

A Functional Language for Logarithmic Space

Peter Møller Neergaard*

Mitchom School of Computer Science
Brandeis University
Waltham, MA 02454, USA
`turtle@achilles.linearity.org`

Abstract. More than being just a tool for expressing algorithms, a well-designed programming language allows the user to express her ideas efficiently. The design choices however effect the efficiency of the algorithms written in the languages. It is therefore of importance to understand how such choices effect the expressibility of programming languages.

The paper pursues the very low complexity programs by presenting a first-order function algebra BC_ε^- that captures exactly LF , the functions computable in logarithmic space. This gives insights into the expressiveness of recursion. Moreover, it can be useful for the automatic analysis of programs' resource usage and the separation of complexity classes.

The important technical features of BC_ε^- are (1) a separation of variables into safe and normal variables where recursion can only be done over the latter; (2) linearity of the recursive call; and (3) recursion with a variable step length (course-of-value recursion). Unlike formulations of LF via Turing Machines, BC_ε^- makes no references to outside resource measures, e.g., the size of the memory used. This appears to be the first such characterization of LF -computable functions (not just predicates).

The proof that all BC_ε^- -programs can be evaluated in LF is of separate interest to programmers: it trades space for time and evaluates recursion with at most one recursive call without a call stack.

1 Introduction

Let thy speech be short, comprehending much in a few words.

From the Aprocrypha

Programmers are used to ask “how fast is my algorithm?”, but an equally important question is “how fast is my programming language?”. While Church-Turing thesis states that all general-purpose programming languages can solve the same problems, they do not necessarily do so equally efficiently. For instance, Pippenger [37] proves that LISP with assignments is faster than pure LISP for a particular online permutation problem. Jones [20] proves that with increasing order of higher-order functions, functional programs can solve increasingly larger classes of problems. There is a similar body of work for typed languages,

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

e.g., increasingly complex intersection types allow increasingly larger classes of problems to be solved [23, 31].

Similarly, starting with the seminal papers by Immerman [17] and Bellantoni and Cook [6], the field of *implicit computational complexity* has blossomed providing—to paraphrase Landin—the next 700 languages for mainly polynomial time, e.g., [13, 10, 2, 9, 38, 24, 25, 34, 33]. These languages all capture their respective complexity class by internal limitations rather than through external measures. While this in most cases makes for cumbersome programming languages, the languages provide precise insights into how programming constructs affect complexity. For instance, the present paper gives a language, BC_ε^- , for the functions computable in for logarithmic space (LF) through a syntactic restriction on course-of-value recursion combined with linearity. This suggests duplicate recursive calls as a potential way for programs to break the straightjacket of LF.

These characterizations can be useful for complexity theory: Computer scientists and mathematicians have for decades sought to answer what classes of problems can be computed with a given amount of resources. Particular notorious in this line of research is the separation problems asking whether these classes are different, i.e., are there problems solvable in polynomial time (PF) that are not in LF, or problems in non-deterministic polynomial time (NPF) that are not in PF. While everybody would believe so—and for instance trust the security of their ATM cards on it—sadly nothing has been proved yet.

The traditional characterization of complexity classes via Turing machines is appealing to programmers: the model is easy to understand and standard programming tricks can be used. An external measure is however like splitting coal from diamonds by the look: it the job gets done, but you gain no insight to why the two are different.

In contrast, comparing the present characterization of LF with the characterization B of PF due to Bellantoni and Cook [6],¹ reveals linearity as the key difference between PF and LF. The much, much harder (and much more interesting) question is of course whether any function in PF truly needs nonlinearity.

The implicit characterizations also hold the promise of a Shangri-La where compilers reason about the complexity of the programs they compile, e.g., [15]. Such compilers would be useful for global computing where a host machine needs guarantees about the running time of a job shipped from different computer. BC_ε^- could be instrumental for more refined bounds on functional programs.

1.1 BC_ε^- in a Nutshell

On surface, BC_ε^- is a slight (hence the ε in the name) generalization of the function algebra BC^- developed by Murawski and Ong [32]. Its merits are that it highlights the limitations of LF, shows the linearity as a potential distinction between LF and PF, and is less cumbersome to use as course-of-value recursion is more natural than primitive recursion used in [32, 6].

¹ B uses a more restricted recursion principle than BC_ε^- . One can however use the recursion principle of BC_ε^- without breaking the PF soundness or completeness of B.

Like its ancestors B and BC^- , BC_ε^- uses a boiled down version of primitive recursion which we (for lack of a shorter name) call *safe affine course-of-value recursion*: (1) in the recursive step, recursion over the recursive value is prohibited (*safe recursion*), (2) the recursive value can be used only once in the recursive step function (*affinity*), but (3) we can skip steps in the recursive chain (*course-of-value recursion* rather than *primitive recursion*). Point (1) is accomplished through a division of the functions arguments into *normal* and *safe* arguments (divided by $:$); the recursion can only be done over the normal arguments and the recursive value is provided to the step function as a safe function. This is originally due to Bellantoni and Cook [6].

In effect, BC_ε^- has a very limited course-of-values as a different step length is allowed at each step of the recursion, e.g., we can recurse through² $f(10001111 :)$, $f(1000 :)$, $f(10 :)$, $f(1 :)$, $f(0 :)$ where the step length is 4, 2, 1, 1. This is accomplished by having a separate function computing the step length from the normal arguments. Syntactically, we express that through the following recursion principle

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

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

1.2 Related Work

Bellantoni and Cook's B [6] captures exactly the functions computable in polynomial time. BC_ε^- is an affine version of B due to Murawski and Ong [32]. It is an attempt to connect B with Girard's characterization of PF through light logic [10, 2, 3] (later it was established that such a connection does not exist as B is only in PF when evaluated call-by-value, see [5, 30]). While Ong and Mairson establish that BC^- can be evaluated in LF (this appears in the unpublished [30]), it is unclear whether BC^- is LF complete. This is solved by BC_ε^- which is both LF-complete and sound; for the soundness we reuse the basic idea of Ong and Mairson, but make it clearer by establishing a connection to tail recursion.

The difference between BC_ε^- and BC^- is the recursion principle. BC_ε^- uses course-of-value recursion, while BC^- uses the more restricted primitive recursion. Combined with affinity this is a true party-killer as it imposes a quantum effect on the recursive result: you can ask for its value, but in doing so you lose the value. Thus, only one bit of information can determine the control in the next step of the recursion. This seems to prevent BC^- from being LF-complete.

To elaborate on the possible LF-incompleteness of BC^- , it is useful to consider the extremely simple characterization of LF due to Jones [18]. He establishes that the problem $DAG \cap DGAP$ is LF-complete under rudimentary reductions [1]. The problem can be stated as follows: given a directed acyclic graph where each

² Here, and throughout, function arguments for BC_ε^- are given in binary.

vertex has at most one outgoing edge; is there a path from the first to the last vertex? Notice that this is easily checkable in LF and it is easy to express the algorithm as a BC_ε^- -function. The author however conjectures that it cannot be done in BC^- .

The reader might faint and think “why another characterization?” when the literature already contains a plethora characterizations of the LF-computable *predicates* or, at best, size-bounded functions. The difference between predicates and functions is however not at all immaterial: We can easily express all LF-computable functions as a predicate on numbers, e.g., addition is $\{(x, y, z) \mid z = x + y\}$. However, due to the sublinear bound on space, we cannot not construct a Turing machine that computes a witness given some of the parameters. With simple restrictions, BC_ε^- therefore goes further and characterize the full class of computable functions: after all most people use computers for more than yes/no-questions. Some of the existing characterizations are:

- (1). Bellantoni [7] shows that restricting B to unary numbers is exactly the non-size increasing LF-computable functions. This in particular includes all LF-computable predicates. The result fundamentally stems from unary numbers using only logarithmic space when represented binary. It does therefore not cast any light on the difference between PF and LF.
- (2). Goerdt [11] characterizes LF-predicates (and other complexity classes) in finite model theory. Jones [20] recasts the results in terms of read-only (`cons-less`) functional program using only `fold` to do iteration. Unlike BC_ε^- they cannot produce big output (being read-only). On a technical note, it is interesting to see that they allow pairs of values; this is a data type that does not seem to be directly encodable³ in BC_ε^- as the affinity prevents accessing both parts of the pair.
- (3). Voda [38] and Kristiansen [24] have characterizations of the LF-computable predicates based on restricted imperative programs with loops.

Finally a comparison should be made to the standard approach to separation through complete problems, i.e., problems that are provably the hardest within a complexity class. Like characterizations through implicit computational complexity complete problems say something intrinsic about the nature of the complexity class. They do however not necessarily tell how the computation is limited; this is why the author find implicit characterizations more appealing.

A full version of this extended abstract is available as a technical report [29] with SML implementations of the programs [28].

2 Preliminaries and BC_ε^-

We assume that the reader is familiar with standard complexity theoretic notions like complexity classes, hardness, and complete problems [35, 19]. We also

³ As we later prove that all LF Turing machines are representable in BC_ε^- an indirect encoding of pairs exists.

assume working knowledge of standard ML [36]. Being computer scientists, the concrete representation of numbers is of importance. We use \mathbb{N}_1 to denote the natural numbers in unary, i.e., a string 1s with length n , and \mathbb{N}_2 for natural numbers in binary. In both cases, we use ε to explicitly denote the empty string corresponding to 0. We write bitstrings of the bits b_1, \dots, b_k as $b_k \cdots b_1$ with b_k the most significant bit. The length function on numbers is $|\cdot|$. We use $\mathbb{N}^{k,l}$ for $\mathbb{N}^k \times \mathbb{N}^l$.

We first introduce our LF-language BC_ε^- . It is given as a function algebra. We divide the function arguments into *normal* and *safe* arguments (syntactically distinguished by a $:$) where recursion is only possible over the former and the latter can be used at most once. use a

Definition 1. *We use \mathbb{N}_2 as out input and output domain.*

- (1). *We define the set of base functions to be the functions: $\mathbf{0}(:) = \varepsilon$; $\mathbf{p}(: \varepsilon) = \varepsilon$ and $\mathbf{p}(: yb) = y'$; $\pi_j^{m,n}(x_1, \dots, x_m : x_{m+1}, \dots, x_{m+n}) = x_j$ with $1 \leq j \leq m+n$; $\mathbf{s}_0(: y) = y\mathbf{0}$; $\mathbf{s}_1(: y) = y\mathbf{1}$; and $\mathbf{c}(: y_1\mathbf{1}, y_2, y_3) = y_2$ and $\mathbf{c}(: y_1, y_2, y_3) = y_3$ otherwise.*
- (2). *Let the following functions be given: $f : \mathbb{N}_2^{m,n} \rightarrow \mathbb{N}_2$, $g_1, \dots, g_m : \mathbb{N}_2^{m,0} \rightarrow \mathbb{N}_2$, and $h_1 : \mathbb{N}_2^{m,n_1} \rightarrow \mathbb{N}_2, \dots, h_n : \mathbb{N}_2^{m,n_n} \rightarrow \mathbb{N}_2$. Let $n \geq n_1 + \dots + n_n$ and define safe affine composition of the functions as the following function in $\mathbb{N}_2^{m,n} \rightarrow \mathbb{N}_2$:*

$$(f \circ \langle g_1, \dots, g_m : h_1, \dots, h_n \rangle)(x_1, \dots, x_m : y_1, \dots, y_n) = f(g_1(\mathbf{x} :), \dots, g_m(\mathbf{x} :), h_1(\mathbf{x} : \mathbf{y}_1), \dots, h_n(\mathbf{x} : \mathbf{y}_n))$$

where $\mathbf{x} = x_1, \dots, x_m$ and $\mathbf{y}_1, \dots, \mathbf{y}_n$ is a division of the variables y_1, \dots, y_n such that each y_i occurs at most once in any of the vectors $\mathbf{y}_1, \dots, \mathbf{y}_n$.

- (3). *Given functions $g : \mathbb{N}_2^{m-1,n} \rightarrow \mathbb{N}_2$, $h_0, h_1 : \mathbb{N}_2^{m,1} \rightarrow \mathbb{N}_2$, $d_0, d_1 : \mathbb{N}_2^{m,0} \rightarrow \mathbb{N}_2$, we define the safe affine course-of-value recursion of the functions (written $\mathbf{rec}(g, h_0, \delta_0, h_1, \delta_1)$) to be the function $f : \mathbb{N}_2^{m,n} \rightarrow \mathbb{N}_2$ defined as follows:*

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

- (4). *The function algebra BC_ε^- is the least set of functions over the integers \mathbb{N}_2 containing the base functions and closed under safe affine composition and safe affine course-of-value recursion.*

Remark 1. The function algebra \mathbf{B} of [6] is obtained by dropping the affinity in the composition and recursion scheme, i.e., by using

$$(f \circ \langle g_1, \dots, g_m : h_1, \dots, h_n \rangle)(\mathbf{x} : \mathbf{y}) = f(g_1(\mathbf{x} :), \dots, g_m(\mathbf{x} :), h_1(\mathbf{x} : \mathbf{y}), \dots, h_n(\mathbf{x} : \mathbf{y}))$$

and letting $f = \text{rec}(g, h_0, d_0, h_1, d_1)$ be

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

Notation 1. When f is nullary (i.e., a constant), we write f for $f \circ \langle : \rangle : \mathbb{N}^{m,n} \rightarrow \mathbb{N}$ for any m and n . We write $\text{rec}(g, h, \delta)$ for $\text{rec}(g, h, \delta, h, \delta)$.

The syntactic restrictions impose a bound on the size of the output.

Lemma 1 ([6, Lem. 4.1]). *Let $f \in BC_\varepsilon^-$ be function. There is a monotone polynomial q_f such that*

$$|f(\mathbf{x} : \mathbf{y})| \leq q_f(|\mathbf{x}|) + \max_i |y_i| \quad . \quad (1)$$

3 LF Completeness

We will now show that BC_ε^- is complete for LF-computations. We do so by showing that given any LF Turing machine we can define the following function in BC_ε^- :

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

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

Notation 2. We consider Turing machines with a read-only input tape (denoted I) of length n , a work tape (denoted W) with $O(\log n)$ symbols, and a write-only output tape (denoted O) infinite to the right. We assume that the program will never try to move beyond the limits of the tapes. The machine’s program consists of labeled instructions $l : I$ consecutively numbered as $0, \dots, S-1$ where I is one of the following: left_I , left_W , right_I , right_W , $\text{if}_I \text{ then } l' \text{ else } l''$, $\text{if}_W \text{ then } l' \text{ else } l''$, and $\text{write}_W b$ and $\text{write}_I b$ with $b = 0, 1$. We assume that the program idles on a single state when there is no more output.

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

We can now turn to the function T ; the function is presented in pseudo-code in Fig. 1. The remainder of this section is devoted to showing that the function can be encoded in BC_ε^- . This has two parts: (1) representing the functions in Fig. 1 in BC_ε^- , (2) turning the recursion over multiple variables into a recursion over a single variable. Starting with the last point, we observe the following

$$\begin{aligned}
T(-1, s, w, i, j, x) &= 1 \\
T(t, s, w, i, j, x) &= \text{let } b_I = \text{bit}(i : x) \text{ in} \\
&\quad \text{let } b_w = \text{bit}(j : \text{unary2bin}(w :)) \text{ in} \\
&\quad T(t - 1, S(b_w, b_I, s :), W(w, j, s :), I(i, s :), J(j, s :), x :) \\
S(b_w, b_I, s :) &= \begin{cases} l' & \text{if } I_T = \text{if}_T \text{ then } l' \text{ else } l'' \text{ and } b_T = 1 \\ l'' & \text{if } I_s = \text{if}_T \text{ then } l' \text{ else } l'' \text{ and } b_T = 0 \\ s + 1 & \text{otherwise} \end{cases} \\
W(w, j, s :) &= \begin{cases} \text{bin2unary}(\text{set}_b(j, \text{unary2bin}(w :)) :) & \text{if } I_s = \text{write}_w b \\ w & \text{otherwise} \end{cases} \\
I(i, s :) &= \begin{cases} i + 1 & \text{if } I_s = \text{right}_I \\ i - 1 & \text{if } I_s = \text{left}_I \\ i & \end{cases} \\
J(j, s :) &= \begin{cases} j + 1 & \text{if } I_s = \text{right}_w \\ j - 1 & \text{if } I_s = \text{left}_w \\ j & \end{cases} \\
\text{set}_b(n, b_k \cdots b_0 :) &= \begin{cases} b_k \cdots b_{|m|+1} b b_{|m|-1} \cdots b_0 & \text{when } 0 \leq k \leq |m| \\ b b_{k-1} \cdots b_0 & \text{otherwise} \end{cases}
\end{aligned}$$

Fig. 1. The function T simulating a LF Turing machine.

bounds: $0 \leq s < S$, $0 \leq i < n$, $0 \leq j < \log_2 n \leq n$. Since the Turing machine cannot twice be in the same state without doing an infinite loop, the maximum number of steps the machine can take before idling infinitely is $N = S \cdot n \cdot n \cdot n^c = S \cdot n^{c+2}$. We can encode the tuple (t, s, w, i, j) uniquely as the number: $t \cdot N + (((w \cdot n) + i) \cdot n + j) \cdot S + s$. Consequently, each step is represented by decreasing the encoding with $N + (((w - w') \cdot n) + (i - i')) \cdot n + (j - j') \cdot S + (s - s')$ where w' , i' , j' , and s' represent the new state. By using unary representation this corresponds to a simple right shift—easily caught with the course-of-value recursion scheme.⁴ We therefore show how to do the functions in Fig. 1 using unary numbers.

Proposition 1. *Let m and n be numbers in binary. Right shift $\text{shift}^R(m : n)$ of m by $|n|$ and selection of bit $|n|$ from m are definable in BC_ε^c .*

⁴ Note that binary representation will not work as the difference between two numbers is not necessarily a string of most significant bits.

Proposition 2. *Let b be either 0 or 1. The following function is representable in BC_ε^c :*

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

Note that set_b does not change the length of the output. We then continue with arithmetic:

Proposition 3. *Let m and n be integers in unary notation. Then the following functions are representable in BC_ε^c : (1) $\text{plus}(m : n) = m + n$, (2) $\text{minus}(n : m) = \max(m - n, 0)$, (3) $\text{mult}(m, n :) = m \cdot n$, (4) $\text{div}(m, n :) = \lfloor m/n \rfloor$ where $n \neq 0$, (5) $\text{mod}(m, n :) = m \bmod n$ where $n \neq 0$, (6) $\text{zero?}(m :) = 1$ iff $m = 0$, and (7) $\text{<?}(m, n :) = 1$ iff $m < n$.*

The representations are straightforward (see [29]). It is however worth noticing how the course-of-value recursion principle of BC_ε^c yields a very simple definition of division:

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

An unfortunate limitation of the syntactic restrictions of BC_ε^c is that even though only one of the branches is taken in a conditional, the two branches cannot share any safe arguments. In a recursion this prevents using a conditional to choose between two different actions on the recursive result. At the cost of having the conditional controlled by a normal variable, we can regain a conditional that shares the variables of the two branches:

Lemma 2. *Let $f, g : \mathbb{N}^{0,1} \rightarrow \mathbb{N}$ be two functions representable in BC_ε^c . Then the following function is representable in BC_ε^c : $\text{COND}_{f,g}(m : n) = f(: n)$ when $m \bmod 2 = 1$ and $\text{COND}_{f,g}(m : n) = g(: n)$ when $m \bmod 2 = 0$.*

Notation 3. We use all capital letters for COND since it is macro over BC_ε^c -functions rather than a function in itself.

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

Lemma 3. *There is a BC_ε^c function $\text{rev} : \mathbb{N}_2 \times \mathbb{N}_1 \rightarrow \mathbb{N}$ that reverses a bit string, i.e., given a number n in unary $\text{rev}(b_k \cdots b_0, n :) = b_0 b_1 \cdots b_{k-n}$.*

Even though we are using unary numbers in the recursion, updating the work tape is simpler to describe if the work tape is in binary. We therefore introduce functions to convert between binary and unary notation. This involves computing the logarithm which the author has failed to represent in BC^c .

Lemma 4. *Let m be an integer in unary notation. Then there is a BC_ε -function $\log(m :) = \lfloor \log_2 m \rfloor + 1$ when $m \geq 0$ and $\log(0 :) = 0$.*

Proof. We define \log as follows:

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

which is implemented by the following BC_ε program:

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

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

Lemma 5. *Let $m, n \in \mathbb{N}$ be integers coded in unary representation. The following functions are representable in BC_ε : (1) $\mathbf{power?}(m, n :) = 1$ if, and only if, m is a power of n , (2) $\mathbf{largest_power}(m, n :) = \max(\{n^i \leq m \mid i \in \mathbb{N}\} \cup \{0\})$*

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

Proposition 4.

- (1). *The function $\mathbf{unary2bin} : \mathbb{N}_1 \rightarrow \mathbb{N}_2$ is representable in BC_ε .*
- (2). *Let $\mathbf{bin2unary}(m, n :)$ be the unary representation of m provided that $m \leq n$. The function $\mathbf{bin2unary} : \mathbb{N}_2 \times \mathbb{N}_1 \rightarrow \mathbb{N}_1$ is representable in BC_ε .*

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

$$\mathbf{unary2bin}(m :) = \mathbf{rev}(\mathbf{unary2bin}'(m :), 1 :)$$

$$\mathbf{unary2bin}'(m :) = \begin{cases} 1 & \text{if } m = \varepsilon \\ \mathbf{s}_0(\mathbf{unary2bin}'(\lfloor m/2 \rfloor :)) & \text{if } m \geq 1, \text{ mod}(m, 2 :) = 0 \\ \mathbf{s}_1(\mathbf{unary2bin}'(\lfloor m/2 \rfloor :)) & \text{if } m \geq 1, \text{ mod}(m, 2 :) = 1 \end{cases}$$

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

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

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

Correctness follows by induction on m . □

Proposition 5. *The function T given in Fig. 1 is representable in BC_ε^c .*

Proof. Except for the step functions S, W, I, and J, we have seen how to represent the functions in Fig. 1 in BC_ε^c . We have also seen how to turn the multivariable recursion into recursion over one variable.

As for the functions S, W, I, and J, the conditions, e.g., “ $I_s = \text{right}_T$ ” can all statically be turned into comparisons on the numerical value of s using $<?$. In W we exploit that the number of configurations of the work tape is $2^{c \cdot \log_2 n} = n^c$. We augment W with a fourth argument, the input tape x . Taking w' to be the updated work tape `setb(j, unary2bin(w :))`, we can therefore compute its unary representation as

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

T is now obtainable through affine composition. □

We conclude with the following corollary:

Corollary 1. *For any LF Turing Machine T, there is BC_ε^c function $f_T : \mathbb{N}^{1,0} \rightarrow \mathbb{N}$ computing the same function: T on input x produces the output tape y if, and only if, $f_T(x :) = 1y$.*

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

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

Proof. The function f_T is constructed immediately as

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

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

Precise time and space bounds of course depend on how the BC_ε^c are represented; we consider only outputting their textual representation. An inspection

of the construction in the proof shows that S, W, I, and J can be output with one scan over the program. For each instruction we need to output the unary representation of the instruction number. (In the case of S we might also need to output the unary representation of the label.) This can be done in time $O(S^2)$. The $O(\log S)$ space uses arises from keeping track of the instruction.

4 LF Soundness

Having shown that BC_ε^- is LF complete, we continue to show that it is also LF sound. We reuse a construction by Ong and Mairson in the unpublished work [30]. The construction depends on two facts:

- (1). The output and all intermediate values are bound by (1); this is established in [6].⁵ We therefore need only a counter of size $O(\log |\mathbf{x}; \mathbf{y}|)$ to iterate over the bits of the output.
- (2). One bit of the output can be produced storing only one bit of the safe arguments and a constant number of bits from the normal arguments. This is straightforward except for the case of safe affine recursion.

In the case of safe of affine recursion we use a form of computational amnesia where we evaluate the recursion without storing the call stack until we can produce a bit. We remember the bit and—as there is no stack of return addresses—restart the recursion from the top. We will however be able to produce a bit after fewer recursive calls. We illustrate this with the parity function $P = \text{rec}(g, h_0, h_1)$:

$$g(:) = 0 \quad h_0(x : r) = \text{co}\langle : r, 1, 0 \rangle \quad h_1(x : r) = \text{co}\langle : r, 0, 1 \rangle .$$

A standard evaluation of $P(10111 :)$ would result in the following unwinding

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

With the computational amnesia we go through

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

We *remember* the result (and its recursion depth, 5) and *restart* the computation of $P(10111 :)$. We now only need to go through the calls

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

as we can lookup the stored value of $P(\varepsilon :)$. We remember that the value 1 was found at recursion depth 4. We restart the computation and keep repeating until we eventually find $P(10111 :) = 0$.

⁵ In [6, 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.

```

type bit = int option
type input = int -> bit
type program-m-n =  $\underbrace{\text{input} \rightarrow \dots \rightarrow \text{input}}_{m+n \text{ times}} \rightarrow \text{int} \rightarrow \text{bit}$ 

fun zero (bt : int) = NONE
fun succ0 (y1 : input) (bt : int) =
  if bt = 0 then SOME 0 else y1 (bt - 1)
fun succ1 (y1 : input) (bt : int) =
  if bt = 0 then SOME 1 else y1 (bt - 1)
fun pred (y1 : input) (bt : int) = y1 (bt + 1)
fun cond (y1 : input) (y2 : input) (y3 : input) (bt : int) =
  let fun boolFromBit NONE = false
      | boolFromBit (SOME b) = (b = 1)
  in
    if boolFromBit (y1 0) then y2 bt else y3 bt
  end
(* Translation of  $\pi_j^{m,n}$ ; first when  $j \leq m$  and then when  $m > j$  *)
fun proj-m-n-j (x1 : input) ... (xm : input)
  (y1 : input) ... (yn : input) (bt : int) = xj bt
fun proj-m-n-j (x1 : input) ... (xm : input)
  (y1 : input) ... (yn : input) (bt : int) = y(j - m) bt

```

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

Our goal in this section is to formalize the idea outlined above and prove that it stays within LF. The core is to show that we in LF can find any given bit of the output by only storing a fixed number of bits of the inputs to each subexpression. We do this by translating each BC_ε^r -expression into a SML-expression that finds any given bit of the output by querying individual bits of the input.

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

Finally, the translation of safe affine course-of-value recursion is given in Fig. 4. The construction embodies the idea outlined above: we start out computing the requested bit at depth 0. If there is a request for a bit from the recursive call, the function `recursiveCall` takes care of updating our goal and restarting the computation by raising the exception `Restartn` (where n is a unique identifier of the recursion). When we succeed in computing the goal bit—either because we reached the base case or because the recursive call was cached—we update the cache kept in `result`. We keep starting the search from depth 0 until we have succeed in computing the bit at depth 0. The last remaining part,

```

fun comp (f : program- $M-N$ )
  (g1 : program- $m-0$ ) ... (gM : program- $m-0$ )
  (h1 : program- $m-n_1$ ) ... (hN : program- $m-n_N$ )
  (x1 : input) ... (xm : input)
  (y1 : input) ... (yn : input) (bt : int) =
let fun X1 bt = g1 x1 ... xm bt
  :
  fun XM bt = gM x1 ... xm bt
  fun Y1 bt = h1 x1 ... xm yi1,1 ... yi1,l1 bt
  :
  fun YN bt = hN x1 ... xm yiN,1 ... yiN,lN bt
in
  f X1 ... XM Y1 ... YN bt
end

```

Fig. 3. Translation of safe affine composition into an SML function. The division of the variables y_1, \dots, y_n between the functions h_1, \dots, h_N computing safe arguments is given by the vectors of indices i_1, \dots, i_N . They satisfy the relation $\{i_{1,1}, \dots, i_{1,l_1}, i_{2,1}, \dots, i_{N-1,l_{N-1}}, i_{N,1}, \dots, i_{N,l_N}\} \subseteq \{1, \dots, n\}$ and that there is at most one $i_{j,j'}$ for any given number in $\{1, \dots, n\}$.

is to notice that we can find the length of the displacement function by simply querying for the bits 0, 1, 2, etc. until we reach the index where the bit is not-existing.

We capture the above discussion in the following definition of the ML function $[f]$ which computes a requested bit of the BC_ε function f :

Definition 2. Let f be a function of BC_ε with m normal and n safe arguments. The ML function $[f]$ of type $\text{program-}m-n$ is defined inductively based on Figs. 2-4: $[0] = \text{zero}$, $[s_b] = \text{succb}$, $[p] = \text{pred}$, $[c] = \text{cond}$, $[\pi_j^{m,n}] = \text{proj-}m-n-j$, $[f \circ \langle g_1, \dots, g_M : h_1, \dots, h_N \rangle] = \text{comp } [g_1] \cdots [g_M] [h_1] \cdots [h_N]$, and $[\text{rec}(g, h_0, \delta_0, h_1, \delta_1)] = \text{saferec } [g] [h_0] [\delta_0] [h_1] [\delta_1]$.

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

- The functions are tail recursive as defined, e.g., by Jones [20]. Consequently there is a fixed maximal depth of the stack for any program.
- Each stack entry is representable in logarithmic space on the inputs.
- The function succeeds and returns the correct bit provided that the input functions are correct and succeed without raising an exception $\text{Restart}n$.

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

Notation 4.

```

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

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

```

Fig. 4. Translations of safe affine recursion

- We adapt the convention of Barendregt [4] and use \equiv for the syntactic equivalence of functions, i.e., $e \equiv e'$ when the expressions e and e' are equivalent up to α -equivalence.
- We assign an *abstraction label* a to each function abstraction; we write $\text{fn}^a \ x \Rightarrow e$. We will refer to the abstraction labels as pairs of the function name and the parameter, e.g., `succ0` above has two abstraction labels: $\langle \text{succ0}, y1 \rangle$ and $\langle \text{succ0}, \text{bit} \rangle$.
We recall that when evaluating the program every function applied—even when it is the result of evaluating an expression—is constructed by one of the abstraction occurrences $\text{fn}^a \ x \Rightarrow e$ in the program.

We want to establish that the program is tail recursive. Various notions exist in the literature [21, 22, 8, 20]. We follow Jones [20] and adapt a slightly more liberal than the usual notion: we allow the call stack to grow as long as there is never two stack entries for the same abstraction.

Definition 3.

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

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

The partial order in the above definition ensures that the program can be evaluated without having more than one entry per abstraction on the call stack. Rather than a fully formal proof based on the formal definition of ML [27], we rely on an informal understanding of ML evaluation. Furthermore, we make the following idealized assumptions about ML: (1) Arithmetic can be done with arbitrary precision. In particular none of the functions, $+$, $-$, $*$, $=$, and $\langle \rangle$, raise any exceptions. (2) The memory is arbitrarily large. In particular, the function `ref` will never raise an exception. (3) The basic functions are implemented tail recursively, i.e, $+$, $-$, $*$, $=$, $\langle \rangle$, $\#$, $!$, and `ref` are all tail recursive.

We can now establish that the functions used in the composition fulfill the conditions for being evaluated tail recursively.

Proposition 6. Let $F \in \mathbb{N}^{m,n}$ be a function of BC_{ε}^* . Its encoding into an SML expression is $\lceil F \rceil \equiv \text{fn}^{a_1} \ \text{in}p1 \Rightarrow \dots \Rightarrow \text{fn}^{a_n} \ \text{in}pn \Rightarrow \text{fn}^{a_{n+1}} \ \text{bt} \Rightarrow e$ of type `program-m-n`. We consider an application $P = \lceil F \rceil \ \text{exp}1 \ \dots \ \text{exp}l$ where

$l = m + n$. Each of the argument expressions \mathbf{exp}_i can evaluate to abstractions with abstraction labels in the set A_i . Furthermore, none of the abstraction labels in $A_1 \cup \dots \cup A_l$ occurs in \mathbf{P} . There exists a partial order \succeq on the abstraction labels with the following properties:

- (1). Consider any abstraction $\mathbf{fn}^{a'} \mathbf{x} \Rightarrow e'$ in \mathbf{P} and any application $e'_1 e'_2$ occurring in e' . For any abstraction $\mathbf{fn}^{a''} x \Rightarrow e''_1$ that e'_1 can evaluate to, we have either (a) $a' \succ a''_1$ or (b) $a' = a''_1$ and the application is in tail position.
- (2). There is an abstraction label \top_\succeq that is the successor of all other abstraction labels that occur in $\lceil F \rceil$ and $A_1 \cup \dots \cup A_l$.
- (3). It holds that $a_1 \succ \dots \succ a_{l+1}$.
- (4). None of the elements $a'_i \in A_i$ for $1 \leq i \leq l$ has a predecessor in \succeq .

Proof. We use induction on the structure of F inspecting each case. □

Proposition 7. Given any BC_ε^c -function $F \in \mathbb{N}^{M,N}$, its SML-encoding $\lceil F \rceil$ can be evaluated in logarithmic space, i.e., for any $x_1, \dots, x_M, y_1, \dots, y_N \in \mathbb{N}$ we can compute $F(x_1, \dots, x_M : y_1, \dots, y_N)$ using only logarithmic workspace.

Proof. This is accomplished by iteratively (and therefore tail recursively) query $\lceil F \rceil$ bit by bit. Using Prop. 6 the full program is tail recursive. □

In proving the correctness of the algorithm, the only tricky case is the safe affine course-of-value recursion. We start with a technical lemma that simply establishes the notation for the evaluation of safe affine recursion.

Lemma 6. Let a BC_ε^c -function $F = \mathbf{rec}(g, h_0, h_1, d_0, d_1)$ be given. For any normal and safe inputs $x_1 = b_k \dots b_1, x_2, \dots, x_M, y_1, \dots, y_N$ there exists $r \geq 0$ and $D_0, \dots, D_r \geq 1$ such that

- (1). $D_r = 0$
- (2). $D_{i-1} = D_i + d_{b_{D_i}}(z_i, x_2, \dots, x_M) + 1$ when $0 \leq i \leq r - 1$.
- (3). $D_0 \geq m$
- (4). $D_i < m$ when $i \geq 1$
- (5). $F(x_1, \dots, x_M : y_1, \dots, y_N) =$

$$h_{b_{D_r+1}}(z_r, x_2, \dots, x_M :$$

$$h_{b_{D_{r-1}+1}}(z_{r-1}, x_2, \dots, x_M : \dots$$

$$h_{b_{D_1+1}}(z_1, x_2, \dots, x_M : g(x_2, \dots, x_M : y_1, \dots, y_N)) \dots))$$

where $z_r = b_k \dots b_{D_r+2}$.

Notation 5. We write $\mathbf{e} \rightsquigarrow v$ when \mathbf{e} evaluates to v without raising an exception.

Definition 4. Let $v = b_k \dots b_1 \in \mathbb{N}$ be a number.

- (1). The representation of bit i of v is (a) NONE when $i \geq k$ (b) SOME b_{i+1} when $0 \leq i \leq k - 1$.

- (2). The SML-expression `e` of type `int -> bool option` bitwise represents v if, and only if, for any expression e' of type `int` where $e' \rightsquigarrow i$ either (a) $e' \rightsquigarrow R$ where R is the representation of bit i of v , or (b) the evaluation of the application `e e'` raises an exception.

Lemma 7. Evaluating `findLength e` returns $|v|$ when expression `e` bitwise represent v .

We can then state the main proposition for the correctness:

Proposition 8. Let F be a BC_ε -function and let $x_1, \dots, x_M, y_1, \dots, y_N$ be the normal and safe inputs, resp. Let $x_1, \dots, x_M, y_1, \dots, y_N$ be bitwise represented by the SML-expressions `x1, ..., xM, y1, ..., yN`. Assume that `x1, ..., xM` cannot raise exceptions,

- (1). `[F] x1 ... xM y1 ... yN` bitwise evaluates $F(x_1, \dots, x_M : y_1, \dots, y_N)$ raising only exceptions raised by `y1, ..., yN`.
- (2). In evaluating `[F] x1 ... xM y1 ... yN` each of the functions `y1, ..., yN` are called at most once.

Proof. Formally, the two claims are proven simultaneously by induction on the structure of F . For the second claim note that the base functions use each argument as most once, that safe affine composition splits the variables linearly, and that safe affine recursion only uses the safe variables in the base cases.

For the first claim, the only interesting case is safe affine recursion. We use the characterization in Lem. 6 to find the recursion depth r . We do induction over i where $0 \leq i \leq r$ to simultaneously prove that in the evaluation of `saferec`

- (1). `x'` is bitwise correct implementation of z_i .
- (2). the program repeats the same sequence of calls to `recursiveCall`. This is due to the determinism of ML.
- (3). `#depth (!goal)` represents one of the levels established in Lem. 6 in the calls to `recursiveCall`.
- (4). when `#bit (!result) \rightsquigarrow j` and `#depth (!result) \rightsquigarrow D_i` for some i where $0 \leq i \leq r$ and then `#bit (!result)` holds the j th bit for the recursive call at $i + 1$,
- (5). `recursiveCall d` bitwise represents the recursive call.
- (6). if the computation reaches the body of `loop2` requesting the bit corresponding to D_i then either an exception is raised or the computation eventually reaches the end of `loop1` with the requested bit.

In the inductive step, the claim is immediate if `h` finds the correct bit without calling `recursiveCall`. Otherwise, `recursiveCall` is called, `goal` is updated, `Restartn` raised, and the computation reaches the body of `loop2`. The claim follows from the induction hypothesis.

Now, `saferec` is correct by the last point. The gory details are in [29]. □

We notice that the encoding can be produced by scanning over the BC_ε^- -expression keeping a pointer on where we are in the expression and a counter on the number of safe recursions. As we have now proven that the encoding is correct and can be evaluated in LF, we conclude the following.

Theorem 1. *For any function definable in the function algebra BC_ε^- , there is a Turing Machine evaluating the function in LF. Moreover, the Turing Machine can be constructed from the function expression in logarithmic space in the size of the BC_ε^- -expression.*

5 Conclusion and Future Work

We have presented the first implicit characterization of the LF-computable functions. The characterization is appealing because it with two simple measures, affinity and a division into safe and normal arguments, obtains a well-known complexity class. When compared to Bellantoni and Cook's B it highlights linearity as a potential distinction between LF and PF.

A problem with the existing work on automatic analysis of resource usage, is that it only derives bounds that are polynomial. It is likely that BC_ε^- could serve as a first step toward a more realistic analysis. A natural first step would be to allow the conditional to share arguments between two branches. This could be achieved by moving to a resource conscious type system in the style of Hofmann [14] with a linear logic like additive operator.

Another line of research is to continue the investigation of the connection to linear logic which originally motivated BC_ε^- . A first question is whether there is a direct encoding into light affine logic [10, 2, 3] of safe affine course-of-value recursion. This could suggest a lighter than light fragment of light logic corresponding to logarithmic space. Finally, it would be worth investigating how these restrictions affect the context semantics [12, 26] for the proofnets.

Acknowledgments

I am indebted to my advisor Harry Mairson for his encouragement and inspiration, as well as his willingness to comment on all stages of the work. I deeply appreciate the very detailed comments I received from Olivier Danvy and Steve Homer. I value the support from Assaf Kfoury and the Church Project. I thank Alan Bawden, Peter Clote, Neil Jones, Steven Lindell, and Stan Wainer for persistently and patiently answering my questions on existing characterizations of LF.

References

- [1] E. Allender and V. Gore. Rudimentary reductions revisited. Technical Report 91-2, Center for Discrete Mathematics and Theoretical Computer Science (DIMACS) at Rutgers University, New Jersey, 1991.

- [2] A. Asperti. Light affine logic. In *Proc. 13th Ann. IEEE Symp. Logic in Comput. Sci.*, pages 300–308, 1998.
- [3] 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.
- [4] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, revised edition, 1984.
- [5] 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.
- [6] S. Bellantoni and S. A. Cook. A new recursion-theoretic characterization of the polytime functions. *Computational Complexity*, 2:97–110, 1992.
- [7] S. J. Bellantoni. *Predicative Recursion and Computational Complexity*. PhD thesis, University of Toronto, Sept. 30 1992.
- [8] W. D. Clinger. Proper tail recursion and space efficiency. In *Proc. ACM SIGPLAN '98 Conf. Prog. Lang. Design & Impl.*, pages 174–185, 1998.
- [9] P. Clote. A safe recursion scheme for exponential time. In A. Adian and A. Nerode, editors, *Proc. of the 4th International Symposium on Logical Foundations of Computer Science, LFCS '97*, volume 1234 of *LNCS*, pages 44–52, Yaroslavl, Russia, July 1997. Springer-Verlag.
- [10] J.-Y. Girard. Light linear logic. *Inform. & Comput.*, 143:175–204, 1998.
- [11] A. Goerdt. Characterizing complexity classes by higher type primitive recursive definitions. *Theoret. Comput. Sci.*, 100(1):45–66, June 1992.
- [12] G. Gonthier, M. Abadi, and J.-J. Lévy. Linear logic without boxes. In *Proc. 7th Ann. IEEE Symp. Logic in Comput. Sci.*, pages 223–34. IEEE Comput. Soc. Press, 1992.
- [13] M. Hofmann. *Type Systems for Polynomial-time Computation*. PhD thesis, Technischen Universität Darmstadt, 1998.
- [14] M. Hofmann. Linear type and non-size-increasing polynomial time computation. In *Proc. 14th Ann. IEEE Symp. Logic in Comput. Sci.*, July 1999.
- [15] M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *Conf. Rec. POPL '03: 30th ACM Symp. Princ. of Prog. Langs.*, pages 185–197, 2003.
- [16] *Proc. Workshop on Implicit Computational Complexity*, July 2002.
- [17] N. Immerman. Languages that capture complexity classes. *SIAM Journal of Computing*, 16(4):760–778, Aug. 1987.
- [18] N. D. Jones. Space-bounded reducibility among combinatorial problems. *J. Comput. System Sci.*, 11(1):68–85, Aug. 1975.
- [19] N. D. Jones. *Computability and Complexity: From a Programming Perspective*. Foundations of Computing. MIT Press, 1997.
- [20] N. D. Jones. The expressive power of higher-order types, or life without CONS. *J. Funct. Programming*, 11(1):55–94, Jan. 2001.
- [21] A. J. Kfoury. Recursion versus iteration at higher-orders. In *Foundations of Software Technology and Theoret. Comput. Sci., 17th Conf.*, volume 1346 of *LNCS*, pages 57–73. Springer-Verlag, Dec. 1997.
- [22] A. J. Kfoury. Recursion, tail-recursion and iteration at higher-orders. Notes for a paper, 1999.
- [23] A. J. Kfoury, H. G. Mairson, F. A. Turbak, and J. B. Wells. Relating typability and expressibility in finite-rank intersection type systems. In *Proc. 1999 Int'l Conf. Functional Programming*, pages 90–101. ACM Press, 1999.

- [24] L. Kristiansen. New recursion-theoretic characterizations of well-known complexity classes. In ICC '02 [16].
- [25] Y. Lafont. Soft linear logic and polynomial time. *Theoret. Comput. Sci.* to appear.
- [26] H. G. Mairson. From Hilbert spaces to Dilbert spaces: Context semantics made simple. In *22nd Conference on Foundations of Software Technology and Theoretical Computer Science*, 2002.
- [27] R. Milner, M. Tofte, R. Harper, and D. B. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.
- [28] P. Møller Neergaard. An example SML implementation of a LOGSPACE linear BC evaluator, 2003–2004.
- [29] P. Møller Neergaard. BC_ϵ : A recursion-theoretic characterization of LOGSPACE. Technical report, Brandeis University, Mar. 2004. Preliminary version.
- [30] P. Møller Neergaard and H. G. Mairson. How light is safe recursion? Translations between logics of polynomial time. Unpublished note, 2003.
- [31] P. Møller Neergaard and H. G. Mairson. Types, potency, and impotency: Why nonlinearity and amnesia make a type system work. In *Proc. 9th Int'l Conf. Functional Programming*. ACM Press, 2004.
- [32] 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.
- [33] I. Oitavem. A term rewriting characterization of the functions computable in polynomial space. *Archive for Mathematical Logic*, 41(1):35–47, jan 2002.
- [34] I. Oitavem. Characterizing NC with tier 0 pointers. *Mathematical Logic Quarterly*, 50(1):9–17, jan 2004.
- [35] C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, Reading, Mass., 1994.
- [36] L. C. Paulson. *ML for the Working Programmer (2nd ed.)*. Cambridge University Press, 1996.
- [37] N. Pippenger. Pure versus impure lisp. In *Conf. Rec. POPL '96: 23rd ACM Symp. Princ. of Prog. Langs.*, pages 104–109, 1996.
- [38] P. J. Voda. Two simple intrinsic characterizations of main complexity classes. In ICC '02 [16].