The strength of non-size increasing computation

Martin Hofmann*

Abstract

We study the expressive power non-size increasing recursive definitions over lists. This notion of computation is such that the size of all intermediate results will automatically be bounded by the size of the input so that the interpretation in a finite model is sound with respect to the standard semantics. Many well-known algorithms with this property such as the usual sorting algorithms are definable in the system in the natural way. The main result is that a characteristic function is definable if and only if it is computable in time $O(2^{p(n)})$ for some polynomial p.

The method used to establish the lower bound on the expressive power also shows that the complexity becomes polynomial time if we allow primitive recursion only. This settles an open question posed in [1, 6].

The key tool for establishing upper bounds on the complexity of derivable functions is an interpretation in a finite relational model whose correctness with respect to the standard interpretation is shown using a semantic technique.

Keywords: computational complexity, higher-order functions, finite model, semantics
AMS Classification: 03D15, 03C13, 68Q15, 68Q55

1 Introduction

Consider the following recursive definition of a function on lists:

$$\begin{aligned}&\texttt{twice}(\mathsf{nil}) = \mathsf{nil}\\&\texttt{twice}(\mathsf{cons}(x,l)) = \mathsf{cons}(\mathsf{tt},\mathsf{cons}(\mathsf{tt},\texttt{twice}(l)))\end{aligned} \tag{1}$$

Here nil denotes the empty list, $\mathsf{cons}(x,l)$ denotes the list with first element x and remaining elements l. tt,ff are the members of a type T of truth values. We have that $\texttt{twice}(l)$ is a list of length $2\cdot|l|$ where $|l|$ is the length of l. Now consider

$$\begin{aligned}&\mathsf{exp}(\mathsf{nil}) = \mathsf{cons}(\mathsf{tt},\mathsf{nil})\\&\mathsf{exp}(\mathsf{cons}(x,l)) = \texttt{twice}(\mathsf{exp}(l))\end{aligned} \tag{2}$$

We have $|\mathsf{exp}(l)| = 2^{|l|}$ and further iteration leads to elementary growth rates.

This shows that innocuous looking recursive definitions can lead to enormous growth. In order to prevent this from happening it has been suggested in [2, 9] to rule out definitions like (2) above, where a recursively defined function, here `twice`, is applied to the result of a recursive call. Indeed, it has been shown that such discipline restricts the definable functions

[*] Fachbereich Mathematik, TU Darmstadt,Schlossgartenstr. 7, 64289 Darmstadt, Germany, mhofmann@mathematik.tu-darmstadt.de

to the polynomial-time computable ones and moreover every polynomial-time computable *function* admits a definition in this style.

Many naturally occurring *algorithms*, however, do not fit this scheme. Consider, for instance, the definition of insertion sort:

$$\begin{aligned}
&\mathtt{insert}(x, \mathsf{nil}) = \mathsf{cons}(x, \mathsf{nil})\\
&\mathtt{insert}(x, \mathsf{cons}(y, l)) = \mathsf{if}\ x \leq y\ \mathsf{then}\ \mathsf{cons}(x, \mathsf{cons}(y, l))\ \mathsf{else}\ \mathsf{cons}(y, \mathtt{insert}(x, l))\\
&\mathtt{sort}(\mathsf{nil}) = \mathsf{nil}\\
&\mathtt{sort}(\mathsf{cons}(x, l)) = \mathtt{insert}(x, \mathtt{sort}(l))
\end{aligned} \tag{3}$$

Here just as in (2) above we apply a recursively defined function (`insert`) to the result of a recursive call (`sort`), yet no exponential growth arises.

It has been argued in [3] and [6] that the culprit is definition (1) because it defines a function that increases the size of its argument and that non size-increasing functions can be arbitrarily iterated without leading to exponential growth.

In [3] a number of partly semantic criteria were offered which allow one to recognise when a function definition is non size-increasing. In [6] we have given syntactic criteria based on linearity (bound variables are used at most once) and a so-called resource type $\diamond$ which counts constructor symbols such as "cons" on the left hand side of an equation.

This means that cons becomes a ternary function taking one argument of type $\diamond$, one argument of some type A (the head) and a third argument of type $\mathsf{L}(A)$, the tail. There being no closed terms of type $\diamond$ the only way to apply cons is within a recursive definition; for instance, we can write

$$\begin{aligned}
&\mathsf{append}(\mathsf{nil}, l_2) = l_2\\
&\mathsf{append}(\mathsf{cons}(d, a, l_1), l_2) = \mathsf{cons}(d, a, \mathsf{append}(l_1, l_2)
\end{aligned} \tag{4}$$

Alternatively, we may write

$$\mathsf{append}(l_1, l_2) = \mathsf{match}\ l\ \mathsf{with}\ \mathsf{nil}{\Rightarrow}l_2 \mid \mathsf{cons}(d, a, l_1'){\Rightarrow}\mathsf{cons}(d, \mathsf{append}(l_1, l_2) \tag{5}$$

We notice that the following attempted definition of `twice` is illegal as it violates linearity (the bound variable d is used twice):

$$\begin{aligned}
&\mathtt{twice}(\mathsf{nil}) = \mathsf{nil}\\
&\mathtt{twice}(\mathsf{cons}(d, x, l)) = \mathsf{cons}(d, \mathsf{tt}, \mathsf{cons}(d, \mathsf{tt}, \mathtt{twice}(l)))
\end{aligned} \tag{6}$$

The definition of `insert`, on the other hand, is in harmony with linearity provided that `insert` gets an extra argument of type $\diamond$ and, moreover, we assume that the inequality test returns its arguments for subsequent use.

The main result of [6] and [1] was that all functions thus definable by *structural recursion* are polynomial-time computable even when higher-order functions are allowed. In [7] it has been shown that general-recursive first-order definitions admit a translation into a fragment of the programming language C without dynamic memory allocation ("malloc") which on the one hand allows one to automatically construct imperative implementations of algorithms on lists which do not require extra space or garbage collection. More precisely, this translation maps the resource type $\diamond$ to the C-type `void *` of pointers. The cons function is translated into the C-function which extends a list by a given value using a provided piece of memory. It is proved that the pointers arising as denotation of terms of type $\diamond$ always point to free memory space which can thus be safely overwritten.

This translation also demonstrates that all definable functions are computable on a Turing machine with linearly bounded work tape and an unbounded stack (to accommodate general recursion) which by a result of Cook[1] [4] equals the complexity class $DTIME(2^{O(n)})$. It was also shown in [7] that any such function admits a representation.

In the presence of higher-order functions the translation into C breaks down as C does not have higher-order functions. Of course, higher-order functions can be simulated as closures, but this then requires arbitrary amounts of space as closures can grow proportionally to the runtime. In a system based on structural recursion such as [6] this is not a problem as the runtime is polynomially bounded there. The hitherto open question of complexity of general recursion with higher-order functions is settled in this paper and shown to require a polynomial amount of space only in spite of the unbounded runtime.

We thus demonstrate that a function is representable with general recursion and higher-order functions iff it is computable in polynomial space and an unbounded stack or equivalently (by Cook's result) in time $O(2^{p(n)})$ for some polynomial p. The lower bound of this result also demonstrates that indeed all characteristic functions of problems in P are definable in the structural recursive system. This settles a question left open in [1, 6].

In view of the results presented in this paper, these systems of non size-increasing computation thus provide a very natural connection between complexity theory and functional programming. There is also a connection to finite model theory in that—as will be shown below—programs admit a sound interpretation in a finite model. This improves upon earlier combinations of finite model theory with functional programming [5] where interpretation in a finite model was achieved in a brute-force way by changing the meaning of constructor symbols, e.g. successor of the largest number N was defined to be N itself. In those systems it is the responsibility of the programmer to account for the possibility of cut-off when reasoning about the correctness of programs. In the systems studied here linearity and the presence of the resource types automatically ensure that cutoff never takes place. Formally, it is shown that the standard semantics in an infinite model agrees with the interpretation in a certain finite model for all well-formed programs.

Another piece of related work is Jones' [8] where the expressive power of cons-free higher-order programs is studied. It is shown there that first-order cons-free programs define polynomial time , whereas second-order programs define EXPTIME. This shows that the presence of "cons", tamed by linearity and the resource type changes the complexity-theoretic strength. While loc. cit. also involves Cook's abovementioned result (indeed, this result was brought to the author's attention by Neil Jones) the other parts of the proof are quite different.

[1] This result asserts that if $L(n) > \log(n)$ then $DTIME(2^{O(L(n))})$ equals the class of functions computable by a Turing machine with an $L(n)$-bounded R/W-tape and an unbounded stack.

2 Syntax and typing rules

The terms of the languag are given by the following grammar:

$e ::=$	x	variable
$\mid$	$f(e_1,\dots,e_n)$	function application
$\mid$	tt,ff	boolean constant
$\mid$	$\mathsf{if}\ e\ \mathsf{then}\ e'\ \mathsf{else}\ e''$	conditional
$\mid$	$e_1 \otimes e_2$	pairing
$\mid$	nil	empty list
$\mid$	$\mathsf{cons}(e_1,e_2,e_3)$	cons with res. arg.
$\mid$	$\mathsf{match}\ e_1\ \mathsf{with}\ \mathsf{nil}{\Rightarrow}e_2 \mid \mathsf{cons}(d,h,t){\Rightarrow}e_3$	list elimination
$\mid$	$\mathsf{match}\ e_1\ \mathsf{with}\ x\otimes y{\Rightarrow}e_2$	pair elim.
$\mid$	$\lambda x.e$	linear lambda abstraction
$\mid$	$e_1 e_2$	linear function application

The match constructs as well as λ bind variables.

The *types* are given by the following grammar.

$$A ::= \mathsf{T} \mid \Diamond \mid \mathsf{L}(A) \mid A_1 \otimes A_2 \mid A_1 \multimap A_2$$

Here T is the type of truth values, $\mathsf{L}(A)$ stands for lists with entries of type A, $A_1 \otimes A_2$ is the type of pairs with first component of type A_1 and second component of type A_2. The type $A_1 \multimap A_2$ is the type of functions from A_1 to A_2, and finally $\Diamond$ is the resource type. The *heap-free* types contain T and are closed under $\otimes$. Variables of heap-free type may be used more than once as described by rule CONTR below.

In [7] also tree types and disjoint union types were considered. We refrain from doing so here for the sake of simplicity. However, it has been checked that all the constructions presented here carry over to this richer setting.

A *signature* Σ maps a finite set of function symbols to expressions of the form $(A_1,\dots,A_n)\to B$ where $A_1 \dots A_n$ and B are types.

A *typing context* Γ is a finite function from variables to types; if $x \notin \mathrm{dom}(\Gamma)$ then we write $\Gamma, x{:}A$ for the extension of Γ with $x \mapsto A$. More generally, if $\mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Delta) = \emptyset$ then we write Γ, Δ for the disjoint union of Γ and Δ. If such notation appears in the premise or conclusion of a rule below it is implicitly understood that these disjointness conditions are met. We write $e[x/y]$ for the term obtained from e by replacing all occurrences of the free variable y in e by x after suitable renaming of bound variables so as to prevent capture. We consider terms modulo renaming of bound variables.

Let Σ be a signature. The *typing judgment* $\Gamma \vdash_\Sigma e : A$ read "expression e has type A in typing context Γ and signature Σ" is defined by the following rules.

$$\frac{x \in \mathrm{dom}(\Gamma)}{\Gamma \vdash_\Sigma x : \Gamma(x)} \qquad (\textsc{Var})$$

$$\frac{\Sigma(f) = (A_1,\dots,A_n)\to B \qquad \Gamma_i \vdash_\Sigma e_i : A_i \,\text{for}\ i = 1\dots n}{\Gamma_1,\dots,\Gamma_n \vdash_\Sigma f(e_1,\dots,e_n) : B} \qquad (\textsc{Sig})$$

$$\frac{\Gamma, x{:}A, y{:}A \vdash_\Sigma e : B \qquad A\ \text{heap-free}}{\Gamma, x{:}A \vdash_\Sigma e[x/y] : B} \qquad (\textsc{Contr})$$

$$\frac{c \in \{\mathsf{tt}, \mathsf{ff}\}}{\Gamma \vdash_\Sigma c : \mathsf{T}} \tag*{(\textsc{Const})}$$

$$\frac{\Gamma \vdash_\Sigma e : \mathsf{T} \qquad \Delta \vdash_\Sigma e' : A \qquad \Delta \vdash_\Sigma e'' : A}{\Gamma, \Delta \vdash_\Sigma \mathsf{if}\ e\ \mathsf{then}\ e'\ \mathsf{else}\ e'' : A} \tag*{(\textsc{If})}$$

$$\frac{\Gamma \vdash_\Sigma e : A \qquad \Delta \vdash_\Sigma e' : B}{\Gamma, \Delta \vdash_\Sigma e \otimes e' : A \otimes B} \tag*{(\textsc{Pair})}$$

$$\frac{\Gamma \vdash_\Sigma e : A \otimes B \qquad \Delta, x{:}A, y{:}B \vdash_\Sigma e' : C}{\Gamma, \Delta \vdash_\Sigma \mathsf{match}\ e\ \mathsf{with}\ x \otimes y \Rightarrow e' : C} \tag*{(\textsc{Split})}$$

$$\Gamma \vdash_\Sigma \mathsf{nil} : \mathsf{L}(A) \tag*{(\textsc{Nil})}$$

$$\frac{\Gamma_d \vdash_\Sigma e_d : \Diamond \qquad \Gamma_h \vdash_\Sigma e_h : A \qquad \Gamma_t \vdash_\Sigma e_t : \mathsf{L}(A)}{\Gamma_d, \Gamma_h, \Gamma_t \vdash_\Sigma \mathsf{cons}(e_d, e_h, e_t) : \mathsf{L}(A)} \tag*{(\textsc{Cons})}$$

$$\frac{\begin{array}{l} \Gamma \vdash_\Sigma e : \mathsf{L}(A) \\ \Delta \vdash_\Sigma e_{\mathsf{nil}} : B \\ \Delta, d{:}\Diamond, h{:}A, t{:}\mathsf{L}(A) \vdash_\Sigma e_{\mathsf{cons}} : B \end{array}}{\Gamma, \Delta \vdash_\Sigma \mathsf{match}\ e\ \mathsf{with}\ \mathsf{nil} \Rightarrow e_{\mathsf{nil}} \mid \mathsf{cons}(d, h, t) \Rightarrow e_{\mathsf{cons}} : B} \tag*{(\textsc{List-Elim})}$$

$$\frac{\Gamma, x{:}A \vdash_\Sigma e : B}{\Gamma \vdash \lambda x.e : A \multimap B} \tag*{(\textsc{Lam})}$$

$$\frac{\Gamma \vdash_\Sigma e_1 : A \multimap B \qquad \Delta \vdash e_2 : A}{\Gamma, \Delta \vdash_\Sigma e_1 e_2 : B} \tag*{(\textsc{App})}$$

Application of function symbols is linear in the sense that several operands must in generalnot share common free variables. This is because of the implicit side condition on juxtaposition of contexts mentioned above. In view of rule Contr, however, variables of a heap-free type may be shared and moreover thesame free variable may appear in different branches of a case distinction as follows e.g. from the form of rule If. It follows by standard type-theoretic techniques that type checking for this system is decidable in linear time. More precisely, we have a linear time computable function which given a context Γ, a term e in normal form[2], and a type A either returns a minimal subcontext Δ of Γ such that $\Delta \vdash e : A$ or returns "failure" in the case where $\Gamma \vdash e : A$ does not hold. This function can be defined by primitive recursion over e.

A *program* consists of a signature Σ and for each symbol $f : (A_1, \dots, A_n) \to B$ contained in Σ a term e_f such that $x_1{:}A_1, \dots, x_n{:}A_n \vdash_\Sigma e_f : B$.

[2] i.e. one that does not contain instance of match applied to constructors ($\mathsf{nil}, \mathsf{cons}, \otimes$) or λ-abstractions in applied position

3 Denotational semantics

In order to specify the purely functional meaning of programs we introduce a denotational semantics following [10].

A partially ordered set $D = (D, \leq)$ is a *complete partial order*, cpo for short, if each increasing chain $x_0 \leq x_1 \leq \dots$ has a least upper bound $\bigvee_i x_i$ in D. A function from cpo D to cpo E is continuous if it is monotone and preserves these least upper bounds. Any set forms a (discrete) cpo. If D is a cpo its lifting $D_\perp$ is formed by freely adjoining a least element $\perp$. For cpos D and E we have their cartesian product $D \times E$ with the component-wise ordering. We write (x, y) for the pair with components x and y and if $p = (x, y)$ we write $p.1 = x$ and $p.2 = y$ for the first and second projections. We assume that × associates to the right so that e.g. the second component of $p \in D \times E \times F$ is obtained as $p.2.1$. We have the continuous function space $D \to E$ consisting of continuous functions from D to E with the point-wise ordering. Elements of $D \to E$ may be defined using λ-notation if continuity is ensured. For instance, if $e \in E$ the expression $\lambda x.e$ denotes the constant function in $D \to E$.

The cpo $\mathsf{L}(D)$ consists of finite lists of elements of D with lists of equal length ordered component-wise and lists of different length being incomparable. We use the notation $[]$ for the empty list, $a :: l$ for the list with first element a and remaining elements l, we write $[a_1, \dots, a_n]$ for the list with members $a_1, \dots, a_n$ and $l_1 @ l_2$ for the concatenation of lists l_1 and l_2. We write $|l|$ for the length of a list l.

We assign a cpo to each type by

$$\begin{array}{lll} [\![\mathsf{T}]\!] = \{\mathsf{tt}, \mathsf{ff}\} & [\![\Diamond]\!] = \{0\} & [\![\mathsf{L}(A)]\!] = \mathsf{L}([\![A]\!]) \\ [\![A \otimes B]\!] = [\![A]\!] \times [\![B]\!] & [\![A \multimap B]\!] = [\![A]\!] \to [\![B]\!]_\perp & \end{array}$$

To each program $P = (\Sigma, (e_f)_{f \in \mathrm{dom}(\Sigma)})$ we can now associate a mapping $[\![P]\!]$ such that $[\![P]\!](f)$ is a continuous map from $[\![A_1]\!] \times \dots \times [\![A_n]\!]$ to $[\![B]\!]_\perp$ for each $f : (A_1, \dots, A_n) \to B$.

This meaning is given in the standard fashion as the least fixpoint of an appropriate compositionally defined operator, as follows.

A *valuation* of a context Γ is a function η such that $\eta(x) \in [\![\Gamma(x)]\!]$ for each $x \in \mathrm{dom}(\Gamma)$; a valuation of a signature Σ is a function ρ such that $\rho(f) \in [\![A_1]\!] \times \dots \times [\![A_n]\!] \to [\![B]\!]_\perp$ whenever $f \in \mathrm{dom}(\Sigma)$.

To each expression e such that $\Gamma \vdash_\Sigma e : A$ we assign a function mapping a valuation η of Γ and a valuation ρ of Σ to an element $[\![e]\!]_{\eta,\rho} \in [\![A]\!]$ in the obvious way, i.e. function symbols and variables are interpreted according to the valuations; basic functions and expression formers are interpreted by the eponymous set-theoretic operations, ignoring the arguments of type $\Diamond$ in the case of constructor functions. The formal definition of $[\![-]\!]_{\eta,\rho}$ is by induction on terms.

Here are a few representative clauses.

$$\llbracket x \rrbracket_{\eta,\rho} = \eta(x)$$
$$\llbracket f(e_1,\dots,e_n) \rrbracket_{\eta,\rho} = \rho(f)(\llbracket e_1 \rrbracket_{\eta,\rho},\dots,\llbracket e_n \rrbracket_{\eta,\rho})$$
$$\llbracket \mathsf{cons}(e_1,e_2,e_3) \rrbracket_{\eta,\rho} = \llbracket e_2 \rrbracket_{\eta,\rho} :: \llbracket e_3 \rrbracket_{\eta,\rho}$$
$$\begin{aligned}&\llbracket \mathsf{match}\ e\ \mathsf{with}\ \mathsf{nil}{\Rightarrow}e_1 \mid \mathsf{cons}(d,h,t){\Rightarrow}e_2 \rrbracket_{\eta,\rho}\\ &\quad = \llbracket e_1 \rrbracket_{\eta,\rho}\\ &\quad \text{when } \llbracket e \rrbracket_{\eta,\rho} = [] \text{ and}\\ &\quad = \llbracket e_2 \rrbracket_{\eta[d\mapsto 0,h\mapsto v_h,t\mapsto v_t],\rho}\\ &\quad \text{when } \llbracket e \rrbracket_{\eta,\rho} = v_h :: v_t\end{aligned}$$
$$\llbracket \lambda x.e \rrbracket_{\eta,\rho}(v) = \llbracket e \rrbracket_{\eta[x\mapsto v],\rho}$$
$$\llbracket e_1 e_2 \rrbracket_{\eta,\rho} = \llbracket e_1 \rrbracket_{\eta,\rho}(\llbracket e_2 \rrbracket_{\eta,\rho})$$

A *program* $(\Sigma,(e_f)_{f\in\mathrm{dom}(\Sigma)})$ is interpreted as the least upper bound of the following (pointwise) increasing sequence of valuations: $\rho_0(f)(\vec{v}) = \bot$ and

$$\rho_{i+1}(f)(v_1,\dots,v_n) = \llbracket e_f \rrbracket_{\rho_i,\eta} \tag{7}$$

where $\eta(x_i) = v_i$, for any $f \in \mathrm{dom}(\Sigma)$. Notice that $\rho = \bigvee_i \rho_i$ satisfies

$$\rho(f)(v_1,\dots,v_n) = \llbracket e_f \rrbracket_{\rho,\eta} \tag{8}$$

and is minimal with this property.

We stress that this order-theoretic semantics does not say anything about computational complexity. Its *only* purpose is to pin down the functional denotations of programs so that we can formally state what it means to implement a function. Accordingly, the resource type is interpreted as a singleton set, $\otimes$ and $\multimap$ are interpreted as ordinary product and function space disregarding linearity.

If f is a function symbol in defined in a program P that is clear from the surrounding context then we may abbreviate $\llbracket P \rrbracket(f)$ to $\llbracket f \rrbracket$.

3.1 Examples

Reverse:

$\mathtt{rev\_aux} : (\mathsf{L(N)},\mathsf{L(N)}){\to}\mathsf{L(N)}$
$\mathtt{reverse} : (\mathsf{L(N)}){\to}\mathsf{L(N)}$
$e_{\mathtt{rev\_aux}}(l, acc) = \mathsf{match}\ l\ \mathsf{with}\ \mathsf{nil}{\Rightarrow}acc \mid \mathsf{cons}(d,h,t){\Rightarrow}\mathtt{rev\_aux}(t,\mathsf{cons}(d,h,acc))$
$e_{\mathtt{reverse}}(l) = \mathtt{rev\_aux}(l,\mathsf{nil})$

Insertion sort

```
insert : $(\diamond,\mathsf{N},\mathsf{L(N)})\to\mathsf{L(N)}$
sort : $(\mathsf{L(N)})\to\mathsf{L(N)}$
$e_{\mathtt{insert}}(d,a,l) = \mathsf{match}\ l\ \mathsf{with}$
    $\mathsf{nil}\Rightarrow\mathsf{nil}$
    $\mid \mathsf{cons}(d',b,t)\Rightarrow\mathsf{if}\ a \le b$
       $\mathsf{then}\ \mathsf{cons}(d,a,\mathsf{cons}(d',b,t))$
       $\mathsf{else}\ \mathsf{cons}(d,b,\mathtt{insert}(d',a,t))$
$e_{\mathtt{sort}}(l) = \mathsf{match}\ l\ \mathsf{with}$
    $\mathsf{nil}\Rightarrow\mathsf{nil}$
    $\mid \mathsf{cons}(d,a,t)\Rightarrow\mathtt{insert}(d,a,\mathtt{sort}(t))$
```

Apply a function to the tail of a list

```
AppTail : $(A \multimap A, \mathsf{L}(A)) \to \mathsf{L}(A)$
$e_{\texttt{AppTail}}(f, a, l)$ = match $l$ with
    nil⇒nil
    | cons$(d, b, t)$⇒match $t$ with nil⇒cons$(d, f(b)$, nil$)$
      | cons$(d', b', t')$⇒AppTail(cons$(d', b', t')$))
```

Composing all functions in a list

```
ComposeList : $(\mathsf{L}((\Diamond \otimes A) \multimap A)) \to A \multimap A$
$e_{\texttt{ComposeList}}(l, a)$ = match $l$ with
    nil⇒$\lambda a.a$
    | cons$(d, f, t)$⇒$\lambda a.f(d \otimes$ ComposeList$(t)(a))$
```

Higher-order tail recursion

```
Contrived : $(A, A \multimap A) \to A$
$e_{\texttt{Contrived}}(x, f)$ = if p$(x)$ then $f(x)$
    else if q$(x)$ then Contrived(a$(x)$, $\lambda y.$g$(f($g$(x)))$)
        else Contrived(b$(x)$, $\lambda y.$h$(f($h$(x)))$)
```

In the last example, $\mathtt{p}, \mathtt{q} : (A) \to \mathsf{T}$ and $\mathtt{a}, \mathtt{b}, \mathtt{g}, \mathtt{h} : (A) \to A$ are arbitrary function symbols defined independently or indeed simultaneously with Contrived. The point of the example is that under a functional evaluation strategy the intermediate term denoting the currently accumulated function grows arbitrarily. Many more examples are given in [6, 7].

4 Expressivity

In this section we characterise the functions of type $(\mathsf{L}(\mathsf{T})) \to \mathsf{L}(\mathsf{T})$ definable in the system. We will say nothing about higher-order functionals definable in the system, notice, however, that a first-order function may involve a higher-order functional as part of its definition. This situation is encompassed by our characterisation.

Let us write W for the type $\mathsf{L}(\mathsf{T})$ and T for the set $\{\mathsf{tt}, \mathsf{ff}\}$ and W for the set $T^* = [\![\mathsf{L}(\mathsf{T})]\!] = [\![\mathsf{W}]\!]$. For a set A we define $\mathsf{L}_n(A) = \{w \in A^* \mid |w| = n\}$ as the set of lists of length n over A. We write $W_n = \mathsf{L}_n(T)$ so that $W_n \subseteq W$. Elements of W_n will be identified with the set $\{0, \dots, 2^n - 1\}$ using the binary encoding. E.g. $W_5 \ni [\mathsf{ff}, \mathsf{tt}, \mathsf{tt}, \mathsf{ff}, \mathsf{ff}] = 12$.

If $A_1, \dots, A_n, B$ are types and $f : [\![A_1]\!] \times \cdots \times [\![A_n]\!] \to [\![B]\!]_\perp$ is a function then we say that f is *representable* if there exists a program containing a function symbol $\mathtt{f} : (A_1, \dots, A_n) \to B$ such that $[\![\mathtt{f}]\!] = f$. Our aim in this section is to prove the following result.

Theorem 4.1 *Let $f : W \to W$ be a function such that $|f(w)| \leq |w|$ and such that $f(x)$ is computable in time $O(2^{p(|x|)})$ for some polynomial p. Then f is representable.*

Definition 4.2 Let $s : \mathbb{N} \to \mathbb{N}$ be a function with $s(n) < 2^n$ and $k \in \mathbb{N}$ be a number. A (k, s)-*storage device* is given by the following data:

- a set $S = [\![\mathsf{S}]\!]$ for some type S

- a family of subsets $S_n \subseteq S$ for $n \in \mathbb{N}$.
- a representable function (a constant) $\mathit{init} :\to S$, i.e. there is $\mathtt{init} : ()\to\mathtt{S}$ with $[\![\mathtt{init}]\!] = \mathit{init}$,
- a representable function $\mathit{read} : W \times W \times S \to W \times W \times T \times S$, i.e., there is $\mathtt{read} : (\mathsf{L(T)}, \mathsf{L(T)}, \mathtt{S})\to\mathsf{L(T)} \otimes \mathsf{L(T)} \otimes \mathsf{T} \otimes \mathtt{S}$ with $[\![\mathtt{read}]\!] = \mathit{read}$,
- a representable function $\mathit{write} : W \times W \times T \times S \to W \times W \times S$, i.e., there is ...

such that for all $n \in \mathbb{N}$ and $w, w_1, w_2, w_3 \in W_{kn}$, $a, a' \in W_n$ and $s \in S_n$ the following are satisfied:

- $\mathit{init}() \in S_n$
- $\mathit{read}(w, a, s) = (w', a', b, s')$ implies $w' \in W_{kn}, a' \in W_n, s' = s$
- $\mathit{write}(w, a, b, s) = (w', a', s')$ implies $w' \in W_{kn}, a' \in W_n, s' \in S_n$
- $\mathit{read}(w_1, a, \mathit{write}(w_2, a, b, s).2.2) = b$ provided that $a < s(n)$
- $\mathit{read}(w_1, a, \mathit{write}(w_2, a', b, s).2.2) = \mathit{read}(w_3, a, s)$ provided that $a, a' < s(n)$ and $a \neq a'$.

□

This means that an element of S_n is capable of holding $s(n)$ bits of information. The call $\mathit{read}(w, a, s)$ reads the a-th bit contained in s; the call $\mathit{write}(w, a, b, s)$ sets it to b when $a < s(n)$. Otherwise, the behaviour of these functions is left unspecified.

The first argument w plays the role of a "scratch pad"; its contents are unimportant; it is used as an item of auxiliary space to perform reading and writing. Both *read* and *write* return an equally long list for possible subsequent use as a scratch pad. Similarly, the address a and (in case of *read* the store s itself) are being returned as part of the result. In a linear setting this is crucial as otherwise these arguments would be lost.

Lemma 4.3 *Let $c \in \mathbb{N}$ be a constant. There is a $(0, \lambda n.c)$-storage device.*

Proof. For $n \in \mathbb{N}$ we put $S = S_n = T^c$

We put

$$
\begin{aligned}
&\mathit{init} = (\mathtt{tt}, \ldots, \mathtt{tt})\\
&\mathit{read}(w, a, s) = (w, a, b_a, s), \text{ if } a < c\\
&\mathit{read}(w, a, s) = (w, a, \mathtt{tt}, s), \text{ otherwise}\\
&\mathit{write}(w, a, b, s) = (w, a, (b_0, \ldots, b_{a-1}, b, b_{a+1}, \ldots, b_{c-1})), \text{ if } a < c\\
&\mathit{write}(w, a, b, s) = (w, a, s), \text{ otherwise}
\end{aligned}
$$

when $s = (b_0, \ldots, b_{c-1})$.

We have $S = [\![\mathsf{S}]\!]$ where $\mathsf{S} = \mathsf{T} \otimes \ldots \otimes \mathsf{T})$ with c factors. Since c is a constant we can "hardwire" all possible c addresses, i.e., we use a case distinction on a of depth $\log(c)$ to distinguish all possible different values of a. We omit the details. □

The key to larger sizes is the following lemma which shows how to "hide" information inside a (constant) function:

Lemma 4.4 *Let* S *be any type and put* $S = [\![\mathsf{S}]\!]$*. There is a representable functional*

$$\Phi : \mathsf{L}(S) \longrightarrow (W \to \mathsf{L}(S)) \times W \tag{9}$$

with the property

$$\Phi(l) = (f, w) \Rightarrow |w| = |l| \wedge \forall w'. |w'| = |l| \Rightarrow f(w') = l \tag{10}$$

Proof. The following program represents f:

```
$e_\Phi(l) = \mathsf{match}\ l\ \mathsf{with}$
  $\mathsf{nil} \Rightarrow (\lambda x.\mathsf{nil}) \otimes \mathsf{nil}$
  $\mathsf{cons}(d, s, l') \Rightarrow \mathsf{match}\ \Phi(l')\ \mathsf{with}$
    $f \otimes w \Rightarrow$
      $(\lambda x.\mathsf{match}\ x\ \mathsf{with}$
        $\mathsf{nil} \Rightarrow \mathsf{nil}$
        $\mathsf{cons}(d', b, w') \Rightarrow \mathsf{cons}(d', s, f(w')))$
      $\otimes \mathsf{cons}(d, \mathsf{tt}, w)$
```

$\square$

The idea is that if $\Phi(l) = (f, w)$ then f holds all the information contained in l yet the abstract space (in the form of $\diamond$-values) occupied by l is returned as w. Of course, in order to read the information contained in f we need an argument of size $|l|$.

Lemma 4.5 *If there exists a* (k, s)*-storage device then there exists a* $(k+1, \lambda n.n \cdot s(n))$*-storage device.*

Proof. Suppose the storage device of size s is given by the sets $S_n \subseteq S$ and the functions *init*, *read*, *write*. We define the desired storage device on

$$S' = W \to \mathsf{L}(S)_\perp = [\![\mathsf{L}(\mathsf{T}) \multimap \mathsf{L}(\mathsf{S})]\!] \tag{11}$$

where $[\![\mathsf{S}]\!] = S$ and

$$S'_n = \{f \mid \forall w \in W_n . f(w) \in \mathsf{L}_n(S_n)\} \subseteq S' \tag{12}$$

We put

$$\begin{aligned} &\mathit{init}'([]) = [] \\ &\mathit{init}'(x :: w) = \mathit{init}() :: \mathit{init}'(w) \end{aligned}$$

so that $\mathit{init}' \in S'$.

Notice that we have $\mathit{init}' = [\![\texttt{init'}]\!]$ where

$$e_{\texttt{init'}} = \lambda w.\mathsf{match}\ w\ \mathsf{with}\ \mathsf{nil} \Rightarrow \mathsf{nil} \mid \mathsf{cons}(d, x, w_1) \Rightarrow \mathsf{cons}(d, \texttt{init}(), \texttt{init'}(w_1))$$

The definition of *read′* will be given as a sequence of intermediate results assuming the existence of certain helper functions whose definition we omit.

For *read′*(w, a, f) we start with $w, a \in W$ and $f \in S'$. We intend that $w \in W_{(k+1)n}$, $a \in W_n$, $f \in S'_n$ for some $n \in \mathbb{N}$.

We split w into w_1, w_2 such that $|w_1| + |w_2| = |w|$ and $|w_1|/|w_2| = 1/k$. If this is impossible we immediately produce some default result. Notice that if $|w| = (k+1)n$ as intended then such decomposition is possible and $|w_1| = |a| = n, |w_2| = kn$. We now apply f to w_1 yielding $l \in \mathsf{L}(S)$, actually $l \in \mathsf{L}_n(S_n)$ in case $f \in S'_n$. We decompose l into $l_1, l_2 \in \mathsf{L}(S), s \in S, d \in \diamond$ where $l_1 @ [s] @ l_2 = l$ and s is the $(a \bmod |a|)$-th entry of l. We let a_1 be $a \operatorname{div} |a|$ where $|a_1| = |a| = n$ and call $\mathrm{read}(w_2, a_1, s)$. This yields the desired boolean value b which forms the main result of *read′*(w_2, a, s). The other return values comprise s and a list w'_2 with $|w'_2| = kn = |w_2|$. From s, l_1, l_2, d we reconstruct l and then—using Lemma 4.4—we reconstruct f and obtain w'_1 with $|w'_1| = |w_1| = n$. We return $w'_1 @ w'_2, a_1, b, f$.

The definition of *write′* is analogous. □

Proof of Theorem 4.1 Suppose that $f : W \to W$ is a function such that $f(l)$ is computable on a Turing machine M in time $2^{p(|l|)}$. Let k be the degree of p. By Lemmas 4.3 and 4.5 there exists a $(k, \lambda n.p(2kn))$-storage device S.

This means that in the presence of a list $w \in W_{n/2}$ serving as a scratch pad we can store $p(n)$ bits.

Starting from the input presented as an element $l \in W$ where $n = |l|$ we first construct by recursion on l an element $(w, l') \in W_{n/2} \times \mathsf{L}_{n/2}(T \times T)$ such that l' contains the entire information of l. Notice that this is possible as a diagonal map $\mathit{diag} : T \to T \times T$ with $\mathit{diag}(x) = (x, x)$ is definable by $e_{\mathsf{diag}}(x) = \mathsf{if}\ x\ \mathsf{then}\ \mathsf{tt} \otimes \mathsf{tt}\ \mathsf{else}\ \mathsf{ff} \otimes \mathsf{ff}$. Alternatively, we can use rule CONTR.

Thus w can be used as a scratch pad for the storage device to store the required amount of $p(n)$ bits occurring as work tape inscriptions. Additionally we can simulate an unbounded stack by general recursion, see [7] for details.

Thus, by Cook's result [4] the function f is representable. □

The above can also be used to solve a question left open in [1, 6]. In those papers a restriction of the described language has been studied which confines recursion to *structural recursion.* This means that the function symbols are totally ordered; in the function body e_f may contain function symbols less or equal to f only. Moreover, if e_f mentions f then f must have one argument x_i of type $\mathsf{L}(A)$ for some A and e_f must be of the form

$$e_f(\dots, x_i, \dots) = \mathsf{match}\ x_i\ \mathsf{with}\ \mathsf{nil} \Rightarrow e_{\mathsf{nil}} \mid \mathsf{cons}(d, h, t) \Rightarrow e_{\mathsf{cons}}$$

where e_{nil} does not contain f and e_{cons} contains f at most once and then with argument x_i equal to t.

Theorem 4.6 *Let $f : W \to W$ be a function such that $|f(l)| \leq |l|$ for all $l \in W$. Then f is representable using structural recursion alone iff f is computable in polynomial time.*

Proof. The "only if" direction is the main result of [1, 6]. For the other direction we use a Lemma from [6] which states that if $g : \mathsf{L}(A) \to \mathsf{L}(A)$ is representable and moreover

$|g(x)| = |x|$ then for any polynomial p the function $\lambda x.g^{p(|x|)}(x)$ is representable, too. In order, then, to represent f we package up a storage device $s \in S$, a scratch pad, and the input, into a single list over some appropriate type A, say $A = T \otimes S \otimes T$. Using for g the appropriately coded one-step function of a Turing machine then yields the result. □

We will now provide a corresponding upper bound on expressivity:

Theorem 4.7 *If $f : W \to W$ is representable then $f(l)$ is computable on a deterministic Turing machine in time $O(2^{p(|l|)})$ for some polynomial p.*

The proof of this result is based on two intuitions: Firstly, due to the linear typing discipline the size of all intermediate results is a priori bounded by a function of the size of the input. Second, linear functions can be simulated as argument-result pairs if one allows for nondeterminism: when constructing a linear function one guesses an argument and stores it together with the corresponding result. When applying such a linear function, one checks whether the actual argument agrees with the previously guessed one and in this case returns the precomputed result. Otherwise, the result is undefined.

To make this precise we construct an appropriate finite relational model for the language and show that evaluation in that finite model yields the same result as evaluation in the official order-theoretic (infinite) model.

Let $N \in \mathbb{N}$ be a fixed parameter. We define finite sets $(\!|A|\!)$ together with functions $|-|_A : (\!|A|\!) \to \{0, \dots, N\}$ for types A inductively as follows.

$$\begin{array}{ll}
(\!|\Diamond|\!) = \{0\} & |0|_\Diamond = 1 \\
(\!|\mathsf{T}|\!) = \{\mathtt{tt}, \mathtt{ff}\} & |x|_\mathsf{T} = 0 \\
(\!|\mathsf{L}(A)|\!) = \{w \in \mathsf{L}((\!|A|\!)) \mid |w|_{\mathsf{L}(A)} \leq N\} & |[a_1, \dots, a_n]|_{\mathsf{L}(A)} = n + \sum_{i=1}^n |a_i|_A \\
(\!|A \otimes B|\!) = \{x \in (\!|A|\!) \times (\!|B|\!) \mid |x|_{A\otimes B} \leq N\} & |(a,b)|_{A\otimes B} = |a|_A + |b|_B \\
(\!|A \multimap B|\!) = (\!|A|\!) \times (\!|B|\!) & |(a,b)|_{A\multimap B} = |b|_B \mathbin{\dot{-}} |a|_A
\end{array}$$

For context Γ we define

$$(\!|\Gamma|\!) = \{\eta \mid \forall x \in \mathrm{dom}(\Gamma).\eta(x) \in (\!|\Gamma(x)|\!) \wedge |\eta|_\Gamma \leq N\} \qquad |\eta|_\Gamma = \textstyle\sum_{x\in\mathrm{dom}(\Gamma)} |\eta(x)|_{\Gamma(x)}$$

When we use, e.g., $|x|_{A\otimes B}$ in the definition of $(\!|A \otimes B|\!)$ it refers to the defining expression for $|-|_{A\otimes B}$ given afterwards. The "modified difference" $x \mathbin{\dot{-}} y$ is defined as $x - y$ if $x > y$ and 0 otherwise. Notice that for nonnegative numbers x, y, z one has $x + y \geq z$ iff $x \geq z \mathbin{\dot{-}} y$.

For $U \subseteq (\!|A|\!)$ we define $|U|_A = \max_{a\in U} |a|_A$.

A *relational valuation* of a signature Σ assigns to each $f : (A_1, \dots, A_r) \to B$ declared in Σ a relation

$$\rho(f) \subseteq (\!|A_1 \otimes \dots \otimes A_r|\!) \times (\!|B|\!) \tag{13}$$

such that $(a_1, \dots, a_n)\rho(f)b$ implies $|b|_B \leq \sum_{i=1}^n |a_i|_{A_i}$.

Given relational valuation ρ of Σ we define a relation

$$(\!|e|\!)_\rho \subseteq (\!|\Gamma|\!) \times (\!|A|\!) \tag{14}$$

by induction on a typing derivation $\Gamma \vdash_\Sigma e : A$ as follows:

$$\llparenthesis \Gamma \vdash x : \Gamma(x) \rrparenthesis_\rho = \{(\eta, v) \mid v = \eta(x)\} \tag{\textsc{Var}}$$

$$\begin{aligned}
&\llparenthesis \Gamma_1, \dots, \Gamma_r \vdash f(e_1, \dots, e_r) : B \rrparenthesis_\rho = \{(\eta, v) \mid \\
&\quad \eta = \eta_1 \uplus \cdots \uplus \eta_r \wedge \\
&\quad \textstyle\bigwedge_i \eta_i \llparenthesis \Gamma_i \vdash e_i : A_i \rrparenthesis_\rho v_i \wedge (v_1, \dots, v_r) \rho(f) v\}
\end{aligned} \tag{\textsc{Sig}}$$

$$\begin{aligned}
&\llparenthesis \Gamma, x{:}A \vdash e : B \rrparenthesis_\rho = \{(\eta, v) \mid \\
&\quad \eta \uplus [y \mapsto \eta(x)] \llparenthesis \Gamma, x{:}A, y{:}A \vdash e : B \rrparenthesis_\rho v\}
\end{aligned} \tag{\textsc{Contr}}$$

$$\llparenthesis \Gamma \vdash c : \mathsf{T} \rrparenthesis_\rho = \{(\eta, v) \mid \eta \in \llparenthesis \Gamma \rrparenthesis, v = [\![c]\!]\} \tag{\textsc{Const}}$$

$$\begin{aligned}
&\llparenthesis \Gamma, \Delta \vdash \mathsf{if}\ e\ \mathsf{then}\ e'\ \mathsf{else}\ e'' : A \rrparenthesis_\rho = \{(\eta, v) \mid \\
&\quad \eta = \eta_1 \uplus \eta_2 \wedge (\\
&\qquad \eta_1 \llparenthesis \Gamma \vdash e : \mathsf{T} \rrparenthesis_\rho \mathsf{tt} \wedge \eta_2 \llparenthesis \Delta \vdash e' : A \rrparenthesis_\rho v \vee \\
&\qquad \eta_1 \llparenthesis \Gamma \vdash e : \mathsf{T} \rrparenthesis_\rho \mathsf{ff} \wedge \eta_2 \llparenthesis \Delta \vdash e'' : A \rrparenthesis_\rho v)\}
\end{aligned} \tag{\textsc{If}}$$

$$\llparenthesis \Gamma \vdash \mathsf{nil} : \mathsf{L}(A) \rrparenthesis_\rho = \{(\eta, []) \mid \eta \in \llparenthesis \Gamma \rrparenthesis\} \tag{\textsc{Nil}}$$

$$\begin{aligned}
&\llparenthesis \Gamma_d, \Gamma_h, \Gamma_t \vdash \mathsf{cons}(e_d, e_h, e_t) : \mathsf{L}(A) \rrparenthesis_\rho = \{(\eta, v_h :: v_t) \mid \\
&\quad \eta = \eta_d \uplus \eta_h \uplus \eta_t \wedge \\
&\quad \eta_d \llparenthesis \Gamma_d \vdash e_d : \diamond \rrparenthesis_\rho 0 \wedge \\
&\quad \eta_h \llparenthesis \Gamma_h \vdash e_h : A \rrparenthesis_\rho v_h \wedge \\
&\quad \eta_t \llparenthesis \Gamma_t \vdash e_t : \mathsf{L}(A) \rrparenthesis_\rho v_t\}
\end{aligned} \tag{\textsc{Cons}}$$

$$\begin{aligned}
&\llparenthesis \Gamma, \Delta \vdash \mathsf{match}\ e\ \mathsf{with}\ \mathsf{nil} \Rightarrow e_{\mathsf{nil}} \mid \mathsf{cons}(d, h, t) \Rightarrow e_{\mathsf{cons}} : B \rrparenthesis_\rho = \{(\eta, v) \mid \\
&\quad \eta = \eta_1 \uplus \eta_2 \\
&\quad \eta_1 \llparenthesis \Gamma \vdash e : \mathsf{L}(A) \rrparenthesis_\rho [] \wedge \eta_2 \llparenthesis \Delta \vdash e_{\mathsf{nil}} : B \rrparenthesis_\rho v \vee \\
&\quad \eta_1 \llparenthesis \Gamma \vdash e : \mathsf{L}(A) \rrparenthesis_\rho v_h :: v_t \wedge \\
&\qquad \eta_2 [d \mapsto 0, h \mapsto v_h, t \mapsto v_t] \llparenthesis \Delta, d{:}\diamond, h{:}A, t{:}\mathsf{L}(A) \vdash e_{\mathsf{cons}} : A \rrparenthesis_\rho v\}
\end{aligned} \tag{\textsc{List-Elim}}$$

$$\begin{aligned}
&\llparenthesis \Gamma, \Delta \vdash e_1 \otimes e_2 : A \otimes B \rrparenthesis_\rho = \{(\eta, (v_1, v_2)) \mid \\
&\quad \eta_1 \llparenthesis \Gamma \vdash e_1 : A \rrparenthesis_\rho v_1 \wedge \eta_2 \llparenthesis \Delta \vdash e_2 : B \rrparenthesis_\rho v_2\}
\end{aligned} \tag{\textsc{Pair}}$$

$$\begin{aligned}
&\llparenthesis \Gamma, \Delta \vdash \mathsf{match}\ e_1\ \mathsf{with}\ x \otimes y \Rightarrow e_2 : C \rrparenthesis_\rho = \{(\eta, v) \mid \\
&\quad \eta = \eta_1 \uplus \eta_2 \wedge \\
&\quad \eta_1 \llparenthesis \Gamma \vdash e_1 : A \otimes B \rrparenthesis_\rho (v_1, v_2) \wedge \\
&\quad \eta_2 [x \mapsto v_1, y \mapsto v_2] \llparenthesis \Delta, x{:}A, y{:}B \vdash e_2 : C \rrparenthesis_\rho v\}
\end{aligned} \tag{\textsc{Split}}$$

$$\begin{aligned}
&\llparenthesis \Gamma \vdash \lambda x.e : A \multimap B \rrparenthesis_\rho = \{(\eta, (a, b)) \mid \\
&\quad \eta[x \mapsto a] \llparenthesis \Gamma, x{:}A \vdash e : B \rrparenthesis_\rho b\}
\end{aligned} \tag{\textsc{Lam}}$$

$$\begin{aligned}
&\llparenthesis \Gamma, \Delta \vdash e_1 e_2 : B \rrparenthesis_\rho = \{(\eta, b) \mid \\
&\quad \eta = \eta_1 \uplus \eta_2 \\
&\quad \eta_1 \llparenthesis \Gamma \vdash e_1 : A \multimap B \rrparenthesis (a, b) \wedge \\
&\quad \eta_2 \llparenthesis \Delta \vdash e_2 : A \rrparenthesis a\}
\end{aligned} \tag{\textsc{App}}$$

The thus defined interpretation of a program is non size-increasing in the following sense.

Lemma 4.8 *If ρ is a relational valuation of Σ and $\Gamma \vdash_\Sigma e : A$ then whenever $\eta \llparenthesis \Gamma \vdash e : A \rrparenthesis_\rho a$ one has $|a|_A \leq |\eta|_\Gamma$.*

Proof. Direct induction on typing derivations. □

For a given program P the mapping which sends ρ to the relational valuation

$$f \mapsto \llparenthesis x_1{:}A_1, \dots, x_n{:}A_n \vdash e_f : B \rrparenthesis_\rho \tag{15}$$

is clearly monotone (with respect to inclusion) so that we can define the relational semantics of a program as the least fixpoint of this functional which in view of the finiteness of the domains is actually reached after a finite number of iterations starting from the relational valuation assigning the empty relation to each function symbol.

We write $\llparenthesis P\rrparenthesis(f)$ or simply $\llparenthesis f\rrparenthesis$ for the thus obtained interpretation of a function symbol f in some program P. Since the empty relation is a relational valuation and by the previous lemma the semantics maps relational valuations to relational valuations, the thus defined semantics of a program is also a relational valuation, i.e., non size-increasing.

Proposition 4.9 *Suppose that P is a program containing some function symbol $f : (\mathsf{W})\to\mathsf{W}$ and let $l \in W$ where $|l| \leq N$ (recall that N is a fixed parameter). Notice that in this case $l \in \llbracket \mathsf{L}(T)\rrbracket$ as well as $l \in \llparenthesis \mathsf{L}(T)\rrparenthesis$. Then $l\llparenthesis f\rrparenthesis l' \iff \llbracket f\rrbracket(l) = l'$ for all $l' \in W$.*

This means in particular that $\llparenthesis f\rrparenthesis$ is a partial function.

Before we prove this result let us remark that it allows us to evaluate any function $f : (\mathsf{L}(\mathsf{T}))\to\mathsf{L}(\mathsf{T})$ in a finite amount of time (regardless of its termination behaviour under an evaluation strategy based on rewriting) by computing $\llparenthesis f\rrparenthesis$ for appropriate parameter N. We will later estimate the amount of time required for this so as to obtain the desired characterisation. Let us first come to the proof of the proposition, though:

Proof. For each $n \leq N$ we define inductively a family of simulation relations

$$\sim^n_A \subseteq \llbracket A\rrbracket \times \{U \subseteq \llparenthesis A\rrparenthesis \mid U \neq \emptyset \wedge |U|_A \leq n\} \tag{16}$$

between elements of $\llbracket A\rrbracket$ and nonempty subsets of $\llparenthesis A\rrparenthesis$ of size $\leq n$. Recall that $|U|_A = \max_{x\in U} |x|_A$.

To simplify the notation we introduce the following shorthands: if $U \subseteq \llparenthesis A\rrparenthesis$ and $V \subseteq \llparenthesis B\rrparenthesis$ then $U \times V := \{(a,b) \mid a \in U \wedge b \in V\}$ We have $U \times V \subseteq \llparenthesis A \otimes B\rrparenthesis$ iff $|U|_A + |V|_B \leq N$ and in this case $|U \times V|_{A\otimes B} = |U|_A + |V|_B$.

If $U \subseteq \llparenthesis A\rrparenthesis$ and $V \subseteq \llparenthesis \mathsf{L}(A)\rrparenthesis$ then $U{::}V := \{a :: w \mid a \in U \wedge w \in V\}$ We have $U{::}V \subseteq \llparenthesis \mathsf{L}(A)\rrparenthesis$ iff $|U|_A + |V|_{\mathsf{L}(A)} + 1 \leq N$ and in this case $|U :: V|_{\mathsf{L}(A)} = |U|_A + |V|_B + 1$.

If $U \subseteq \llparenthesis A \multimap B\rrparenthesis$ and $V \subseteq \llparenthesis A\rrparenthesis$ then $U(V) := \{b \mid \exists a \in V.(a,b) \in U\}$ We have $|U(V)|_B \leq |U|_{A\multimap B} + |V|_A$.

We formally extend $\sim^n_A$ by putting $\bot \sim^n_A \emptyset$. Notice that whenever $x \in \llbracket A\rrbracket \cup \{\bot\}$ and $x \sim^n_A U$ and $x \neq \bot$ then $U \neq \emptyset$.

The defining clauses are now given as follows.

$$\begin{array}{lcl}
\multicolumn{3}{c}{\mathsf{tt} \sim^n_{\mathsf{T}} \{\mathsf{tt}\} \quad \mathsf{ff} \sim^n_{\mathsf{T}} \{\mathsf{ff}\} \quad 0 \sim^{n+1}_{\diamond} \{0\} \quad [] \sim^n_{\mathsf{L}(A)} \{[]\}} \\
(a,b) \sim^n_{A\otimes B} W & \iff & \exists n_1, n_2, U, V. n_1 + n_2 = n \\
 & & \quad \wedge\ a \sim^{n_1}_A U \wedge b \sim^{n_2}_B V \wedge W = U \times V \\
f \sim^n_{A\multimap B} U & \iff & \forall n_1, x. n + n_1 \leq N \\
 & & \quad \wedge\ x \sim^{n_1} V \Rightarrow f(x) \sim^{n+n_1}_B U(V) \\
x :: l \sim^n_{\mathsf{L}(A)} W & \iff & \exists n_1, n_2, U, V. n_1 + n_2 + 1 \leq n \\
 & & \quad \wedge\ x \sim^{n_1}_A U \wedge l \sim^{n_2}_{\mathsf{L}(A)} V \wedge W = U :: V
\end{array}$$

Notice that if $m \leq n \leq N$ then $x \sim^m_A U$ implies $x \sim^n_A U$. Notice also that if A is heap-free and $x \sim_A U$ then U has at most one element; exactly one if $x \neq \bot$. We write $\eta \sim^n_\Gamma U$ for $\eta \in \llbracket\Gamma\rrbracket$ and $U \subseteq \llparenthesis\Gamma\rrparenthesis$ if there exist $\mathrm{dom}(\Gamma)$-indexed families $(n_x)_x, (U_x)_x$ such that $\sum_{x\in\mathrm{dom}(\Gamma)} n_x \leq n$

and $U = \prod_{x \in \mathrm{dom}(\Gamma)} U_x$ and $\eta(x) \sim^{n_x}_{\Gamma(x)} U_x$ for all $x \in \mathrm{dom}(\Gamma)$. If X, Y are sets and $f : X \to Y$ and $U \subseteq X$ we define

$$f(U) := \{y \in Y \mid \exists x \in U. y \in f(x)\} \tag{17}$$

Similarly, if $\Gamma \vdash e : A$ and $U \subseteq (\!|\Gamma|\!)$ we define

$$(\!|\Gamma \vdash e : A|\!)_{U,\rho} = \{b \mid \exists \eta \in U. \eta (\!|\Gamma \vdash e : A|\!)_\rho b\} \tag{18}$$

Suppose that we are given a domain-theoretic valuation ψ and a relational valuation ρ of a given signature Σ. We will write $\psi \sim \rho$ to mean that for each function symbol $f : (A_1, \dots, A_r) \to B$ declared in Σ and whenever $n = n_1 + \cdots + n_r \leq N$ one has $\bigwedge_i u_i \sim^{n_i}_{A_i} U_i \Longrightarrow \psi(f)(u_1, \dots, u_r) \sim^n_B \rho(f)^n(U_1, \dots, U_r)$ We now have the following sublemma :

Sublemma: *Suppose that $\psi \sim \rho$. If $\Gamma \vdash_\Sigma e : A$ and $\eta \sim^n_\Gamma U$ then $[\![e]\!]_{\eta,\psi} \sim^n_A (\!|\Gamma \vdash e : A|\!)_{U,\rho}$*

Proof of sublemma: By induction on typing derivations. For rule VAR we use the fact that U is nonempty.

Rule SIG follows from the assumption made on ψ and ρ.

Rule CONTR uses the fact that elements of heap-free type have zero size as well as the observation that whenever $v \sim_A U$ for heap-free A then U has at most one element which implies that whenever $\eta \sim_{\Gamma,x:A,y:A} U$ where $U_x = U_y$ and $\eta \in U$ then $\eta = \eta[y \mapsto \eta(x)]$. These are the only two properties of heap-free types used thus allowing for possible extensions. All other cases are direct.

□

Now let ψ_0 be the valuation defined by $\psi_0(f)(\vec{x}) = \bot$ and ρ_0 be the relational valuation that assigns the empty relation to each function symbol. Clearly, $\psi_0 \sim \rho_0$ and so the sublemma shows that $\psi_m \sim \rho_m$ for all m where

$$\begin{aligned} \psi_{m+1}(f)(v_1, \dots, v_r) &= [\![e_f]\!]_{[x_1 \mapsto v_1, \dots, x_r \mapsto v_r], \psi_m} \\ \rho_{m+1}(f)(v_1, \dots, v_r) &= (\!|e_f|\!)_{[x_1 \mapsto v_1, \dots, x_r \mapsto v_r], \rho_m} \end{aligned} \tag{19}$$

As already mentioned, in view of the finiteness of the sets $(\!|A|\!)$ there exists m_0 such that $(\!|P|\!)(f) = \rho_{m_0}(f)$ for all $f \in \mathrm{dom}(\Sigma)$. Therefore, $\forall m \geq m_0. \rho_m \sim (\!|P|\!)$.

Now, $[\![P]\!](f) = \bigvee_m \rho_m(f) = \bigvee_{m \geq m_0} \rho_m(f)$. It is readily seen by induction on types that each relation $\sim^n_A$ is continuous in the sense that $\forall i. x_i \sim^n_A U$ implies $(\bigvee_i x_i) \sim^n_A U$ assuming of course that the x_i form an ascending chain. We have thus proved that $[\![P]\!] \sim (\!|P|\!)$ which yields the desired result when specialised to the type $\mathsf{L}(\mathsf{T})$. □

The idea is now to compute for a given N the iterations ρ_m by stepwise updating a big value table holding the relations $\rho_m(f)$.

To estimate the size of such a value table we must estimate the number of elements of the sets $(\!|A|\!)$. Writing $\#X$ for the cardinality of set X we have

$$\begin{array}{lll} \log \#(\!|\mathsf{T}|\!) = 1 & \log \#(\!|\diamond|\!) = 0 & \log \#(\!|\mathsf{L}(A)|\!) \leq N \log \#(\!|A|\!) \\ \log \#(\!|A \otimes B|\!) \leq \log \#(\!|A|\!) + \log \#(\!|B|\!) & & \log \#(\!|A \multimap B|\!) \leq \log \#(\!|A|\!) + \log \#(\!|B|\!) \end{array} \tag{20}$$

Therefore, for a given program P we can find a polynomial p such that $\log \#(\!|A|\!) \leq p(N)$ for each type A occurring in P.

The space required to store a relational valuation for P in the relational model is therefore $O(2^{p(N)})$ where the hidden constant involves the number and arities of function symbols.

Now, using the definition of $(\!|\Gamma \vdash e : A|\!)$ the computation of ρ_{m+1} given a value table for ρ_m and space to hold ρ_{m+1} can be performed with $O(p(N))$ extra space which would be required e.g. to hold particular elements of $(\!|A|\!)$.

In order to compute $(\!|P|\!)$ we maintain space for two value tables initialising both with the empty relational valuation. If at any time one of the two tables holds ρ_m we perform the necessary computations to achieve that the other one holds ρ_{m+1}. Thereafter, ρ_m is not needed anymore so that we can overwrite it with ρ_{m+2} and so forth, until no more changes take place and we have found $(\!|P|\!)$.

Since $\rho_m \subseteq \rho_{m+1}$ the number of iterations is $O(2^{p(N)})$ as well (in the worst case each iteration adds one single tuple to ρ), so that we have given a $\mathit{DTIME}(O(2^{p(N)}))$ algorithm for computing $(\!|P|\!)$ hence $[\![P]\!](f)(l)$ for $f : (\mathsf{L}(\mathsf{T}))\to\mathsf{L}(\mathsf{T})$ when $|l| \leq N$. This concludes the proof of Theorem 4.7.

References

[1] Klaus Aehlig and Helmut Schwichtenberg. A syntactical analysis of non-size-increasing polynomial time computation. In *Proceedings of the Fifteenth IEEE Symposium on Logic in Computer Science (LICS '00), Santa Barbara*, 2000.

[2] Stephen Bellantoni and Stephen Cook. New recursion-theoretic characterization of the polytime functions. *Computational Complexity*, 2:97–110, 1992.

[3] Vuokko-Helena Caseiro. *Equations for Defining Poly-time Functions.* PhD thesis, University of Oslo, 1997. Available by ftp from `ftp.ifi.uio.no/pub/vuokko/0adm.ps`.

[4] Stephen A. Cook. Linear-time simulation of deterministic two-way pushdown automata. *Information Processing*, 71:75–80, 1972.

[5] Andreas Goerdt. Characterizing complexity classes by higher type primitive recursive definitions. *Theoretical Computer Science*, 100:45–66, 1992.

[6] Martin Hofmann. Linear types and non size-increasing polynomial time computation. To appear in Theoretical Computer Science. See `www.dcs.ed.ac.uk/home/papers/icc.ps.gz` for a draft. An extended abstract has appeared under the same title in Proc. Symp. Logic in Comp. Sci. (LICS) 1999, Trento, 2000.

[7] Martin Hofmann. A type system for bounded space and functional in-place update. *Nordic Journal of Computing*, 2001. To appear, see `www.dcs.ed.ac.uk/home/mxh/papers/nordic.ps.gz` for a draft. An extended abstract has appeared in *Programming Languages and Systems*, G. Smolka, ed., Springer LNCS, 2000.

[8] Neil Jones. The Expressive Power of Higher-Order Types or, Life without CONS. *Journal of Functional Programming*, 2001. to appear.

[9] Daniel Leivant. Stratified Functional Programs and Computational Complexity. In *Proc. 20th IEEE Symp. on Principles of Programming Languages*, 1993.

[10] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction.* MIT Press, 1993.