\text{MID} : \text{ID}$  above; then

$$\begin{aligned} \text{MID } [T \otimes \gamma][\mathcal{P} \otimes \delta] : \\ !(T \multimap T \otimes \gamma \multimap T \otimes \gamma) \multimap !(\mathcal{P} \multimap \mathcal{P} \otimes \delta \multimap \mathcal{P} \otimes \delta) \multimap \\ \S(T \otimes \gamma \multimap \dots \multimap T \otimes \gamma \multimap \\ \mathcal{P} \otimes \delta \multimap (T \otimes \gamma)^{(4)} \otimes \mathcal{P} \otimes \delta) \end{aligned}$$

and so (using a construction in [2] for coding Turing machines):

$$\begin{aligned} \text{MID } [T \otimes \gamma][\mathcal{P} \otimes \delta] \\ !(\lambda e : T. \lambda u \otimes v : T \otimes \gamma. e \otimes (\text{cons}' u v)) \\ !(\lambda e : \mathcal{P}. \lambda u \otimes v : \mathcal{P} \otimes \delta. e \otimes (\text{cons}'' u v)) \\ : \S(T \otimes \gamma \multimap \dots \multimap T \otimes \gamma \multimap \\ \mathcal{P} \otimes \delta \multimap (T \otimes \gamma)^{(4)} \otimes \mathcal{P} \otimes \delta) \end{aligned}$$

and finally,

$$\begin{aligned} \S(x (\# \otimes n_s)(\# \otimes n_e)(\# \otimes n_{d_1})(\# \otimes n_{d_2})(\text{nop} \otimes n_c)) \\ [\text{MID } [T \otimes \gamma][\mathcal{P} \otimes \delta]/x] \\ : \S((T \otimes \gamma)^{(4)} \otimes \mathcal{P} \otimes \delta) \end{aligned}$$

The  $n_z$  are the *nil* for the stacks,  $\# : T$  a trivial stack symbol, and  $\text{nop} : \mathcal{P}$  a trivial basic command. It is straightforward to  $\lambda$ -abstract over appropriate parameters to derive a term *cycle* :  $\text{ID} \multimap \text{ID}$ .

**Iterator:** We now need to iterate *cycle* to carry out the entire computation, but the size of the iterator is a polynomial in the length of the integer inputs. This polynomial  $Q$  is readily computable, in a compositional fashion, from the bounds given by polynomial  $P$  derived from the Bellantoni-Cook theorem for BC, which measures the size of output as a fixed polynomial in the length of the inputs (see equation (2) in section 3). Because the LAL simulation operates at the “bit” level, we can safely take  $Q(\vec{x}, \vec{y}) = cP(\vec{x}, \vec{y})^k$  for some small integers  $c, k$ . The proof that such integers exist is a tedious induction following the lines of the derivation of polynomial  $P$ .

**THEOREM 3.** *Let  $f(\vec{x}; \vec{y})$  be a BC-program whose output is bounded by a polynomial of degree  $k$ . Let  $\mathbf{B}^* = \forall \alpha. !(\mathbf{B} \multimap \alpha \multimap \alpha) \multimap \S(\alpha \multimap \alpha)$ . Then for some constant  $c$ , there exists an LAL proofnet  $\Pi$ , defined compositionally from the structure of  $f$ , with type  $!\mathbf{B}^* \multimap \dots \multimap !\mathbf{B}^* \multimap \S^{c \log k} \mathbf{B}^*$ , such that  $\Pi$  simulates the computation of  $f$ , and (modulo a constant factor) the number of proofnet reductions required for normalization of  $\Pi$  (with inputs) is bounded by the number of reduction steps in the CBV evaluation of  $f$ .*

The translation we have described, from BC programs to LAL proofnets which simulate a kind of SECD machine, satisfy the basic requirement of a compositional encoding: the

translation of the whole is made up from the translation of the constituent parts. However, the translation is *unsatisfying* because it outputs proofnets that look like souped-up Turing machines, which they are. More specifically, unlike the translations of Murawski/Ong and Roversi, normal and safe variables have the same type, and do not seem to be treated differently.

Since the major conceptual idea of BC is the distinguishing of the normal from the safe, where is this distinction found in the equivalent LAL proofnets? The answer is: *in the iterator*. If we (illegally) primitively recursed on a safe variable, the translator would break because the iterator would be *too weak* to complete the computation, which would stop in mid-stream.

## 6. CONCLUSION AND OPEN PROBLEMS

In this paper, we investigated the difficulty of constructing a compositional translation between the Bellantoni-Cook functional combinator language BC, and the Girard-Asperti Light Affine Logic (LAL), two well-known examples of polynomial-time programming languages.

We first analyzed the natural inductive-type translation, due to Murawski and Ong, of  $BC^-$ , the linear fragment of BC, and showed it could be evaluated in LOGSPACE, which is most probably not PTIME—thus while the coding is elegant, its computational expressiveness is likely insufficient. We proceeded to establish that an extension of their approach is doomed to fail because it contradicts the normalization theorem for LAL, which gives explicit bounds on the size of possible output.

The lack of such an extension exhibits the fundamental difference in strength between light logics and the function algebra BC: the latter is *only* polynomial-time when evaluated call-by-value. (Thus the solution to this logical puzzle is based squarely on a programming language intuition.)

As a consequence, in order to provide a translation for the entirety of BC, we required a mechanism that could capture the BC evaluation order within the LAL encoding. This need motivated the virtually microcoded construction in the previous section, where we developed a compositional translation inspired by the SECD-machine. The technical anomaly that arises in attempting the essentially impossible extension of the inductive-type approach to the full BC is that LAL modalities  $!$  and  $\S$  appear in the wrong places; when a function that we need has type  $!\alpha \multimap \S \S \alpha$  instead of type  $!\alpha \multimap !\alpha$ , it can no longer be iterated.

The technical solution of this anomaly is to have only one, big iteration: that of a machine clock on computation states—the Bellantoni-Cook bounds on output explain exactly how much power that iterator needs. Some may question the beauty of this solution, but it is unquestionably a compositional translation.

Notwithstanding the progress made here, there are still more questions than answers when it comes to the relationship between polynomial-time languages. If we are to genuinely understand what polynomial time really means, we must be able to explain how programming languages which capture that notion can simulate each other. Among many questions are the following:

- How do these results relate to Hofmann’s operator algebra [11] and FALL [20]? Due to the additives, Hofmann’s system has the same feature as BC that

not all evaluation strategies are polynomial time. We thus conjecture that Hofmann’s system will exhibit the same behavior as BC and need the SECD-inspired compositional encoding. We also believe that the coding of BC in FALL (see [20]) is prone to the same problem as LAL that precludes a single-typed compositional encoding of BC.<sup>6</sup>

- A more difficult problem is to investigate the translation light logics into BC. It is not clear to us whether one can do fundamentally better than the Turing Machine encoding as it appears that one will have to rely on Gödel numbering the proof-nets. Is it really possible to represent a proofnet as a giant integer, and simulate normalization by arithmetic on that integer?
- Can some sort of light CPS-conversion be used to control evaluation order, thus generating an alternate (and perhaps prettier) solution? We have tried working with this approach, and still have encountered the problems with linear modalities that we describe above.
- Following the evaluation of  $BC^*$  in LOGSPACE and its inductive-type embedding in LAL, is there a good characterization of the fragment of LAL that normalizes in LOGSPACE?

Donald Knuth was once quoted as having said, “I’m glad I got into computer science early, when the problems were easier.” Having done exactly that, Church and Turing basically had to deal with merely getting their “computations” to run, and any slowdown in their simulations was acceptable. Computer scientists have often run aground answering the question, “does the technique scale *up*?”—but here, in contrast, we want to scale *down* to time-bounded computation. And in that more constrained world, the restrictions on data representation and data access present formidable obstacles to the translation between programming languages having similar expressive power.

**Acknowledgements:** We appreciate the lengthy and fruitful discussions we have had with Alan Bawden, Luca Roversi, Luke Ong, and Kazushige Terui in the research leading to this paper.

## 7. REFERENCES

- [1] A. Asperti. Light affine logic. In *Proc. 13th Ann. IEEE Symp. Logic in Comput. Sci.*, pages 300–308, 1998.
- [2] A. Asperti and L. Roversi. Intuitionistic light affine logic (proof-nets, normalization complexity, expressive power, programming notation). *ACM Transactions on Computational Logic*, 3(1):1–39, Jan. 2002.
- [3] A. Beckmann and A. Weiermann. A term rewriting characterization of the polytime functions and related complexity classes. *Archive for Mathematical Logic*, 36:11–30, 1996.
- [4] S. Bellantoni and S. Cook. A new recursion-theoretic characterization of the polytime functions. *Computational Complexity*, 2:97–110, 1992.
- [5] R. Bird, G. Jones, and O. de Moor. More haste, less speed: lazy versus eager evaluation. *J. Funct. Programming*, 7(5):541–547, Sept. 1997.
- [6] H. B. Curry and R. Feys. *Combinatory Logic*, volume 1. North-Holland, 1958.
- [7] J.-Y. Girard. Linear logic. *Theoret. Comput. Sci.*, 50:1–102, 1987.
- [8] J.-Y. Girard. Light linear logic. *Inform. & Comput.*, 143:175–204, 1998.
- [9] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and types*, volume 7 of *Cambridge tracts in theoretical computer science*. Cambridge University Press, 1989.
- [10] M. Hofmann. Linear types and non-size-increasing polynomial time computation. *Theoret. Comput. Sci.* To appear.
- [11] M. Hofmann. *Type Systems for Polynomial-time Computation*. PhD thesis, Technischen Universität Darmstadt, 1998.
- [12] W. A. Howard. The formulae-as-types notion of construction. In J. R. Hindley and J. P. Seldin, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 479–490. Academic Press, 1980. An earlier version was privately circulated in 1969.
- [13] N. D. Jones. The expressive power of higher-order types, or life without CONS. *J. Funct. Programming*, 11(1):55–94, Jan. 2001.
- [14] P. J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, Jan. 1964.
- [15] D. Leivant. Finitely stratified polymorphism. *Inform. & Comput.*, 93(1):93–113, July 1991.
- [16] D. Leivant. Stratified functional programs and computational complexity. In *Conf. Rec. 20th Ann. ACM Symp. Princ. of Prog. Langs.*, pages 325–333, 1993.
- [17] P. Møller Neergaard and H. G. Mairson. LAL is square: Representation and expressiveness in light affine logic. In *Proc. Workshop on Implicit Computational Complexity*, July 2002.
- [18] 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.
- [19] L. Roversi. Light affine logic as a programming language: a first contribution. *Int’l J. Foundations Comput. Sci.*, 11(1):113–152, 2000.
- [20] L. Roversi. Flexible Affine Light Logic. Submitted, 2001.
- [21] K. Terui. Light affine lambda calculus and polytime strong normalization. In *Proc. 16th Ann. IEEE Symp. Logic in Comput. Sci.*, pages 209–220, June 2001.

<sup>6</sup>The paper [20] claims the existence of such an encoding, but does not handle the simulation of safe composition correctly.