A syntactical analysis of non-size-increasing polynomial time computation

Klaus Aehlig[*]   Helmut Schwichtenberg

Abstract

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.

1   Introduction

Recent research [3, 4, 9] has provided many characterizations of Ptime (the class of all functions computable in polynomial time) by means of appropriate restrictions of the terms in Gödel's T [7]. Consider the following definition of an exponentially growing function.

$$\texttt{double}([\,]) := [\,] \qquad\qquad \texttt{exp}([\,]) := [\texttt{tt}]$$
$$\texttt{double}([\,a \mid \ell\,]) := [\,a, a \mid \texttt{double}(\ell)\,] \qquad \texttt{exp}([\,a \mid \ell\,]) := \texttt{double}(\texttt{exp}(\ell))$$

Approaches based on predicative recursion [19, 3, 16] argue that the exponential growth in this example is due to the way double is called: the previous value $\exp(\ell)$ of the outer recursion is the recursive argument to the inner recursion.

Although such approaches can capture all *functions* computable in polynomial time, many natural *algorithms* are excluded, particularly if they involve nested recursions. Standard examples are sorting algorithms like insertion sort, which has a similar recursive structure as exp:

$$\texttt{insert}(a, [\,]) := [a]$$
$$\texttt{insert}(a, [\,b \mid \ell\,]) := [\,b' \mid \texttt{insert}(a', \ell)\,] \quad a', b' \text{ a permutation of } a, b \text{ with } b' \le a'$$
$$\texttt{sort}([\,]) := [\,]$$
$$\texttt{sort}([\,a \mid \ell\,]) := \texttt{insert}(a, \texttt{sort}(\ell))$$

Caseiro [5] 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 [10] 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 to define non-size-increasing `Ptime` functions. The basic idea is that if a function increases the size of its argument (like a successor function), then one has to "pay" for the increase by providing a "token", i.e., a term of type $\diamond$.

Hofmann [10] 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 more insight and yields an explicit construction of time bound polynomials. Although Hofmann's proof is constructive and thus also contains bounds, these are deeply hidden.

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 semantical size measure [10, §3.2] (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 [1] 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 $\vartheta\left(\cdot\right)$.

- 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 paper [10] is of course the starting point of ours. In the journal version [11] 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 [15], in a paper called "The expressive power of higher-order types or, life without CONS", directly relates programming languages and complexity theory. In a previous paper [14] (see also [13]) he had characterized the power of first-order read-only

programs in complexity terms; the results are extended [15] to arbitrary data orders and tail recursive programs. As Jones notes, the root for his `Ptime` result (for data order 0) is an early paper of Cook [6] on "two-way auxiliary pushdown automata". However, no characterization of `Ptime` in terms of recursion with unlimited data orders is obtained.

Jones work is certainly related to the present one: 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 [18] give a λ-calculus characterization of `Ptime`. The main novelty is 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). This leads to a nice characterization of `Ptime`. Leivant [17] gives particularly elegant proofs of characterizations of Jones [14] for `Ptime` and of Beckmann and Weiermann [2] for the elementary functions. He also treats a control-based subcalculus of the system in [4], and — more importantly — obtains a characterization of `Pspace` along the same lines. This is achieved by the notions of an "input driven" term, i.e., 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 [10], can be defined as $\mathbf{L}(\mathbf{B})$. The type $\diamond$ may be interpreted as a pointer to free memory [8]; Lemma 4.3 will show that no closed terms of type $\diamond$ exist. Canonical terms of datatypes 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 meaning of those types is different. Each component of a tensor product $\tau \otimes \rho$ can be used, whereas in the case of an ordinary pair one can choose only one component to be used. 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 should express that the variable x has type τ.

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

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

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

The idea for the $\{\cdot\}$ term construct is taken from Joachimski and Matthes [12]. 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 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 $\lambda \vec{x}^{\vec{\tau}}.\, t$ are defined as usual, so $\lambda x^\tau, y^\rho.\, t$ is an abbreviation for $\lambda x^\tau.(\lambda y^\rho.\, t)$. Terms that only differ in the naming of bound variables are identified.

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

$$
\begin{array}{ll}
\mathbf{tt} & \mathbf{B} \\
\mathbf{ff} & \mathbf{B} \\
\mathbf{nil}_\tau & \mathbf{L}(\tau) \\
\mathbf{cons}_\tau & \Diamond \multimap \tau \multimap \mathbf{L}(\tau) \multimap \mathbf{L}(\tau) \\
\otimes_{\tau,\rho} & \tau \multimap \rho \multimap \tau \otimes \rho
\end{array}
$$

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:

$$\overline{\Gamma, x^\tau \vdash x^\tau} \qquad\qquad\qquad\qquad\qquad\qquad\qquad (Var)$$

$$\frac{c \text{ of type } \tau}{\Gamma \vdash c^\tau} \qquad\qquad\qquad\qquad\qquad\qquad\qquad (Const)$$

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

$$\frac{\Gamma_1 \vdash t^{\tau \multimap \rho} \qquad \Gamma_2 \vdash s^\tau}{\Gamma_1, \Gamma_2 \vdash (ts)^\rho} \tag{$\multimap^-$}$$

$$\frac{\Gamma \vdash t^\tau \qquad \Gamma \vdash s^\rho}{\Gamma \vdash \langle t, s \rangle^{\tau \times \rho}} \tag{$\times^+$}$$

$$\frac{\Gamma \vdash t^{\tau \times \rho}}{\Gamma \vdash (t\mathbf{tt})^\tau} \tag{$\times_1^-$}$$

$$\frac{\Gamma \vdash t^{\tau \times \rho}}{\Gamma \vdash (t\mathbf{ff})^\rho} \tag{$\times_0^-$}$$

$$\frac{\Gamma_1 \vdash t^{\mathbf{B}} \qquad \Gamma_2 \vdash s^\tau \qquad \Gamma_2 \vdash r^\tau}{\Gamma_1, \Gamma_2 \vdash (t \langle s, r \rangle)^\tau} \tag{$\mathbf{B}^-$}$$

$$\frac{\Gamma_1 \vdash t^{\tau \otimes \rho} \qquad \Gamma_2, x^\tau, y^\rho \vdash s^\sigma}{\Gamma_1, \Gamma_2 \vdash (t(\lambda x^\tau, y^\rho.s))^\sigma} \tag{$\otimes^-$}$$

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

It is crucial that the step term in $(\mathbf{L}(\tau)^-)$ is closed. Otherwise it would not be possible to define reduction rules in such a way that typed terms reduce to typed terms (Lemma 3.5), 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, e.g., 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, i.e., 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.

---

[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 $\lambda x, y, z.zxy$.

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 $(-\!\circ^+)$ could as well have been written

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

requiring that the bound variable does not occur in the context afterwards. In our formulation it is easier to recognize weakening as a derived 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, e.g., if r, s are terms then so is $r\{s\}$, rather then 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 Joachimski and Matthes [12]. But in the present rather simple situation this seems to be an 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 $(-\!\circ^-)$, 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 $(-\!\circ^-)$, $(\times_0^-)$, $(\times_1^-)$, $(\mathbf{B}^-)$, $(\otimes^-)$ 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 $(-\!\circ^+)$ that $\Gamma_2 \vdash (\lambda x^\tau, y^\rho.s)^{\tau-\!\circ\rho-\!\circ\sigma}$. $\square$

---

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

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 $\cdot \vdash \cdot$ relation, we have

Lemma 2.8 (Head form). *Assume $\Gamma \vdash t^\tau$. 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 serious 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 correct with respect to the set-theoretic semantics (cf. Hofmann [11, §2.1]). In order to control the effects of recursion we allow reduction of an iteration only if the argument is already calculated, i.e., if the argument is a list.

Definition 3.1 (Lists). *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 the $d_1, \ldots, d_n$ stand for arbitrary terms of type $\diamond$.

It should be noted that we could also have required the $\vec{d}$ to be variables and get 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, i.e., contain free variables of types other than $\diamond$.

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

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

Although they look quite similar, the rules $\langle t, s \rangle \, \mathbf{tt} \mapsto t$ and $\mathbf{tt} \, \langle t, s \rangle \mapsto t$ actually have a 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 $\mathbf{tt}$. Also notice the different typings of the two rules.

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

$$\frac{t \mapsto t'}{t \to t'} \qquad \frac{t \to t'}{ts \to t's} \qquad \frac{s \to s'}{ts \to ts'}$$

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, i.e., 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 $(\multimap^-)$. Similar if $t = \langle t_1, t_2 \rangle$. $\qquad\square$

Lemma 3.5 (Subject reduction). *If $\Gamma \vdash t^\tau$ and $t \to t'$, then $\Gamma \vdash t'^\tau$.*

Proof. Induction on $t \to t'$ shows that only conversions have to be considered. The only non-trivial case is handled in Lemma 3.4. $\qquad\square$

Next we show that we have sufficiently many reduction rules, i.e., that normal terms have the expected form. As the size of a term is its number of free variables, we also have to consider non-closed terms. As $\diamond$ is the only base type without closed terms, we can restrict ourselves to *almost closed* terms, i.e., to 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, $\mathtt{tt}$ or $\mathtt{ff}$, 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 $\mathtt{tt}$ or $\mathtt{ff}$, $\mathbf{nil}_\tau$ or $\otimes_{\tau,\rho}$,

8

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 $\mathbf{cons}_\tau dar\vec{t}$ we apply the induction hypothesis to r yielding that t is of the form $\ell\vec{t}$ with a *list* ℓ. So if $\vec{t}$ would consist of more then 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 actually is 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, i.e., 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 chains

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 (since the λ disappears). The same is true for the reductions due to projections and the "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*, i.e., if we reduce $\mathbf{cons}_\tau d_1^\diamond r_1^\tau (\dots (\mathbf{cons}_\tau d_n^\diamond r_n^\tau \mathbf{nil}_\tau)) \{h\} t$ in $n+1$ steps to $h d_1^\diamond r_1^\tau (\dots (h d_n^\diamond r_n^\tau t))$ then the $\{\cdot\}$'s disappear! So by making them "sufficiently heavy" the total weight of the term reduces. As however the weight necessary for the $\{\cdot\}$-construct depends not only on the term within the braces, but also on the length n of the numeral we iterate on, we cannot assign this term a fixed number; instead we assign a polynomial(ly bound function) in Definition 4.5. But what to plug in? Noting that we have n terms $d_1^\diamond, \dots, d_n^\diamond$ of type $\diamond$ and remembering the idea that terms of type $\diamond$ should contain free variables (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 $\vec{d}$ are applied to one another, 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 $\mathbf{cons}_\tau d_1^\diamond r_1^\tau (\dots (\mathbf{cons}_\tau d_n^\diamond r_n^\tau \mathbf{nil}_\tau)) \{h\} t$ 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).

---

[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

So we use three different measures for terms: the number of free variables, which corresponds to the "size function" in Hofmann's work [10]; 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).

For instance, if t is $t_1 t_2$, then 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 obvious. □

Lemma 4.3. *Every term of type* $\diamond$ *contains a free variable, i.e., 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 $t[t_1/x]t_2 \ldots t_n$. Similarly for $\langle r, s \rangle \vec{t}$, where we apply the induction hypothesis to $rt_2 \ldots t_n$ or $st_2 \ldots t_n$ depending whether t_1 is tt or ff (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 $t_2 \ldots t_n$. In the cases $\mathsf{tt}\vec{t}$ and $\mathsf{ff}\vec{t}$, by typing, t_1 has to be of the form $\langle r, s \rangle$ and we can apply the induction hypothesis to (say) $rt_2 \ldots t_n$. □

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", i.e., a resource that is not yet present.

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, i.e., $X_n(m) = \min\{n, m\}$. $\mathbb{N}$ 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 $\vartheta(t) \in \mathbb{N}^{\mathrm{poly}}$ by recursion on t.

$$\vartheta(x) := \vartheta(c) := 0$$

$$\vartheta(ts) := \begin{cases} \vartheta(t) + X_n \cdot \vartheta(h) + X_n \cdot |h| & \text{if } t \text{ is a list with } n \text{ entries} \\ & \text{and } s \text{ is of the form } \{h\} \\ \vartheta(t) + \vartheta(s) & \text{otherwise} \end{cases}$$

$$\vartheta(\lambda x^\tau . t) := \vartheta(t)$$
$$\vartheta(\langle t, s\rangle) := \sup_{\preccurlyeq}\{\vartheta(t), \vartheta(s)\}$$
$$\vartheta(\{h\}) := X \cdot \vartheta(h) + X \cdot |h|$$

We write $\vartheta(\vec{t})$ 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 non-trivial 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'. We distinguish cases whether r is a list or not.

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 from the fact that t is typed we know that at most one of the terms r and r' contains the variable x free. 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. $\qquad\square$

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

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

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 $r \to r'$. We distinguish whether r is a list or not.

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

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

12

Subcase r is not a list. Then

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

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

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

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 $n+1 \leq |\mathrm{FV}\,(\mathbf{cons}_\tau d^\diamond a\ell\,\{s\}\,r)| \leq N$. Therefore we have

$$
\begin{aligned}
&&&\vartheta\,(\mathbf{cons}_\tau d^\diamond a\ell\,\{s\}\,r)\,(N) \\
&+ && |\mathbf{cons}_\tau d^\diamond a\ell\,\{s\}\,r| \\[6pt]
={} && &\vartheta\,(d)\,(N)+\vartheta\,(a)\,(N)+\vartheta\,(\ell)\,(N) \\
&&&+\,(n+1)\vartheta\,(s)\,(N)+(n+1)\,|s|+\vartheta\,(r)\,(N) \\
&+ && 1+|d|+|a|+|\ell|+|r|
\end{aligned}
$$

and on the other hand

$$
\begin{aligned}
&&&\vartheta\,(sd^\diamond a(\ell\,\{s\}\,r))\,(N) \\
&+ && |sd^\diamond a(\ell\,\{s\}\,r)| \\[6pt]
={} && &\vartheta\,(s)\,(N)+\vartheta\,(d)\,(N)+\vartheta\,(a)\,(N)+\vartheta\,(\ell)\,(N) \\
&&&+\,n\cdot\vartheta\,(s)\,(N)+n\cdot|s|+\vartheta\,(r)\,(N) \\
&+ && |s|+|d|+|a|+|\ell|+|r|
\end{aligned}
$$

which obviously is strictly smaller.

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

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

The remaining subcases are similar to the last one. $\qquad\square$

Corollary 4.9 (Hofmann, 1999). *If $\emptyset \vdash t^{\mathbf{L}(\mathbf{B}) \multimap \cdots \multimap \mathbf{L}(\mathbf{B}) \multimap \mathbf{L}(\mathbf{B})}$, then t denotes a function computable in polynomial many steps.*

Proof. We only treat unary functions. 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. $\qquad\square$

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 rather 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.

Example 4.10 (Insertion sort). Let τ be a type equipped with a linear ordering. Assume that this ordering is represented by the term $\lhd_\tau$ of type $\tau \multimap \tau \multimap \mathbf{B} \otimes (\tau \otimes \tau)$, i.e., $\lhd_\tau ts \to^* \otimes_{\mathbf{B}, \tau \otimes \tau} \mathbf{tt}(\otimes_{\tau,\tau} ts)$, if t is "smaller" than s, and $\lhd_\tau ts \to^* \otimes_{\mathbf{B}, \tau \otimes \tau} \mathbf{ff}(\otimes_{\tau,\tau} ts)$ otherwise.

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

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

Now we can immediately define a function of type $\mathbf{L}(\tau) \multimap \diamond \multimap \tau \multimap \mathbf{L}(\tau)$ inserting an element at the correct position in a given sorted list. At each step we compare the given element 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.

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

Then insertion sort is defined as usual:

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

Counting braces, as $\vartheta(\lhd'_\tau) = \vartheta(\lhd_\tau)$, 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 (the proof of) [10, §4.3] we may conclude that many "natural orderings", e.g., 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 $\lhd_\tau ts \mapsto \otimes_{\mathbf{B},\tau\otimes\tau}\mathbf{tt}(\otimes_{\tau,\tau}ts)$, if t is "smaller" than s, and $\lhd_\tau ts \mapsto \otimes_{\mathbf{B},\tau\otimes\tau}\mathbf{ff}(\otimes_{\tau,\tau}ts)$ otherwise. With $|\lhd_\tau| := 4$ the above theory 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 which 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 [8] show that indeed all these functions (and hence in particular all `Ptime`-predicates) are already *within* the present system.

Here we shall briefly sketch an approach to obtain all `Ptime` functions, that might give 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 the Turing-tape, but are *from a technical 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, adding the following constants with their respective types

$\circ$	ι	**p**	$\iota \multimap \iota$
$\mathfrak{s}_0$	$\iota \multimap \iota$	**iszero**	$\iota \multimap (\mathbf{B} \otimes \iota)$
$\mathfrak{s}_1$	$\iota \multimap \iota$	**head**	$\iota \multimap (\mathbf{B} \otimes \iota)$

15

$\mapsto$ is enriched by

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

$$
\begin{array}{lll}
\mathbf{head}\circ & \mapsto & \otimes_{\mathbf{B},\iota}\mathtt{ff}\circ \\
\mathbf{head}(\mathfrak{s}_0 t^\iota) & \mapsto & \otimes_{\mathbf{B},\iota}\mathtt{ff}(\mathfrak{s}_0 t^\iota) \\
\mathbf{head}(\mathfrak{s}_1 t^\iota) & \mapsto & \otimes_{\mathbf{B},\iota}\mathtt{tt}(\mathfrak{s}_1 t^\iota)
\end{array}
$$

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 $\mathfrak{s}_{i_1}(\dots(\mathfrak{s}_{i_n}\circ))$ *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 to $\mathbf{iszero}$ and $\mathbf{head}$ still has length 1. Then Lemmata 4.2 and 4.3 remain valid.

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

To show that every $\mathtt{Ptime}$ 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 $\{S_0, \dots, S_{N-1}\}$, working over the alphabet $\{0, 1\}$) with the symbols $i_0 \dots i_k$ before and including the head and the symbols $j_0 \dots j_{k'}$ followed by the non-visited positions after the head and with current state S_m by

$$
\begin{aligned}
&\otimes(\mathfrak{s}_{i_k}(\dots(\mathfrak{s}_{i_0}\circ))) \\
&(\otimes(\mathfrak{s}_{j_0}(\dots(\mathfrak{s}_{j_{k'}}\circ)))) \\
&(\otimes t_1(\otimes \dots(\otimes t_{n-1} t_n))))
\end{aligned}
$$

with n being the smallest number such that $N \leq 2^n$ and each of the $\vec{t}$ being $\mathtt{tt}$ or $\mathtt{ff}$ such that this 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 shown by Hofmann [10, §4.3]) completes the (sketched) proof.

5.2 Trees

We sketch how our technique applies to the more complex data type of binary labeled trees; this is to be hoped for, as Hofmann's original method is capable of dealing with them [11, §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 with 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 $\{\cdot, \cdot\}$

- a new typing rule

$$
\frac{\Gamma \vdash t^{\mathbf{T}(\tau,\rho)} \qquad \emptyset \vdash s^{\Diamond \multimap \tau \multimap \circ \sigma \multimap \circ \sigma \multimap \circ \sigma} \qquad \emptyset \vdash r^{\rho \multimap \circ \sigma}}{\Gamma \vdash (t\,\{s, r\})^{\sigma}} \qquad (\mathbf{T}(\tau, \rho)^-)
$$

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

$$
\begin{array}{ll}
\mathbf{leaf}_{\tau,\rho} t\,\{s, r\} \mapsto rt & \\
\mathbf{tree}_{\tau,\rho} dat_1 t_2\,\{s, r\} \mapsto sda(t_1\,\{s, r\})(t_2\,\{s, r\}) & \quad t_1, t_2 \text{ trees}
\end{array}
$$

We extend the definition of the length by

$$
|\{s, r\}| := |r|
$$

and the definition of the polynomial bound by

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

$$
\vartheta\left(ts\right) \quad := \begin{cases}
\vartheta\left(t\right) + X_n \vartheta\left(h\right) + X_n\,|h| & \text{if } t \text{ is a list with } n \text{ entries} \\
& \text{and } s \text{ is of the form } \{h\} \\[1.2em]
\vartheta\left(t\right) + X_n \cdot \vartheta\left(s'\right) + (X_n + 1) \cdot \vartheta\left(r'\right) & \\
+ X_n \cdot |s'| + X_n \cdot |r'| & \text{if } t \text{ is a tree with } n \text{ nodes} \\
& \text{and } s \text{ is of the form } \{s', r'\} \\[1.2em]
\vartheta\left(t\right) + \vartheta\left(s\right) & \text{otherwise}
\end{cases}
$$

Then the theory in section 4 remains valid, with identical proofs. The new non-trivial subcase in Theorem 4.8 is

$$
\mathbf{tree}_{\tau,\rho} dat_1 t_2\,\{s, r\} \mapsto sda(t_1\,\{s, r\})(t_2\,\{s, r\})
$$

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 $n_1+n_2+1 \leq |\text{FV}\,(\mathbf{tree}_{\tau,\rho}dat_1t_2\,\{s,r\})| \leq N$, and hence

$$
\begin{aligned}
& \vartheta\,(\mathbf{tree}_{\tau,\rho}dat_1t_2\,\{s,r\})\,(N) \\
+\;& |\mathbf{tree}_{\tau,\rho}dat_1t_2\,\{s,r\}| \\
=\;& \vartheta\,(d)\,(N) + \vartheta\,(a)\,(N) + \vartheta\,(t_1)\,(N) + \vartheta\,(t_2)\,(N)+ \\
+\;& (n_1+n_2+1)\vartheta\,(s)\,(N) + (n_1+n_2+1+1)\vartheta\,(r)\,(N) \\
+\;& (n_1+n_2+1)\,|s| + (n_1+n_2+1)\,|r| \\
+\;& 1 + |d| + |a| + |t_1| + |t_2| + |r|
\end{aligned}
$$

and

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

which is strictly smaller.

It should be noted that, once iteration on labeled trees is available, "divide and conquer" recursion can be implemented: 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

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

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 $\mathfrak{g}(\ell) = \left\langle \ell, \tilde{\mathfrak{f}}\ell \right\rangle$. Iterating a function that applies $\tilde{\mathfrak{f}}$ to (the label of) every leaf of a tree sufficiently (i.e., linearly) often [10, §4.3] we can define a function $\mathfrak{f}^D\colon \mathbf{L}(\tau) \multimap \mathbf{T}(\tau, \mathbf{L}(\tau))$ such that

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

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 one with smaller

elements

$$\lambda x^\tau, l^{\mathbf{L}(\tau)} \quad l\{\lambda y_0^\diamond, y_1^\tau, p^{\tau\otimes\mathbf{L}(\tau)\otimes\tau} \quad p(\lambda x^\tau, p^{\mathbf{L}(\tau)\otimes\mathbf{L}(\tau)} \quad p(\lambda l', l''$$
$$\lhd_\tau xy_1 \; (\lambda z^{\mathbf{B}}, p^{\tau\otimes\rho} \quad p(\lambda x^\tau, y_1^\tau$$
$$z\langle \; \otimes_{\tau,\mathbf{L}(\tau)\otimes\mathbf{L}(\tau)}x(\otimes_{\mathbf{L}(\tau),\mathbf{L}(\tau)}(\mathbf{cons}_\tau y_0 y_1 l')l''),$$
$$\otimes_{\tau,\mathbf{L}(\tau)\otimes\mathbf{L}(\tau)}x(\otimes_{\mathbf{L}(\tau),\mathbf{L}(\tau)}l'(\mathbf{cons}_\tau y_0 y_1 l''))\rangle)))\}$$
$$\otimes_{\tau,\mathbf{L}(\tau)\otimes\mathbf{L}(\tau)}x(\otimes_{\mathbf{L}(\tau),\mathbf{L}(\tau)}\mathbf{nil}_\tau\mathbf{nil}_\tau)$$

- 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 in the correct way with the middle element

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

  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

[1] Klaus Aehlig and Helmut Schwichtenberg. A syntactical analysis of non-size-increasing polynomial time computation. In *Proceedings 15'th Symposium on Logic in Computer Science (LICS 2000)*, pages 84–91, 2000.

[2] Arnold Beckmann and Andreas Weiermann. Characterizing the elementary recursive functions by a fragment of Gödel's T. *Archive for Mathematical Logic*, 39(7):475–492, 2000.

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

[4] Stephen Bellantoni, Karl-Heinz Niggl, and Helmut Schwichtenberg. Higher type recursion, ramification and polynomial time. *Annals of Pure and Applied Logic*, 104:17–30, 2000.

[5] Vuokko-Helena Caseiro. *Equations for Defining Poly-Time Functions*. PhD thesis, Department of Computer Science, University of Oslo, 1997. Available at www.ifi.uio.no/ ftp/publications/research-reports/.

[6] Stephen A. Cook. Characterizations of push-down machines in terms of time bounded computers. *Journal of the Association for Computing Machinery*, 18:4–18, 1971.

[7] Kurt Gödel. Über eine bisher noch nicht benützte Erweiterung des finiten Standpunkts. *Dialectica*, 12:280–287, 1958.

[8] Martin Hofmann. A type system for bounded space and functional in-place update. To appear: Nordic Journal of Programming. An extended abstract has appeared in 'Programming Languages and Systems' (Proc. ESOP 2000), G. Smolka, ed., Springer LNCS, 2000.

[9] Martin Hofmann. 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, 1998.

[10] Martin Hofmann. Linear types and non-size-increasing polynomial time computation. In *Proceedings 14'th Symposium on Logic in Computer Science (LICS'99)*, pages 464–473, 1999.

[11] Martin Hofmann. Linear types and non-size-increasing polynomial time computation. To appear: Theoretical Computer Science, 2000.

[12] Felix Joachimski and Ralph Matthes. Short proofs of normalisation for the simply-typed λ-calculus, permutative conversions and Gödel's T. To appear: Archive for Mathematical Logic, 1999.

[13] Neil D. Jones. *Computability and Complexity from a Programming Perspective*. MIT Press, 1997.

[14] Neil D. Jones. LOGSPACE and PTIME characterized by programming languages. *Theoretical Computer Science*, 228:151–174, 1999.

[15] Neil D. Jones. The expressive power of higher-order types or, life without CONS. *J. Functional Programming*, 11(1):55–94, 2001.

[16] Daniel Leivant. A foundational delineation of computational feasibility. In *Proceedings 6'th Symposium on Logic in Computer Science (LICS'91)*, 1991.

[17] Daniel Leivant. Applicative control and computational complexity. In J. Flum and M. Rodriguez-Artalejo, editors, *Computer Science Logic (Proceedings of the Thirteenth CSL Conference)*, number 1683 in LNCS, pages 82–95. Springer Verlag, Berlin, Heidelberg, New York, 1999.

[18] Daniel Leivant and Jean-Yves Marion. Lambda calculus characterization of polytime. *Fundamenta Informaticae*, 19:167–184, 1993.

[19] Harold Simmons. The realm of primitive recursion. *Archive for Mathematical Logic*, 27:177–188, 1988.