UNIVERSITY OF OSLO
Department of informatics

An Equational Characterization of the Poly-time Functions on any Constructor Data Structure

Vuokko-Helena Caseiro

Research report 226

ISBN 82-7368-141-6
ISSN 0806-3036

December 1996

An Equational Characterization of the Poly-time Functions on any Constructor Data Structure

Vuokko-Helena Caseiro
Department of Informatics, University of Oslo, P.B. 1080 Blindern, N-0316 Oslo, Norway
Tel: +4722852405 Fax: +4722852401 E-mail: vuokko@ifi.uio.no

December 1996

Abstract

We give a purely syntactical, equational characterization of the poly-time functions on any constructor data structure (free algebra). The equations defining a function f have the shape of simple patterns: $(f\,(c\,y_1 \ldots y_m)\,x_2 \ldots x_n) = r$, where c is a constructor, $y_1,\ldots,y_m$, $x_2,\ldots$, x_n are different variables. There are restrictions on the right-hand sides (rhs) r. The first restrictions concern the general shape of calls to mutually recursive functions, and they imply that we recur on first argument.

To express the two main restrictions on rhs we use a concept of "critical position" which is closely related to the notion "safe" of Bellantoni and Cook, and to the "tiers" of Leivant. A function f's i'th argument position is critical iff in this position f may have access to the result of a recursive call. Then the two main restrictions are (there will be some exceptions for if-then-else, projections and unary addition):

1. The first position of every recursive function is noncritical.
2. Every rhs is linear in all variables from critical positions in the lhs.

Say that a function g on input $X_1, \ldots, X_k$ "doubles" X_i iff the length of $(g\,\overline{X})$ is at least twice the length of X_i. The purpose of (1) and (2) is to forbid doubling of arguments in critical positions. (1) forbids doubling by recursion (which otherwise would have been possible for $i = 1$). (2) forbids explicit doubling of a variable from position i.

1 Introduction and Summary

We consider equations defining functions on data structures built from constructors, e.g. sorting a list constructed from nullary nil and binary cons by using an intermediate binary tree constructed from ternary bic (value, two subtrees) and nullary emp:

$$
\begin{array}{lcl}
\mathsf{treesort}\ l & = & \mathsf{flatten}\ (\mathsf{maketree}\ l\,) \\
\mathsf{maketree\ nil} & = & \mathsf{emp} \\
\mathsf{maketree}\ (\mathsf{cons}\ x\ y) & = & \mathsf{insert}\ (\mathsf{maketree}\ y)\ x \\
\mathsf{insert\ emp}\ x & = & \mathsf{bic}\ x\ \mathsf{emp\ emp} \\
\mathsf{insert}\ (\mathsf{bic}\ v\ l\ r)\ x & = & \mathsf{if}\ (\mathsf{lesseq}\ x\ v)\ \ (\mathsf{bic}\ v\ (\mathsf{insert}\ l\ x)\ r)\ \ (\mathsf{bic}\ v\ l\ (\mathsf{insert}\ r\ x)) \\
\mathsf{flatten\ emp} & = & \mathsf{nil} \\
\mathsf{flatten}\ (\mathsf{bic}\ v\ l\ r) & = & \mathsf{append}\ (\mathsf{flatten}\ l)\ (\mathsf{cons}\ v\ (\mathsf{flatten}\ r)) \\
\mathsf{append\ nil}\ z & = & z \\
\mathsf{append}\ (\mathsf{cons}\ x\ y)\ z & = & \mathsf{cons}\ x (\mathsf{append}\ y\ z)
\end{array}
$$

where if has boolean first argument and is defined by $\mathsf{if\ true}\, x\, y = x$, $\mathsf{if\ false}\, x\, y = y$. Another example is exponentiating a unary number built from nullary 0 and unary succ:

$$
\begin{array}{lcllcl}
\mathsf{exp}_1\ (\mathsf{succ}\ x) & = & \mathsf{double}_1\ (\mathsf{exp}_1\ x) & \mathsf{exp}_1\ \mathsf{0} & = & \mathsf{succ}\ \mathsf{0} \\
\mathsf{double}_1\ (\mathsf{succ}\ x) & = & \mathsf{succ}\ (\mathsf{succ}\ (\mathsf{double}_1\ x)) & \mathsf{double}_1\ \mathsf{0} & = & \mathsf{0}
\end{array}
$$

These two equation sets are examples of "canonical systems", i.e. equation sets where each function f is defined by equations $f\,(c\, y_1 \ldots y_m)\, x_2 \ldots x_n = r$, where c is a constructor, the y_i's and x_j's are different variables, and r is a term with variables among the left-hand side (lhs) variables (the lhs "$\mathsf{treesort}\, l$" is considered shorthand notation).

Our problem is: Can we give syntactical criteria on the right-hand sides (rhs) of canonical equations so that 1) the defined functions are guaranteed to be poly-time, and 2) *any* poly-time function is definable by such equations? And as a side goal, can we be so light-handed that a natural definition like treesort satisfies the criteria?

To the first part of the problem we have already given some answers in [3]. The second part will be answered positively here, it's the main result of the present report. And it turns out that a modified treesort will satisfy the criteria.

Bellantoni and Cook [1] have given a characterization of the poly-time functions on binary numbers, which may be read as an equational characterization of the poly-time functions on any constructor data structure with constructors of arity less than two. In [4], Leivant has studied arbitrary constructor data structures and has given equational characterizations of complexity classes, but for constructors of arity greater than one, his classes exceed poly-time. As far as we know, our problem (2) has sofar been open.

As we read [1], [4], the key idea is to control recursion by saying that, input on which we do recursion is of a different nature than input which is the result of a recursive call; and it should be forbidden to do recursion on the result of a recursive call. E.g. exp_1 recurs on its argument and that's ok since exp_1 doesn't ever receive the result of a recursive call as argument. double_1

also recurs on its argument and that's *not* ok since there's an equation where double_1's argument is the result of a recursive call, $(\mathsf{exp}_1\, x)$.

We will exploit the same idea. We formalize the distinction between different kinds of inputs by defining a partition of argument positions into *critical* and *noncritial.* The main property is that the critical argument positions of a function f are exactly those positions that (directly or indirectly) in some rhs are filled with the result of a recursive call. So e.g. exp_1's position is noncritical, double_1's position is critical. The definition was first given in [3] along with a simple algorithm that, given a canonical system, finds the critical positions.

Following the idea of [1], [4], we should now forbid recursion on arguments in critical positions. So we will, but *why* do we do it? Considering again the example of exp_1, we understand that the real problem with double_1 is not that double_1 recurs on its input, but that double_1 *doubles* its input[1]. If instead of double_1 we used some other definition of doubling, then exp_1 would still be an exponential function. And considering the treesort example we see that both append and insert recur on critical, and intuitively that's ok since these functions *don't* double their input. So we come up with the rule of "Don't Double Criticals".

Basically, there are two ways for a function f to double an argument. The first is by doing recursion on the argument, as double_1 does. The second way of doubling is by having a rhs *nonlinear in the variable* from the related lhs position, and then in some way or other using constructors of arity at least two to put the variable copies together. E.g. $\mathsf{double}_2\, x = \mathsf{cons}\, x\, (\mathsf{cons}\, x\, \mathsf{nil})$. Without constructors of arity at least two, this way of doubling doesn't work; therefore Bellantoni and Cook didn't need to consider it. Leivant didn't consider the second way of doubling either, and in our opinion that explains why his classes (in general) exceed poly-time.

In [3] we formulated the *DDC* (Don't Double Criticals) canonical systems based on the idea of avoiding both ways of doubling criticals, and furthermore on an analysis of needed arguments (from [2]). We showed that any function definable in a *DDC* system is guaranteed to be poly-time. In this report, we will define a class of particularly simple, purely syntactical *DDC* systems, called the $DDC_{\mathsf{if},\pi_i,+}$ systems. In these systems we have dropped the analysis of needed arguments and instead we treat if-then-else (if) specially. A system is $DDC_{\mathsf{if},\pi_i,+}$ if the following hold (and in the formal definition given later we also allow some exceptions for if-then-else and projections):

i) No Inner Doubling: For every equation, for every (mutually) recursive call $(g\, t_1 \ldots t_m)$ in the rhs: Each t_i is built from variables and constructors, and $(g\, t_1 \ldots t_m)$ is linear in all variables.

[1] f on input $X_1, \ldots X_n$ "doubles" its i'th argument if $|f\,\overline{X}| \geq 2|X_i|$.

ii) Recursion on First Argument For each equation $(f\,(c\,y_1\ldots y_m)\,x_2\ldots x_n) = r$, let $(g_1\,t_{1,1}\ldots t_{1,a_{g_1}}),\ldots,(g_k\,t_{k,1}\ldots t_{k,a_{g_k}})$ be the (mutually) recursive calls in r, then $k \leq m$ and each $t_{i,1}$ is a y_j, and $t_{1,1},\ldots,t_{k,1}$ are all different.

iii) Recursion on Noncritical For every recursive function f except unary addition, f's first position is noncritical.

iv) Linear in Critical Every rhs is linear in every critical variable (i.e. in every variable from a lhs critical position).

In the treesort example, (i) - (iv) are ok except that insert doesn't satisfy (ii) and (iii), append doesn't satisfy (iii).

(i) and (ii) concern the "inside", i.e. the *arguments* of (mutually) recursive calls. Intuitively, we have only been reasoning about "outside doubling", but also "inside doubling" can be dangerous, e.g. the exponential function $\mathsf{exp}_2\,(\mathsf{succ}\,x)\,y = \mathsf{exp}_2\,x\,(\mathsf{cons}\,y\,y)$, $\mathsf{exp}_2\,0\,y = y$. Our choice in (i) is to forbid all inside doubling. (ii) implies that we recur on first argument, so (iii) becomes easy to state.

We want to show that *any* poly-time function can be defined by a $DDC_{\mathsf{if},\pi_i,+}$ system, and to do this, we'd like to have a simple, generic machine working on constructor terms. Leivant has already suggested such a machine, but his machine can do too much (double the contents of a register) in a single step. However, by modifying his machine, we obtain our "small step term machine" (SST machine). We then simulate poly-time computations on SST machines by $DDC_{\mathsf{if},\pi_i,+}$ systems. Loosely, we a) compute the length l of the input in unary[2], b) compute a polynomial $p(l)$, c) recur on $p(l)$, simulating one machine step in each "round". Note that's in (a), passing from arbitrary terms (with constructors of arity greater than one) to unary numbers, that we need unary addition to be able to recur on critical. The only difficulty in the simulation is that naturally, one would do *careful* recursion on *critical* to define the "one step" function in (c) (like insert and append do). However, often enough, such recursion can be mimicked by a recursion on noncritical.

This kind of mimicking of recursion on critical can be used more generally - applied to treesort's insert and append, treesort becomes $DDC_{\mathsf{if},\pi_i,+}$.

2 Preliminaries: Function Definitions

Given three disjoint sets, of variables, of constructors with arity and of functions with arity greater than zero, respectively, we define *terms* in the usual way: A variable is a term, and if $t_1,\ldots,t_n$ are terms and h is a constructor

[2]Throughout this report, the length of a constructor term t is the number of constructors in t.

or a function, then $(h\,t_1 \dots t_n)$ is a term. Furthermore $(h\,t_1 \dots t_n)$ is an *application* with h as *head* and the t_i's as *arguments* of h. s is a *subterm* of t if s is t, or if t is an application $(h\,t_1 \dots t_m)$ and s is a subterm of some t_i. We will assume that there's at least one nullary constructor. A *constructor term* is a term built only from constructors.

Define a *canonical equation system* to be a set of equations such that each function f is defined by

$$(f\,(c\,y_1 \dots y_m)\,x_2 \dots x_n) = r$$

where $n \geq 1, m \geq 0$ and there's one equation for each constructor c, where $y_1, \dots y_m, x_2, \dots, x_n$ all are different variables and r is a term with variables among $y_1, \dots y_m, x_2, \dots, x_n$. We consider only finite systems. All our equations will be in this form. As shorthand notation, sometimes we instead define a function f by composition, $(f\,x_1 \dots x_n) = t$, where $x_1, \dots, x_n$ are different variables and t is a term with variables among $x_1, \dots, x_n$. Often, we define functions just for some constructors (e.g. append only on *lists*), then formally, the rhs of the remaining equations can be taken as some nullary constructor.

Let a canonical (equation) system be given. If a function g occurs in the rhs of an equation for f (i.e. f occurs in the lhs) then f *calls* g. If there is a sequence $f_1, f_2, \dots, f_n$ $(n \geq 1)$ of different functions such that f_1 calls f_2, $\dots, f_n$ calls f_1 then each f_i is *recursive*, and every two functions from the sequence are *mutually recursive*. In an equation $e : l = r$ for a function f, if in r there is a subterm t such that t is $(g\,t_1 \dots t_n)$ and g and f are mutually recursive, then t is a *recursive call term in* e, and t has *arguments* $t_1, \dots, t_n$.

3 A "Don't Double Criticals" System

3.1 Critical Positions (from [3])

We intend to define the critical positions such that these are exactly those argument positions that may receive the result of recursive calls. So the naive definition of a critical position is: Argument position number i in function f is critical iff in some equation e's rhs, f's i'th argument is a recursive call term in e. But there are two complications about this: 1) That f's' i'th argument t_i isn't itself a recursive call term, but t_i has a *proper subterm* that is a recursive call term (e.g. if f is double_1 and $\mathsf{exp}_3\,(\mathsf{succ}\,x) = \mathsf{double}_1\,(\mathsf{succ}\,(\mathsf{exp}_3\,x)))$. Also in this case, f's i'th position will be defined to be critical. 2) That arguments are passed from one function (in lhs) to another (in rhs). Then criticality should be "remembered". It's because of this second complication that we will first define critical *variables* and then critical *positions*. Formal definitions now follow:

Definitions

Let a canonical system be given. Note that the definition of critical variables and positions is with respect to this system, but to simplify notation, we don't mark this explicitly. Given an equation $e : (f\,(c\,y_1 \ldots y_m)\,x_2 \ldots x_n) = r$, and a set of positions $u \subseteq \{1, \ldots, n\}$, we define the variable set from e corresponding to u:

$$W_u^e = \{y_j \mid 1 \in u \text{ and } 1 \leq j \leq m\} \cup \{x_i \mid i \in u \text{ and } 2 \leq i \leq n\}$$

E.g. let e_1, e_2 be the equations for append (in Sect. 1), then $W_{\{1\}}^{e_1} = \emptyset, W_{\{1\}}^{e_2} = \{x, y\}$.

Definition 1 (critical variables in an equation)

- Let $e : \mathit{lhs} = \mathit{rhs}$ be an equation in the given canonical system. If there is a subterm $(f\,t_1 \ldots t_m)$ of rhs such that t_i $(1 \leq i \leq m)$ has a subterm which is a recursive call term in e or a critical variable in e, then this induces that in every equation e' for f: Every $v \in W_{\{i\}}^{e'}$ is a *critical variable in* e'.
- A variable is noncritical if it cannot be demonstrated to be critical.

Definition 2 (critical positions) For an n-ary function f defined by k equations $e_1, \ldots, e_k$: Position i, $1 \leq i \leq n$, is *critical* iff every $v \in (W_{\{i\}}^{e_1} \cup \cdots \cup W_{\{i\}}^{e_k})$ is a critical variable.[3]

Note that we haven't said anything about the positions of the constructors.

Consider the $\mathsf{treesort}$ example. The critical positions are the first position in insert, the second and third positions in if, both positions in append.

3.2 If-then-else, Projections, Unary Addition

Define $\mathsf{if}\;\mathsf{true}\,x\,y = x, \mathsf{if}\;\mathsf{false}\,x\,y = y$. Consider a call $t = (\mathsf{if}\,t_1\,t_2\,t_3)$. To compute t, t_1 and either t_2 or t_3 are needed, and the output of t is either t_2 or t_3. We want to define terms to reflect this. Let c_0 be some nullary constructor. Define the function TB with input a term and output a set of terms (TB means Test and Branch):

$$\begin{array}{lcl}
\mathit{TB}(x) & = & \{x\} \quad \text{for every variable } x \\
\mathit{TB}(k\,t_1 \ldots t_m) & = & \{(k\,t'_1 \ldots t'_m) \mid t'_1 \in \mathit{TB}(t_1), \ldots, t'_m \in \mathit{TB}(t_m)\} \\
 & & \text{for } k \text{ a constructor or a function different from } \mathsf{if} \\
\mathit{TB}(\mathsf{if}\,t_1\,t_2\,t_3) & = & \{(\mathsf{if}\,t'_1\,t'_2\,c_0) \mid t'_1 \in \mathit{TB}(t_1), t'_2 \in \mathit{TB}(t_2)\} \cup \\
 & & \{(\mathsf{if}\,t'_1\,c_0\,t'_3) \mid t'_1 \in \mathit{TB}(t_1), t'_3 \in \mathit{TB}(t_3)\}
\end{array}$$

[3]Either all or none of the variables in $W_{\{i\}}^{e_1} \cup \cdots \cup W_{\{i\}}^{e_k}$ are critical.

Define another function $B(t)$ (B means Branch) in the same way, except that

$$B(\mathsf{if}\, t_1\, t_2\, t_3) = \{(\mathsf{if}\, c_0\, t_2'\, c_0) \mid t_2' \in B(t_2)\} \cup \{(\mathsf{if}\, c_0\, c_0\, t_3') \mid t_3' \in B(t_3)\}$$

Define the *i'th projection* π_i by $\pi_i\,(c\,x_1 \ldots x_m) = x_i$ for every nonnullary constructor c and $1 \leq i \leq m$. A term s is a *projection sequence (p.s.)* if $s = \pi_{i_1}\,(\pi_{i_2}\,(\ldots(\pi_{i_n}\,v)\ldots))$ such that $n \geq 0$ and v is a variable[4]. Let t be a term, let s be a particular occurrence of a subterm of t: If s is a p.s., then *s is a p.s. in t*; if s is a p.s. and moreover either s is t or the father of π_{i_1} (when we view t as a tree) is not a projection, then *s is a maximal projection sequence in t*.

Define addition of unary numbers by $+\,\mathsf{0}\,y = y,\ +(\mathsf{succ}\,x)\,y = \mathsf{succ}\,(+\,x\,y)$.

3.3 $DDC_{\mathsf{if},\pi_i,+}$

A term t is *linear* in a term s if t has at most one (occurrence of a) subterm s.

Definition 3 ($DDC_{\mathsf{if},\pi_i,+}$ system) A canonical system is $DDC_{\mathsf{if},\pi_i,+}$ if the following hold:

No Inner Doubling (*NID*) In every equation e, for every recursive call term $(g\,t_1 \ldots t_m)$ in e: Every t_i is built from only variables, constructors and projections, and $(g\,t_1 \ldots t_m)$ is linear in every maximal projection sequence.

Recursion on First Argument (*ROFA*) For every equation $e : f\,(c\,y_1 \ldots y_m)\,\overline{x} = r$, for every $r' \in TB(r)$, let $RCT_{r'}$ be the set of terms t such that t is a particular occurrence of a recursive call term in e and this occurrence t is in r': Every $t \in RCT_{r'}$ has a variable y_i as first argument, and if $t_1 \in RCT_{r'}$ and $t_2 \in RCT_{r'}$ then t_1 and t_2 have different first arguments.

Recursion on Noncritical (*RON*) For every function f different from $+$, f's first position is noncritical.

Linear in Critical (*LIC*) For every rhs r, for every $r' \in B(r)$: r' is linear in every maximal projection sequence s such that s ends with a critical variable.

In the $\mathsf{treesort}$ example, *NID*, *ROFA*, and *LIC* are satisfied, and *RON* is satisfied for all functions except append and insert.

In [3], the *DDC* systems were defined, and we showed that every function definable in a *DDC* system is poly-time (the *length* of a constructor term t

[4]Note that every variable is a projection sequence.

is the number of constructors in t). Every $DDC_{\mathsf{if},\pi_i,+}$ system is obviously a DDC system[5] except that in DDC, $+$'s first position could not be critical. However it's easy to see that if we enlarge a DDC system by allowing $+$'s first position to be critical, the system still guarantees that only poly-time functions are definable[6]. So: Every function in a $DDC_{\mathsf{if},\pi_i,+}$ system is poly-time.

Note why we didn't choose to let every recursive definition have a simple "primitive recursive form": $f\,(c\,\overline{y})\,\overline{x} = h\,\overline{y}\,\overline{x}\,(f\,y_1\,\overline{x})\ldots(f\,y_m\,\overline{x})$. Here *NID* and *ROFA* are satisfied, and *RON* remains as a restriction. But *LIC* cannot be satisfied if there are critical variables and $h \neq \mathsf{if}$. However, [1] and [4] use "primitive recursive form" definitions. Intuitively that works well when constructors have arity less than two since then the only way of doubling is by recursion (so there's no need for *LIC*).

Theorem 1 *For any poly-time function f on a constructor data structure there's a* $\mathrm{DDC}_{\mathit{if},\pi_i,+}$ *system that defines f.*

In the next section, we will prove Theorem 1 by adapting Leivant's idea of simulating machines. As a corollary, the proof shows that if every constructor has arity less than two, then *RON*'s exception for $+$ is not needed.

4 Simulating a Poly-time Machine in $DDC_{\mathsf{if},\pi_i,+}$

Choose a set $C_0 = \{c_0, \ldots, c_p\}$ of constructors, including the nullary constructor $\#$. A *small step term machine* (SST machine) M over C_0 consists of

1. a finite set $S = \{s_0, \ldots, s_n\}$ of *states*, of which s_0 is the *initial state* and s_n is the *final state*
2. an infinite set $P = p_0, p_1, \ldots$ of registers, that contain constructor terms over C_0
3. a finite, nonempty set $H = \{h_0, \ldots, h_m\}$ of *heads*, where $m - 1 \geq$ maximal arity of any constructor. The heads will be considered as "pointers" to registers as well as variables over the nonnegative numbers,
4. a finite set of *commands* s. t. for each state s_j there's exactly one command

A command is one of the following:

[5] Choose trivial fit units except $\{1,2\}, \{1,3\}$ for if, and choose trivial output units except $\{2\}, \{3\}$ for if. Then every fit tree $\tau \in \tau(t)$ corresponds to a unique $t' \in TB(t)$ and vice versa. Analogously for output trees and B.

[6] See [3]: add with trivial units is *poly-basic* and *PBO'* holds. Apply Theorem 15.

branch $(s_j h_i s_{j_0} \dots s_{j_p})$ For (fixed) states $s_j, s_{j_0} \dots s_{j_p}$, (fixed) head h_i: When in state s_j, if the value of p_{h_i} has a $c_k \in C_0$ as head, then switch to state s_{j_k}.

construct $(s_j c_k s_r)$ For states s_j, s_r, constructor c_k of arity $l \geq 0$: When in state s_j, if $h_0, \dots, h_{l-1}$ point to different registers, then store in register p_{h_0} the term resulting from applying c_k to the values of $p_{h_0}, \dots, p_{h_{l-1}}$, then store $\#$ in $p_{h_1}, \dots, p_{h_{l-1}}$. Switch to state s_r.

destruct $(s_j s_r)$ For states s_j, s_r: When in state s_j, let the value of p_{h_0} be a term $(c_k\, a_0 \dots a_{l-1})$. If $l \geq 1$ and the heads $h_0, \dots h_{l-1}$ point to different registers, then store a_i in p_{h_i} $(0 \leq i \leq l-1)$. Switch to state s_r.

move head right $(s_j h_i s_r)$ For states s_j, s_r, head h_i: When in state s_j, let head h_i point to the next register, and switch to state s_r.

move head left $(s_j h_i s_r)$ For states s_j, s_r, head h_i: When in state s_j, if h_i doesn't point to p_0 then let head h_i point to the previous register. Switch to state s_r.

swap $(h_i h_j s_k s_r)$ For states s_k, s_r, heads h_i, h_j: When in state s_k, let h_i point to h_j's register and let h_j point to h_i's register, and switch to state s_r.

do nothing (s_n) When in state s_n don't do anything.

M is deterministic. As a special case (use only nullary constructors) one obtains an ordinary turing machine with a one-way infinite tape.

A *configuration* is a tuple $(h_0, \dots, h_m, s_i, u_0, \dots, u_k)$ such that $h_0, \dots, h_m$ are the values of the heads, s_i is the state, $u_0, \dots, u_k$ are the values of the first $k+1$ registers where k is such that a) there's a head pointing to p_k or the value of p_k is not $\#$, and b) for every register p_j, $j > k$: There's no head pointing to p_j and the value of p_j is $\#$.

An SST machine M *computes* a a $k+1$-ary function f on C_0 terms if $f(x_0, \dots, x_k) = y$ iff when M starts in configuration $(0, \dots, 0, s_0, x_0, \dots, x_k)$, then in a finite number of steps, M reaches configuration $(0, \dots, 0, s_n, y)$.

Our SST machine is a modification of the deterministic register machine of Leivant in [4]. His machine has only a *finite* set of registers and no heads. His commands are *branch* -the same as ours; *construct* - store in register p the result of applying constructor c to the contents of registers $p_1, \dots, p_k$, and change state; j*-destruct* - store in register p the j'th immediate subterm of the term in register q, and change state.

We couldn't use Leivant's register machine, since his construct and destruct commands are too strong. E.g. the subterms to be combined in "construct" may all be taken from the same register p and the result put in

p, and so in one single step, the contents of a register may be doubled. In this way e.g. the *exponential* function exp_2 defined by $\mathsf{exp}_2\,(\mathsf{succ}\,x)\,y = \mathsf{exp}_2\,x\,(\mathsf{cons}\,y\,y)$, $\mathsf{exp}_2\,\mathsf{0}\,y = y$, may be computed in *polynomially* many machine steps as follows: Define the following Leivant machine M: M uses the constructors 0, succ, nil, cons. M has four states, s_b, s_c, s_d, s_{stop}, and we start in s_b and terminate in s_{stop}. M has two registers p_1, p_2. There are three commands:

- When in state s_b, if the head of the value of register p_1 is succ, then switch to state s_c, else switch to state s_{stop}.
- When in state s_c, store in register p_2 the term resulting from applying cons to the values of of register p_2 and p_2, and switch to state s_d.
- When in state s_d, store in register p_1 the first immediate subterm of the term in p_1, if it exists, and switch to state s_b.

Starting in state s_b with $(\mathsf{succ}^n\,\mathsf{0})$ in register p_1 and some L in register p_2, in $3 \times n + 1$ steps we reach s_{stop} with a term of length $\geq 2^{n+1} - 1$ in register p_2[7].

So instead we have introduced the SST machine. There's no implicit duplication, but instead the SST machine has infinitely many registers accessed by a finite number of heads (in this way, one can still double the contents of a register, but in many small steps).

4.1 Simulating One Step in an SST Machine

Choose an SST machine M with constructor set C_0, states $s_0, \ldots, s_n$, registers $p_0, p_1, \ldots$, heads $h_0, \ldots, h_m$.

We will define a canonical system O_M over C_0 and nullary 0, nil, true, false, *, unary succ, binary cons[8]. We code each state s_i in unary by $\mathsf{succ}^i\,\mathsf{0}$, we code the values of the heads in unary.

As list abbreviations we use $[e_1, \ldots, e_n]$ for $\mathsf{cons}\,e_1\,(\mathsf{cons}\,e_2 \ldots (\mathsf{cons}\,e_n\,\mathsf{nil}) \ldots)$, and we use $[e_1, \ldots, e_n \mid L]$ for $\mathsf{cons}\,e_1\,(\mathsf{cons}\,e_2 \ldots (\mathsf{cons}\,e_n\,L) \ldots)$.

We code a configuration $(h_0, \ldots, h_m, s, u_0, \ldots, u_j)$ by any list $[h_0, \ldots, h_m, s, u_0, \ldots, u_k]$ such that $k \geq j$ and every p_j with $j > k$ contains $\#$. So the representation of a configuration might actually be longer than the real configuration. (In the simulation below, we "keep on to" all registers we have ever touched.)

Below are some common functions in O_M. l-different tests if l unary numbers are different. Each $\mathsf{head?}_c$ tests if the input starts with constructor c or not. We need equality eq only on unary numbers and *. np means "next pair" and the intended use is to call np repeatedly to produce sequences of pairs $[k, 0], [k-1, 1], \ldots, [0, k], [*, k], [*, k] \ldots$

[7]Note: Leivant defined the length of a contructor term t to be the height of t as a tree.

[8]It's implicit that if these special constructors 0 etc. are in C_0, they must have the same arity there.

$$
\begin{array}{lcl}
\mathsf{if}\,\mathsf{true}\,x\,y & = & x\\
\mathsf{if}\,\mathsf{false}\,x\,y & = & y\\
\mathsf{append}\,\mathsf{nil}\,z & = & z\\
\mathsf{append}\,(\mathsf{cons}\,x\,y)\,z & = & \mathsf{cons}\,x(\mathsf{append}\,y\,z)\\
\\
\mathsf{rev}\,\mathsf{nil} & = & \mathsf{nil}\\
\mathsf{rev}\,(\mathsf{cons}\,x\,y) & = & \mathsf{append}\,(\mathsf{rev}\,y)\,[x]\\
\\
\mathsf{np}\,(\mathsf{cons}\,x\,y) & = & r_1\,x\,y\\
r_1\,(\mathsf{succ}\,x)\,y & = & [x, \mathsf{succ}\,(\pi_1\,y)]\\
r_1\,\mathsf{0}\,y & = & [* \mid y]\\
r_1\,*\,y & = & [* \mid y]\\
\\
\mathsf{l\text{-}different}\,h_0\ldots h_{l-1} & = & \mathsf{if}\,(\mathsf{eq}\,h_0\,h_1)\,\mathsf{false}\\
 & & (\mathsf{if}\,(\mathsf{eq}\,h_0\,h_2)\,\mathsf{false}\,(\ldots\\
 & & (\mathsf{if}\,(\mathsf{eq}\,h_0\,h_l)\,\mathsf{false}\\
 & & (\mathsf{if}\,(\mathsf{eq}\,h_1\,h_2)\,\mathsf{false}\,(\ldots\\
 & & (\mathsf{if}\,(\mathsf{eq}\,h_{l-2}\,h_{l-1})\,\mathsf{false}\,\mathsf{true})\ldots)))\ldots))\\
\\
\pi_i\,(c\,x_1\ldots x_n) & = & x_i \quad \text{for all } i,n \text{ such that } 1 \leq i \leq n\\
\\
\mathsf{head?}_c\,(\mathsf{c}\,\overline{y}) & = & \mathsf{true} \quad \text{for all } c \in C\\
\mathsf{head?}_c\,(\mathsf{d}\,\overline{y}) & = & \mathsf{false}, \quad \text{for all } d, c \in C, d \neq c\\
\\
\mathsf{eq}\,\mathsf{0}\,x & = & \mathsf{head?}_0\,x\\
\mathsf{eq}\,(\mathsf{succ}\,y)\,x & = & h_{\mathsf{succ}}\,x\,y\\
\mathsf{eq}\,*\,x & = & \mathsf{head?}_*\,x\\
\mathsf{h}_{\mathsf{succ}}\,(\mathsf{succ}\,z)\,y & = & \mathsf{eq}\,y\,z\\
h_{\mathsf{succ}}\,\mathsf{0}\,y & = & \mathsf{false}
\end{array}
$$

Below, **onestep** expects the representation of a configuration.

$$
\begin{array}{lcl}
\mathsf{onestep}\,(\mathsf{cons}\,h_0\,L) & = & \mathsf{b}_1\,L\,h_0\\
\mathsf{b}_1\,(\mathsf{cons}\,h_1\,L)\,h_0 & = & \mathsf{b}_2\,L\,h_0\,h_1\\
\mathsf{b}_2\,(\mathsf{cons}\,h_2\,L)\,h_0\,h_1 & = & \mathsf{b}_3\,L\,h_0\,h_1\,h_2\\
\vdots & & \\
\mathsf{b}_{m+1}\,(\mathsf{cons}\,s\,L)\,h_0\ldots h_m & = & \mathsf{if}\,(\mathsf{eq}\,s\,\mathsf{0})\,t_0\\
 & & (\mathsf{if}\,(\mathsf{eq}\,s\,(\mathsf{succ}\,\mathsf{0}))\,t_1\ \ldots\\
 & & (\mathsf{if}\,(\mathsf{eq}\,s\,(\mathsf{succ}^n\,\mathsf{0}))\,t_n\,\mathsf{false})\ldots)
\end{array}
$$

In the following we will define each t_j.

4.1.1 branch $s_j h_i s_{j_0} \dots s_{j_p}$

Then t_j is $\mathsf{s}_j\text{-}\mathsf{branch}\, h_0\, L\, h_1 \dots h_{l-1}\, h_{l+1} \dots h_m\, \mathsf{0}\, \mathsf{nil}$, where $l \geq 0$. Let $\overline{h} = h_1 \dots h_{l-1}\, h_{l+1} \dots h_m$ in

$$
\begin{array}{lcl}
\mathsf{s}_j\text{-}\mathsf{branch}\,(\mathsf{succ}\, n)\, L\, \overline{h}\, h'\, L' & = & \mathsf{s}_j\text{-}\mathsf{b}\, L\, n\, \overline{h}\,(\mathsf{succ}\, h')\, L' \\
\mathsf{s}_j\text{-}\mathsf{branch}\, \mathsf{0}\;\, L\, \overline{h}\, h'\, L' & = & \mathsf{s}_j\text{-}\mathsf{finish}\text{-}\mathsf{branch}\, L\, \overline{h}\, h'\, L' \\
\mathsf{s}_j\text{-}\mathsf{b}\,(\mathsf{cons}\, x\, L)\, n\, \overline{h}\, h'\, L' & = & \mathsf{s}_j\text{-}\mathsf{branch}\, n\, L\, \overline{h}\, h'\,(\mathsf{cons}\, x\, L') \\
\mathsf{s}_j\text{-}\mathsf{finish}\text{-}\mathsf{branch}\,(\mathsf{cons}\, x\, L)\, \overline{h}\, h'\, L' & = & [h_1, \dots, h_{i-1}, h', h_{i+1}, \dots, h_m, \mathit{state} \mid \\
 & & (\mathsf{append}\,(\mathsf{rev}\, L')\,(\mathsf{cons}\, x\, L))]
\end{array}
$$

where *state* is the following term:

$$\mathsf{if}\,(\mathsf{head?}_{c_0}\, x)\, s_{j_0}\;\; (\mathsf{if}\,(\mathsf{head?}_{c_1}\, x)\, s_{j_1}\;\; \dots (\mathsf{if}\,(\mathsf{head?}_{c_p}\, x)\, s_{j_p}\;\; \mathsf{false}) \dots)$$

4.1.2 construct $s_j c_k s_r$

Then according to the arity l of c_k, t_j is $\mathsf{l}\text{-}\mathsf{construct}\, h_0 \dots h_m\, s_r\,(c_k\, \mathsf{0} \dots \mathsf{0})\, L$.
If $l = 0$ then

$$\mathsf{0}\text{-}\mathsf{construct}\, h_0 \dots h_m\, s_r\, c\, L = \mathsf{insert}\, h_0\, L\, h_1 \dots h_m\, s_r\, c\, \mathsf{0}\, \mathsf{nil}$$

If $l = 1$ then

$$\mathsf{1}\text{-}\mathsf{construct}\, h_0 \dots h_m\, s_r\, c\, L = (\mathsf{1}\text{-}\mathsf{extract}\, L\;\, [h_0, 0]\, h_1 \dots h_m\, s_r\, c\, \#\, \mathsf{nil})$$

If $l > 1$ then (below, there are l #'s)

$$
\begin{array}{lcl}
\mathsf{l}\text{-}\mathsf{construct}\, h_0 \dots h_m\, s_r\, c\, L & = & \mathsf{if}\,(\mathsf{l}\text{-}\mathsf{different}\, h_0 \dots h_{l-1}) \\
 & & (\mathsf{l}\text{-}\mathsf{extract}\, L\;\, [h_0, 0]\, [h_1, 0] \dots [h_{l-1}, 0]\, h_l \dots h_m\, s_r\, c\, \# \dots \#\, \mathsf{nil}) \\
 & & [h_0, \dots, h_m, s_r \mid L]
\end{array}
$$

Let $\overline{h} = h_l \dots h_m$ in $\mathsf{l}\text{-}\mathsf{extract}$.

$$
\begin{array}{l}
\mathsf{l}\text{-}\mathsf{extract}\,(\mathsf{cons}\, x\, L)\, p_0\, p_1 \dots p_{l-1}\, \overline{h}\, s_r\, c\, a_0 \dots a_{l-1}\, L' = \\
\mathsf{if}\,(\mathsf{eq}\,(\pi_1\, p_0)\, 0)\;\, (\mathsf{l}\text{-}\mathsf{extract}\, L\,(\mathsf{np}\, p_0) \dots (\mathsf{np}\, p_{l-1})\, \overline{h}\, s_r\, c\, x\, a_1 \dots a_{l-1}\, [\# \mid L']) \\
(\mathsf{if}\,(\mathsf{eq}\,(\pi_1\, p_1)\, 0)\;\, (\mathsf{l}\text{-}\mathsf{extract}\, L\;\, (\mathsf{np}\, p_0) \dots (\mathsf{np}\, p_{l-1})\, \overline{h}\, s_r\, c\, a_0\, x\, a_2 \dots a_{l-1}\, [\# \mid L']) \\
\vdots \\
(\mathsf{if}\,(\mathsf{eq}\,(\pi_1\, p_{l-1})\, 0)\;\, (\mathsf{l}\text{-}\mathsf{extract}\, L\,(\mathsf{np}\, p_1)\; \dots (\mathsf{np}\, p_{l-1})\, \overline{h}\, s_r\, c\, a_0 \dots a_{l-2}\, x\, [\# \mid L']) \\
(\mathsf{l}\text{-}\mathsf{extract}\, L\;\, (\mathsf{np}\, p_0)\; \dots (\mathsf{np}\, p_{l-1})\, \overline{h}\, s_r\, c\, a_0 \dots a_{l-1}\, [x \mid L'])) \dots)
\end{array}
$$

$$
\begin{array}{l}
\mathsf{l}\text{-}\mathsf{extract}\, \mathsf{nil}\, p_0 \dots p_{l-1}\, h_l \dots h_m\;\, s_r\, c\, a_0 \dots a_{l-1}\, L' = \\
\mathsf{insert}\,(\pi_1\,(\pi_2\, p_0))\,(\mathsf{rev}\, L')\,(\pi_1\,(\pi_2\, p_1))\; \dots (\pi_1\,(\pi_2\, p_{l-1}))\, h_l \dots h_m\, s_r\,(\mathsf{join}\, c\, a_0 \dots a_{l-1})\, \mathsf{0}\, \mathsf{nil}
\end{array}
$$

join is defined for every nonnullary constructor c as:

$$\mathsf{join}\,(c\, x_0 \dots x_{l-1})\, a_0 \dots a_{l-1} = c\, a_0 \dots a_{l-1}$$

Let $\overline{h} = h_1 \dots h_m$ in insert.

$$
\begin{array}{lcl}
\mathsf{insert}\,(\mathsf{succ}\, n)\, L\, \overline{h}\, s_r\, a\, h'\, L' & = & \mathsf{ins}\, L\, n\, \overline{h}\, s_r\, a\,(\mathsf{succ}\, h')\, L' \\
\mathsf{insert}\, \mathsf{0}\, L\, \overline{h}\, s_r\, a\, h'\, L' & = & [h', \overline{h}, s_r \mid (\mathsf{append}\,(\mathsf{rev}\, L')\,(\mathsf{cons}\, a\,(\pi_2\, L)))] \\
\mathsf{ins}\,(\mathsf{cons}\, x\, L)\, n\, \overline{h}\, s_r\, a\, h'\, L' & = & \mathsf{insert}\, n\, L\, \overline{h}\, s_r\, a\, h'\,(\mathsf{cons}\, x\, L')
\end{array}
$$

4.1.3 destruct $s_j s_r$

Then t_j is $\mathsf{destruct}\, h_0 \ldots h_m\, L\, s_r$. Let $\overline{h} = h_1 \ldots h_m$ in the definition of destruct, find and fi:

$$\begin{array}{rcl}
\mathsf{destruct}\, h_0 \ldots h_m\, L s_r & = & \mathsf{if}\,(\mathsf{l\text{-}different}\, h_0 \ldots h_{l-1}) \\
& & (\mathsf{find}\, h_0\, L\, \overline{h}\, s_r\, \mathsf{0}\, \mathsf{nil}) \\
& & [h_0, \ldots, h_m, s_r \mid L]
\end{array}$$

$$\begin{array}{rcl}
\mathsf{find}\,(\mathsf{succ}\, y)\, L\, \overline{h}\, s_r\, h'\, L' & = & \mathsf{fi}\, L\, y\, \overline{h}\, s_r\, (\mathsf{succ}\, h')\, L' \\
\mathsf{find}\, \mathsf{0}\, L\, \overline{h}\, s_r\, h'\, L' & = & \mathsf{pick1}\, L\, h'\, \overline{h}\, s_r\, L' \\
\mathsf{fi}\,(\mathsf{cons}\, x\, L)\, y\, \overline{h}\, s_r\, h'\, L' & = & \mathsf{find}\, y\, L\, \overline{h}\, s_r\, h'\,(\mathsf{cons}\, x\, L')
\end{array}$$

Let $\overline{h} = h_0 \ldots h_m$ in the definition of pick1 and pick2:

$$\begin{array}{rcl}
\mathsf{pick1}\,(\mathsf{cons}\, a\, L)\, \overline{h}\, s_r\, L' & = & \mathsf{pick2}\, a\, \overline{h}\, s_r\, (\mathsf{append}\,(\mathsf{rev}\, L')\,(\mathsf{cons}\, \#\, L)) \\
\mathsf{pick2}\,(c\, a_0 \ldots a_{l-1})\, \overline{h}\, s_r\, L & = & \mathsf{l\text{-}putin}\, L\, [h_0, \mathsf{0}] \ldots [h_{l-1}, \mathsf{0}]\, h_l \ldots h_m\, s_r\, a_0 \ldots a_{l-1}\, \mathsf{nil} \\
\mathsf{pick2}\, c\, \overline{h}\, s_r\, L & = & [\overline{h}, s_r \mid L] \quad \text{nullary } c
\end{array}$$

Let $\overline{h} = h_l \ldots h_m$ in the definition of l-putin:

$$\begin{array}{l}
\mathsf{l\text{-}putin}\,(\mathsf{cons}\, x\, L)\, \overline{p}\, \overline{h}\, s_r\, a_0 \ldots a_{l-1}\, L' = \\
\mathsf{if}\,(\mathsf{eq}\,(\pi_1\, p_0)\, \mathsf{0})\;\; (\mathsf{l\text{-}putin}\, L\,(\mathsf{np}\, p_0) \ldots (\mathsf{np}\, p_{l-1})\, \overline{h}\, s_r\, \#\, a_1 \ldots a_{l-1}\,(\mathsf{cons}\, a_0\, L')) \\
(\mathsf{if}\,(\mathsf{eq}\,(\pi_1\, p_1)\, \mathsf{0})\;\; (\mathsf{l\text{-}putin}\, L\,(\mathsf{np}\, p_0) \ldots (\mathsf{np}\, p_{l-1})\, \overline{h}\, s_r\, a_0\, \#\, a_2 \ldots a_{l-1}\,(\mathsf{cons}\, a_1\, L')) \\
\vdots \\
(\mathsf{if}\,(\mathsf{eq}\,(\pi_1\, p_{l-1})\, \mathsf{0})\;\; (\mathsf{l\text{-}putin}\, L\,(\mathsf{np}\, p_0) \ldots (\mathsf{np}\, p_{l-1})\, \overline{h}\, s_r\, a_0 \ldots \#\,(\mathsf{cons}\, a_{l-1}\, L')) \\
(\mathsf{l\text{-}putin}\, L\,(\mathsf{np}\, p_0) \ldots (\mathsf{np}\, p_{l-1})\, \overline{h}\, s_r\, a_0 \ldots a_{l-1}\,(\mathsf{cons}\, x\, L'))) \ldots)
\end{array}$$

$$\mathsf{l\text{-}putin}\, \mathsf{nil}\, \overline{p}\, \overline{h}\, s_r\, a_0 \ldots a_{l-1}\, L' = [\pi_1\,(\pi_2\, p_0), \ldots, \pi_1\,(\pi_2\, p_{l-1}), \overline{h}, s_r \mid (\mathsf{rev}\, L')]$$

4.1.4 move head right $h_i s_j s_r$

Then t_j is
$[h_0, \ldots, h_{i-1}, (\mathsf{succ}\, h_i), h_{i+1}, \ldots, h_m, s_r \mid \mathsf{if}\;(\mathsf{lesseq}\,(\mathsf{noelem}\, L)\,(\mathsf{succ}\, h_i))\;\; (\mathsf{append}\, L\, [\#])\;\; L]$
where

$$\begin{array}{rcl}
\mathsf{noelem}\,(\mathsf{cons}\, x\, l) & = & \mathsf{succ}\,(\mathsf{noelem}\, l) \\
\mathsf{noelem}\, \mathsf{nil} & = & \mathsf{0} \\
\mathsf{lesseq}\, \mathsf{0}\, y & = & \mathsf{true} \\
\mathsf{lesseq}\,(\mathsf{succ}\, x)\, y & = & \mathsf{if}\,(\mathsf{head?}_0\, y)\, \mathsf{false}\,(\mathsf{lesseq}\, x\,(\pi_1\, y))
\end{array}$$

4.1.5 move head left $h_i s_j s_r$

Then t_j is $[h_0, \ldots, h_{i-1}, (\pi_1\, h_i), h_{i+1}, \ldots, h_m, s_r \mid L]$.

4.1.6 swap $h_i h_j s_k s_r$

If $i < j$ then t_j is $[h_0, \ldots, h_{i-1}, h_j, h_{i+1}, \ldots, h_{j-1}, h_i, h_{j+1}, \ldots, h_m, s_j \mid L]$, else if $i > j$ then t_j is $[h_0, \ldots, h_{j-1}, h_i, h_{j+1}, \ldots, h_{i-1}, h_j, h_{i+1}, \ldots, h_m, s_j \mid L]$, else if $i = j$ then t_j is $[h_0, \ldots, h_m, s_j \mid L]$.

4.1.7 do nothing s_n

Then t_n is $[h_0, \ldots, h_m, s_n \mid L]$.

Lemma 1 *Let $[h_0, \ldots, h_m, s_i, \overline{u}]$ be a representation of an M configuration. We have that* $\mathsf{onestep}\,[h_0, \ldots, h_m, s_i, \overline{u}] = [h'_0, \ldots, h'_m, s_j, \overline{v}]$ *iff M passes from the first to the second of the corresponding configurations.*

4.2 Simulating Poly-time Computations by a Canonical System

Let M be an SST machine over a constructor set C_0, that computes a function $f(x_0, \ldots, x_k)$ in time (with a number of steps) less than or equal to $a(|x_0|+\cdots+|x_k|)^b$, where $|t|$ is the number of constructors in the constructor term t, a and b are positive integers.

We will define a canonical system S_M to consist of the following equations plus the equations in O_M (onestep's position is critical!). q-lengthsum is defined for $q = k$ and for every positive q less than or equal to the maximal arity of any $c \in C_0$.

$\mathsf{f}\ \overline{x}$	$=$	$\mathsf{take}\ (\mathsf{succ}^{m+2}\ 0)\ (\mathsf{compute}\ (\mathsf{pol}_{a,b}\ (\text{k+1-lengthsum}\ \overline{x}))\ \overline{x})$
$\mathsf{compute}\ 0\ x_0 \ldots x_k$	$=$	$[0, \ldots, 0, 0, x_0, \ldots, x_k]$ m+1+1 $0's$
$\mathsf{compute}\ (\mathsf{succ}\ y)\ x_0 \ldots x_k$	$=$	$\mathsf{onestep}\ (\mathsf{compute}\ y\ x_0 \ldots x_k)$
$\mathsf{pol}_{a,0}\ x$	$=$	$\mathsf{succ}^a\ 0$
$\mathsf{pol}_{a,i+1}\ x$	$=$	$\times\ x\ (\mathsf{pol}_{a,i}\ x)$ note that $\mathsf{pol}_{a,i}\ x = \mathsf{succ}^{a(\lvert x\rvert-1)^i}\ 0$
$\mathsf{take}\ (\mathsf{succ}\ x)\ y$	$=$	$\mathsf{take}\ x\ (\pi_2\ y)$
$\mathsf{take}\ 0\ y$	$=$	$\pi_1\ y$
$\text{q-lengthsum}\ x_1 \ldots x_q$	$=$	$+\ (\mathsf{length}\ x_1)\ (+\ (\mathsf{length}\ x_2) \ldots \mathsf{length} x_q) \ldots)$ $q \geq 2$
$\text{1-lengthsum}\ x$	$=$	$\mathsf{length}\ x$
$\mathsf{length}\ c$	$=$	$\mathsf{succ}\ 0$
$\mathsf{length}\ (c\ x)$	$=$	$\mathsf{succ}\ (\mathsf{length}\ x)$
$\mathsf{length}\ (c\ x_1 \ldots x_q)$	$=$	$\mathsf{succ}\ (\text{q-lengthsum}\ x_1 \ldots x_q)$ $q \geq 2$
$\times\ 0\ y$	$=$	0
$\times\ (\mathsf{succ}\ x)\ y$	$=$	$+\ y\ (\times\ x\ y)$
$+\ 0\ y$	$=$	y
$+\ (\mathsf{succ}\ x)\ y$	$=$	$\mathsf{succ}\ (+\ x\ y)$

Lemma 2 *For all C_0 terms $x_0, \ldots, x_k$: $f(x_0, \ldots, x_k) = y$ iff $\mathsf{f} x_0 \ldots x_k = y$.*

Proof of Lemma 2 Using Lemma 1 repeatedly plus "do nothing" in s_n we get that M has a computation of length l starting with a configuration $(0,\ldots,0,s_0,\overline{x})$ and ending with a configuration $(h_0,\ldots,h_m,s_n,\overline{u})$ iff for every $d \geq l$ and for some nonnegative number of #'s, $(\mathsf{compute}\,(\mathsf{succ}^d\,0)\,\overline{x}) = [h_0,\ldots,h_m,s_n,\overline{u},\overline{\#}]$.

So we obtain that for all C_0 terms $x_0,\ldots,x_k$: $f(x_0,\ldots,x_k) = y$ iff $\mathsf{f}\,x_0\ldots x_k = y$. ○

4.3 Simulating Poly-time Computations in $\boldsymbol{DDC}_{\mathsf{if},\pi_i,+}$

S_M isn't $DDC_{\mathsf{if},\pi_i,+}$ since *NID* doesn't hold (the use of np in l-extract and l-putin offends), *ROFA* doesn't hold (s_j-branch, insert, find, eq offend), *RON* doesn't hold (the input to onestep is critical, so all functions below onestep have only critical positions).

Define a canonical system S to be a *preDDC*$_{\mathsf{if},\pi_i,+}$ system if the following are satisfied for S: *NID*, *LIC*, and

pre1: For every equation $e : l = r$, for every $r' \in TB(r)$: There's at most one recursive call term t in e such that t is a subterm of r'.

pre2: The length of any recursion is bounded by the length of the input, i.e. for any call $(f_1\,X_1\ldots X_n)$ (where $X_1,\ldots,X_n$ are constructor terms) that provokes a call sequence $f_1 \to f_2 \to \cdots \to f_k$ such that every pair f_i, f_j is mutually recursive, we have $k \leq \sum_{i=1}^{n} |X_i|$.

pre3: There's a nonnegative constant δ such that for any n-ary f in S, for any constructor terms $X_1,\ldots,X_n$, let r be the rhs of the equation such that $(f\,\overline{X})$ matches the lhs, let σ be the matching substitution, then for any subterm $t = (g\,t_1\ldots t_m)$ of r such that $g \neq \mathsf{if}$ we have that $\sum_{i=1}^{m} |t_i\sigma| \leq \sum_{i=1}^{n} |X_i| + \delta$.

Lemma 3 *Divide S_M in two at onestep, i.e. let $F_1 = \{$ f, compute, $pol_{a,0}$, $\ldots$, $pol_{a,b}$, take, q-lengthsum's, length, $\times$, $+$, π_1, π_2 $\}$, let F_2 consist of all the functions that define onestep for machine M*[9]*. Then the equations for F_1 make up an incomplete* DDC$_{\mathit{if},\pi_i,+}$ *system, and with a minor adjustment for np, the equations for F_2 make up a* preDDC$_{\mathit{if},\pi_i,+}$ *system.*

Proof of Lemma 3 Obviously F_1's system is $DDC_{\mathsf{if},\pi_i,+}$ (only $+$ has critical positions) except that the definition of onestep is lacking.

The system for F_2 is almost *preDDC*$_{\mathsf{if},\pi_i,+}$ - the only problem is that *NID* isn't satisfied in l-extract and l-putin since they use np. But instead of np we can explicitly test for whether each pair p is $[\mathsf{succ}\,x \mid y]$ or $[0 \mid y]$ or $[* \mid y]$ before doing the recursive call to l-extract (or l-putin). Then in the

[9] $F_1 \cap F_2 = \{\pi_1, \pi_2\}$

first case use $[\pi_1\,(\pi_1\,p) \mid \mathsf{succ}\,(\pi_1\,(\pi_2\,p))]$, in the second and third case use $[* \mid (\pi_2\,p)]$. For example, 2-extract's first test and then-branch are

$\mathsf{if}\,(\mathsf{eq}\,(\pi_1\,p_0)\,0)$ Note: As before
$\mathsf{if}\,(\mathsf{head?}_*\,(\pi_1\,p_1))$ Note: New from here onwards
$(\mathsf{if}\,(\mathsf{head?}_*\,(\pi_1\,p_2))$
$(\mathsf{2\text{-}extract}\,L\,[* \mid (\pi_2\,p_0)]\,[* \mid (\pi_2\,p_1)]\,[* \mid (\pi_2\,p_2)]\,\overline{s})$
$(\mathsf{2\text{-}extract}\,L\,[* \mid (\pi_2\,p_0)]\,[* \mid (\pi_2\,p_1)]\,[\pi_1\,(\pi_1\,p_2) \mid \mathsf{succ}\,(\pi_1\,(\pi_2\,p_2))]\,\overline{s}))$
$(\mathsf{if}\,(\mathsf{head?}_*\,(\pi_1\,p_2))$
$(\mathsf{2\text{-}extract}\,L\,[* \mid (\pi_2\,p_0)]\,[\pi_1\,(\pi_1\,p_1) \mid \mathsf{succ}\,(\pi_1\,(\pi_2\,p_1))]\,[* \mid (\pi_2\,p_2)]\,\overline{s})$
$(\mathsf{2\text{-}extract}\,L\,[* \mid (\pi_2\,p_0)]\,[\pi_1\,(\pi_1\,p_1) \mid \mathsf{succ}\,(\pi_1\,(\pi_2\,p_1))]\,[\pi_1\,(\pi_1\,p_2) \mid \mathsf{succ}\,(\pi_1\,(\pi_2\,p_2))]\,\overline{s}))$

where $\overline{s} = \overline{h}\,s_r\,c\,x\,a_1 \ldots a_2\,[\# \mid L']$.

Then *NID*, *LIC*, **pre1**, **pre2**, are satisfied, and as for **pre3**, by inspection, the arguments to functions in rhs are variables, constructors, projections, rev, append, join, - δ may be taken as $4m + n + 5$. ○

Lemma 4 *For every* preDDC$_{\mathsf{if},\pi_i,+}$ *system S there's a* DDC$_{\mathsf{if},\pi_i,+}$ *system R such that R mimicks S, i.e. for every n-ary f in S there's an $n+2$-ary function $f^\bullet$ in R such that for all constructor terms $X_1, X_2, X_3, \ldots, X_{n+2}$: If $|X_1| = |X_2| > \sum_{i=3}^{n+2} |X_i|$ then $(f\,X_3 \ldots X_{n+2}) = (f^\bullet\,X_1 \ldots\ldots X_{n+2})$.*

Proof of Lemma 4 Let a *preDDC*$_{\mathsf{if},\pi_i,+}$ system S be given. The proof idea is: In order to obtain *ROFA* and *RON* we will provide each function with two new noncritical arguments, do recursion on the first of them and keep the length of the original arguments in the second. Formally:

We define a canonical system R: If S has if, then R has if. For every constructor c in S, R has a function $\mathsf{head?}_c$ (as before $\mathsf{head?}_c\,(\mathsf{c}\,\overline{y}) = \mathsf{true}$, and for $d \neq c$: $\mathsf{head?}_c\,(\mathsf{d}\,\overline{y}) = \mathsf{false}$). For each n-ary f in S, $f \neq \mathsf{if}$, defined by equations of the form $f\,(c\,y_1 \ldots y_m)\,x_2 \ldots x_n = r_i$, where $r_0, \ldots, r_p$ are the rhs's, there's an $n+2$-ary function $f^\bullet$ in R, defined by:

$$\begin{aligned} f^\bullet\,(\mathsf{succ}\,z_1)\,z_2\,y\,x_2 \ldots x_n \;=\;& \mathsf{if}\ (\mathsf{head?}_{c_0}\,y)\ r_0^* \\ & (\mathsf{if}\ (\mathsf{head?}_{c_1}\,y)\ r_1^* \ldots \\ & (\mathsf{if}\ (\mathsf{head?}_{c_p}\,y)\ r_p^*\ \mathsf{false}) \ldots) \end{aligned}$$

where for each subterm t of some of $r_0, \ldots, r_p$, t^* is

$$\begin{aligned} x_i^* &= x_i \\ y_i^* &= \pi_i\,y \\ (c\,t_1 \ldots t_k)^* &= k\,t_1^* \ldots t_k^*, \quad c \text{ a constructor} \\ (g\,t_1 \ldots t_k)^* &= g^\bullet\,z_1\,(\mathsf{succ}^\delta\,z_2)\,t_1^* \ldots t_k^*, \quad g,\,f \text{ mutually recursive} \\ (g\,t_1 \ldots t_k)^* &= g^\bullet\,(\mathsf{succ}^\delta\,z_2)\,(\mathsf{succ}^\delta\,z_2)\,t_1^* \ldots t_k^*, \quad g,\,f \text{ not mutually recursive, } g \neq \mathsf{if} \\ (\mathsf{if}\,t_1\,t_2\,t_3)^* &= \mathsf{if}\,t_1^*\,t_2^*\,t_3^* \end{aligned}$$

We show that R is a *DDC*$_{\mathsf{if},\pi_i,+}$ system:

- *NID* holds for R since *NID* holds for S and since projections and constructors are allowed in arguments to recursive call terms.
- *ROFA* holds by the shape of the equations and **pre1**.
- *RON* holds since the first position of every $f^\bullet$ is noncritical.
- *LIC* holds for R since *LIC* holds for S and since the first two positions of every $f^\bullet$ are noncritical.

R mimicks S since the first two arguments for $f^\bullet$ are large enough not to disturb the intended definitions, i.e.: Call any $n+2$-ary $f^\bullet$ in S with input $X_1, \ldots, X_{n+2}$ such that $|X_1| = |X_2| > \sum_{i=1}^{n+2} |X_i|$. Consider any possible function call $g^\bullet Y_1 \ldots Y_{m+2}$ ($g^\bullet \neq \mathsf{if}$) in this computation. If $g^\bullet \overline{Y}$ is a nonrecursive call (i.e. $g^\bullet$ was called by an $h^\bullet$ not mutually recursive with $g^\bullet$) then $|Y_1| = |Y_2| > \sum_{i=1}^{m+2} |Y_i|$. If $g^\bullet \overline{Y}$ is a recursive call then by **pre2**, Y_1 has a succ as head.

Proof of Theorem 1 Let f be a poly-time function on a constructor set C_0'. Then there's an SST machine M over $C_0 = C_0' \cup \{\#\}$ that computes f in poly-time. Define a system S_M as in Subsect. 4.2. Then by Lemma 2, S_M defines f. Replace $\mathsf{compute}$'s second rhs by $\mathsf{onestep}^\bullet\, z\, z\, (\mathsf{compute}\, y\, \overline{x})$ where z is some bound on the length of the output of $(\mathsf{compute}\, y\, \overline{x})$ plus one, e.g. a term representing $2(\sum_{i=0}^{k} |x_i| + (m+1)|y| + m + n + 4)$. Then by Lemmas 3 and 4, S_M is $DDC_{\mathsf{if},\pi_i,+}$. ○

5 Some Remarks on Natural Definitions

We have shown that in principle, any poly-time function on a constructor data structure may be defined by a $DDC_{\mathsf{if},\pi_i,+}$ system. But we also think that many functions may be defined in a natural way by a $DDC_{\mathsf{if},\pi_i,+}$ system. $DDC_{\mathsf{if},\pi_i,+}$ offers natural features like mutually recursive functions (e.g. length and $\mathsf{q\text{-}lengthsum}$'s), choice between different recursive calls ($\mathsf{treesort}$'s insert), combination of different recursive calls ($\mathsf{treesort}$'s $\mathsf{flatten}$), use of constructors and projections inside recursive calls (take, s_j-branch).

But it is a problem that *careful recursion on critical* is not allowed - many functions do use this. E.g. both $\mathsf{treesort}$ and our machine simulation work by having two levels of functions. High level functions like $\mathsf{maketree}$, $\mathsf{flatten}$, $\mathsf{compute}$ recur on noncritical - doubling and tripling their (noncritical) input as they please. Low level functions like insert, append, $\mathsf{onestep}$ that only do some modifications to their (partially critical) input, typically taking things apart and putting them back together a little bit differently. But also to do such simple things naturally, some simple recursion is needed.

However, often it's possible to solve this problem as we did for $\mathsf{onestep}$ in the proof of Theorem 1. E.g. in the $\mathsf{treesort}$ example, both append and insert

are $preDDC_{\mathsf{if},\pi_i,+}$, so if we replace flatten's call to append by a "suitable" call to $\mathsf{append}^\bullet$ (i.e. with large enough first and second argument), and replace maketree's call to insert by a suitable call to $\mathsf{insert}^\bullet$, then treesort becomes $DDC_{\mathsf{if},\pi_i,+}$.

References

[1] S. Bellantoni, S. Cook: *A new recursion-theoretic characterization of the polytime functions*, 24th Annual ACM STOC (1992), 283-293

[2] V.H. Caseiro: *Some general criteria on equations to guarantee poly-time functions*, Research Report November 1996, available at `http://www.ifi.uio.no/~ftp/publications/research-reports/VHCaseiro-1.ps`

[3] V.H. Caseiro: *Criticality conditions on equations to ensure poly-time functions*, Research Report November 1996, available at `http://www.ifi.uio.no/~ftp/publications/research-reports/VHCaseiro-2.ps`

[4] D. Leivant: *Stratified functional programs and computational complexity*, 20th ACM Symposium on Principles of Programming Languages, 1993