Linear Ramified Higher Type Recursion and Parallel Complexity

Klaus Aehlig[1,⋆], Jan Johannsen[2,⋆⋆], Helmut Schwichtenberg[1,⋆⋆⋆], and
Sebastiaan A. Terwijn[3,†]

[1] Mathematisches Institut, Ludwig-Maximilians-Universität München,
Theresienstraße 39, 80333 München, Germany
{aehlig,schwicht}@rz.mathematik.uni-muenchen.de
[2] Institut für Informatik, Ludwig-Maximilians-Universität München
Oettingenstraße 67, 80538 München, Germany
jjohanns@informatik.uni-muenchen.de
[3] Department of Mathematics and Computer Science, Vrije Universiteit Amsterdam,
De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands
terwijn@cs.vu.nl

Abstract. A typed lambda calculus with recursion in all finite types is defined such that the first order terms exactly characterize the parallel complexity class NC. This is achieved by use of the appropriate forms of recursion (concatenation recursion and logarithmic recursion), a ramified type structure and imposing of a linearity constraint.

Keywords: higher types, recursion, parallel computation, NC, lambda calculus, linear logic, implicit computational complexity

1 Introduction

One of the most prominent complexity classes, other than polynomial time, is the class NC of functions computable in parallel polylogarithmic time with a polynomial amount of hardware. This class has several natural characterizations in terms of circuits, alternating Turing machines, or parallel random access machines as used in this work. It can be argued that NC is the class of efficiently parallalizable problems, just as polynomial time is generally considered as the correct formalization of feasible sequential computation.

Machine-independent characterizations of computational complexity classes are not only of theoretical, but recently also of increasing practical interest. Besides indicating the robustness and naturalness of the classes in question, they also provide guidance for the development of programming languages [11].

[⋆] Supported by the DFG Graduiertenkolleg "Logik in der Informatik"

[⋆⋆] Supported by the DFG Emmy Noether-Programme under grant No. Jo 291/2-1

[⋆⋆⋆] The hospitality of the Mittag-Leffler Institute in the spring of 2001 is gratefully acknowledged.

[†] Supported by a Marie Curie fellowship of the European Union under grant no. ERB-FMBI-CT98-3248

R. Kahle, P. Schroeder-Heister, and R. Stärk (Eds.): PTCS 2001, LNCS 2183, pp. 1–21, 2001.
© Springer-Verlag Berlin Heidelberg 2001

The earliest such characterizations, starting with Cobham's function algebra for polynomial time [9], used recursions with explicit bounds on the growth of the defined functions. Function algebra characterizations in this style of parallel complexity classes, among them NC, were given by Clote [8] and Allen [1].

More elegant *implicit* characterizations, i.e., without any explicitly given bounds, but instead using logical concepts like ramification or tiering, have been given for many complexity classes, starting with the work of Bellantoni and Cook [4] and Leivant [14] on polynomial time. In his thesis [2], Bellantoni gives such a characterization of NC using a ramified variant of Clote's recursion schemes. A different implicit characterization of NC, using tree recursion, was given by Leivant [15], and refined by Bellantoni and Oitavem [6]. Other parallel complexity classes, viz. parallel logarithmic and polylogarithmic time, were given implicit characterizations by Bellantoni [3], Bloch [7] and Leivant and Marion [16].

In order to apply the approach within the functional programming paradigm, one has to consider functions of higher type, and thus extend the function algebras by a typed lambda calculus. To really make use of this feature, it is desirable to allow the definition of higher type functions by recursion. Higher type recursion was originally considered by Gödel [10] for the analysis of logical systems. Systems with recursion in all finite types characterizing polynomial time were given by Bellantoni et al. [5] and Hofmann [12], based on the first-order system of Bellantoni and Cook [4].

We define an analogous system that characterizes NC while allowing an appropriate form of recursion, viz. logarithmic recursion as used by Clote [8] and Bellantoni [2], in all finite types. More precisely, our system is a typed lambda calculus which allows two kinds of function types, denoted $\sigma \multimap \tau$ and $\sigma \to \tau$, and two sorts of variables of the ground type ι, the *complete* ones in addition to the usual ones, which are called incomplete for emphasis. A function of type $\sigma \to \tau$ can only be applied to complete terms of type σ, i.e., terms containing only complete free variables.

It features two recursion operators LR and CR, the latter corresponding to Clote's [8] concatenation recursion on notation, which can naturally only be applied to first-order functions. The former is a form of recursion of logarithmic length characteristic of all function algebra representations of NC, and here can be applied to functions of all linear types, i.e., types only built up using ι and $\multimap$. The function being iterated, as well as the numerical argument being recurred on have to be complete, i.e., the type of LR is $\sigma \multimap (\iota \to \sigma \multimap \sigma) \to \iota \to \sigma$ for linear σ.

Our analysis clearly reveals the different roles played by the two forms of recursion in characterizing NC: Logarithmic recursion controls the runtime, in that the degree of the polylogarithm that bounds the runtime depends only on the number of occurrences of LR. On the other hand, concatenation recursion is responsible for parallelism; the degree of the polynomial bounding the amount of hardware used depends only on the number of occurrences of CR (and the number of occurences of the constant $\#$.)

The crucial restriction in our system, justifying the use of linear logic notation, is a linearity constraint on variables of higher types: all higher type variables in a term must occur at most once.

The main new contribution in the analysis of the complexity of the system is a strict separation between the term, i.e., the program, and the numerical context, i.e., its input and data. Whereas the runtime may depend polynomially on the former, it may only depend polylogarithmically on the latter.

To make use of this conceptual separation, the algorithm that unfolds recursions computes, given a term and context, a recursion-free term *plus a new context*. In particular, it does not substitute numerical parameters, as this would immediately lead to linear growth, but only uses them for unfolding; in some cases, including the reduction of CR, it extends the context. This way, the growth of terms in the elimination of recursions is kept under control. In earlier systems that comprised at least polynomial time this strict distinction was not necessary, since the computation time there may depend on the *input* superlinearly. Note that any reasonable form of computation will depend at least linearly on the size of the *program*.

A direct extension to higher types of the first-order system of Bellantoni [2] would have a constant for concatenation recursion of linear type $(\iota \multimap \iota) \multimap \iota \multimap \iota$. This causes problems in our analysis because the amount of hardware required depends exponentially on the number of CR in a term, thus we must not allow duplications of this constant during the unfolding of LR. The only way to avoid this is by giving CR the more restrictive typ $(\iota \to \iota) \multimap \iota \to \iota$. This weaker form of concatenation recursion nevertheless suffices to include all of NC, when the set of base functions is slightly extended.

Finally, in order to be able to handle numerals in parallel logarithmic time, we use a tree data structure to store numerals during the computation. Whereas trees are used as the principal data structure in other characterizations of parallel complexity classes [15,16], our system works with usual binary numerals, and trees are only used in the implementation.

2 Clote's Function Algebra for NC

Clote [8] gives a function algebra characterization of NC using two recursion schemes. The class $\mathbf{A}$ is defined as the least class of functions that contain the constant 0, projections $\pi^n_j(x_1, \dots, x_n) = x_j$, the binary successors s_0, s_1, bit test *bit*, binary length $|x| := \lceil \log_2(x+1) \rceil$, and # where $x \# y = 2^{|x|\cdot|y|}$, and is closed under composition and the following two forms of recursion:

A function f is defined by *concatenation recursion on notation* (CRN) from functions g, h_0, h_1 if

$$\begin{aligned} f(0, \vec{x}) &= g(\vec{x}) \\ f(s_i(y), \vec{x}) &= s_{h_i(y,\vec{x})}(f(y, \vec{x})) \ , \end{aligned}$$

and f is defined from g, h_0, h_1 and r by weak bounded recursion on notation (WBRN) if there is F such that

$$\begin{aligned} F(0, \vec{x}) &= g(\vec{x}) \\ F(s_i(y), \vec{x}) &= h_i(y, \vec{x}, F(y, \vec{x})) \\ F(y, \vec{x}) &\leq r(y, \vec{x}) \\ f(y, \vec{x}) &= F(|y|, \vec{x}) \ . \end{aligned}$$

Theorem 1 (Clote [8]). *A number-theoretic function f is in $\boldsymbol{A}$ if and only if f is in NC.*

It is easy to see that the recursion scheme WBRN can be replaced by the following scheme: f is defined from g, h and r by *bounded logarithmic recursion* if

$$\begin{aligned} & f(0, \vec{x}) = g(\vec{x}) \\ & f(y, \vec{x}) = h(y, \vec{x}, f(H(y), \vec{x})) && \text{for } y > 0 \\ & f(y, \vec{x}) \leq r(y, \vec{x}) \ , \end{aligned}$$

where $H(n) := \left\lfloor \frac{n}{2^{\lceil |n|/2 \rceil}} \right\rfloor$ has length about half the length of n. Both forms of recursion produce $\log |y|$ iterations of the step function. We shall denote the function algebra with bounded logarithmic recursion by $\mathbf{A}$ as well.

3 Formal Definition of the System

We use simple types with two forms of abstraction over a single base type ι, i.e., our types are given by the grammar

$$\sigma, \tau ::= \iota \mid \sigma \multimap \tau \mid \sigma \to \tau \ ,$$

and we call the types that are built up from ι and $\multimap$ only the *linear* types.

As the intended semantics for our base type are the binary numerals we have the constants 0 of type ι and s_0 and s_1 of type $\iota \multimap \iota$. Moreover we add constants len of type $\iota \multimap \iota$ and bit of type $\iota \multimap \iota \multimap \iota$ for the corresponding base functions of $\mathbf{A}$.

The functionality of the base function # is split between two constants, a unary # of type $\iota \to \iota$ to produce growth, and sm of type $\iota \multimap \iota \multimap \iota \multimap \iota$ that performs the multiplication of lengths without producing growth. The intended semantics, reflected in the conversion rules below, is $\#n = 2^{|n|^2}$ and $sm(w, a, b) = 2^{|a| \cdot |b|} \bmod 2^{|w|}$.

In order to embed $\mathbf{A}$ into the system, we need two more constants drop of type $\iota \multimap \iota \multimap \iota$ and half of type $\iota \multimap \iota$, intended to denote the functions $drop(n, m) = \left\lfloor \frac{n}{2^{|m|}} \right\rfloor$ and H, respectively.

We allow case-distinction for arbitrary types, so we have a constant d_σ of type $\iota \multimap \sigma \multimap \sigma \multimap \sigma$ for *every* type σ. Recursion is added to the system via the constant LR and parallelism via the constant CR. Their types are

$$\begin{array}{lll} \mathsf{CR} & : & (\iota \to \iota) \multimap \iota \to \iota \\ \mathsf{LR}_\sigma & : & \sigma \multimap (\iota \to \sigma \multimap \sigma) \to \iota \to \sigma \qquad \text{for } \sigma \text{ linear} \end{array}$$

Terms are built from variables and constants via abstraction and typed application. We have incomplete variables of every type, denoted by x, y, ... and complete variables of ground type, denoted by $\mathbf{x}$, $\mathbf{y}$, ... All our variables and terms have a fixed type and we add type superscripts to emphasize the type: x^σ, $\mathbf{x}^\iota$, t^σ.

Corresponding to the two kinds of function types, there are two forms of abstraction

$$(\lambda x^\sigma . t^\tau)^{\sigma \multimap \tau} \qquad \text{and} \qquad (\lambda \mathbf{x}^\iota . t^\tau)^{\iota \to \tau}$$

and two forms of application

$$\left(t^{\sigma \multimap \tau}\, s^\sigma\right)^\tau \qquad \text{and} \qquad (t^{\sigma \to \tau}\, s^\sigma)^\tau \ ,$$

where in the last case we require s to be complete, and a term is called *complete* if all its free variables are. It should be noted that, although we cannot form terms of type $\sigma \to \tau$ with $\sigma \neq \iota$ directly via abstraction, it is still important to have that type in order to express, for example, that the first argument of LR must not contain free incomplete variables.

In the following we omit the type subscripts at the constants d_σ and LR_σ if the type is obvious or irrelevant. Moreover we identify α-equal terms. As usual application associates to the left. A *binary numeral* is either 0, or of the form $\mathsf{s}_{i_1}(\ldots(\mathsf{s}_{i_k}(\mathsf{s}_1 0)))$. We abbreviate the binary numeral $(\mathsf{s}_1 0)$ by 1.

The semantics of ι as binary numerals (rather than binary words) is given by the conversion rule $\mathsf{s}_0\ 0 \mapsto 0$. In the following definitions we identify binary numerals with the natural number they represent. The base functions get their usual semantics, i.e., we add conversion rules $\mathsf{len}\, n \mapsto |n|$, $\mathsf{drop}\, n\, m \mapsto \mathit{drop}(n, m)$, $\mathsf{half}\, n \mapsto H(n)$, $\mathsf{bit}\, n\, i \mapsto \left\lfloor \frac{n}{2^i} \right\rfloor \bmod 2$, $\mathsf{sm}\, w\, m\, n \mapsto \mathit{sm}(w, m, n)$. Moreover, we add the conversion rules

$$\begin{array}{lll} \mathsf{d}_\sigma\, 0 & \mapsto & \lambda x^\sigma\, y^\sigma\, .\, x \\ \mathsf{d}_\sigma\, (\mathsf{s}_i n) & \mapsto & \lambda x_0^\sigma\, x_1^\sigma\, .\, x_i \\ \#\, n & \mapsto & \mathsf{s}_0{}^{|n|^2}\, 1 \\ \mathsf{CR}\, h\, 0 & \mapsto & 0 \\ \mathsf{CR}\, h\, (\mathsf{s}_i\, n) & \mapsto & \mathsf{d}_{(\iota \multimap \iota)}\ (h\, (\mathsf{s}_i n))\, \mathsf{s}_0\, \mathsf{s}_1\, (\mathsf{CR}\, h\, n) \\ \mathsf{LR}\, g\, h\, 0 & \mapsto & g \\ \mathsf{LR}\, g\, h\, n & \mapsto & h\, n\, (\mathsf{LR}\, g\, h\, (\mathsf{half}\, n)) \end{array}$$

Here we always assumed that n, m and $\mathsf{s}_i\, n$ are binary numerals, and in particular that the latter does not reduce to 0. In the last rule, n has to be a binary numeral different from 0.

As usual the reduction relation is the closure of $\mapsto$ under all term forming operations and equivalence is the symmetric, reflexive, transitive closure of the reduction relation. As all reduction rules are correct with respect to the intended semantics and obviously all closed normal terms of type ι are numerals, closed terms t of type ι have a unique normal form that we denote by t^{nf}.

As usual, lists of notations for terms/numbers/ ... that only differ in successive indices are denoted by leaving out the indices and putting an arrow over the notation. It is usually obvious where to add the missing indices. If not we add a dot wherever an index is left out. Lists are inserted into formulae "in the natural way", e.g., $\overrightarrow{hm.} = hm_1, \ldots, hm_k$ and $x\overrightarrow{t} = ((x\,t_1)\ldots t_k)$ and $|g| + \overrightarrow{|s|} = |g| + |s_1| + \ldots + |s_k|$. Moreover, by abuse of notation, we denote lists consisting of maybe both, complete and incomplete variables also by $\overrightarrow{x}$.

As already mentioned, we are not interested in all terms of the system, but only in those fulfilling a certain linearity condition.

Definition 1. *A term t is called* linear*, if every variable of higher type in t occurs at most once.*

Since we allow that the variable x does not occur in $\lambda x.t$, our linear terms should correctly be called *affine*, but we keep the more familiar term *linear*.

4 Completeness

Definition 2. *A term $t : \overrightarrow{\iota} \to \iota$ denotes the function $f(\overrightarrow{x})$ if for every $\overrightarrow{n}$, $t\overrightarrow{n}$ reduces to the numeral $f(\overrightarrow{n})$.*

We will sometimes identify a term with the function it denotes.

In order to prove that our term system can denote all functions in NC, we first have to define some auxiliary terms. We define $\mathsf{ones} := \mathsf{CR}(\lambda \mathbf{x}.1)$, then we have that $\mathsf{ones}\, n = 2^{|n|} - 1$, i.e., a numeral of the same length as n consisting of ones only. We use this to define

$$\leq_\ell := \lambda \mathbf{y}\, b\,.\, \mathsf{bit}\,(\mathsf{ones}\, \mathbf{y})\,(\mathsf{len}\, b)\ ,$$

so that $\leq_\ell\ m\,n$ is the characteristic function of $|m| \leq |n|$. We will write $\leq_\ell$ infix in the following. It is used to define

$$\mathsf{max}_\ell := \lambda \mathbf{a}\, b\,.\, \mathsf{d}\,(\mathbf{a} \leq_\ell b)\, b\, \mathbf{a}$$

computing the longer of two binary numerals.

Next we define $\mathsf{rev} := \lambda x.\mathsf{CR}(\lambda \mathbf{i}.\mathsf{bit}\, x\, \mathbf{i})$, so that $\mathsf{rev}\, m\, n$ returns the $|n|$ least significant bits of m reversed. Finally we define the binary predecessor as $\mathsf{p} := \lambda x.\mathsf{drop}\, x\, 1$.

Theorem 2. *For every function $f(\vec{x})$ in $\boldsymbol{A}$, there is a closed linear term t_f of type $\vec{\iota} \to \iota$ that denotes f.*

Proof. The proof follows the lines of Bellantoni's [2] completeness proof for his two-sorted function algebra for NC.

We will use the following fact: for every function $f \in \mathbf{A}$ there is a polynomial q_f such that for all $\vec{n}$, $|f(\vec{n})| \leq q_f(\overrightarrow{|n|})$. To prove the theorem, we will prove the following stronger claim:

> For every $f(\vec{x}) \in \mathbf{A}$, there is a closed linear term t_f of type $\iota \to \vec{\iota} \multimap \iota$ and a polynomial p_f such that for every $\vec{n}$, $t_f\, w\, \vec{n}$ reduces to $f(\vec{n})$ for all w with $|w| \geq p_f(\overrightarrow{|n|})$.

The claim implies the theorem, since by use of the constant $\#$ and the term max_ℓ, we can define terms $w_f : \vec{\iota} \to \iota$ such that for all $\vec{n}$, $|w_f\, \vec{n}| \geq p_f(\overrightarrow{|n|})$.

We prove the claim by induction on the definition of f in the function algebra $\mathbf{A}$ with bounded logarithmic recursion.

If f is any of the base functions $0, s_i, |.|\,, bit$, then we let $t_f := \lambda \mathbf{w}.c$ where c is the corresponding constant of our system, and for $f = \pi^n_j$ we let $t_f := \lambda \mathbf{w}\ \vec{x}.x_j$. In these cases we can set $p_f = 0$, and the claim obviously holds.

If f is $\#$, then we set $t_f := \lambda \mathbf{w}\,.\,\mathsf{sm}\,\mathbf{w}$. It holds that $t_f\, w\, a\, b = a\#b$ as long as $|a| \cdot |b| < |w|$, so we set $p_f(x, y) = x \cdot y + 1$.

If f is defined by composition, $f(\vec{x}) = h(\overrightarrow{g(\vec{x})})$, then by induction we have terms $t_h, \vec{t_g}$ and polynomials $p_h, \vec{p_g}$. We define $t_f := \lambda \mathbf{w}\vec{x}.t_h\mathbf{w}(\overrightarrow{t_g\mathbf{w}\vec{x}})$ and $p_f(\vec{x}) := p_h(\overrightarrow{q_g(\vec{x})}) + \overrightarrow{p_g(\vec{x})}$. The claim follows easily from the induction hypothesis.

Now let f be defined by CRN from g, h_0, h_1, and let t_g, t_{h_i} be given by induction. First we define a function h that combines the two step functions into one, by

$$h := \lambda \mathbf{w}\, y\,.\,\mathsf{d}\, y\, (t_{h_0}\, \mathbf{w}\, (\mathsf{p}\, y))\, (t_{h_1}\, \mathbf{w}\, (\mathsf{p}\, y))$$

then we use this to define a function f' that computes an end-segment of $f(y, \vec{x})$ reversed, using CR, by

$$\begin{aligned} \mathsf{aux} &:= \lambda \mathbf{w}\, y\, \vec{x}\, \mathbf{z}\,.\, d\, (\mathbf{z} \leq_\ell y)\, \big(h\, \mathbf{w}\, (\mathsf{drop}\, y\, (\mathsf{p}\, \mathbf{z}))\, \vec{x}\big) \\ &\qquad\qquad\qquad\qquad \big(\mathsf{bit}\, (t_g\, \mathbf{w}\, \vec{x})\, (|z| - |y| - 1)\big) \\ f' &:= \lambda \mathbf{w}\, y\, \vec{x}\,.\, CR\, (\mathsf{aux}\, \mathbf{w}\, y\, \vec{x})\ , \end{aligned}$$

where $|z| - |y| - 1$ is computed as $\mathsf{len}\, (\mathsf{drop}\, \mathbf{z}\, (\mathsf{s}_1\, y))$. Finally, the computed value is reversed, and t_f is defined by

$$t_f := \lambda \mathbf{w}\, y\, \vec{x}\,.\, \mathsf{rev}\, (f'\, \mathbf{w}\, y\, \vec{x}\, \mathbf{w})\, \mathbf{w}\ .$$

In order for this to work, w has to be large enough for g and the h_i to be computed correctly by the inductive hypothesis, thus p_f needs to maximize p_g and the p_{h_i}. Also, w has to be long enough for the concatenation recursion in the definition of f' to actually compute all bits of $f(y, \overrightarrow{x})$, so $|w|$ has to be larger than $|y| + |g(\overrightarrow{x})|$. All this is guaranteed if we set

$$p_f(y, \overrightarrow{x}) := p_g(\overrightarrow{x}) + \sum_{i=1,2} p_{h_i}(y, \overrightarrow{x}) + y + q_g(\overrightarrow{x}) + 1 \ .$$

Finally, let f be defined by bounded logarithmic recursion from g, h and r,

$$\begin{aligned} &f(0, \overrightarrow{x}) = g(\overrightarrow{x}) \\ &f(y, \overrightarrow{x}) = h(y, \overrightarrow{x}, f(H(y), \overrightarrow{x})) && \text{for } y > 0 \\ &f(y, \overrightarrow{x}) \leq r(y, \overrightarrow{x}) \ , \end{aligned}$$

and let t_g and t_h be given by induction. In order to define t_f, we cannot use logarithmic recursion on y since y is incomplete. Instead we simulate the recursion on y by a recursion on a complete argument.

We first define a function Y that yields the values $H^{(k)}(y)$ that are needed in the recursion as

$$\begin{aligned} S &:= \lambda \mathbf{u}\, v^{\iota \multimap \iota}\, y \,.\, \big(\mathsf{d}\,(\mathbf{u} \leq_\ell \mathbf{z})\, y\, (\mathsf{half}\,(v\, y))\big) \\ Y &:= \lambda \mathbf{z}\, \mathbf{w} \,.\, \mathsf{LR}_{\iota \multimap \iota}\, (\lambda y . y)\, S\, \mathbf{w} \ . \end{aligned}$$

We now use this function to define a term f' computing f by recursion on a complete argument $\mathbf{z}$ by

$$\begin{aligned} T &:= \lambda \mathbf{u}\, v^{\iota \multimap \overrightarrow{\iota} \multimap \iota}\, y\, \overrightarrow{x} \,.\, \Big(\mathsf{d}\,((Y\, \mathbf{u}\, \mathbf{w}\, y) = 0)\, (t_g\, \mathbf{w}\, y\, \overrightarrow{x}) \\ &\qquad\qquad\qquad (t_h\, \mathbf{w}\, (Y\, \mathbf{u}\, \mathbf{w}\, y)\, \overrightarrow{x}\, (v\, y\, \overrightarrow{x}))\Big) \\ f' &:= \lambda \mathbf{w}\, \mathbf{z} \,.\, \mathsf{LR}_{\iota \multimap \overrightarrow{\iota} \multimap \iota}\, (\lambda\, y\, \overrightarrow{x} \,.\, 0)\, T\, \mathbf{z} \end{aligned}$$

where the test $x = 0$ is implemented as $\mathsf{d}\,(\mathsf{bit}\,(\mathsf{s}_0\, x)\, (\mathsf{len}\, x))\, 1\, 0$. Finally, t_f is defined by identifying the complete arguments in f':

$$t_f := \lambda \mathbf{w} \,.\, f'\, \mathbf{w}\, \mathbf{w}$$

To show the correctness of this definition, define

$$p_f(y, \overrightarrow{x}) := 2y + p_h(y, \overrightarrow{x}, q_r(y, \overrightarrow{x})) + p_g(\overrightarrow{x})$$

and fix y, $\overrightarrow{x}$ and w with $|w| \geq p_f(|y|\,, \overrightarrow{|x|})$.

Note that the only values of z for which the function Y is ever invoked during the computation are $H^{(k)}(w)$ for $0 \leq k \leq ||y||$, and that for these values of z,

$Y(z,w,y)$ varies over the values $H^{(k)}(y)$. By a downward induction on k we show that for these values of z,

$$(f'(w,z,y,\vec{x}) = f(Y(z,w,y),\vec{x}) \ .$$

This implies the claim for t_f, since $Y(w,w,y) = y$.

The induction basis occurs for $k = ||y||$, where $V(z,w,y) = 0$. Since $|w| \geq 2\,|y|$, we have $z > 0$, thus the recursive step in the definition of f' is used, and the first branch of the case distinction is chosen. Therefore the equality follows from the fact that w is large enough for t_g to compute g correctly.

In the inductive step, we use the fact that $Y(H(z),w,y) = H(Y(z,w,y))$, and that w is large enough for t_h to compute h correctly. Since for $z = H^{(k-1)}(w)$ we have $Y(z,w,y) > 0$, we get

$$\begin{aligned}
f'(w,z,y,\vec{x}) &= t_h(w, Y(z,w,y), \vec{x}, f'(w,H(z),y,\vec{x})\\
&= t_h(w, Y(z,w,y), \vec{x}, f(Y(H(z),w,y),\vec{x})\\
&= t_h(w, Y(z,w,y), \vec{x}, f(H(Y(z,w,y)),\vec{x})\\
&= h(Y(z,w,y), \vec{x}, f(H(Y(z,w,y)),\vec{x}))\\
&= f(Y(z,w,y),\vec{x})
\end{aligned}$$

where the second equality holds by the induction hypothesis. This completes the proof of the claim and the theorem. □

5 Soundness

Definition 3. *The length $|t|$ of a term t is inductively defined as follows: For a variable x, $|x| = 1$, and for any constant c other than d, $|c| = 1$, whereas $|\mathsf{d}| = 3$. For complex terms we have the usual clauses $|r\,s| = |r| + |s|$ and $|\lambda x.r| = |r| + 1$.*

The length of the constant d is motivated by the desire to decrease the length of a term in the reduction of a d-redex.

Note that due to our identification of natural numbers with binary numerals, the notation $|n|$ is ambiguous now. Nevertheless, in the following we will only use $|n|$ as the term length defined above which for numerals n differs from the binary length only by one.

Definition 4. *For a list $\vec{n}$ of numerals, define $|\vec{n}| := \max(\overrightarrow{|n|})$.*

Definition 5. *A context is a list of pairs (x,n) of variables (complete or incomplete) of type ι and numerals, where all the variables are distinct. If $\vec{x}$ is a list of distinct variables of type ι and $\vec{n}$ a list of numerals of the same length, then we denote by $\vec{x};\vec{n}$ the context $\overrightarrow{(x,n)}$.*

Definition 6. *For every symbol c of our language and term t, $\sharp_c(t)$ denotes the number of occurrences of c in t. For obvious aesthetic reasons we abbreviate $\sharp_\#(t)$ by $\#(t)$.*

Definition 7. *A term t is called* simple *if t contains none of the constants $\#$, CR or LR.*

Bounding the Size of Numerals

Lemma 1. *Let t be a simple, linear term of type ι and $\overrightarrow{x}; \overrightarrow{n}$ a context, such that all free variables in t are among $\overrightarrow{x}$. Then for $t^* := t[\overrightarrow{x} := \overrightarrow{n}]^{\mathrm{nf}}$ we have $|t^*| \leq |t| + |\overrightarrow{n}|$.*

Proof. By induction on $|t|$. We distinguish cases according to the form of t.

Case 1: t is $x\,\overrightarrow{r}$ for a variable x. Since x must be of type ι, $\overrightarrow{r}$ must be empty, and t^* is just one of the numerals in $\overrightarrow{n}$.

Case 2: t is $c\,\overrightarrow{r}$ for a constant c. Here we have four subcases, depending on the constant c.

Case 2a: c is 0, so $\overrightarrow{r}$ is empty and t is already normal.

Case 2b: c is s_i, so t is $c\,r$ for a term r of type ι. Let $r^* := r[\overrightarrow{x} := \overrightarrow{n}]^{\mathrm{nf}}$, by the induction hypothesis we have $|r^*| \leq |r| + |\overrightarrow{n}|$, and therefore we get $|t^*| \leq |r^*| + 1 \leq |t| + |\overrightarrow{n}|$.

Case 2c: c is one of the constants len, half, drop, bit or sm, so t is $c\,r\,\overrightarrow{s}$ for terms $r, \overrightarrow{s}$ of type ι. Let $r^* := r[\overrightarrow{x} := \overrightarrow{n}]^{\mathrm{nf}}$, by the induction hypothesis we have $|r^*| \leq |r| + |\overrightarrow{n}|$, and therefore we get $|t^*| \leq |r^*| \leq |t| + |\overrightarrow{n}|$.

Case 2d: c is d_σ, so t is $\mathsf{d}_\sigma\, s\, u_0\, u_1\, \overrightarrow{v}$, where s is of type ι and u_i are of type σ. Depending on the last bit i of the value of $s[\overrightarrow{x} := \overrightarrow{n}]$, t reduces to the shorter term $t' = u_i\,\overrightarrow{v}$, to which we can apply the induction hypothesis obtaining the normal form t^* with $|t^*| \leq |t'| + |\overrightarrow{n}| < |t| + |\overrightarrow{n}|$.

Case 3: t is $(\lambda x.r)\, s\, \overrightarrow{s}$. Here we have two subcases, depending on the number of occurrences of x in r.

Case 3a: x occurs at most once, then the term $t' := r[x := s]\,\overrightarrow{s}$ is smaller than t, and we can apply the induction hypothesis to t'.

Case 3b: x occurs more than once, and thus is of type ι. Then s is of type ι, so we first apply the induction hypothesis to s, obtaining $s^* := s[\overrightarrow{x} := \overrightarrow{n}]^{\mathrm{nf}}$ with $|s^*| \leq |s| + |\overrightarrow{n}|$. Now we let $t' := r\,\overrightarrow{s}$, and we apply the induction hypothesis to t' and the context $\overrightarrow{x}, y; \overrightarrow{n}, s^*$, so we get

$$|t^*| \leq |t'| + |\overrightarrow{n}, s^*| \leq |t'| + |s| + |\overrightarrow{n}| \ .$$

The last case, where t is $\lambda x.r$, cannot occur because of the type of t. □

Data Structure

We represent terms as parse trees, fulfilling the obvious typing constraints. The number of edges leaving a particular node is called the out-degree of this node. There is a distinguished node with in-degree 0, called the root. Each node is stored in a record consisting of an entry `cont` indicating its kind, plus some pointers to its children. We allow the following kinds of nodes with the given restrictions:

- Variable nodes representing a variable x. Variable nodes have out-degree 0. Every variable has a unique name and an associated register $\mathtt{R}[x]$.

- Abstraction nodes λx representing the binding of the variable x. Abstraction nodes have out-degree one, and we denote the pointer to its child by `succ`.
- For each constant c, there are nodes representing the constant c. These nodes have out-degree 0.
- Application nodes @ representing the application of two terms. The obvious typing constraints have to be fulfilled. We denote the pointers to the two children of an application node by `left` and `right`.
- Auxiliary nodes κ_i representing the composition of type one. These nodes are labeled with a natural number i, and each of those nodes has out-degree either 2 or 3. They will be used to form 2/3-trees (as e.g. described by Knuth [13]) representing numerals during the computation. We require that any node reachable from a $\kappa_.$-node is either a $\kappa_.$ node as well or one of the constants $\mathtt{s}_0$ or $\mathtt{s}_1$.
- Auxiliary nodes κ' representing the identification of type-one-terms with numerals (via "applying" them to 0). The out-degree of such a node, which is also called a "numeral node", either is zero, in which case the node represents the term 0, or the out-degree is one and the edge starting from this node either points to one of the constants $\mathtt{s}_0$ or $\mathtt{s}_1$ or to a $\kappa_.$ node.
- Finally, there are so-called dummy nodes $\diamond$ of out-degree 1. The pointer to the child of a dummy node is again denoted by `succ`. Dummy nodes serve to pass on pointers: a node that becomes superfluous during reduction is made into a dummy node, and any pointer to it will be regarded as if it pointed to its child.

A tree is called a *numeral* if the root is a numeral node, all leaves have the same distance to the root and the label i of every κ_i node is the number of leaves reachable from that node. By standard operations on 2/3-trees it is possible in sequential logarithmic time to

- split a numeral at a given position i.
- find out the i'th bit of the numeral.
- concatenate two numerals.

So using κ' and $\kappa_.$ nodes is just a way of implementing "nodes" labeled with a numeral allowing all the standard operations on numerals in logarithmic time. Note that the length of the label i (coded in binary) of a κ_i node is bounded by the logarithm of the number of nodes.

Normalization Algorithms and Their Complexity

Lemma 2. *Let t be a simple, linear term of type ι and $\vec{x}; \vec{n}$ a context such that all free variables in t are among the $\vec{x}$. Then the normal form of $t[\vec{x} := \vec{n}]$ can be computed in time $O(|t| \cdot \log |\vec{n}|)$ by $O(|t| \cdot |\vec{n}|)$ processors.*

Proof. We start one processor for each of the nodes of the parse-tree of t, with a pointer to this node in its local register. The registers associated to the variables $\vec{x}$ in the context contain pointers to the respective numerals $\vec{n}$, and the registers associated to all other variables are initialized with a `NULL` pointer.

The program operates in rounds, where the next round starts once all active processors have completed the current round. The only processors that will ever do something are those at the application or variable nodes. Thus all processors where $\texttt{cont} \notin \{@, x, \mathsf{d}\}$ can halt immediately. Processors at d nodes do not halt because they will be converted to variable nodes in the course of the reduction.

The action of a processor at an application node in one round depends on the type of its sons. If the right son is a dummy node, i.e., $\texttt{right.cont} = \diamond$, then this dummy is eliminated by setting $\texttt{right} := \texttt{right.succ}$. Otherwise, the action depends on the type of the left son.

- If $\texttt{left.cont} = \diamond$, then eliminate this dummy by setting $\texttt{left} := \texttt{left.succ}$.
- If $\texttt{left.cont} = \lambda x$, then this β-redex is partially reduced by copying the argument $\texttt{right}$ into the register $\texttt{R}[x]$ associated to the variable x. The substitution part of the β-reduction is then performed by the processors at variable nodes. Afterwards, replace the @ and λx nodes by dummies by setting $\texttt{cont} := \diamond$, $\texttt{left.cont} := \diamond$ and $\texttt{succ} := \texttt{left}$.
- If $\texttt{left.cont} \in \{\mathsf{s}_i, \mathsf{len}, \mathsf{half}\}$ and the right son is a numeral, $\texttt{right.cont} = \kappa'$, then replace the current node by a dummy, and let $\texttt{succ}$ point to a numeral representing the result. In the case of s_i and half, this can be implemented by 2/3-tree operations using sequential time $O(\log |\overrightarrow{n}|)$.
  In the case of len, the result is equal to the number i of leaves of the numeral argument. This value is read off the topmost κ_i node, and a numeral of that value is produced. Since i is a number of length $O(\log |\overrightarrow{n}|)$, this can also be done in sequential time $O(\log |\overrightarrow{n}|)$.
- If $\texttt{left.cont} = @$, $\texttt{left.left.cont} \in \{\mathsf{drop}, \mathsf{bit}\}$ and $\texttt{right}$ and $\texttt{left.right}$ both point to numerals, then again replace the current node by a dummy, and let $\texttt{succ}$ point to a numeral representing the result, which again can be computed by 2/3-tree operations in time $O(\log |\overrightarrow{n}|)$.
- If $\texttt{left.cont} = \texttt{left.left.cont} = @$, $\texttt{left.left.left.cont} = \mathsf{sm}$ and all of $\texttt{right}$, $\texttt{left.right}$ and $\texttt{left.left.right}$ point to numerals, then again the current node is replaced by a dummy with $\texttt{succ}$ pointing to the result.
  To compute the result, the lengths i and j are read off the second and third argument, and multiplied. As i and j are $O(\log |\overrightarrow{n}|)$ bit numbers, this can be done in parallel time $O(\log \log |\overrightarrow{n}|)$ by $O(\log^3 |\overrightarrow{n}|)$ many processors.
  The product $i \cdot j$ is compared to the length of the first argument; let the maximum of both be k. Now the result is a numeral consisting of a one followed by k zeroes, which can be produced in parallel time $\log^2 k$ by $O(k)$ many processors using the square-and-multiply method, which suffices since $k \leq O(\log |\overrightarrow{n}|)$.
- Finally, if $\texttt{left.cont} = \mathsf{d}$ and $\texttt{right.cont} = \kappa'$, then extract the last bit b of the numeral at $\texttt{right}$, and create two new variables x_0 and x_1. Then reduce the d-redex by replacing the current node and the right son by abstraction nodes, and the left son by a variable node, i.e., setting $\texttt{cont} := \lambda x_0$, $\texttt{right.cont} := \lambda x_1$, $\texttt{succ.right}$, $\texttt{succ.succ} := \texttt{left}$ and $\texttt{left.cont} := x_b$.

A processor with $\texttt{cont} = x$ only becomes active when $\texttt{R}[x] \neq \texttt{NULL}$, and what it does then depends on the type of x.

If x is not of ground type, then the variable x occurs only in this place, so the substitution can be safely performed by setting $\mathtt{cont} := \diamond$ and $\mathtt{succ} := \mathtt{R}[x]$.

If x is of type ι, the processor waits until the content of register $\mathtt{R}[x]$ has been normalized, i.e., it acts only if $\mathtt{R}[x].\mathtt{cont} = \kappa'$. In this case, it replaces the variable node by a dummy, and lets $\mathtt{succ}$ point to a newly formed copy of the numeral in $\mathtt{R}[x]$. This copy can be produced in parallel time $O(\log|\overrightarrow{n}|)$ by $|\overrightarrow{n}|$ processors, since the depth of any numeral is bounded by $\log|\overrightarrow{n}|$.

Concerning correctness, note that the tree structure is preserved, since numerals being substituted for type ι variables are explicitly copied, and variables of higher type occur at most once. Obviously, no redex is left when the program halts.

For the time bound, observe that every processor performs at most one proper reduction plus possibly some dummy reductions. Every dummy reduction makes one dummy node unreachable, so the number of dummy reductions is bounded by the number of dummy nodes generated. Every dummy used to be a proper node, and the number of nodes is at most $2\,|t|$, so this number is bounded by $2\,|t|$. Thus at most $4\,|t|$ reductions are performed, and the program ends after at most that many rounds. As argued above, every round takes at most $O(\log|\overrightarrow{n}|)$ operations with $O(|\overrightarrow{n}|)$ many additional processors. □

The next lemma is the key to show that all terms can be normalized in NC: it shows how to eliminate the constants #, LR and CR. As mentioned in the introduction, we have to distinguish between the program, i.e., the term we wish to normalize, and its input, given as a context. The runtime and length of the output term may depend polynomially on the former, but only polylogarithmically on the latter.

Since an ordinary $O(\cdot)$-analysis is too coarse for the inductive argument, we need a more refined asymptotic analysis. Therefore we introduce the following notation:

$$f(n) \lesssim g(n) \;:\Longleftrightarrow\; f(n) \le (1 + o(1))g(n)\;,$$

or equivalently $\limsup_{n\to\infty} \frac{f(n)}{g(n)} \le 1$.

Lemma 3. *Let t be a linear term of linear type and $\overrightarrow{x};\overrightarrow{n}$ a context with all free variables of $t[\overrightarrow{x} := \overrightarrow{n}]$ incomplete. Then there are a term* $\mathrm{simp}(t,\overrightarrow{x};\overrightarrow{n})$ *and a context $\overrightarrow{y};\overrightarrow{m}$ such that* $\mathrm{simp}(t,\overrightarrow{x};\overrightarrow{n})[\overrightarrow{y} := \overrightarrow{m}]$ *is simple and equivalent to $t[\overrightarrow{x} := \overrightarrow{n}]$, and which can be computed in time*

$$T(|\overrightarrow{n}|) \;\lesssim\; 2^{\sharp_{\mathsf{LR}}(t)} \cdot |t| \cdot (2^{\#(t)} \cdot \log|\overrightarrow{n}|)^{\sharp_{\mathsf{LR}}(t)+2}$$

by

$$P(|\overrightarrow{n}|) \;\lesssim\; |t| \cdot |\overrightarrow{n}|^{2^{\#(t)}(\sharp_{\mathsf{CR}}(t)+2)} \cdot (\log|\overrightarrow{n}|)^{2^{\#(t)}(\sharp_{\mathsf{LR}}(t)+1)}$$

processors, such that

$$|\mathrm{simp}(t,\overrightarrow{x};\overrightarrow{n})| \;\lesssim\; |t| \cdot \left(2^{\#(t)} \cdot \log|\overrightarrow{n}|\right)^{\sharp_{\mathsf{LR}}(t)} \quad \textit{and} \quad |\overrightarrow{m}| \;\lesssim\; |\overrightarrow{n}|^{2^{\#(t)}}\;.$$

The proof is somewhat lengthy, so we sketch it first:

We start by describing the algorithm. It searches for the head-redex and reduces it in the obvious way (and then continues in the same way until the term is normal): in the case of a ground-type β-redex enlarge the context, in the case of a higher type β-redex reduce it in the term; in the case of LR the step term has to be unfolded only logarithmically many times, so we can just form a new term, whereas in the case of CR we have to use parallelism. However, in this case the result of every processor is just a single bit, so the results can be collected efficiently and returned to the *context* (whereas in the case of LR the result is a term of higher type). Note the crucial interplay between the length of the term, the size of the context, the running time and the number of processors needed; therefore we have to provide all four bounds simultaneously.

After the description of the algorithm, a long and tedious (but elementary) calculation follows showing that all bounds indeed hold in every case. The structure of the proof is always the same: in the interesting cases a numerical argument has to be evaluated in order to be able to reduce the redex (i.e., the numeral we recurse on, or the numeral to be put in the context in the case of a ground type β-redex). Then the induction hypothesis yields the size of this numeral and also the amount of time and processors needed. Then calculate the length of the unfolded term. The induction hypothesis for this term yields the amount of time and processors needed for the final computation, and also the bounds for the final output. Summing up all the times (calculation of the numeral, unfolding, final computation) one verifies that the time bound holds as well.

Proof (of lemma 3). By induction on $\sharp_{\mathsf{LR}}(t)$, with a side-induction on $|t|$ show that the following algorithm does it:

By pattern matching, determine in time $O(|t|)$ the form of t, and branch according to the form.

- If t is a variable or one of the constants 0 or d, then return t and leave $\overrightarrow{x};\overrightarrow{n}$ unchanged.
- If t is $c\,\overrightarrow{s}$, where c is one of the constants s_i, drop, bit, len or sm then recursively simplify $\overrightarrow{s}$, giving $\overrightarrow{s^*}$ and contexts $\overrightarrow{y_j};\overrightarrow{m_j}$, and return $c\,\overrightarrow{s^*}$ and $\overrightarrow{\overrightarrow{y}};\overrightarrow{\overrightarrow{m}}$.
- If t is $\mathsf{d}\,r\,\overrightarrow{s}$, then simplify r giving r' and $\overrightarrow{y};\overrightarrow{m}$. Compute the numeral $r^* := r'[\overrightarrow{y} := \overrightarrow{m}]^{\mathrm{nf}}$, and reduce the redex $\mathsf{d}\,r^*$, giving t', and recursively simplify $t'\,\overrightarrow{s}$ with context $\overrightarrow{x};\overrightarrow{n}$.
- If t is $\#\,r$ then simplify r giving r' and $\overrightarrow{y};\overrightarrow{m}$. Compute the numeral $r^* := r'[\overrightarrow{y} := \overrightarrow{m}]^{\mathrm{nf}}$, and return a new variable y' and the context $y'; 2^{|r^*|^2}$.
- If t is $\mathsf{CR}\,h\,r$, then simplify r giving r' and $\overrightarrow{y};\overrightarrow{m}$, and compute the numeral $r^* := r'[\overrightarrow{y} := \overrightarrow{m}]^{\mathrm{nf}}$.

  Spawn $|r^*|$ many processors, one for each leaf of r^*, by moving along the tree structure of r^*. The processor at bit i of r^* simplifies $h\,\mathbf{z}$ in the context $\overrightarrow{x},\mathbf{z};\overrightarrow{n},\lfloor r^*/2^i \rfloor$ (with $\mathbf{z}$ a new complete variable), giving a term h_i and context $\overrightarrow{y_i};\overrightarrow{m_i}$, then he computes $h_i^* := h_i[y_i := m_i]^{\mathrm{nf}}$, retaining only the lowest order bit b_i.

The bits $\overrightarrow{b}$ are collected into a 2/3-tree representation of a numeral m, which is output in the form of a new variable z and the context $z; m$.
- t is $\mathsf{LR}\, g\, h\, m\, \overrightarrow{s}$ then simplify m, giving m' and $\overrightarrow{x_m}; \overrightarrow{n_m}$. Normalize m' in the context $\overrightarrow{x}, \overrightarrow{x_m}; \overrightarrow{n}, \overrightarrow{n_m}$, giving m^*. Form k numerals $m_i = \mathsf{half}^i(m^*)$ and sequentially simplify $\overrightarrow{h\, m.}$, giving $\overrightarrow{h'}$. (Of course, more precisely simplify $h\, x$ for a new variable x in the context extended by $x; m_i$.) Then form the term
$$t' := h'_0(h'_1 \dots (h'_k\, g))\, \overrightarrow{s}$$
and simplify it.
- If t is of the form $\lambda x.r$ then recursively simplify r.
- If t is of the form $(\lambda x.r)\, s\, \overrightarrow{s}$ and x occurs at most once in r then recursively simplify $r[x := s]\, \overrightarrow{s}$.
- If t is of the form $(\lambda x.r)\, s\, \overrightarrow{s}$ and x occurs several times in r, then simplify s giving s' and a context $\overrightarrow{y}; \overrightarrow{m}$. Normalize s' in this context giving the numeral s^*. Then simplify $r\, \overrightarrow{s}$ in the context $\overrightarrow{x}, x; \overrightarrow{n}, s^*$.

For correctness, note that **in the case** $\mathsf{d}\, r\, \overrightarrow{s}$ simplifying r takes time
$$\lesssim\ 2^{\sharp_{\mathsf{LR}}(r)} \cdot |r| \cdot \left(2^{\#(r)} \cdot \log|\overrightarrow{n}|\right)^{\sharp_{\mathsf{LR}}(r)+2}$$
and uses
$$\lesssim\ |r| \cdot |\overrightarrow{n}|^{2^{\#(r)}(\sharp_{\mathsf{CR}}(r)+2)} (\log|\overrightarrow{n}|)^{2^{\#(r)}(\sharp_{\mathsf{LR}}(r)+1)}$$
many processors. For the output we have $|r'| \lesssim |r| \cdot \left(2^{\#(r)} \cdot \log|\overrightarrow{n}|\right)^{\sharp_{\mathsf{LR}}(r)}$ and $|\overrightarrow{m}| \lesssim |\overrightarrow{n}|^{2^{\#(r)}}$. Hence the time used to normalize r' (using the algorithm of lemma 2) is $O(|r'| \cdot \log|\overrightarrow{m}|)$, which is (order of)
$$|r| \cdot \left(2^{\sharp_{\mathsf{LR}}(r)} \cdot \log|\overrightarrow{n}|\right)^{\sharp_{\mathsf{LR}}(r)+1}$$
and the number of processors needed is $O(|r'| \cdot |\overrightarrow{m}|) \leq |r| \cdot |\overrightarrow{n}|^{2^{\#(r)}+1}$. Finally, to simplify $t'\,\overrightarrow{s}$ we need time
$$\lesssim\ 2^{\sharp_{\mathsf{LR}}(\overrightarrow{s})} \cdot (|\overrightarrow{s}| + 3) \cdot (2^{\#(\overrightarrow{s})} \cdot \log|\overrightarrow{n}|)^{\sharp_{\mathsf{LR}}(\overrightarrow{s})+2}$$
and the number of processors is
$$\lesssim\ (|\overrightarrow{s}| + 3)\, |\overrightarrow{n}|^{2^{\#(\overrightarrow{s})}(\sharp_{\mathsf{CR}}(\overrightarrow{s})+2)} (\log \overrightarrow{n})^{2^{\#(\overrightarrow{s})}(\sharp_{\mathsf{LR}}(\overrightarrow{s})+1)}$$
Summing up gives an overall time that is
$$\lesssim\ 2^{\sharp_{\mathsf{LR}}(t)} \cdot (|r| + |\overrightarrow{s}| + 3) \cdot \left(2^{\#(t)} \cdot \log|\overrightarrow{n}|\right)^{\sharp_{\mathsf{LR}}(t)+2}$$
which is a correct bound since $|\mathsf{d}| = 3$. Maximizing gives that the overall number of processors is
$$\lesssim\ |t| \cdot |\overrightarrow{n}|^{2^{\#(t)}(\sharp_{\mathsf{CR}}(t)+2)} (\log|\overrightarrow{n}|)^{2^{\#(t)}(\sharp_{\mathsf{LR}}(t)+1)}$$

The length of the output term is

$$\lesssim\; (|\overrightarrow{s}|+3)\cdot\left(2^{\#(\overrightarrow{s})}\cdot\log|\overrightarrow{n}|\right)^{\sharp_{\mathsf{LR}}(\overrightarrow{s})}$$

and the size of the output context is $\lesssim |\overrightarrow{n}|^{2^{\#(\overrightarrow{s})}}$, which suffices.

In the case $\# r$ we obtain the same bounds for simplification and normalization of r as in the previous case. For r^* we get

$$|r^*| = O(|r'|+|\overrightarrow{m}|)\;\lesssim\;|\overrightarrow{n}|^{2^{\#(r)}}$$

Computing the output now takes time

$$\log|r^*|^2 = 2^{\#(r)+1}\cdot\log|\overrightarrow{n}|$$

and

$$|r^*|^2\;\lesssim\;|\overrightarrow{n}|^{2^{\#(r)+1}}$$

many processors. Thus the overall time is

$$\lesssim\; 2^{\sharp_{\mathsf{LR}}(r)}\cdot|r|\cdot\left(2^{\#(r)+1}\cdot\log|\overrightarrow{n}|\right)^{\sharp_{\mathsf{LR}}(r)+2}$$

and the number of processors is

$$\lesssim\; |r|\cdot|\overrightarrow{n}|^{2^{\#(r)+1}(\sharp_{\mathsf{CR}}(r)+2)}\cdot(\log|\overrightarrow{n}|)^{2^{\#(r)}(\sharp_{\mathsf{LR}}(r)+1)}\;.$$

The length of the output term is 1, and the size of the output context is bounded by $|r^*|^2+1\lesssim|\overrightarrow{n}|^{2^{\#(r)+1}}$, which implies the claim.

In the case $\mathsf{CR}\,h\,r$ note that the arguments $h\colon\iota\to\iota$ and $r\colon\iota$ both have to be present, since t has to be of linear type (and $\mathsf{CR}\colon(\iota\to\iota)\multimap\iota\to\iota$). We obtain the same bounds for simplification and normalization of r and the length of the numeral r^* as in the previous case. Spawning the parallel processors and collecting the result in the end each needs time $\log|r^*| = 2^{\#(r)}\cdot|\overrightarrow{n}|$. The main work is done by the $|\overrightarrow{n}|^{2^{\#(r)}}$ many processors that do the simplification and normalization of the step terms. Each of them takes time

$$\begin{aligned}&\lesssim 2^{\sharp_{\mathsf{LR}}(h)}\cdot(|h|+1)\cdot\left(2^{\#(h)}\cdot\log|\overrightarrow{n},r^*|\right)^{\sharp_{\mathsf{LR}}(h)+2}\\&\lesssim 2^{\sharp_{\mathsf{LR}}(h)}\cdot(|h|+1)\cdot\left(2^{\#(h)+\#(r)}\cdot\log|\overrightarrow{n}|\right)^{\sharp_{\mathsf{LR}}(h)+2}\end{aligned}$$

and a number of sub-processors satisfying

$$\begin{aligned}&\lesssim(|h|+1)\cdot|\overrightarrow{n},r^*|^{2^{\#(h)}(\sharp_{\mathsf{CR}}(h)+2)}\cdot(\log|\overrightarrow{n},r^*|)^{2^{\#(h)}(\sharp_{\mathsf{LR}}(h)+1)}\\&\lesssim(|h|+1)\cdot|\overrightarrow{n}|^{2^{\#(h)+\#(r)}(\sharp_{\mathsf{CR}}(h)+2)}\cdot(2^{\#(r)}\log|\overrightarrow{n}|)^{2^{\#(h)}(\sharp_{\mathsf{LR}}(h)+1)}\end{aligned}$$

$$\lesssim (|h|+1)\cdot|\overrightarrow{n}|^{2^{\#(h)+\#(r)}(\sharp_{\mathsf{CR}}(h)+2)}\cdot(\log|\overrightarrow{n}|)^{2^{\#(h)+\#(r)}(\sharp_{\mathsf{LR}}(h)+1)}$$

to compute h_i and $\overrightarrow{y_i};\overrightarrow{m_i}$ with

$$\begin{aligned}|h_i| &\lesssim (|h|+1)\cdot\Big(2^{\#(h)}\cdot\log|\overrightarrow{n},r^*|\Big)^{\sharp_{\mathsf{LR}}(h)}\\ &\lesssim (|h|+1)\cdot\Big(2^{\#(h)+\#(r)}\cdot\log|\overrightarrow{n}|\Big)^{\sharp_{\mathsf{LR}}(h)}\end{aligned}$$

and

$$|\overrightarrow{m_i}|\ \lesssim\ |\overrightarrow{n},r^*|^{2^{\#(h)}}\ \lesssim\ |\overrightarrow{n}|^{2^{\#(h)+\#(r)}}$$

Now the normal form h_i^* is computed in time

$$O(|h_i|\cdot\log|\overrightarrow{m_i}|)\ \lesssim\ (|h|+1)\cdot\Big(2^{\#(h)+\#(r)}\cdot\log|\overrightarrow{n}|\Big)^{\sharp_{\mathsf{LR}}(h)+1}$$

by

$$O(|h_i|\cdot|\overrightarrow{m_i}|)\ \lesssim\ (|h|+1)\,|\overrightarrow{n}|^{2^{\#(h)+\#(r)}+1}\cdot(\log|\overrightarrow{n}|)^{2^{\#(h)+\#(r)}(\sharp_{\mathsf{LR}}(h)+1)}$$

many sub-processors. Summing up the times yields that the overall time is

$$\begin{aligned}&\lesssim 2^{\sharp_{\mathsf{LR}}(r)}\cdot|r|\cdot\Big(2^{\#(r)}\cdot\log|\overrightarrow{n}|\Big)^{\sharp_{\mathsf{LR}}(r)+2}\\ &\qquad+2^{\sharp_{\mathsf{LR}}(h)}\cdot(|h|+1)\cdot\Big(2^{\#(h)+\#(r)}\cdot\log|\overrightarrow{n}|\Big)^{\sharp_{\mathsf{LR}}(h)+2}\\ &\lesssim 2^{\sharp_{\mathsf{LR}}(t)}\cdot(|r|+|h|+1)\cdot\Big(2^{\#(t)}\cdot\log|\overrightarrow{n}|\Big)^{\sharp_{\mathsf{LR}}(t)+2}\end{aligned}$$

The number of sub-processors used by each of the $|r^*|$ processes is

$$\lesssim\ (|h|+1)\cdot|\overrightarrow{n}|^{2^{\#(h)+\#(r)}(\sharp_{\mathsf{CR}}(h)+2)}\cdot(\log|\overrightarrow{n}|)^{2^{\#(h)+\#(r)}(\sharp_{\mathsf{LR}}(h)+1)}$$

and multiplying this by the upper bound $|\overrightarrow{n}|^{2^{\#(r)}}$ on the number of processes yields that the bound for the total number of processor holds. The output term is of length 1, and the length of the output context is bounded by $|r^*|\lesssim|\overrightarrow{n}|^{2^{\#(r)}}$.

In the case $\mathsf{LR}\,g\,h\,m\,\overrightarrow{s}$ note that, as t has linear type, all the arguments up to and including the m have to be present. Moreover, h is in a complete position, so it cannot contain incomplete free variables, therefore neither can do any of the $\overrightarrow{h'}$; so t' really is linear. Due to the typing restrictions of the LR the step functions $\overrightarrow{hm.}$ have linear type. So in all cases we're entitled to recursively call the algorithm and to apply the induction hypothesis. For calculating m^* we have the same bounds as in the previous cases. We have $k\sim\log|m^*|\lesssim 2^{\#(m)}\log|\overrightarrow{n}|$. The time needed for calculating the $\overrightarrow{h'}$ is

$$\le k2^{\sharp_{\mathsf{LR}}(h)}(|h|+1)(2^{\#(h)}\log|\overrightarrow{n},m^*|)^{\sharp_{\mathsf{LR}}(h)+2}$$

$$\lesssim 2^{\sharp_{\mathsf{LR}}(h)}(|h|+1)(2^{\#(h)+\#(m)}\log|\overrightarrow{n}|)^{\sharp_{\mathsf{LR}}(h)+3}$$

For the length $\left|\overrightarrow{h'}\right|$ we have

$$\begin{aligned}\left|\overrightarrow{h'}\right| &\lesssim (|h|+1)\left(2^{\#(h)}\cdot\log|\overrightarrow{n},m^*|\right)^{\sharp_{\mathsf{LR}}(h)}\\ &\lesssim (|h|+1)\left(2^{\#(h)+\#(m)}\log|\overrightarrow{n}|\right)^{\sharp_{\mathsf{LR}}(h)}\end{aligned}$$

and the length of the numerals $\overrightarrow{\overrightarrow{n}}$ in the contexts output by the computation of the $\overrightarrow{h'}$ is bounded by $|\overrightarrow{n},m^*|^{2^{\#(h)}}\lesssim(|\overrightarrow{n}|^{2^{\#(m)}})^{2^{\#(h)}}=|\overrightarrow{n}|^{2^{\#(m)+\#(h)}}$. For the length of t' we have

$$\begin{aligned}|t'| &\leq k\left|\overrightarrow{h'}\right|+|g|+\overrightarrow{|s|}\\ &\lesssim (|h|+|g|+\overrightarrow{|s|}+1)\left(2^{\#(h)+\#(m)}\log|\overrightarrow{n}|\right)^{\sharp_{\mathsf{LR}}(h)+1}\end{aligned}$$

So the final computation takes time

$$\begin{aligned}&\lesssim\ 2^{\sharp_{\mathsf{LR}}(t')}\,|t'|\,(2^{\#(t')}\log\left|\overrightarrow{n},\overrightarrow{\overrightarrow{n}}\right|)^{\sharp_{\mathsf{LR}}(t')+2}\\ &\lesssim\ 2^{\sharp_{\mathsf{LR}}(g)+\overrightarrow{\sharp_{\mathsf{LR}}(s)}}(|h|+|g|+\overrightarrow{|s|}+1)(2^{\#(h)+\#(m)}\log|\overrightarrow{n}|)^{\sharp_{\mathsf{LR}}(h)+1}\\ &\qquad\cdot(2^{\#(g)+\overrightarrow{\#(s)}}\cdot 2^{\#(m)+\#(h)}\log|\overrightarrow{n}|)^{\sharp_{\mathsf{LR}}(g)+\overrightarrow{\sharp_{\mathsf{LR}}(s)}+2}\\ &\lesssim\ 2^{\sharp_{\mathsf{LR}}(g)+\overrightarrow{\sharp_{\mathsf{LR}}(s)}}(|h|+|g|+\overrightarrow{|s|}+1)(2^{\#(t)}\log|\overrightarrow{n}|)^{\sharp_{\mathsf{LR}}(h)+1+\sharp_{\mathsf{LR}}(g)+\overrightarrow{\sharp_{\mathsf{LR}}(s)}+2}.\end{aligned}$$

So summing up all the times one verifies that the time bound holds. The number of processors needed in the final computation is

$$\begin{aligned}&\lesssim |t'|\cdot\left|\overrightarrow{n},\overrightarrow{\overrightarrow{n}}\right|^{2^{\#(t')}(\sharp_{\mathsf{CR}}(t')+2)}(\log\left|\overrightarrow{n},\overrightarrow{\overrightarrow{n}}\right|)^{2^{\#(t')}(\sharp_{\mathsf{LR}}(t')+1)}\\ &\lesssim (|h|+|g|+\overrightarrow{|s|}+1)\left(2^{\#(h)+\#(m)}\log|\overrightarrow{n}|\right)^{\sharp_{\mathsf{LR}}(h)+1}\\ &\quad\cdot\left(|\overrightarrow{n}|^{2^{\#(m)+\#(h)}}\right)^{2^{\#(t')}(\sharp_{\mathsf{CR}}(t')+2)}\\ &\quad\cdot\left(2^{\#(m)+\#(h)}\log|\overrightarrow{n}|\right)^{2^{\#(t')}(\sharp_{\mathsf{LR}}(t')+1)}\\ &\lesssim (|h|+|g|+\overrightarrow{|s|}+1)\cdot|\overrightarrow{n}|^{2^{\#(m)+\#(h)+\#(t')}(\sharp_{\mathsf{CR}}(t')+2)}\\ &\quad\cdot\left(2^{\#(m)+\#(h)}\log|\overrightarrow{n}|\right)^{\sharp_{\mathsf{LR}}(h)+1+2^{\#(t')}(\sharp_{\mathsf{LR}}(t')+1)}\\ &\lesssim |t|\cdot|\overrightarrow{n}|^{2^{\#(t)}(\sharp_{\mathsf{CR}}(t)+2)}\cdot\left(2^{\#(m)+\#(h)}\log|\overrightarrow{n}|\right)^{2^{\#(t')}(\sharp_{\mathsf{LR}}(h)+\sharp_{\mathsf{LR}}(t')+2)}\\ &\lesssim |t|\cdot|\overrightarrow{n}|^{2^{\#(t)}(\sharp_{\mathsf{CR}}(t)+2)}\cdot(\log|\overrightarrow{n}|)^{2^{\#(m)+\#(h)+\#(t')}(\sharp_{\mathsf{LR}}(h)+\sharp_{\mathsf{LR}}(t')+2)}\end{aligned}$$

The context finally output is bounded by $\lesssim\left|\overrightarrow{n},\overrightarrow{\overrightarrow{n}}\right|^{2^{\#(t')}}\lesssim|\overrightarrow{n}|^{2^{\#(m)+\#(h)+\#(t')}}$. The length of the final output is bounded by

$$\lesssim |t'|\cdot(2^{\#(t')}\log\left|\overrightarrow{n},\overrightarrow{\overrightarrow{n}}\right|)^{\sharp_{\mathsf{LR}}(t')}$$

$$\begin{aligned}
&\lesssim (|h| + |g| + \overrightarrow{|s|} + 1)(2^{\#(h)+\#(m)} \log|\overrightarrow{n}|)^{\sharp_{\mathsf{LR}}(h)+1} \\
&\qquad\qquad\qquad\qquad \cdot (2^{\#(t')+\#(m)+\#(h)} \log|\overrightarrow{n}|)^{\sharp_{\mathsf{LR}}(t')} \\
&\lesssim (|h| + |g| + \overrightarrow{|s|} + 1)(2^{\#(t')+\#(m)+\#(h)} \log|\overrightarrow{n}|)^{\sharp_{\mathsf{LR}}(t')+\sharp_{\mathsf{LR}}(h)+1}
\end{aligned}$$

So all bounds hold in this case.

In the case $\lambda x.r$ note that due to the fact that t has linear type x has to be incomplete, so we're entitled to use the induction hypothesis.

In the case $(\lambda x.r)\, s\, \overrightarrow{s}$ **with *several* occurrences of** x in r note that due to the fact that t is linear, x has to be of ground type (since higher type variables are only allowed to occur once). The time needed to calculate s' is bounded by

$$2^{\sharp_{\mathsf{LR}}(s)}\, |s|\, (2^{\#(s)} \log|\overrightarrow{n}|)^{\sharp_{\mathsf{LR}}(s)+2}$$

and the number of processors is not too high. For the length of s' we have

$$|s'| \;\lesssim\; |s| \cdot (2^{\#(s)} \log|\overrightarrow{n}|)^{\sharp_{\mathsf{LR}}(s)}$$

and $|\overrightarrow{m}| \lesssim |\overrightarrow{n}|^{2^{\#(s)}}$. So the time for calculating s^* is bounded by

$$\lesssim\; |s'| \log|\overrightarrow{n}, \overrightarrow{m}| \;\lesssim\; |s| \cdot (2^{\#(s)} \log|\overrightarrow{n}|)^{\sharp_{\mathsf{LR}}(s)+1}$$

For the length of the numeral s^* we have

$$|s^*| \le |s'| + |\overrightarrow{n}, \overrightarrow{m}| \;\lesssim\; |\overrightarrow{n}|^{2^{\#(s)}}$$

So the last computation takes time

$$2^{\sharp_{\mathsf{LR}}(r\,\overrightarrow{s})} \cdot |r\,\overrightarrow{s}| \left(2^{\#(r\,\overrightarrow{s})+\#(s)} \log|\overrightarrow{n}|\right)^{\sharp_{\mathsf{LR}}(r\,\overrightarrow{s})+2}.$$

Summing up, the time bound holds. The number of processors needed for the last computation is bounded by

$$\begin{aligned}
&\lesssim |r\,\overrightarrow{s}| \cdot \left(|\overrightarrow{n}|^{2^{\#(s)}}\right)^{2^{\#(r\,\overrightarrow{s})}(\sharp_{\mathsf{CR}}(r\,\overrightarrow{s})+2)} \left(2^{\#(s)} \log|\overrightarrow{n}|\right)^{2^{\#(r\,\overrightarrow{s})}(\sharp_{\mathsf{LR}}(r\,\overrightarrow{s})+1)} \\
&\lesssim |t| \cdot |\overrightarrow{n}|^{2^{\#(s)+\#(r\,\overrightarrow{s})}(\sharp_{\mathsf{CR}}(r\,\overrightarrow{s})+2)} (\log|\overrightarrow{n}|)^{2^{\#(s)+\#(r\,\overrightarrow{s})}(\sharp_{\mathsf{LR}}(r\,\overrightarrow{s})+1)}
\end{aligned}$$

The context finally output is bounded by $|\overrightarrow{n}, s^*|^{2^{\#(r\,\overrightarrow{s})}} \lesssim |\overrightarrow{n}|^{2^{\#(s)+\#(r\,\overrightarrow{s})}}$.

In all other cases the bounds trivially hold. □

We conclude that a term of linear type can be simplified by an NC algorithm, where the degree of the runtime bound only depends on the number of occurrences of LR, and the degree of the hardware bound only depends on the number of occurrences of # and CR. More precisely, we have the following corollary.

Corollary 1. *The term* $\mathrm{simp}(t, \vec{x}; \vec{n})$ *and the new context* $\vec{y}; \vec{m}$ *in the above lemma can be computed in time*

$$T(|\vec{n}|) \;\leq\; O((\log|\vec{n}|)^{\sharp_{\mathsf{LR}}(t)+2})$$

by a number of processors satisfying

$$P(|\vec{n}|) \;\leq\; O(|\vec{n}|^{2^{\#(t)}(\sharp_{\mathsf{CR}}(t)+3)})\ .$$

Theorem 3. *Let t be a linear term of type $\vec{\iota} \to \iota$. Then the function denoted by t is in NC.*

Proof. Let $\vec{n}$ be an input, given as 2/3-tree representations of numerals, and $\vec{\mathbf{x}}$ complete variables of type ι. Using Lemma 3, we compute $t' := \mathrm{simp}(t\,\vec{\mathbf{x}}, \vec{\mathbf{x}}; \vec{n})$ and a new context $\vec{y}; \vec{m}$ with $|t'| \leq (\log|\vec{n}|)^{O(1)}$ and $|\vec{m}| \leq |\vec{n}|^{O(1)}$ in time $(\log|\vec{n}|)^{O(1)}$ by $|\vec{n}|^{O(1)}$ many processors.

Then using Lemma 2 we compute the normal form $t'[\vec{y} := \vec{m}]^{\mathrm{nf}}$ in time $O(|t'| \cdot \log|\vec{m}|) = (\log|\vec{n}|)^{O(1)}$ by $O(|t'|\,|\vec{m}|) = |\vec{n}|^{O(1)}$ many processors.

Hence the function denoted by t is computable in polylogarithmic time by polynomially many processors, and thus is in NC. □

From Theorems 2 and 3 we immediately get our main result:

Corollary 2. *A number-theoretic function f is in NC if and only if it is denoted by a linear term of our system.*

References

1. B. Allen. Arithmetizing uniform NC. *Annals of Pure and Applied Logic*, 53(1):1–50, 1991.
2. S. Bellantoni. *Predicative Recursion and Computational Complexity.* PhD thesis, University of Toronto, 1992.
3. S. Bellantoni. Characterizing parallel time by type 2 recursions with polynomial output length. In D. Leivant, editor, *Logic and Computational Complexity*, pages 253–268. Springer LNCS 960, 1995.
4. S. Bellantoni and S. Cook. A new recursion-theoretic characterization of the polytime functions. *Computational Complexity*, 2:97–110, 1992.
5. S. Bellantoni, K.-H. Niggl, and H. Schwichtenberg. Higher type recursion, ramification and polynomial time. *Annals of Pure and Applied Logic*, 104:17–30, 2000.
6. S. Bellantoni and I. Oitavem. Separating NC along the δ axis. Submitted, 2001.
7. S. Bloch. Function-algebraic characterizations of log and polylog parallel time. *Computational Complexity*, 4:175–205, 1994.
8. P. Clote. Sequential, machine independent characterizations of the parallel complexity classes $ALogTIME$, AC^k, NC^k and NC. In S. Buss and P. Scott, editors, *Feasible Mathematics*, pages 49–69. Birkhäuser, 1990.
9. A. Cobham. The intrinsic computational difficulty of functions. In *Proceedings of the second International Congress on Logic, Methodology and Philosophy of Science*, pages 24–30, 1965.

10. K. Gödel. Über eine bisher noch nicht benützte Erweiterung des finiten Standpunktes. *Dialectica*, 12:280–287, 1958.
11. M. Hofmann. Programming languages capturing complexity classes. *ACM SIGACT News*, 31(2), 2000. Logic Column 9.
12. M. Hofmann. Safe recursion with higher types and BCK-algebra. *Annals of Pure and Applied Logic*, 104:113–166, 2000.
13. D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, 2nd edition, 1998.
14. D. Leivant. Stratified functional programs and computational complexity. In *Proc. of the 20th Symposium on Principles of Programming Languages*, pages 325–333, 1993.
15. D. Leivant. A characterization of NC by tree recurrence. In *Proc. 39th Symposium on Foundations of Computer Science*, pages 716–724, 1998.
16. D. Leivant and J.-Y. Marion. A characterization of alternating log time by ramified recurrence. *Theoretical Computer Science*, 236:193–208, 2000.