A Syntactical Analysis of Non-Size-Increasing Polynomial Time Computation

KLAUS AEHLIG and HELMUT SCHWICHTENBERG
Universität München

A syntactical proof is given that all functions definable in a certain affine linear typed λ-calculus with iteration in all types are polynomial time computable. The proof provides explicit polynomial bounds that can easily be calculated.

Categories and Subject Descriptors: F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*Lambda calculus and related systems*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems

General Terms: Theory, Languages

Additional Key Words and Phrases: Complexity, lambda calculus, linear logic

1. INTRODUCTION

Recent research [Bellantoni and Cook 1992; Bellantoni et al. 2000; Hofmann 1998] has provided many characterizations of the class PTIME of all polynomial time computable functions by means of appropriate restrictions of the terms in Gödel's T [Gödel 1958]. Consider the following definition of an exponentially growing function:

$$\begin{array}{rl} \mathtt{double}([\,]) := [\,] & \qquad \mathtt{exp}([\,]) := [\mathrm{tt}] \\ \mathtt{double}([a \mid \ell\,]) := [a, a \mid \mathtt{double}(\ell)\,], & \qquad \mathtt{exp}([a \mid \ell\,]) := \mathtt{double}(\mathtt{exp}(\ell)). \end{array}$$

Approaches based on predicative recursion [Simmons 1988; Bellantoni and Cook 1992; Leivant 1991] argue that the exponential growth in this example is due to the way `double` is called: the previous value $\mathtt{exp}(\ell)$ of the outer recursion is the recursive argument to the inner recursion.

Although predicative recursion characterizations can capture all *functions* computable in polynomial time, many natural *algorithms* are excluded,

Klaus Aehlig was supported by the Ph.D. program "Logik in der Informatik" of the "Deutsche Forschungsgemeinshaft."
Authors' addresses: Mathematisches Institut, Universität München, Theresienstraße 39, 80333 Munich, Germany; email: {aehlig, schwicht}@mathematik.uni-muenchen.de.

particularly if they involve nested recursions. Standard examples are sorting algorithms like insertion sort, which has a similar recursive structure as exp:

$$\begin{aligned}
\mathtt{insert}(a, [\,]) &:= [a] \\
\mathtt{insert}(a, [\,b \mid \ell\,]) &:= [\,b' \mid \mathtt{insert}(a', \ell)\,], \quad a', b' \text{ a permutation of } a, b \text{ with } b' \leq a', \\
\mathtt{sort}([\,]) &:= [\,] \\
\mathtt{sort}([\,a \mid \ell\,]) &:= \mathtt{insert}(a, \mathtt{sort}(\ell)).
\end{aligned}$$

Caseiro [1997] studied many related examples and reached some (partially semantic) criteria for algorithms in order to ensure polynomial time complexity. For the insertion sort algorithm the essential point is that insert does not increase the size of its input. Hofmann [1999] took up this line of research and formulated a new term system with a special type $\diamond$ of tokens that accommodates nested recursion, but only allows one to define non-size-increasing PTIME functions. The basic idea is that if a function (like a successor) increases the size of its argument, then one has to "pay" for the increase by providing a token, that is, a term of type $\diamond$.

Hofmann [1999] proved this result by means of inherently semantic concepts, such as the set-theoretic interpretation of terms. We present a new proof of his main result, which apart from being simpler provides additional insight and yields a construction of time-bound polynomials which is more explicit than Hofmann's.

The method developed here has several benefits:

—A reduction relation is defined in such a way that the term system is closed under it. Therefore calculations can be performed *within* the system.

—We not only show that every definable function is polytime, but give explicit polynomial bounds for the number of reduction steps that can be determined easily for any given term.

—Hofmann's [1999, §3.2] semantical size measure (minimal upper bound of growth) is replaced by the syntactic concept of the number of free variables. Hence the role of the $\diamond$-type becomes more transparent, as we will show (in Lemma 4.3) that there are no closed terms of this type.

A preliminary version [Aehlig and Schwichtenberg 2000] of this work has already been published. Apart from giving more technical details and elaborated proofs, the following aspects are added:

—The previous estimate of the number of reduction steps referred to a fixed reduction strategy. Here we show that this requirement can be relaxed somewhat, without loosing the sharp estimate on the length of reduction sequences. The main tool is an appropriate modification of the size measure $\vartheta(\cdot)$.

—It is shown how the approach covers more complex data structures such as binary labeled trees, for which iteration involves two recursive calls.

Hofmann's [1999] work is of course the starting point of ours. In the journal version of that work [Hofmann 2000a] some new aspects are added, among

others the definition and justification of a similar system which captures PSPACE, and an operator for divide-and-conquer recursion. We shall show in Section 5.2 that our simplified approach can deal with the latter; however, it is not known whether it also suffices for Hofmann's characterization of PSPACE.

Jones [2001] directly related programming languages and complexity theory. In previous works [Jones 1999, 1997] he had characterized the power of first-order read-only programs in complexity terms, building on Cook [1971]; the results are extended to arbitrary data orders and tail recursive programs.

Jones' work is certainly related to our current work. In his approach it is important whether constructors of structured data are allowed or not. Moreover, Jones' paper is written from a broader programming point of view, giving special emphasis to the program control structure (general recursion, tail recursion, or primitive recursion). The present paper (for simplicity) concentrates on the effects of higher-order primitive recursion (i.e., Gödel's T).

Leivant and Marion [1993] gave a λ-calculus characterization of PTIME. The main novelty was the use of the concept of "tiers" in the sense of different representations of data: as words (i.e., terms of base type), or as Church-like abstraction terms (expressions of higher type). Leivant [1999] gave particularly elegant proofs of characterizations of Jones [1999] for PTIME and of Beckmann and Weiermann [2000] for the elementary functions. He also treated a control-based subcalculus of the system in Bellantoni et al. [2000], and—more importantly—obtained a characterization of PSPACE along the same lines. This was achieved by the notions of an "input-driven" term, that is, a term with the property that no recursion argument has a free variable bound in the term, and the more special notion of a "solitary" term, where every higher-type abstracted variable has at most one occurrence in the kernel. The solitary terms are the ones used for his characterization of PTIME. As with all extensional characterizations of PTIME, the question remains as to whether interesting PTIME algorithms can be represented directly.

2. TYPES AND TERMS

Definition 2.1 (*Finite Types*). The set of linear types is defined inductively as

$$\rho, \tau \ ::= \ \diamond \mid \mathbf{B} \mid \tau \multimap \rho \mid \tau \otimes \rho \mid \tau \times \rho \mid \mathbf{L}(\tau).$$

So types are built from the base type $\mathbf{B}$ of Booleans and a special type $\diamond$. $\mathbf{L}(\tau)$ denotes the list-type over τ. The type of binary numerals, as used by Hofmann [1999], can be defined as $\mathbf{L}(\mathbf{B})$. The type $\diamond$ may be interpreted as a pointer to free memory [Hofmann 2000b]; Lemma 4.3 will show that no closed terms of type $\diamond$ exist. Canonical terms of data-types may still contain free variables of that type. The type $\diamond \multimap \tau \multimap \mathbf{L}(\tau) \multimap \mathbf{L}(\tau)$ of the $\mathbf{cons}_\tau$ function together with the linear typing discipline guarantees that the length of lists and the number of free variables coincide.

$\tau \otimes \rho$ and $\tau \times \rho$ both represent ordered pairs. However, the linear interpreatation of those types is different. A tensor product allows access to each of its components whereas of an ordinary pair one can choose only one component.

As can be seen from Definition 3.2, $\otimes$ corresponds to $\multimap$, while $\times$ corresponds to $\mathbf{B}$.

Terms are built from variables, denoted by x, y, z, and typed constructor symbols c. Each variable has a type, and it is assumed that there are infinitely many variables of each type. The notation x^τ expresses that the variable x has type τ.

Definition 2.2 (*Terms*). The set of terms is inductively defined by

$$r, s, t \ ::= \ x \mid c \mid \lambda x.t \mid \langle t, s\rangle \mid ts \mid \{t\}\,.$$

These terms should be seen as our raw syntax; only correctly typed terms (in the sense of Definition 2.4) represent meaningful functions.

The $\{\cdot\}$ term construct is inspired by Joachimski and Matthes [1999]. Terms of this form appear only as arguments in an $\mathbf{L}(\tau)$ elimination. This explicit marking of the step term in iteration constructs is not only technically convenient but also allows one to directly read off the time complexity: the degree of the bounding polynomial is the nesting depth of braces and the symbols within n braces contribute to the coefficient of X^n (see Definition 4.5 for details).

The notations $t\vec{s}$ and $\lambda\vec{x}.t$ are defined as usual, so $\lambda x, y.t$ is an abbreviation for $\lambda x.(\lambda y.t)$. Terms that only differ in the naming of bound variables are identified. We also use a, d, h to denote terms.

Definition 2.3 (*Constructor Symbols*). The constructor symbols and their types are

$$\begin{array}{ll}
\mathbb{t}, & \mathbf{B},\\
\mathbb{f}, & \mathbf{B},\\
\mathbf{nil}_\tau, & \mathbf{L}(\tau),\\
\mathbf{cons}_\tau & \diamond \multimap \tau \multimap \mathbf{L}(\tau) \multimap \mathbf{L}(\tau),\\
\otimes_{\tau,\rho} & \tau \multimap \rho \multimap \tau \otimes \rho.
\end{array}$$

A *context* is a set of variables. For two contexts Γ_1 and Γ_2 the notation Γ_1, Γ_2 stands for the union, expressing that Γ_1 and Γ_2 are disjoint. We also write x^τ for the singleton context $\{x^\tau\}$.

The next definition states which terms of our raw syntax are correctly typed. $\Gamma \vdash t^\tau$ is to be read as "t is a typed term of type τ with free variables in Γ."

Definition 2.4 ($\Gamma \vdash t^\tau$). The relation $\Gamma \vdash t^\tau$ is inductively defined as follows:

$$\frac{}{\Gamma, x^\tau \vdash x^\tau}\ (\text{Var}), \quad \frac{c \text{ of type } \tau}{\Gamma \vdash c^\tau}\ (\text{Const}),$$

$$\frac{\Gamma \cup \{x^\tau\} \vdash t^\rho}{\Gamma \vdash (\lambda x^\tau.t)^{\tau\multimap\rho}}\ (\multimap^+), \quad \frac{\Gamma_1 \vdash t^{\tau\multimap\rho} \quad \Gamma_2 \vdash s^\tau}{\Gamma_1, \Gamma_2 \vdash (ts)^\rho}\ (\multimap^-),$$

$$\frac{\Gamma \vdash t^\tau \quad \Gamma \vdash s^\rho}{\Gamma \vdash \langle t, s\rangle^{\tau\times\rho}}\ (\times^+), \quad \frac{\Gamma \vdash t^{\tau\times\rho}}{\Gamma \vdash (t\mathbb{t})^\tau}\ (\times_1^-), \quad \frac{\Gamma \vdash t^{\tau\times\rho}}{\Gamma \vdash (t\mathbb{f})^\rho}\ (\times_0^-),$$

$$\frac{\Gamma_1 \vdash t^{\mathbf{B}} \quad \Gamma_2 \vdash s^{\tau} \quad \Gamma_2 \vdash r^{\tau}}{\Gamma_1, \Gamma_2 \vdash (t\ \langle s, r\rangle)^{\tau}}\ (\mathbf{B}^{-}), \quad \frac{\Gamma_1 \vdash t^{\tau\otimes\rho} \quad \Gamma_2, x^{\tau}, y^{\rho} \vdash s^{\sigma}}{\Gamma_1, \Gamma_2 \vdash (t(\lambda x^{\tau}, y^{\rho}.s))^{\sigma}}\ (\otimes^{-}),$$

$$\frac{\Gamma \vdash t^{\mathbf{L}(\tau)} \quad \emptyset \vdash s^{\diamond\multimap\tau\multimap\rho\multimap\rho}}{\Gamma \vdash (t\ \{s\})^{\rho\multimap\rho}}\ (\mathbf{L}(\tau)^{-}).$$

It is crucial that the step term in $(\mathbf{L}(\tau)^{-})$ is closed. Otherwise subject reduction would fail, since free variables in the step terms would be duplicated.

The typing system is based on elimination rules: for every type different to $\diamond$ there is a rule describing how to use terms of this type; in all these rules the elimination is written as application.[1] For the right-hand side of such an application a notation is chosen that expresses the computational behavior of the terms used for the elimination. This avoids duplication of syntax, for example, with the pair $\langle t, s\rangle$ we have a notation expressing that exactly one of the terms t and s will be needed. This syntax is also used for the "if ... then ... else ... " construct $t\ \langle s, r\rangle$.

Data-types, that is, types from which all the stored information can be retrieved by a single use of an object of this type, are introduced by constants. In the system we have two forms of abstraction with special introduction rules: the λ-abstraction and the pair. Note that the pair really is an abstraction, as only part of the stored information can be retrieved by a single use. The tensor $\tau \otimes \rho$, however, is a data-type as a single use can access both components; hence we have a constant $\otimes_{\tau,\rho}$ to introduce it.

This way of introduction allows the relatively simple Definition 3.3 of the reduction relation expressing that we may reduce within data, but not under abstractions.

Immediately obvious from the definition is

PROPOSITION 2.5 (WEAKENING). *If* $\Gamma \vdash t^{\tau}$ *and* $\Gamma \subset \Gamma'$ *then* $\Gamma' \vdash t^{\tau}$.

Rule $(\multimap^{+})$ could as well have been written

$$\frac{\Gamma, x^{\tau} \vdash t^{\rho}}{\Gamma \vdash (\lambda x^{\tau}.t)^{\tau\multimap\rho}}$$

requiring that the bound variable does not occur in the context afterwards. In our formulation it is easier to recognize weakening as a admissible rule. As we identify α-equal terms, weakening holds in the alternative formulation as well and both formulations are in fact equivalent.

It might seem odd that in this calculus a typed term can have an untypable subterm $\{t\}$. An obvious definition would introduce a new form of application, for example, if r, s are terms then so is $r\ \{s\}$, rather than a new term former $\{s\}$. However, the present approach is technically more convenient, since it allows the simple classification of terms according to their head form (cf. Lemma 2.8).

It should be noted that with some technical overhead one could resolve this problem by separating terms and so-called elimination terms, as in work of

[1]It should be noted that the same syntax would appear in the standard implementation in the untyped λ-calculus. For example one should think of $\otimes$ as being $\lambda x, y, z.zxy$.

Joachimski and Matthes [1999]. But in the present rather simple situation this seems to be overkill.

The notation t^τ expresses that there is a Γ such that $\Gamma \vdash t^\tau$. The smallest such Γ is called the set of free variables. By induction on the definition of $\Gamma \vdash t^\tau$ one easily verifies:

LEMMA 2.6. *For each term t there is at most one type τ such that t^τ. In this case there is a smallest Γ with $\Gamma \vdash t^\tau$ which coincides with the set of free variables of t in the usual sense.*

As already mentioned, the main restriction of the typing calculus is linearity, that is, in an application $t_1 t_2$ the free variables of t_1 and t_2 have to be disjoint. This is stated explicitly in the rule $(\multimap^-)$, but holds for all other "applications" as well. More precisely:

LEMMA 2.7 (LINEARITY). *Assume $(t_1 t_2)^\tau$. Then either $t_2 = \{t_2'\}$ with closed t_2' or else there are types τ' and τ'' and disjoint contexts Γ_1, Γ_2 such that $\Gamma_1 \vdash {t_1}^{\tau'}$ and $\Gamma_2 \vdash {t_2}^{\tau''}$.*

PROOF.[2] The last rule of a derivation of $\Gamma \vdash (t_1 t_2)^\tau$ must be one of $(\multimap^-)$, $(\times_0^-)$, $(\times_1^-)$, $(\mathbf{B}^-)$, $(\otimes^-)$, or $(\mathbf{L}(\tau)^-)$. In each of these cases the claim is trivial. For example, in the case $(\otimes^-)$ we conclude by two applications of $(\multimap^+)$ that $\Gamma_2 \vdash (\lambda x^\tau, y^\rho.s)^{\tau \multimap \rho \multimap \sigma}$. □

The fact that all eliminations are written as applications ensures that all typed terms have a uniform appearance. As can easily be verified by induction on the definition of the relation $\vdash$, we have

LEMMA 2.8 (HEAD FORM). *If $\Gamma \vdash t^\tau$ then t is of the form $x\vec{t}$, $c\vec{t}$, $(\lambda x^\rho.s)\vec{t}$, or $\langle s, r\rangle\,\vec{t}$.*

It should be noted that this lemma, although technically trivial, turns out to be crucial for the further development: as we are taking vector notation seriously and only have constants for the introduction of data-types, case distinction according to this lemma (and further according to the constant c in the case $c\vec{t}$) is essentially case distinction according to the "form of the next canonical redex," without the need of defining such a notion.

3. REDUCTIONS

The reduction rules to be defined are all sound with respect to the set-theoretic semantics [Hofmann 2000a, §2.1]. In order to control the effects of recursion we allow reduction of an iteration only if the argument is already calculated, that is, if the argument is a list.

Definition 3.1. *Lists* ℓ (with n entries) are terms of the form

$$\mathbf{cons}_\tau d_1^\diamond a_1^\tau (\ldots (\mathbf{cons}_\tau d_n^\diamond a_n^\tau \mathbf{nil}_\tau)),$$

where $d_1, \ldots, d_n$ stand for arbitrary terms of type $\diamond$.

[2]It should be noted that this is just an inspection of the rules and in particular *no* induction is needed.

It should be noted that we could also have required the $\vec{d}$ to be variables and gotten the same results. However, this definition allows more reductions (see Definition 3.2) and therefore is slightly more flexible when dealing with terms that are not almost closed, that is, contain free variables of types other than $\diamond$.

Definition 3.2 (*Conversions*). $\mapsto$ is defined as

$$\begin{array}{ll}
(\lambda x^{\tau}.t)s & \mapsto t[s/x], \\
\langle t, s\rangle\, \mathbb{tt} & \mapsto t, \\
\langle t, s\rangle\, \mathbb{ff} & \mapsto s, \\
\mathbb{tt}\,\langle t, s\rangle & \mapsto t, \\
\mathbb{ff}\,\langle t, s\rangle & \mapsto s, \\
\otimes_{\tau,\rho} ts(\lambda x^{\tau}, y^{\rho}.r) & \mapsto r[t, s/x, y], \\
\mathbf{nil}_{\tau}\,\{t\}s & \mapsto s, \\
\mathbf{cons}_{\tau} d^{\diamond} a \ell\,\{t\}s & \mapsto t d^{\diamond} a(\ell\,\{t\}s), \quad \ell \text{ a list.}
\end{array}$$

Although they look quite similar, the rules $\langle t, s\rangle\, \mathbb{tt} \mapsto t$ and $\mathbb{tt}\,\langle t, s\rangle \mapsto t$ actually have very different meaning. The first rule says that we can unfold a projection once the argument of the term of $\times$-type is in canonical form, whereas the other rule tells us to take the if-branch, once we know that the conditional evaluates to $\mathbb{tt}$. Also notice the different typings of the two rules.

Definition 3.3. The reduction relation $t \to t'$ is inductively defined as follows:

$$\frac{t \mapsto t'}{t \to t'}\,, \quad \frac{t \to t'}{ts \to t's}\,, \quad \frac{s \to s'}{ts \to ts'}\,.$$

The requirement that we can only unfold one step of an iteration if the argument is a full list (in the sense of Definition 3.1) is crucial for our method of estimating the length of reduction sequences (cf. the introduction to Section 4). This can be seen from the corresponding case in the proof of Theorem 4.8. Apart from this restriction and the requirement that one cannot reduce under abstractions, the reduction strategy is arbitrary.

As usual, we call a term t *normal* if it cannot be further reduced, that is, if there is no t' such that $t \to t'$.

LEMMA 3.4. *If* $\Gamma_1 \cup \{x^{\rho}\} \vdash t^{\tau}$ *and* $\Gamma_2 \vdash s^{\rho}$ *and moreover* Γ_1 *and* Γ_2 *are disjoint, then* $\Gamma_1, \Gamma_2 \vdash (t[s/x^{\rho}])^{\tau}$.

PROOF. Induction on t, using Lemma 2.7: if t is a variable or a constant the claim is obvious. If t is of the form $t = t_1 t_2$ then by Lemma 2.7 either $t[s/x] = t_1[s/x]t_2$ or $t[s/x] = t_1(t_2[s/x])$. We apply the induction hypothesis to the corresponding subterm and can type $t[s/x]$ by the same rule used to type t. If t is of the form $t = \lambda y.r$, then t must be typed due to $(\multimap^{+})$; hence $\Gamma_1 \cup \{x, y\} \vdash r^{\tau'}$ and we may apply the induction hypotheses to r (without loss of generality y does not to occur in Γ_2) and then conclude the claim by $(\multimap^{-})$. The proof is similar if $t = \langle t_1, t_2\rangle$. □

LEMMA 3.5 (SUBJECT REDUCTION). *If* $\Gamma \vdash t^{\tau}$ *and* $t \to t'$, *then* $\Gamma \vdash t'^{\tau}$.

PROOF. Induction on $t \to t'$. The only nontrivial case is handled in Lemma 3.4. □

Next we show that we have sufficiently many reduction rules, that is, that normal terms have the expected form. As the size of a term is its number of free variables, we also have to consider nonclosed terms. Since $\diamond$ is the only base type without closed terms, we can restrict ourselves to *almost closed* terms, which are defined to be terms with free variables of type $\diamond$ only. It should be noted that (for example) an arbitrary list can be encoded as an almost closed term.

PROPOSITION 3.6. *Every normal, almost closed term of type* $\mathbf{L}(\tau)$, $\mathbf{B}$, *or* $\diamond$ *is a list,* $\mathbb{t}$ *or* $\mathbb{f}$, *or a variable of type* $\diamond$, *respectively.*

PROOF. Induction on t and case distinction according to Lemma 2.8: If t is of the form $x\vec{t}$ then x has to be of type $\diamond$ and hence $\vec{t}$ has to be empty (since there is no elimination rule for the type $\diamond$). If t is of the form $(\lambda x.s)\vec{t}$ or $\langle s, r\rangle\,\vec{t}$, then $\vec{t}$ has to be empty as well, for otherwise the term would not be normal. So the interesting case is if t is of the form $c\vec{t}$. In this case we distinguish cases according to c. If c is $\mathbb{t}$ or $\mathbb{f}$, $\mathbf{nil}_\tau$, or $\otimes_{\tau,\rho}$, then $\vec{t}$ can only consist of at most 0, 1, or 2 terms, respectively (for otherwise there were a redex); hence we have the claim (or the term is not of one of the types we consider). In the case $\mathbf{cons}_\tau d\,ar\vec{t}$ the induction hypothesis for r yields that t is of the form $\ell\vec{t}$ with a *list* ℓ. So if $\vec{t}$ consists of more than one term, there would be a redex; but if $\vec{t}$ is a single term, then the whole term would have arrow-type. □

It should be noted that in the above proof the only place where the induction hypothesis is actually needed is to argue that in $\mathbf{cons}_\tau d^\diamond ar$ the subterm r has to be a list as well (and hence the whole term is a list). In particular, these statements need not all be proved simultaneously, but only simultaneously with the statement that every normal, almost closed term of list-type is in fact a list (which is necessary due to the side-condition on the reduction rules for lists). This modularity is a feature of the vector notation, that is, the consistent use of elimination-*rules* written as *applications*: in this way the (syntactical) Lemma 2.8 gives easy access to the canonical redex by showing the corresponding introduction. It should be noted that the use of elimination-*constants* ("iterators") would have messed up this modularity, as in the case of an elimination constant followed by some arguments one would in fact need some form of induction hypothesis stating that the argument to be eliminated is introduced canonically.

4. LENGTH OF REDUCTION SEQUENCES

Before continuing, we give a sketch of the main idea of the proof in order to motivate the following definitions.

Since the system is linear, β-reduction reduces the number of symbols (the λ disappears). The same is true for the reductions due to projections and "if ... then ... else." So the only case where the number of symbols is increased is the case of iteration. However, if we unfold an iteration *completely*, that is, if we reduce $\mathbf{cons}_\tau d_1^\diamond r_1^\tau(\ldots(\mathbf{cons}_\tau d_n^\diamond r_n^\tau \mathbf{nil}_\tau))\{h\}\,t$ in $n+1$ steps to $hd_1^\diamond r_1^\tau(\ldots(hd_n^\diamond r_n^\tau t))$ then the $\{\cdot\}$'s disappear. So by making them "sufficiently heavy" the total weight

of the term reduces. For the $\{\cdot\}$-construct the weight has to depend not only on the term within the braces but also on the length n of the numeral we iterate on. So instead of a fixed number we assign a polynomial(ly bound function) to this term. But what argument will this polynomial get? Noting that we have n terms $d_1^\diamond, \ldots, d_n^\diamond$ of type $\diamond$ and remembering the idea that terms of type $\diamond$ always contain a free variable (as will be proved in Proposition 4.3), we find that the number of free variables is just the right upper bound for the length n (since the $\vec{d}$ appear in a joint application their free variables have to be disjoint).

This would lead to a proof that every reduction strategy that *always unfolds iterations completely* is polynomially bounded. In order to get results for *every* reduction strategy we notice that within the subterm $\mathbf{cons}_\tau d_1^\diamond r_1^\tau(\ldots(\mathbf{cons}_\tau d_n^\diamond r_n^\tau \mathbf{nil}_\tau))\{h\}t$ the iteration unfolds at most n times, even if the actual number of free variables in the whole term is larger.[3] Using this information we can limit the assigned polynomial(ly bounded function) to get a measure that actually decreases in every step (Theorem 4.8).

So we use three different measures for terms: the number of free variables, which corresponds to the "size function" in Hofmann's [1999] work; the length, which is the number of symbols of a term that can be accessed (using the interpretation that you can only access one component of an ordinary pair); and the polynomial, which is the upper bound for the complexity of a function.

Definition 4.1 (*Length*). The length $|t|$ of a term t is inductively defined as follows:

$$\begin{aligned} |c| := |x| &:= 1, \\ |ts| &:= |t| + |s|, \\ |\lambda x^\tau . s| &:= |s| + 1, \\ |\langle t, s\rangle| &:= \max\{|t|, |s|\} + 1, \\ |\{t\}| &:= 0. \end{aligned}$$

As the length $|t|$ is essentially used to handle β-redexes, the length of $\{\cdot\}$-terms (which are closed terms!) is of no importance. So for simplicity, and to obtain slightly sharper results (as the length occurs in the Definition 4.5 of the polynomial bound), the value 0 has been chosen. It should be noted that all the results would also hold with any other "reasonable" definition for $|\{t\}|$, such as $|t|$.

LEMMA 4.2. *If* $\Gamma_1 \vdash t^\tau$ *and* $\Gamma_2 \vdash s^\rho$ *and moreover* Γ_1, Γ_2 *are disjoint, then* $|t[s/x]| \leq |t| + |s|$. *In particular,* $|(\lambda x^\tau . t)s| > |t[s/x]|$.

PROOF. Induction on t, using the fact that t is typed and therefore in the case of an application only one of the terms can contain the variable x free (compare Lemma 2.7).

[3]This subterm might not unfold at all if it is positioned under a λ-abstraction that remains in the normal form. We do not allow to reduce under λ-abstractions (see Definition 3.3) since the number of free variables under a λ-abstraction is potentially higher and therefore numerals and lists are potentially longer. This restriction corresponds to not allowing the use of a "potential resource," that is, a resource that is not yet present.

For instance, if t is t_1t_2, then the last rule of a derivation of $\Gamma \vdash (t_1t_2)^\tau$, must be one of $(\multimap^-)$, $(\times_0^-)$, $(\times_1^-)$, $(\mathbf{B}^-)$, $(\otimes^-)$, or $(\mathbf{L}(\tau)^-)$. In each of these cases the claim is obvious. □

LEMMA 4.3. *Every term of type $\diamond$ contains a free variable, that is, if $\Gamma \vdash t^\diamond$, then $\Gamma \neq \emptyset$.*

PROOF. Induction on $|t|$ and case distinction according to Lemma 2.8. The case $x\vec{t}$ is trivial. In the case $(\lambda x.s)\vec{t}$ the $\vec{t}$ cannot be empty (for otherwise the term would have an arrow-type), so we can apply the induction hypothesis to the (by Lemma 4.2) shorter term $t[t_1/x]t_2 \cdots t_n$. Similarly for $\langle r, s\rangle\,\vec{t}$, where we apply the induction hypothesis to $rt_2 \cdots t_n$ or $st_2 \cdots t_n$ depending whether t_1 is $\mathbb{t}$ or $\mathbb{f}$ (note that by *typing* one of these has to be the case) and $\otimes_{\tau,\rho}\vec{t}$, where we again use Lemma 4.2. For $\mathbf{cons}_\tau\vec{t}$ we use the induction hypothesis for t_1. For $\mathbf{nil}_\tau\vec{t}$ use the induction hypothesis for $t_2 \cdots t_n$. In the cases $\mathbb{t}\vec{t}$ and $\mathbb{f}\vec{t}$, by typing, t_1 has to be of the form $\langle r, s\rangle$ and we can apply the induction hypothesis to (say) $rt_2 \cdots t_n$. □

The following corollary is formulated for type 1 functions only. However, if one accepts the number of free variables as a reasonable size measure for definable functions for higher types as well, then the corollary trivially remains valid for *every* definable function.

COROLLARY 4.4 (NON-SIZE-INCREASING PROPERTY). *Every function of type $\mathbf{L}(\tau) \multimap \mathbf{L}(\tau)$ definable by a closed term has the property that the output is not longer than the input.*

PROOF. Lemma 4.3 shows that for closed terms of type $\mathbf{L}(\tau)$ the usual length and the number of free variables coincide (due to the typing $\diamond \multimap \tau \multimap \mathbf{L}(\tau) \multimap \mathbf{L}(\tau)$ of the $\mathbf{cons}_\tau$ function). The number of free variables trivially does not increase when reducing the term to normal form. □

Let $\mathbb{N}$ be the set of natural numbers and $\mathbb{N}^{\mathrm{poly}}$ be the set of all functions from $\mathbb{N}$ to $\mathbb{N}$ that are (pointwise) bounded by some polynomial. We write X for the identity on the natural numbers, $f + g$ and $f \cdot g$ for the pointwise sum and product in $\mathbb{N}^{\mathrm{poly}}$, and X_n for the minimum of the identity and n, that is, $X_n(m) = \min\{n, m\}$. $\mathbb{N}$ is treated as a subset of $\mathbb{N}^{\mathrm{poly}}$ by identifying a natural number with the corresponding constant function. Let $\preccurlyeq$ be the pointwise order on $\mathbb{N}^{\mathrm{poly}}$. Then there exist finite suprema (the pointwise maxima) in $\mathbb{N}^{\mathrm{poly}}$ with respect to $\preccurlyeq$, denoted by $\sup_{\preccurlyeq}\{\cdot\}$.

It seems that by considering functions we leave the realm of syntax. But as we will only use functions that are explicitly defined by pointwise sum, product, maximum, and minimum from the identity and constants we could as well restrict ourselves to these functions only. It should be obvious how these functions can be encoded by some finite (syntactical) object.

Definition 4.5 (*Polynomial Bound $\vartheta(\cdot)$ of a Term*). For each (not necessarily typed) term t we define a polynomial $\vartheta(t) \in \mathbb{N}^{\text{poly}}$ by recursion on t:

$$\begin{aligned}
\vartheta(x) &:= \vartheta(c) := 0,\\
\vartheta(ts) &:= \begin{cases} \vartheta(t) + X_n \cdot \vartheta(h) + X_n \cdot |h| & \text{if } t \text{ is a list with } n \text{ entries}\\ & \text{and } s \text{ is of the form } \{h\},\\ \vartheta(t) + \vartheta(s) & \text{otherwise,} \end{cases}\\
\vartheta(\lambda x^{\tau}.t) &:= \vartheta(t),\\
\vartheta(\langle t, s\rangle) &:= \sup\nolimits_{\preccurlyeq}\{\vartheta(t), \vartheta(s)\},\\
\vartheta(\{h\}) &:= X \cdot \vartheta(h) + X \cdot |h|,
\end{aligned}$$

We write $\vartheta\left(\vec{t}\right)$ for $\sum_{i=1}^{n} \vartheta(t_i)$.

Immediately from the definition (since $X_n \preccurlyeq X$) we get

PROPOSITION 4.6. $\vartheta(ts) \preccurlyeq \vartheta(t) + \vartheta(s)$.

LEMMA 4.7. *If $\Gamma_1 \vdash t^{\tau}$, $\Gamma_2 \vdash s^{\rho}$, and Γ_1, Γ_2 are disjoint, then*

$$\vartheta(t[s/x]) \preccurlyeq \vartheta(t) + \vartheta(s).$$

PROOF. We prove this by induction on t. The only nontrivial case is when t is an application (note that the case $\{h\}$ does not occur since t is typed). So let t be of the form rr'.

If r is a list with n entries then, as t is typed, we know that r' must be of the form $r' = \{h\}$ and therefore closed. $r[s/x]$ is easiliy seen to be again a list with n entries. Therefore we get

$$\begin{aligned}
\vartheta((rr')[s/x]) &= \vartheta(r[s/x]r')\\
&= \vartheta(r[s/x]) + X_n \cdot \vartheta(h) + X_n \cdot |h|\\
&\preccurlyeq \vartheta(r) + \vartheta(s) + X_n \cdot \vartheta(h) + X_n \cdot |h| \quad \text{by IH}\\
&= \vartheta(rr') + \vartheta(s).
\end{aligned}$$

If r is not a list, then at most one of the terms r, r' contains the variable x free for typing reasons. In the case $x \in \mathrm{FV}(r)$ (the other case is handled similarly) we have

$$\begin{aligned}
\vartheta((rr')[s/x]) &= \vartheta(r[s/x]r')\\
&\preccurlyeq \vartheta(r[s/x]) + \vartheta(r') \quad \text{by Proposition 4.6}\\
&\preccurlyeq \vartheta(r) + \vartheta(s) + \vartheta(r') \quad \text{by IH}\\
&= \vartheta(rr') + \vartheta(s).
\end{aligned}$$

This completes the proof. □

THEOREM 4.8. *Assume $\Gamma \vdash t^{\tau}$ and $N \geq |\mathrm{FV}(t)|$ where $N \in \mathbb{N}$. If $t \to t'$; then*

$$\vartheta(t)(N) + |t| > (t')(N) + |t'|.$$

In particular, any *reduction sequence starting from t has length at most*

$$\vartheta\,(t)\,(|\mathrm{FV}\,(t)|) + |t|\,.$$

PROOF. We prove this by induction on the definition of the relation $t \to t'$.

Case $rs \to r's$ via $r \to r'$.

Subcase r is a list with n entries. Then by typing restrictions and from Lemma 4.3 we know that $n \le |\mathrm{FV}\,(t)| \le N$. Also s has to be of the form $s = \{h\}$. r' is again a list with n entries, so

$$\begin{array}{rlll} & \vartheta(rs)(N) & + & |rs| \\ = & \vartheta(r)(N) + n \cdot \vartheta(h)(N) + n \cdot |h| & + & |r| + |s| \\ > & \vartheta(r')(N) + n \cdot \vartheta(h)(N) + n \cdot |h| & + & |r'| + |s| \quad \text{by IH} \\ = & \vartheta(r's)(N) & + & |r's|. \end{array}$$

Subcase r is not a list. Then

$$\begin{array}{rlll} & \vartheta(rs)(N) & + & |rs| & \\ = & \vartheta(r)(N) + \vartheta(s)(N) & + & |r| + |s| & \\ > & \vartheta(r')(N) + \vartheta(s)(N) & + & |r'| + |s| & \text{by IH} \\ \ge & \vartheta(r's)(N) & + & |r's| & \text{by Proposition 4.6.} \end{array}$$

Case $rs \to rs'$ via $s \to s'$. Then r is not a list, since otherwise, s were of the form $\{\cdot\}$ and we must not reduce within braces. Therefore this case can be handled as the second subcase above.

Case $t \mapsto t'$. We distinguish subcases according to the form of the conversion.

Subcase $(\lambda x.\,t)s \mapsto t[s/x]$. We have

$$\begin{array}{rlll} & \vartheta((\lambda x.\,t)s)(N) & + & |(\lambda x.\,t)s| & \\ = & \vartheta\,(t)\,(N) + \vartheta\,(s)\,(N) & + & |t| + 1 + |s| & \\ \ge & \vartheta(t[s/x])(N) & + & |t| + 1 + |s| & \text{by Lemma 4.7} \\ > & \vartheta(t[s/x])(N) & + & |t[s/x]| & \text{by Lemma 4.2.} \end{array}$$

Subcase $\otimes_{\tau,\rho} rs(\lambda x,\, y\,.t) \mapsto t[r, s/x,\, y\,]$ is handled similarly.

Subcase $\mathbf{cons}_\tau d^\diamond a\ell\,\{s\}\,r \mapsto sd^\diamond a(\ell\,\{s\}\,r)$ with ℓ a list with n entries. Then by Lemma 4.3 and the linear typing discipline $n+1 \le |\mathrm{FV}\,(\mathbf{cons}_\tau d^\diamond a\ell\,\{s\}\,r)| \le N$. Therefore we have

$$\begin{array}{rl} & \vartheta\,(\mathbf{cons}_\tau d^\diamond a\ell\,\{s\}\,r)(N) \\ + & |\mathbf{cons}_\tau d^\diamond a\ell\,\{s\}\,r| \\ = & \vartheta\,(d\,)(N) + \vartheta\,(a)(N) + \vartheta\,(\ell)(N) \\ & + (n+1)\vartheta\,(s)(N) + (n+1)\,|s| + \vartheta\,(r)(N) \\ & + 1 + |d\,| + |a| + |\ell| + |r|\,, \end{array}$$

and on the other hand

$$\begin{array}{rl} & \vartheta(sd^\diamond a(\ell\,\{s\}\,r))(N) \\ + & |sd^\diamond a(\ell\,\{s\}\,r)| \\ = & \vartheta(s)(N) + \vartheta(d\,)(N) + \vartheta(a)(N) + \vartheta(\ell)(N) \\ & + n \cdot \vartheta\,(s)\,(N) + n \cdot |s| + \vartheta\,(r)\,(N) \\ & + |s| + |d\,| + |a| + |\ell| + |r|\,, \end{array}$$

which obviously is strictly smaller.

Subcase $\mathtt{tt}\,\langle r, s\rangle \mapsto r$. We have

$$\begin{array}{lll} & \vartheta\,(\mathtt{tt}\,\langle r,s\rangle)(N) & + \ |\mathtt{tt}\,\langle r,s\rangle| \\ = & \sup_{\preccurlyeq}\{\vartheta\,(r)\,,\vartheta\,(s)\}(N) & + \ 1+\max\{|r|\,,|s|\} \\ \geq & \vartheta\,(r)(N) & + \ 1+|r|\,. \end{array}$$

The remaining subcases are similar to the last one. □

COROLLARY 4.9. *If* $\emptyset \vdash t^{\mathbf{L}(\mathbf{B})\multimap\mathbf{L}(\mathbf{B})}$, *then* $t\ell$ *can be reduced to normal form in polynomial many steps in the length of* ℓ *for any almost closed normal* ℓ.

PROOF. Let ℓ be an almost closed normal list. Then $\vartheta\,(t\ell) = \vartheta\,(t)$, as $\vartheta\,(\ell) = 0$. Therefore an upper bound for the number of steps is $\vartheta\,(t)(|\mathrm{FV}\,(\ell)|) + |t| + |\ell|$, which is polynomial in the length of the input ℓ. □

Strictly speaking, the above result does not show that functions definable by closed terms are polynomial time computable, since no machine model has been provided. However, it is obvious how these reductions can be implemented in constant time (with respect to the size of the input): First note that the only terms that have to be duplicated are the step terms of iterations which are *not* part of the input and—as they are closed terms—no part of the input can become substituted into them. Next remark that all β-redexes are linear and hence can be implemented by the usual pointer switching techniques. In the case of $\langle\cdot,\cdot\rangle$ an indirection node is used for the shared variables of the two components. Our reduction strategy does not reduce within a pair until the projection or the "if ... then ... else" statement is resolved. As pairs are not part of the input, the amount of extra work in each resolution step is independent of the input. Hence we have

COROLLARY 4.10 (HOFMANN [1999]). *If* $\emptyset \vdash t^{\mathbf{L}(\mathbf{B})\multimap\mathbf{L}(\mathbf{B})}$, *then* t *denotes a function computable in polynomial time.*

Example 4.11 (INSERTION SORT). Let τ be a type equipped with a linear ordering. Assume that this ordering is represented by a term $\lhd_\tau$ of type $\tau \multimap \tau \multimap \mathbf{B}\otimes(\tau\otimes\tau)$, that is, $\lhd_\tau ts \to^* \otimes_{\mathbf{B},\tau\otimes\tau}\mathtt{tt}(\otimes_{\tau,\tau}ts)$, if t is "smaller" than s, and $\lhd_\tau ts \to^* \otimes_{\mathbf{B},\tau\otimes\tau}\mathtt{ff}(\otimes_{\tau,\tau}ts)$ otherwise.

Using this function, we can define a sort function of type $\tau \multimap \tau \multimap \tau\otimes\tau$ for two elements, that is, a function of two arguments that returns them in the correct order:

$$\lhd'_\tau = \lambda p_1^\tau, p_2^\tau.\ \lhd_\tau\ p_1p_2(\lambda y^{\mathbf{B}}, p^{\tau\otimes\tau}.\ p(\lambda p_1^\tau, p_2^\tau.y\,\langle\otimes_{\tau,\tau}p_1p_2\otimes_{\tau,\tau}p_2p_1\rangle)).$$

Now define a function of type $\mathbf{L}(\tau) \multimap \diamond \multimap \tau \multimap \mathbf{L}(\tau)$ that inserts an element at the correct position in a given sorted list. At each step we compare the argument with the first element of the list, put the smaller element at the beginning of the list, and insert the larger one in the rest of the list.

$$\begin{array}{l} \mathtt{insert} = \lambda l.l\{\lambda x_1^\diamond, y_1^\tau, p^{\diamond\multimap\tau\multimap\mathbf{L}(\tau)}, x_2^\diamond, y_2^\tau.\ \lhd'_\tau\ y_1y_2(\lambda z_1^\tau, z_2^\tau.\,\mathbf{cons}_\tau x_1z_1(px_2z_2))\} \\ \qquad\qquad (\lambda x^\diamond, y^\tau.\,\mathbf{cons}_\tau xy\,\mathbf{nil}_\tau). \end{array}$$

Then insertion sort is defined as usual:

$$\mathtt{sort} = \lambda l^\tau.l\{\lambda x^\diamond, y^\tau, l^{\mathbf{L}(\tau)}.\,\mathtt{insert}lxy\}\mathbf{nil}_\tau.$$

Using $\vartheta(\lhd'_\tau) = \vartheta(\lhd_\tau)$ and counting braces we get

$$\vartheta\,(\mathtt{sort}) = X \cdot \vartheta\,(\mathtt{insert}) + O(X) = X^2 \cdot \vartheta(\lhd'_\tau) + O(X^2) = X^2 \cdot \vartheta\,(\lhd_\tau) + O(X^2).$$

This reflects the fact that insertion sort is quadratic in the number of comparison operations.

By simple modifications of Hofmann's [1999, §4.3] proof we may conclude that many "natural orderings," for example, the normal ordering on binary coded natural numbers, *can* be defined in the given term system.

However, it should be noted that it is *not necessary* that every interesting ordering is definable in the given system. It would also be possible to just add a new symbol $\lhd_\tau$ with the conversion rules $\lhd_\tau ts \mapsto \otimes_{\mathbf{B},\tau\otimes\tau}\mathtt{tt}(\otimes_{\tau,\tau} ts)$, if t is "smaller" than s, and $\lhd_\tau ts \mapsto \otimes_{\mathbf{B},\tau\otimes\tau}\mathtt{ff}(\otimes_{\tau,\tau} ts)$ otherwise. With $|\lhd_\tau| := 4$ the above argument remains valid and shows that there are at most X^2 of the newly introduced conversions in a normalizing sequence. Therefore this theory can be used to calculate the number of calls to a subroutine, even if the subroutine itself is not definable in the given system, or not even polynomial time computable.

5. EXTENSIONS OF THE SYSTEM

The syntactical analysis of the system allows various extensions that we only sketch here, giving sufficient detail to reconstruct the proofs.

5.1 Full Polynomial Time

The system so far only contains non-size-increasing functions, and hence cannot contain all PTIME functions. New results of Hofmann [2001] show that indeed all non-size-increasing PTIME functions (and hence in particular all PTIME *predicates*) are already definable *within* the present system.

Here we shall briefly sketch an approach to obtain all PTIME *functions* in order to provide some insight into the way the restriction to non-size-increasing functions works. Its motivation was to avoid explosion of growth by iterating over already aggregated data-structures. Yet in the definition of PTIME, the only large data-structure of a Turing machine is its tape. Moreover, a Turing machine does not iterate over its tape but instead modifies it locally.

The central idea lies in the observation that size is represented by the number of free variables. Hence, we can add a type ι that allows closed terms for objects that are *semantically* of arbitrary size. On this type, we can then define functions that are *semantically* size-increasing, like the extension of a Turing tape, but are *from a syntactic point of view* non-size-increasing, in that they do not require an argument of type $\diamond$.

Iteration on this new type ι would lead beyond polynomial time, as the number of iterations that a loop (i.e., an $\mathbf{L}(\tau)$-elimination) unfolds to is no longer immediately related to the input (via the number of free variables).

The term system is extended by adding a new ground type ι and the following constants with types:

$$\begin{array}{llll} \circ & \iota & \mathbf{p} & \iota \multimap \iota \\ \mathfrak{s}_0 & \iota \multimap \iota & \mathbf{iszero} & \iota \multimap (\mathbf{B} \otimes \iota) \\ \mathfrak{s}_1 & \iota \multimap \iota & \mathbf{head} & \iota \multimap (\mathbf{B} \otimes \iota) \end{array}$$

$\mapsto$ is enriched by

$$\begin{array}{llll} \mathbf{p}\circ & \mapsto & \circ & \\ \mathbf{p}(\mathfrak{s}_i t^\iota) & \mapsto & t^\iota & \end{array} \qquad \begin{array}{llll} \mathbf{iszero}\circ & \mapsto & \otimes_{\mathbf{B},\iota} \mathtt{tt} \circ & \\ \mathbf{iszero}(\mathfrak{s}_i t^\iota) & \mapsto & \otimes_{\mathbf{B},\iota} \mathtt{ff}(\mathfrak{s}_i t^\iota) & \end{array} \qquad \begin{array}{llll} \mathbf{head}\circ & \mapsto & \otimes_{\mathbf{B},\iota} \mathtt{ff} \circ & \\ \mathbf{head}(\mathfrak{s}_0 t^\iota) & \mapsto & \otimes_{\mathbf{B},\iota} \mathtt{ff}(\mathfrak{s}_0 t^\iota) & \\ \mathbf{head}(\mathfrak{s}_1 t^\iota) & \mapsto & \otimes_{\mathbf{B},\iota} \mathtt{tt}(\mathfrak{s}_1 t^\iota). & \end{array}$$

The definitions of the relations $\Gamma \vdash t^\tau$ and $t \to t'$ remain unchanged. Proposition 2.5, and Lemmata 2.6, 2.8, 3.4, and 3.5 remain valid with almost identical proofs.

We call terms of the form $\mathfrak{s}_{i_1}(\ldots(\mathfrak{s}_{i_n}\circ))$ *short numerals*. As in Proposition 3.6 we show that every normal, almost closed term of type ι is a short numeral (and therefore closed).

The definition of the term length $|t|$ had the property that for all reductions except iteration the length decreased. To retain this property we define $|\mathbf{iszero}| = |\mathbf{head}| = 3$ and keep the rest of Definition 4.1. In particular, every constant different from **iszero** and **head** still has length 1. Then Lemmata 4.2 and 4.3 remain valid.

The definition of $\vartheta(t)$ is unchanged. Then (with identical proof) Lemma 4.7 holds and also the main Theorem 4.8. In particular, the extended system still consists of PTIME functions only.

To show that every PTIME function can be defined by a closed term of type $\mathbf{N} \multimap \iota$, we code the configuration of a Turing machine (with N states $\{S_0, \ldots, S_{N-1}\}$, working over the alphabet $\{0, 1\}$) with the symbols $i_0 \cdots i_k$ before and including the head and the symbols $j_0 \cdots j_{k'}$ followed by the nonvisited positions after the head and with current state S_m by

$$\otimes\big(\mathfrak{s}_{i_k}\big(\cdots\big(\mathfrak{s}_{i_0}\circ\big)\big)\big) \qquad \big(\otimes\big(\mathfrak{s}_{j_0}\big(\cdots\big(\mathfrak{s}_{j_{k'}}\circ\big)\big)\big) \qquad (\otimes t_1(\otimes\cdots(\otimes t_{n-1}t_n))))$$

with n the smallest number such that $N \le 2^n$ and each of the $\vec{t}$ being $\mathtt{tt}$ or $\mathtt{ff}$, so that $\vec{t}$ is the binary coding of m. The closed terms $\mathfrak{s}_0$ and $\mathfrak{s}_1$ extend the Turing tape where necessary, so we can define the one-step function of the Turing machine by sufficiently many case-distinctions. Iterating this one-step function polynomially many times (e.g., as done by Hofmann [1999, §4.3]) completes the (sketched) proof.

5.2 Trees

We sketch how our technique applies to the data type of binary labeled trees, studied also by Hofmann [2000a, §4.3]. Notice that the extension is not completely obvious, since iteration on trees involves *two* recursive calls. It turns

out that the number of free variables in a term still is a good measure for the number of unfoldings of an iteration.

The system is extended by

—a new type constructor $\mathbf{T}(\tau, \rho)$ for trees (with nodes labeled by elements of type τ and leaves labeled with elements of type ρ);

—new constants with their respective types

$$\begin{array}{ll} \mathbf{leaf}_{\tau,\rho} & \rho \multimap \mathbf{T}(\tau, \rho) \\ \mathbf{tree}_{\tau,\rho} & \diamond \multimap \tau \multimap \mathbf{T}(\tau, \rho) \multimap \mathbf{T}(\tau, \rho) \multimap \mathbf{T}(\tau, \rho); \end{array}$$

—a new term constructor $\{\cdot, \cdot\}$ for iteration on trees;

—a new typing rule

$$\frac{\Gamma \vdash t^{\mathbf{T}(\tau,\rho)} \quad \emptyset \vdash s^{\diamond \multimap \tau \multimap \sigma \multimap \sigma \multimap \sigma} \quad \emptyset \vdash r^{\rho \multimap \sigma}}{\Gamma \vdash (t\,\{s, r\})^{\sigma}}\ (\mathbf{T}(\tau, \rho)^{-}).$$

We inductively define the notion of a tree (with n nodes) by

—$\mathbf{leaf}_{\tau,\rho} t^{\rho}$ is a tree (with 0 nodes);

—if t_1 and t_2 are trees (with n_1 and n_2 nodes) then $\mathbf{tree}_{\tau,\rho} d^{\diamond} a^{\tau} t_1 t_2$ is also a tree (with $n_1 + n_2$ nodes).

The conversion relation $\mapsto$ is augmented by

$$\begin{array}{lll} \mathbf{leaf}_{\tau,\rho} t\,\{s, r\} & \mapsto & rt, \\ \mathbf{tree}_{\tau,\rho} d\, a\, t_1 t_2\,\{s, r\} & \mapsto & sda(t_1\,\{s, r\})(t_2\,\{s, r\}) \qquad t_1, t_2 \text{ trees.} \end{array}$$

We extend the definition of the length by $|\{s, r\}| := |r|$ and the definition of the polynomial bound by

$$\vartheta\,(\{s, r\}) := X \cdot \vartheta\,(s) + (X + 1) \cdot \vartheta\,(r) + X \cdot |s| + X \cdot |r|\,,$$

$$\vartheta\,(ts) \quad := \begin{cases} \vartheta(t) + X_n \vartheta(h) + X_n\,|h| & \text{if } t \text{ is a list with } n \text{ entries} \\ & \text{and } s \text{ is of the form } \{h\}, \\ \vartheta(t) + X_n \cdot \vartheta(s') + (X_n + 1) \cdot \vartheta(r') & \text{if } t \text{ is a tree with } n \text{ nodes} \\ +X_n \cdot |s'| + X_n \cdot |r'| & \text{and } s \text{ is of the form } \{s', r'\}, \\ \vartheta(t) + \vartheta(s) & \text{otherwise.} \end{cases}$$

Then the theory in Section 4 remains valid, with identical proofs. The new nontrivial subcase in Theorem 4.8 is

$$\mathbf{tree}_{\tau,\rho} d\, a\, t_1 t_2\,\{s, r\} \mapsto sda(t_1\,\{s, r\})(t_2\,\{s, r\}),$$

where t_1 and t_2 are trees with n_1 and n_2 nodes, respectively. Then by Lemma 4.3 and the linear typing discipline we know that $n_1 + n_2 + 1 \leq$

$|\mathrm{FV}(\mathbf{tree}_{\tau,\rho} d a t_1 t_2 \{s,r\})| \leq N$, and hence

$$
\begin{aligned}
&\vartheta(\mathbf{tree}_{\tau,\rho} d a t_1 t_2 \{s,r\})(N)\\
&+ |\mathbf{tree}_{\tau,\rho} d a t_1 t_2 \{s,r\}|\\
&= \vartheta(d)(N) + \vartheta(a)(N) + \vartheta(t_1)(N) + \vartheta(t_2)(N)\\
&+ (n_1+n_2+1)\vartheta(s)(N) + (n_1+n_2+1+1)\vartheta(r)(N)\\
&+ (n_1+n_2+1)\,|s| + (n_1+n_2+1)\,|r|\\
&+ 1 + |d| + |a| + |t_1| + |t_2| + |r|
\end{aligned}
$$

and

$$
\begin{aligned}
&\vartheta(s d a(t_1\{s,r\})(t_2\{s,r\}))(N)\\
&+ |s d a(t_1\{s,r\})(t_2\{s,r\})|\\
&= \vartheta(s)(N) + \vartheta(d)(N) + \vartheta(a)(N)\\
&+ \vartheta(t_1)(N) + n_1 \cdot \vartheta(s)(N) + (n_1+1)\cdot\vartheta(r)(N) + n_1\cdot|s| + n_1\cdot|r|\\
&+ \vartheta(t_2)(N) + n_2 \cdot \vartheta(s)(N) + (n_2+1)\cdot\vartheta(r)(N) + n_2\cdot|s| + n_2\cdot|r|\\
&+ \ |s| + |d| + |a| + |t_1| + |r| + |t_2| + |r|\,,
\end{aligned}
$$

which is strictly smaller.

It should be noted that, once iteration on labeled trees is available, "divide and conquer" recursion can be implemented: given a data type τ and starting from a "divide function"

$$\mathfrak{f}\colon \tau \multimap \mathbf{L}(\tau) \multimap \tau \otimes \mathbf{L}(\tau) \otimes \mathbf{L}(\tau)$$

we can construct a function $\tilde{\mathfrak{f}}\colon \mathbf{L}(\tau) \multimap \mathbf{T}(\tau, \mathbf{L}(\tau))$ with

$$
\begin{aligned}
\tilde{\mathfrak{f}}(\mathbf{nil}_\tau) &= \mathbf{leaf}_{\tau,\mathbf{L}(\tau)}\mathbf{nil}_\tau,\\
\tilde{\mathfrak{f}}(\mathbf{cons}_\tau d^\diamond a^\tau \ell) &= \mathbf{tree}_{\tau,\mathbf{L}(\tau)} d a' \big(\mathbf{leaf}_{\tau,\mathbf{L}(\tau)}\ell_1\big)\big(\mathbf{leaf}_{\tau,\mathbf{L}(\tau)}\ell_2\big) \quad \text{if } \mathfrak{f} a \ell = a' \otimes \ell_1 \otimes \ell_2.
\end{aligned}
$$

To do so we first construct via iteration over lists a function $\mathfrak{g}\colon \mathbf{L}(\tau) \multimap \mathbf{L}(\tau) \times \mathbf{T}(\tau, \mathbf{L}(\tau))$ with $\mathfrak{g}(\ell) = \langle \ell \tilde{\mathfrak{f}} \ell \rangle$. It should be noted that $\tilde{\mathfrak{f}}$ is defined by case analysis (iteration is only used to emulate the $\mathbf{nil}_\tau$/$\mathbf{cons}_\tau$ case distinction). Iterating a function that applies $\tilde{\mathfrak{f}}$ to (the label of) every leaf of a tree sufficiently (i.e., linearly) often, using algorithms similar to those described by Hofmann [1999, §4.3], we can define a function $\mathfrak{f}^D\colon \mathbf{L}(\tau) \multimap \mathbf{T}(\tau, \mathbf{L}(\tau))$ such that

$$
\begin{aligned}
\mathfrak{f}^D(\mathbf{nil}_\tau) &= \mathbf{leaf}_{\tau,\mathbf{L}(\tau)}(\mathbf{nil}_\tau),\\
\mathfrak{f}^D(\mathbf{cons}_\tau d^\diamond a^\tau \ell) &= \mathbf{tree}_{\tau,\mathbf{L}(\tau)} d a' \big(\mathfrak{f}^D \ell_1\big)\big(\mathfrak{f}^D \ell_2\big) \quad \text{if } \mathfrak{f} a \ell = a' \otimes \ell_1 \otimes \ell_2.
\end{aligned}
$$

A final iteration over the created tree with an appropriate "conquer function" and an appropriate initial value finishes the implementation of "divide and conquer" recursion in the present system.

Following these lines, we can, for example, specify `quicksort` by providing

—a divide function of type $\tau \multimap \mathbf{L}(\tau) \multimap \tau \otimes \mathbf{L}(\tau) \otimes \mathbf{L}(\tau)$ splitting a list into two sublists, one with the elements larger than the given one and the other with

smaller elements

$$\begin{aligned}\lambda x^{\tau}, l^{\mathbf{L}(\tau)}.\quad & l\{\lambda y_0^{\diamond}, y_1^{\tau}, p^{\tau\otimes\mathbf{L}(\tau)\otimes\tau}.\quad p\big(\lambda x^{\tau}, p^{\mathbf{L}(\tau)\otimes\mathbf{L}(\tau)}.\quad p\big(\lambda l', l''.\\ & \lhd_{\tau} x\, y_1\ \big(\lambda z^{\mathbf{B}}, p^{\tau\otimes\rho}.\qquad p\big(\lambda x^{\tau}, y_1^{\tau}.\\ & \qquad z\big\langle \otimes_{\tau,\mathbf{L}(\tau)\otimes\mathbf{L}(\tau)} x\big(\otimes_{\mathbf{L}(\tau),\mathbf{L}(\tau)}(\mathbf{cons}_{\tau}\, y_0\, y_1 l')l''\big),\\ & \qquad\quad \otimes_{\tau,\mathbf{L}(\tau)\otimes\mathbf{L}(\tau)} x\big(\otimes_{\mathbf{L}(\tau),\mathbf{L}(\tau)} l'(\mathbf{cons}_{\tau}\, y_0\, y_1 l'')\big)\big\rangle\big)\big)\big)\big)\}\\ & \otimes_{\tau,\mathbf{L}(\tau)\otimes\mathbf{L}(\tau)} x\big(\otimes_{\mathbf{L}(\tau),\mathbf{L}(\tau)}\mathbf{nil}_{\tau}\mathbf{nil}_{\tau}\big);\end{aligned}$$

—a conquer function of type $\diamond \multimap \tau \multimap \mathbf{L}(\tau) \multimap \mathbf{L}(\tau) \multimap \mathbf{L}(\tau)$ taking two sorted lists and joining them appropriately with the middle element

$$\lambda x^{\diamond}, y^{\tau}, l'^{\mathbf{L}(\tau)}, l''^{\mathbf{L}(\tau)}.\quad \mathrm{app} l'(\mathbf{cons}_{\tau} x\, y\, l''),$$

where app is the append function, defined as usual;

—an initial case for the empty list, $\mathbf{nil}_{\tau}$.

However, as in this *implementation* of "divide and conquer" recursion, for each unfolding step the whole (intermediate) tree structure has to be traversed, it is less efficient than a "native" variant of "divide and conquer," but, as shown, still in polynomial time.

REFERENCES

AEHLIG, K. AND SCHWICHTENBERG, H. 2000. A syntactical analysis of non-size-increasing polynomial time computation. In *Proceedings of the 15th Symposium on Logic in Computer Science* (LICS 2000). IEEE Computer Science Press, Los Alamitos, CA, 84–91.

BECKMANN, A. AND WEIERMANN, A. 2000. Characterizing the elementary recursive functions by a fragment of Gödel's *T*. *Arch. Math. Logic 39*, 7 (Oct.), 475–492.

BELLANTONI, S. AND COOK, S. 1992. A new recursion-theoretic characterization of the polytime functions. *Computat. Complex. 2*, 97–110.

BELLANTONI, S., NIGGL, K.-H., AND SCHWICHTENBERG, H. 2000. Higher type recursion, ramification and polynomial time. *Ann. Pure Appl. Logic 104*, 17–30.

CASEIRO, V.-H. 1997. Equations for defining poly-time functions. Ph.D. thesis, Department of Computer Science, University of Oslo, Oslon, Norway. Available at `www.ifi.uio.no/~ftp/publications/research-reports/`.

COOK, S. A. 1971. Characterizations of push-down machines in terms of time bounded computers. *Journal of the ACM 18*, 4–18.

GÖDEL, K. 1958. Über eine bisher noch nicht benützte Erweiterung des finiten Standpunkts. *Dialectica 12*, 280–287.

HOFMANN, M. 1998. Typed lambda calculi for polynomial-time computation. Habilitation thesis, Mathematisches Institut, TU Darmstadt, Germany. Available under `www.dcs.ed.ac.uk/home/mxh/habil.ps.gz`.

HOFMANN, M. 1999. Linear types and non-size-increasing polynomial time computation. In *Proceedings of the 14th Symposium on Logic in Computer Science* (LICS'99). IEEE Computer Science Press, Los Alamitos, CA, 464–473.

HOFMANN, M. 2000a. Linear types and non-size-increasing polynomial time computation. *Theoret. Comput. Sci.* To appear.

HOFMANN, M. 2000b. A type system for bounded space and functional in-place update. *Nordic J. Comput. 7*, 4 (Winter), 258–289.

HOFMANN, M. 2001. The strength of non size-increasing computation. In *Proceedings of the 3rd International Workshop on Implicit Computational Complexity* (ICC '01) Aarhus, Denmark, May 20–21, 2001), M. Hofmann, Ed. Number NS-01-3 in Notes Series. BRICS, Department of Computer Science, University of Aarhus, Aarhus, Denmark, 19–35.

JOACHIMSKI, F. AND MATTHES, R. 1999. Short proofs of normalisation for the simply-typed λ-calculus, permutative conversions and Gödel's *T*. *Arch. Math. Logic*, To appear.

JONES, N. D. 1997. *Computability and Complexity from a Programming Perspective*. MIT Press, Cambridge, MA.

JONES, N. D. 1999. LOGSPACE and PTIME characterized by programming languages. *Theoret. Comput. Sci. 228*, 1–2 (Oct.), 151–174.

JONES, N. D. 2001. The expressive power of higher-order types or, life without CONS. *J. Funct. Program. 11,* 1, 55–94.

LEIVANT, D. 1991. A foundational delineation of computational feasibility. In *Proceedings of the 6th Symposium on Logic in Computer Science* (LICS'91). IEEE Computer Science Press, Los Alamitos, CA, 2–11.

LEIVANT, D. 1999. Applicative control and computational complexity. In *Proceedings of the 13th International Workshop on Computer Science Logic* (CSL '99), J. Flum and M. Rodriguez-Artalejo, Eds. Lecture Notes in Computer Science, Vol. 1683, Springer Verlag, Berlin, Germany, 82–95.

LEIVANT, D. AND MARION, J.-Y. 1993. Lambda calculus characterization of poly-time. *Fundamenta Informaticae 19*, 1/2 (Sep./Oct.), 167–184.

SIMMONS, H. 1988. The realm of primitive recursion. *Arch. Math. Logic 27*, 2 (Oct.), 177–188.

Received November 2000; revised September 2001; accepted November 2001