Composition Is Almost (but Not Quite) as Good as S-1-1*

Yves Marcoux
EBSI, Université de Montréal,
C.P. 6128, succ. Centre-ville, Montréal (Québec), Canada H3C 3J7
E-mail: yves.marcoux@umontreal.ca

Abstract

We establish a polynomial upper bound on the time complexity of an s-1-1 function in programming systems with a linear time composition function. This improves the double-exponential upper bound of Machtey and Young [9], the only previously known upper bound, and invalidates the belief expressed twice in the literature [8, 19] that it could not be significantly improved. We then show our upper bound to be *tight* by exhibiting a family of acceptable programming systems for which it is optimal. We deduce several bounds on the time complexity of composition functions, s-1-1 functions, and various other semantic transformations of programs, in programming systems with a linear or polynomial time composition function. In particular, we show the existence of an acceptable programming system with a quadratic time composition function, but no subexponential time s-1-1 function. In one interpretation from [11], this last result states that the complexity of a composition function for an effective programming system does not give an upper bound on the complexity of the "task of programming" in that programming system. In contrast, results by Royer [19] indicate that this task is essentially no more complex than computing an s-1-1 function.

1 Introduction

1.1 Basic definitions

Let N be the set of nonnegative integers, and $\mathcal{P}art\mathcal{R}ec$ the class of unary partial recursive functions from N to N. A *programming system* [9] is a (total) surjec-

*This is the correct version of the article with the same title published in *Theoretical Computer Science*, vol. 120, 1993, pp. 169-195. A preliminary version appeared in the *Proceedings of the Fourth Annual Conference on Structure in Complexity Theory*, IEEE Computer Society Press (1989), under the title *Composition is Almost as Good as S-1-1.*

tion from N onto $\mathcal{PartRec}$. If ψ is a programming system and i an integer, ψ_i denotes $\psi(i)$, and we say i *computes* or *is a (ψ-)program for* ψ_i. Two programs are said to be *equivalent* iff they compute the same partial function.

The symbol ϕ denotes a fixed programming system induced by some standard computational model (e.g., the Turing machine [9, 3]), and some fixed encoding of the computing devices of this model. We say a programming system ψ is *effective* iff it corresponds to an interpretable programming language, i.e., iff there exists a Turing machine M such that for all i and x, $\psi_i(x) = M(i, x)$. Intuitively, M takes a ψ-program and a piece of data, and runs the program against that piece of data. We say ψ is *programmable* iff there exists a recursive function t such that for all i, $\psi_{t(i)} = \phi_i$. For any such t, we say ψ is programmable *via* t. Intuitively, t translates programs from a standard formalism (ϕ) into ψ-programs. We say ψ is *acceptable* iff it is both effective and programmable. Effectiveness and programmability are nontrivial and independent properties of programming systems [16, 15]. The programming system ϕ is trivially acceptable.

Acceptable programming systems are abstractions of exactly those algorithm description formalisms that give rise to essentially the same theory of computability as the classical models of computation, such as the Turing machine, the random access machine, and the lambda-calculus (computing over Church's numerals). By Church's thesis, and assuming implicit coding of programs and data, one can view acceptable programming systems as abstractions of *general purpose programming languages*, i.e., formalisms for describing computer actions that have full computational power. In fact, with their semantics appropriately "twisted" to fit the programming system framework, all known general purpose programming languages (APL, PASCAL, etc.) are easily shown to correspond to acceptable programming systems. For this reason, we allow ourselves in this paper to discuss acceptable programming systems in the more intuitive terms of programming languages. In particular, "acceptable programming system" and "programming language" are used interchangeably, although we use the latter mainly in the more intuitive passages.

A *composition function for* (or *instance of* composition *in*) a programming system ψ is a 2-ary function c such that for all programs i and j, $\psi_{c(i,j)} = \psi_i \circ \psi_j = \lambda z.\psi_i(\psi_j(z))$. An *s-1-1 function for* (or *instance of* s-1-1 *in*) ψ is a 2-ary function s such that for all program i and integer x, $\psi_{s(i,x)} = \lambda y.\psi_i(\langle x, y\rangle)$, where $\langle\cdot,\cdot\rangle$ is any fixed pairing function. Since $\mathcal{PartRec}$ is closed under composition and fixing the first parameter, every programming system has *some* composition function and *some* s-1-1 function. Not all programming systems, however, have *recursive* composition and s-1-1 functions (also called *effective* instances of composition and s-1-1). It is well known that an effective programming system is acceptable iff it has either a recursive s-1-1 function or a recursive composition function, iff it has both [16, 9]. Many other characterizations of acceptable programming systems can be found in [19] and [11].

Composition and s-1-1 functions can be seen as implementations of two *pro-*

gramming techniques. From this perspective, they represent actions performed by a programmer in the task of programming. For instance, a composition function corresponds to the programming technique which consists in functionally composing two programs together. This programming technique is quite naturally referred to as "program composition". An s-1-1 function is said to realize the programming technique of "program specialization", because the output of an s-1-1 function is a specialized version of the input program, operating on a fixed value of one of the parameters (the reader may here want to review the formal definition of an s-1-1 function).

1.2 Significance of composition and s-1-1

Program composition is conceptually very close to the idea of "modular programming"; it is thus readily seen to be a natural and important programming technique. Program specialization, on the other hand, may seem more remote from day-to-day life. However, there are two areas in which it can be of direct interest.

First, program specialization can be seen as a generalization of textual substitution, which is but one way of achieving the former. Thus, program specialization could serve as a basis for the study (or even implementation) of other mechanisms of programming languages based on textual substitution, such as parameter binding and Currying [22].

Second, a special case of program specialization, known as *partial evaluation*, is actually quite widely used as an automatic program generation technique. Partial evaluation can be described as program specialization with a concern for the efficiency of output programs. Research in partial evaluation consists in developing "clever" program specializers (i.e., "clever" s-1-1 functions) that return computationally efficient programs. Partial evaluation has proven to be very useful in such diverse fields as computer graphics, database query processing, compiling, compiler generation, and scientific computing [4, 5].

Thus, both composition and s-1-1 correspond to interesting and fundamental programming techniques.

1.3 Overview of this paper

As mentioned in Subsection 1.1, all acceptable programming systems have recursive composition and s-1-1 functions. *A priori*, however, it could be the case that some have *computationally efficient* composition and s-1-1 functions, while others do not.[1] Since composition and s-1-1 functions represent important aspects of the task of programming in a programming system, this would certainly be a criterion for preferring some acceptable programming systems over others.

[1] Note that we are now concerned with the complexity of the composition and s-1-1 functions *themselves*, not, as in partial evaluation, of the programs they return. Unless otherwise stated, "complexity" is to be understood as "Turing machine time complexity".

While all common general purpose programming languages have efficient composition and s-1-1 functions (the reader may check his or her favorite language), constructions such as the one we give in the proof of Theorem 3.1 below show that some acceptable programming systems have only arbitrarily complex composition and s-1-1 functions. Thus, all acceptable programming systems are indeed not equal with respect to the complexities of composition and s-1-1 functions.

However, a few statements can be made about the *relative* complexities of composition and s-1-1 functions *within the same acceptable programming system*. One first such statement is based on an analysis of a classical construction in recursion theory for obtaining a composition function from an s-1-1 function. Machtey, Winklmann, and Young have observed in [8] that the composition function obtained by this construction is essentially no more complex than the s-1-1 function used as a starting point. Thus, for instance, if an acceptable programming system has a linear time s-1-1 function, then it also has a linear time composition function. This indicates that possessing an efficient s-1-1 function is a desirable property for an acceptable programming system, because it guarantees not only an efficient implementation of program specialization, but also one of program composition.

Another statement that can be made is that if an acceptable programming system possesses a linear time composition function, then it also possesses a double-exponential time s-1-1 function. Based on this statement alone, we cannot deduce that possessing an efficient composition function is as desirable a property for acceptable programming systems as possessing an efficient s-1-1 function. It is thus interesting to see if the statement can be strengthened.

The statement is based on the analysis of a construction by Machtey and Young of an s-1-1 function from a composition function [9, Theorem 3.1.2]. Further analysis shows that the obtained s-1-1 function sometimes *requires* double-exponential time to compute. (Both the construction and the analysis are given in Subsection 2.1.) Thus, the statement cannot be strengthened, unless we use a different construction.

In two different instances in the literature [8, 19], the belief has been expressed that the Machtey and Young construction could not be improved significantly. This belief was in both cases based on the intuition that there might exist a programming system for which all s-1-1 functions would be much more complex than some (intuitively, the "easiest") composition function. In [8, p. 53], Machtey, Winklmann, and Young write that it would not be counterintuitive if, in some programming system, the task of computing any s-1-1 function turned out to be significantly more complex than that of computing some composition function, to the point of making the Machtey and Young construction close to optimal. In [19, pp. 39, 152], Royer is more precise and suggests that one could construct an acceptable programming system with a linear time composition function, but no subexponential time s-1-1 function.

Our first main result, Theorem 2.2, invalidates both this belief and the intu-

ition on which it is based. This theorem states that if an acceptable programming system possesses a linear time composition function, then it also possesses a *polynomial* time s-1-1 function, where the degree of the polynomial is essentially the base 2 logarithm of the linear time constant of the composition function. Our proof of Theorem 2.2 is based on an improvement of Machtey and Young's construction.

Our second main result, Theorem 3.1, states that the polynomial upper bound of Theorem 2.2 is optimal for a large family of acceptable programming systems. To prove this, we construct, for each rational[2] constant $q \geq 1$, an acceptable programming system with a composition function computable in time linear with constant q, and for which the upper bound of Theorem 2.2 is optimal.

As a corollary to the proof of Theorem 3.1, we show the existence of an acceptable programming system with a quadratic time composition function, but *no subexponential time s-1-1 function* (Corollary 3.5). Under the common hypothesis that computations requiring more than polynomial time are not feasible practically, this programming system has a practically computable composition function but no practically computable s-1-1 function. In contrast, remember that an acceptable programming system with a polynomial time s-1-1 function is guaranteed to possess a polynomial time composition function.

In the light of these results, we can conclude that having an efficient s-1-1 function is indeed a better property for an acceptable programming system, than having an efficient composition function, although the difference is not as dramatic as foreseen by Machtey, Winklmann, and Young and Royer. Hence, the title of this paper.

The rest of this paper is organized as follows. In Subsection 1.4, we present a setting in which the questions treated in this paper can be expressed in terms of the *task of programming* in a programming system, rather than just realizing program composition and specialization. In Subsection 1.5, we present an alternative interpretation of these questions, which is implicitly used by Machtey, Winklmann, and Young in [8]. Related work is presented in Subsection 1.6, while Subsection 1.7 is devoted to the definitions, conventions, and terminology not presented elsewhere in the paper. Sections 2 and 3 contain, respectively, our upper and lower bound theorems and corollaries. An absolute lower bound, valid for any composition function in any programming system, is presented in Section 4. Finally, in Section 5, we make some concluding remarks and mention open questions.

1.4 A wider setting

We now present a setting in which both the questions treated in this paper and our main results can be interpreted in terms of the *task of programming* in

[2]See footnote 3.

a programming system. This setting is nearly explicit in a number of works, among which [13, 14, 15], and is presented in detail in [11]. We take it as a basis for this paper. Here is a quick and intuitive overview of this setting.

Define $\Phi : \mathcal{PartRec} \times \mathcal{PartRec} \to \mathcal{PartRec}$ such that for all α and β, $\Phi(\alpha, \beta) = \alpha \circ \beta$. We can say that Φ represents the "semantic change" that composition functions perform on their input programs. Similarly, $\Psi : \mathcal{PartRec} \times N \to \mathcal{PartRec}$ satisfying $\Psi(\alpha, x) = \lambda z.\alpha(\langle x, z \rangle)$ for all α and x, represents the "semantic change" performed by s-1-1 functions. A composition function for a programming system ψ is a transformation of programs that *realizes* Φ in ψ, in that it is a function c such that for all i and j, $\psi_{c(i,j)} = \Phi(\psi_i, \psi_j)$. Similarly, an s-1-1 function for ψ is a transformation of programs *and data* that realizes Ψ in ψ, in that it is a function s such that for all p and x, $\psi_{s(p,x)} = \Psi(\psi_p, x)$.

The notion of semantic change has been formally defined in [11, Definition 2.9] (though there labelled *semantic relation*). For the purposes of this paper, it will be sufficient for the reader to consider a semantic change as a mapping of program semantics and/or data to program semantics, in the manner of Φ and Ψ above. Sans-serif names are used to denote semantic changes. The names composition and s-1-1 denote the semantic changes corresponding to composition and s-1-1 respectively. (For the purposes of this paper, the reader may thus think of composition and s-1-1 as identical to the mappings Φ and Ψ above.)

Now, in the same way as composition and s-1-1 functions realize the semantic changes composition and s-1-1, so can other transformations of programs (and/or data) realize other semantic changes. If sc is a semantic change and the function f realizes sc in a programming system ψ, then we say that f is an *instance* of sc *in* ψ. (This is why the expressions *instances of* composition and s-1-1 were presented as alternative names for composition and s-1-1 functions in Subsection 1.1.) If f is recursive, we say it is an *effective instance* of sc *in* ψ.

Several frameworks for expressing classes of semantic changes have been introduced in the literature. Examples of such frameworks are Riccardi's *control structure* [13] and our own *enumeration-verifiable relation* [11]. All these frameworks allow the expression of composition and s-1-1, but they also typically allow the expression of much more complex semantic changes, such as finding a fixed point of a partial function [19, Definition 1.4.1.5] or finding an inverse of a partial function (appropriately defined if the partial function is not one-one and onto N; see [11]).

Now, in the same way as composition and s-1-1 functions can be seen as implementations of programming techniques, so can instances of other semantic changes be seen as implementations of other programming techniques. For instance, if fix is the semantic change corresponding to finding a fixed point of a partial function, then an instance of fix in a programming system ψ can be seen as an implementation in ψ of the technique of programming by taking fixed points.

Some classes of semantic changes are so vast that they can arguably represent

most, if not all, of the programming techniques a programmer might ever want to use. Examples of such classes are Royer's *control structures with a trivial predicate* [19], and our own *enumeration-verifiable relations with an LC predicate* [11]. Because these classes are so encompassing, we can intuitively interpret complexity bounds on the instances of the semantic changes they contain, in terms of the *task of programming* in a programming system: if a programming system possesses an efficient instance of all the semantic changes in the class, then we say the task of programming in that programming system is *practically feasible*; otherwise, we say that task is *not* practically feasible.

This is our motivation for stating our results in terms of control structures with a trivial predicate. For example, Corollary 2.4, which states that "every effective programming system with a linear time instance of composition has a polynomial time instance of any control structure with a trivial predicate", conveys the interpretation that any acceptable programming system that has a linear time composition function is guaranteed to have a practically feasible programming task.

Royer has shown that if an effective programming system has a linear (respectively, polynomial) time instance of s-1-1, then it has a linear (respectively, polynomial) time instance of all control structures with a trivial predicate [19]. Thus, essentially, the complexity of an s-1-1 function gives an upper bound on the complexity of the task of programming in an acceptable programming system.

The intuition, expressed in [8] and [19], that there might exist a programming system for which all s-1-1 functions would be much more complex than some composition function, would imply that even a linear time composition function does *not* guarantee a practically feasible programming task. This, of course, is refuted by our Corollary 2.4. However, our lower bound results (Theorem 3.1 and Corollary 3.5) show that the complexity of a composition function does *not*, contrary to s-1-1, give an upper bound on the task of programming in an acceptable programming system. In this setting, the title of the paper takes on a whole new dimension.

Note that all our results expressed in terms of control structures with a trivial predicate are also valid for enumeration-verifiable relations with an LC predicate, a strictly larger class of semantic changes [11].

The formal definitions of semantic change, control structure with a trivial predicate, and enumeration-verifiable relation with an LC predicate are rather lengthy, and since we do not use them directly, we omit them. The interested reader is referred to [19] and [11].

1.5 Another interpretation

As mentioned above in Subsection 1.1, the possession of either a recursive composition function or a recursive s-1-1 function characterizes acceptable program-

ming systems among the effective ones. But what is the most "natural" characterization?

Machtey, Winklmann, and Young [8], in line with their intuition that, in some programming systems, computing a composition function might be strictly easier than computing any s-1-1 function, suggest that the possession of a recursive composition function is a "simpler" property than the possession of an s-1-1 function, and thus, that the most natural definition of an acceptable programming system is an effective programming system with a recursive composition function.

From this perspective, our results confirm that possessing a recursive composition function is indeed a "simpler" property than possessing an s-1-1 function (even though the extent by which it is simpler is not as great as expected by Machtey, Winklmann, and Young), and that, in this sense, the most natural definition of an acceptable programming system is an effective programming system with a recursive composition function.

1.6 Related work

Riccardi [15] showed that in the context of programmable (and not necessarily acceptable) programming systems, composition is strictly more *expressive* than s-1-1, in that an effective instance of it guarantees an effective instance of strictly more control structures than an effective instance of s-1-1 does. Many other control structures are also studied. In [14], the expressiveness of several control structures is compared in the context of effective (and not necessarily acceptable) programming systems. In these two references, no consideration is given to the complexity of the instances.

In addition to the study of s-1-1 versus control structures with a trivial predicate, one finds in [19] many results on the complexity interrelationship of different control structures in the context of effective programming systems. In particular, s-1-1 is studied in relationship to *padding type* control structures, as well as classes of control structures properly containing the class of control structures with a trivial predicate. Some of these results are improved in [11].

In [2], Hartmanis and Baker study the relationship between the complexities of s-1-1 functions and of *translations* of programs from other programming systems, especially isomorphic translations, in the same programming system. In [1], Hartmanis studies the complexity of isomorphic translations between programming systems in relationship to the complexity of padding functions in these same programming systems.

In [8], Machtey, Winklmann, and Young compare the complexities of implementing various *programming properties* in the same programming system, including paddability and the self-referential property of satisfying Rogers' Fixed-point Theorem [18].

The studies in [14], [15] and [19] put a great deal of emphasis on self-referential properties, especially the control structure KRT (for *Kleene Recur-*

sion Theorem), corresponding to Kleene's original form of the Strong Recursion Theorem [18].

There is a rich Russian and Eastern European literature on the *theory of numberings*; see [7] for a survey.

The term *s-1-1* comes from the special case $m{=}n{=}1$ of Kleene's *s-m-n* theorem [18], where the *s* stands for "substitution".

The expression *programming system* is from [9], the adjective *effective* (for programming systems) is from [19], *programmable* is from [13], *acceptable* probably originates from [17]. Programming systems were introduced in [16], and were called there, as well as in [13, 14, 15, 18, 19], *numberings of the partial recursive functions* or, for short, *numberings*. The expression *indexing of the partial recursive functions* is often encountered, and *Gödel numbering* is used in [8]. The adjective *effective* is replaced by *semi-effective* in [16], by *executable* in [13, 14, 15], and by *universal* in [9, 8]. Acceptable programming systems are called *Gödel numberings* in [16, 23, 2, 1] (note the discrepancy with [8]). They are also referred to simply as *programming systems* in [23].

1.7 Further definitions and conventions

In general, our implicit universe of discourse is N. Thus, any symbol denotes a (nonnegative) integer, unless otherwise stated; however, by default, q and r denote real numbers. The symbols ψ and ϕ always denote programming systems. We denote the base 2 logarithm function by "lg". We use lambda (λ) notation for defining partial functions [18].

For any n, we denote by $|n|$ the *length* of n, i.e., the minimum number of bits required to express n in binary. We note that $|n| = \lceil \lg(n+1) \rceil$ (in particular, $|0| = 0$).

The computational model underlying our discussion is the deterministic multi-tape Turing machine taking integers in binary representation as inputs. When expressing a running time as a function of the length of the input, we use n to denote that length. For instance, if we say a function is computable in time $O(n^2)$, we mean it is computable in time quadratic in the length of the input. By *exponential time*, we mean "time $O(2^{p(n)})$ for some polynomial p". We often shorten "linear (polynomial, etc.) time computable" to simply "linear (polynomial, etc.) time".

We use the pairing function from [20, 19, 21], which we denote by $\langle \cdot, \cdot \rangle$. By convention, for all $i > 2$, $\langle x_1, x_2, \ldots, x_i \rangle$ denotes $\langle x_1, \langle x_2, \ldots, x_i \rangle \rangle$. This pairing function has many convenient properties, among which the following: it is strictly increasing in both arguments; it and its two associated projection functions are computable in linear time; for all a and b, $2 \cdot \max(|a|, |b|) - 1 \leq |\langle a, b \rangle| \leq 2 \cdot \max(|a|, |b|)$, with equality on the left if $|a| < |b|$ and on the right otherwise; for all $x > 1$ and for all y, $\langle x, y \rangle > x$ and $\langle y, x \rangle > x$. We may occasionally use these properties without explicit mention. Note that we rely on *some* pairing function being linear time computable only in Proposition 2.6.

Any pairing function whatsoever could be used in the definition of an s-1-1 function, and all our results would still hold.

A problem that is very seldom dealt with in complexity theory is that of deciding what is the "length of the input" for functions of more than one argument. The reason for this is that, by using any "reasonable" pairing function (like the one we use here), the length of a pair is always kept within a constant factor of the length of the longer component. Hence, the difference made by using, say, the sum of the lengths of the arguments instead of the length of the *paired* arguments, would simply be a constant factor somewhere in the expression obtained for the complexity of the function or the running time of an algorithm for computing it.

In most cases, a constant factor is of no concern. Here, however, we are at times interested by the *exact* value of some linear time constant, and we must pay attention to all multiplicative factors that show up in our analyses. In particular, it turns out that what the "length of many arguments" really is, makes a difference. Among the most natural candidates, let us mention the following:

(a) $|\langle p_1, p_2, \ldots, p_k \rangle|$,

(b) $\max_{1 \leq i \leq k} (|p_i|)$, and

(c) $\sum_{i=1}^{k} |p_i|$, which we denote by $|p_1, p_2, \ldots, p_k|$,

where k is the arity of the function and p_i its i-th argument.

In this paper, we use (c), which seems the most reasonable to us. The use of (b) would change both bounds to $n^{\lg q}$ $(q > 2)$ or $n \log n$ $(q = 2)$, whereas using (a) would change them to $n^{2+\lg q}$ and allow us to use an injective composition function in our proof of Theorem 3.1.

2 The Upper Bound

2.1 Machtey and Young's construction

Before we state and prove our main upper bound theorem, we give for reference Machtey and Young's construction of an s-1-1 function from a composition function [9, Theorem 3.1.2], and briefly argue that it can yield an s-1-1 function that requires double-exponential time to compute, even if the composition function is computable in linear time. Note that Machtey and Young's result was originally stated for effective programming systems only; however, Machtey and Young [10] and Riccardi [13, 15] noted that the construction is also applicable to noneffective programming systems.

Theorem 2.1 (Machtey and Young, 1978) *Every programming system that has a recursive composition function also has a recursive s-1-1 function.*

Proof. **(Machtey and Young's construction)** Suppose ψ is a programming system and c a recursive composition function for ψ. Let q_0 and q_1 be ψ-programs for $\lambda z.\langle 0, z\rangle$ and $\lambda\langle y, z\rangle.\langle y+1, z\rangle$ respectively (remember that ψ is onto $\mathcal{P}art\mathcal{R}ec$; thus, such programs exist).

Let function h be defined as follows:

$$\begin{array}{lcl} h(0) & \stackrel{\mathrm{d}}{=} & q_0, \\ h(x+1) & \stackrel{\mathrm{d}}{=} & c(q_1, h(x)). \end{array}$$

For all x, we say a ψ-program is an "x-inserting program" iff it computes the function $\lambda z.\langle x, z\rangle$. It is easy to verify that for all x, $h(x)$ is an x-inserting program.

Now, define $s(p, x) \stackrel{\mathrm{d}}{=} c(p, h(x))$. Then, for all p and x,

$$\psi(s(p, x)) = \psi_p \circ \lambda z.\langle x, z\rangle = \lambda z.\psi_p(\langle x, z\rangle).$$

Thus, s is an s-1-1 function. Since c is recursive and since the definitions of h and s directly give algorithms to compute them, s is recursive. □

We now argue that the Machtey and Young (MY) construction can yield an s-1-1 function that requires double-exponential time to compute, even if the composition function used as a starting point is computable in linear time.

Let ψ, q_0, c, h, and s be as in the MY construction. Now, suppose $q_0 > 0$ and c is computable in linear time but also satisfies $|c(p, q)| \geq c_0 \cdot |p, q|$ for some constant $c_0 \geq 4$ and for all p and q (the proof of Theorem 3.1 below gives examples of such ψ and c). Then, it is immediate by the definition of h that for all x,

$$|h(x)| \geq c_0^x \geq 2^{2^{|x|}}.$$

By the definition of s, we thus have

$$|s(p, x)| \geq 2^{2^{|p,x|/2}}$$

as soon as $p \leq x$, for all p and x. Hence, s clearly requires double-exponential time to compute, infinitely often.

2.2 Our construction

Our improvement of Machtey and Young's construction is based on two simple ideas for building an "x-inserting program". These ideas are presented in the proof of the next theorem, which constitutes our main upper bound result.

Theorem 2.2 *Suppose ψ is a programming system with an instance of* composition *computable everywhere in time $qn + k$ for some constants $q \geq 1$ and k. If $q > 1$, then ψ has an instance of* s-1-1 *computable in time $O(n^{1+\lg q})$. If $q = 1$, then ψ has an instance of* s-1-1 *computable in time $O(n \log n)$.*

Proof. We present only the case $q > 1$. The case $q = 1$ is proven similarly.

Suppose ψ is a programming system and c is an instance of composition in ψ computable in time $qn+k$ for some constants $q > 1$ and k. It will be convenient to assume, without loss of generality, that $k \geq 1$. We describe an algorithm which, using c, computes an instance of s-1-1 in ψ and runs in time $O(n^{1+\lg q})$. The inputs to the algorithm are p, a ψ-program, and x, an integer. We call s the function computed by the algorithm.

We now give an intuitive outline of how the proof proceeds.

As in the MY construction, on input (p, x), we first build an x-inserting program, then we compose that program to the right of p, and return the result as a value for $s(p, x)$. Like Machtey and Young, we build the x-inserting program by composing copies of a finite number of fixed "base programs" (q_0 and q_1 in their construction) together. Our construction, however, differs in the choice of base programs, and on how we compose them. Informally speaking, each of these two improvements gets rid of one level of exponential.

First improvement The x-inserting program obtained in the MY construction inserts x in front of its argument by first inserting a 0 (which is done by q_0), and then incrementing that 0 by 1, x times (which is done by the x copies of q_1 used in building the x-inserting program). In contrast, we make our x-inserting program insert x bit by bit. To achieve this, we use three base programs, p_0 to p_2, satisfying

$$\begin{array}{lcl} \psi(p_0) & = & \lambda z.\langle 0, z\rangle, \\ \psi(p_1) & = & \lambda\langle y, z\rangle.\langle 2y, z\rangle, \text{ and} \\ \psi(p_2) & = & \lambda\langle y, z\rangle.\langle 2y+1, z\rangle. \end{array} \tag{1}$$

One way to get an x-inserting program using these base programs would be to redefine the function h from the MY construction as follows:

$$\begin{array}{lcl} h(0) & \stackrel{\mathrm{d}}{=} & p_0, \\ h(2x) & \stackrel{\mathrm{d}}{=} & c(p_1, h(x)), \\ h(2x+1) & \stackrel{\mathrm{d}}{=} & c(p_2, h(x)). \end{array}$$

Intuitively, we start with p_0 and compose to the left of it either p_1 or p_2, for each bit of x, depending on the value of the bit. The reader can verify that $h(x)$ is indeed an x-inserting program for all x.

Now since c is computable in linear time, it is easy to deduce from the definition of h that $|h(x)|$ will be at most exponential in $|x|$. Thus, we already have an improvement over the MY construction. However, if c is such that $|c(p, q)| \geq c_0 \cdot |p, q|$ for some sufficiently large constant c_0 and for all p and q (as in the discussion following the presentation of the MY construction in Subsection 2.1), then it is easy to show that $|h(x)|$ will also be at *least* exponential in $|x|$.

Thus, although we get rid of one level of exponential by making the x-inserting program build x bit by bit, we need to do more if we want to achieve polynomial time. This brings us to our second improvement.

Second improvement In the above definitions of h, the base programs that make up the x-inserting program (copies of q_0 and q_1 in the MY construction, and of p_0 to p_2 in ours) are composed "sequentially", that is, the composition function is always called with an argument of the form "(new base program , result of composing everything else so far)". We call this a purely sequential application pattern of the composition function. This application pattern is a natural by-product of the concise recursive definitions we have used for h; however, it is certainly not the only possible one, since function composition is associative. Indeed, our second improvement consists essentially in modifying the last definition of h to make it use a binary tree application pattern of the composition function (divide-and-conquer strategy) instead of a sequential one.

A very informal description of the new algorithm for h is as follows ($x_1, \ldots, x_{|x|}$ denote the successive bits of x, x_1 being the most significant one).

Input: x
Algorithm for h**:**

1. Build a vector v of programs, containing the $|x|$ programs $p_{(1+x_i)}$, $1 \leq i \leq |x|$.
2. Append p_0 to v.
3. **while** v has more than one element, **do**
4. $w \leftarrow$ the empty vector,
5. Using c, compose the programs in v two by two, appending the results to w as they are obtained,
6. $v \leftarrow w$.
7. **end while**
8. Return the only element of v.

□ **Algorithm**

Clearly, because function composition is associative, the programs returned by this version of h are equivalent to those returned by the previous version. Thus, $h(x)$ is an x-inserting program for all x.

We can immediately see that $|h(x)|$ will now be polynomial in $|x|$. Indeed, at each iteration of the **while** loop, the total length of (the programs in) v can only increase by a constant factor dependant on the running time of c, say c_0. Now, clearly, the initial total length of v is bounded by a constant multiple of $|x|$, say $c_1 \cdot |x|$ (for the moment, we will ignore the case $|x| = 0$). Since the

number of iterations in the **while** loop is essentially $\lg(|x|)$, and since $h(x)$ is simply v on exit from the **while** loop, we have for all x,

$$|h(x)| \leq c_0^{\lg(|x|)} \cdot c_1 \cdot |x| = c_1 \cdot |x|^{1+\lg c_0}.$$

The detailed proof below will show that this idea can indeed be used to build a polynomial time s-1-1 function.

It may seem odd that a divide-and-conquer strategy should pay off in performing a series of associative operations. Indeed, the same number of composition operations must be performed, whether they are performed with a sequential application pattern or a binary tree one. Note, however, that a composition function performs a "semantically" associative operation, but not one that is necessarily "textually" associative: if i, j, and k are programs, then $c(i, c(j, k))$ and $c(c(i, j), k)$, though equivalent programs, may very well be two *different* programs.

We now give more precisely, yet somewhat informally, our algorithm for s. Details are presented for implementation on a multi-tape Turing machine.

Input: p, a ψ-program and x, an integer
Algorithm for s:
{We use three "base" ψ-programs, p_0 to p_2, satisfying (1), and a subroutine for the function c which we assume runs in time $qn + k$.}

1. **Initialization phase:** Successively for each bit b of x (starting with the most significant bit), we write the program p_{b+1} on a work tape; then we write p_0. (Adjacent programs are separated by some special symbol.) Let us denote by n_0 the number of programs we write on the work tape during this phase (clearly, $n_0 = |x| + 1$), and by $\pi_{0,j}$ the j-th of these programs ($1 \leq j \leq n_0$).
2. **Iterative phase:** During this phase, the programs on the work tape are composed two by two (using our subroutine for c), repeatedly, until a single program is obtained. During each iteration, the work tape produced as output in the preceding iteration (or in the initialization phase on the first iteration) is used as input to this iteration. (It is clear that two work tapes suffice, serving alternatively as input and output tape, no matter how many iterations take place.)

   Suppose for the moment that b iterations take place, and let i satisfy $1 \leq i \leq b$. Let us denote by n_i the number of programs we write on the output tape during the i-th iteration, and by $\pi_{i,j}$ the j-th of these programs ($1 \leq j \leq n_i$). The exact processing performed during the i-th iteration is as follows. Successively, for each j satisfying $1 \leq j \leq \lfloor n_{i-1}/2 \rfloor$, we compute $c(\pi_{i-1,2j-1}, \pi_{i-1,2j})$, which we write on the output tape

and which thus constitutes program $\pi_{i,j}$. If n_{i-1} is even, the iteration is complete, otherwise, we copy $\pi_{i-1,n_{i-1}}$ at the end of the output tape, where it becomes known as $\pi_{i,\lceil n_{i-1}/2 \rceil}$.

The iterative phase is terminated when an output tape is produced that has only one program on it. In other words, b is the least integer such that $n_b = 1$. We can designate the single program written to the output tape during the last (b-th) iteration by $\pi_{b,1}$.

3. **Termination phase:** The output of the algorithm, $c(p, \pi_{b,1})$, is computed and written on the output tape.

□ **Algorithm**

We should now point out that the head of the output tape must be repositioned at the end of the initialization phase and of each iteration in the iterative phase; this will have to be considered in estimating the running time of the algorithm.

For the analysis of the algorithm, let p and x be fixed, let b be the number of iterations that occur in the iterative phase, and suppose i satisfies $1 \leq i \leq b$. Also, let n_i and $\pi_{i,j}$ (for i satisfying $0 \leq i \leq b$ and j satisfying $1 \leq j \leq n_i$) be as in the description of the algorithm. Since there is only one iterative phase, the term "iteration" will unambiguously designate an iteration in that phase.

It is immediately seen that $n_i = \lceil n_{i-1}/2 \rceil$ and, hence, that $b = \lceil \lg n_0 \rceil$. Thus, our algorithm computes a total function. It is easily shown, by induction on $|x|$, that

$$\psi(\pi_{0,1}) \circ \psi(\pi_{0,2}) \circ \cdots \circ \psi(\pi_{0,n_0}) = \lambda z.\langle x, z \rangle.$$

Also observe that, by associativity of function composition and by the fact that c is a composition function for ψ,

$$\psi(\pi_{i,1}) \circ \psi(\pi_{i,2}) \circ \cdots \circ \psi(\pi_{i,n_i}) = \psi(\pi_{i-1,1}) \circ \psi(\pi_{i-1,2}) \circ \cdots \circ \psi(\pi_{i-1,n_{i-1}}).$$

Thus, we have

$$\psi(\pi_{i,1}) \circ \psi(\pi_{i,2}) \circ \cdots \circ \psi(\pi_{i,n_i}) = \lambda z.\langle x, z \rangle.$$

In particular, $\pi_{b,1}$ (the single program produced during the last iteration), is a ψ-program for $\lambda z.\langle x, z \rangle$. Clearly, then, our algorithm computes an s-1-1 function for ψ.

Respectively denote by T_{init}, T_{loop}, T_{term}, $T_{loop,i}$, and T the execution times of the initialization, iterative, and termination phases, of the i-th iteration, and of the whole algorithm. Also, for i satisfying $0 \leq i \leq b$, let

$$l_i = \sum_{j=1}^{n_i} |\pi_{i,j}|.$$

Clearly, $l_0 \leq T_{init} \leq c_1 n_0$ for some constant c_1.

The following observation follows directly from our hypotheses on c and on our subroutine to compute it.

Observation 2.3 For all a and b, $|c(a,b)| \leq q(|a| + |b|) + k$, and computing $c(a,b)$ in the course of our algorithm can be done in time $c_2(q(|a| + |b|) + k)$ for some constant c_2.

We claim that $l_i \leq q l_{i-1} + n_i k$ and that $T_{loop,i} \leq c_3(q l_{i-1} + n_i k)$ for some constant c_3. Indeed, the i-th iteration consists of "running through c" (at most) all programs $\pi_{i-1,j}$, for j satisfying $1 \leq j \leq n_{i-1}$, and this is done by (at most) n_i applications of c. (The last program is simply copied if n_{i-1} is odd.) Thus, our claim follows from Observation 2.3. (Note that the head of the output tape must be repositioned after each iteration and that adjacent programs on both the input and output tapes are separated by a special symbol; however, our claim does hold, partly because we have taken $k \geq 1$.)

Next, we claim that $n_i \leq 2^{1-i} n_0$. Indeed, remember that $n_i = \lceil n_{i-1}/2 \rceil$. If n_0 is a power of 2, then it is clear that $n_i = 2^{-i} n_0$. If, on the other hand, n_0 is *not* a power of 2, observe that, since $\lambda z.\lceil z/2 \rceil$ is nondecreasing, n_i will certainly be no greater than it would be, were n_0 replaced by the next greater power of 2. Formally, this translates into $n_i \leq 2^{-i} 2^{\lceil \lg n_0 \rceil}$. The last factor being less than $2n_0$, our claim is verified.

From the above two claims, it is easily shown (by induction on i) that $l_i \leq q^i l_0 + 2kn_0 \cdot \sum_{j=1}^{i} 2^{-j} q^{i-j}$. Since the summation is less than q^i, and because $l_0 \leq c_1 n_0$, we obtain

$$l_i \leq c_4 n_0 q^i, \tag{2}$$

where $c_4 = c_1 + 2k$. Similarly, we get $T_{loop,i} \leq c_3 c_4 n_0 q^i$.

Hence, we now have

$$T_{loop} = \sum_{i=1}^{b} T_{loop,i} \leq c_3 c_4 n_0 \cdot \sum_{i=1}^{b} q^i.$$

Using the familiar geometric identity, the last summation becomes

$$\frac{q^{b+1} - q}{q - 1} < \frac{q^2}{q-1} q^{\lg n_0} = \frac{q^2}{q-1} n_0^{\lg q}.$$

Thus, we get $T_{loop} \leq c_5 n_0^{1+\lg q}$, where $c_5 = c_3 c_4 q^2/(q-1)$.

To bound T_{term}, we first use Observation 2.3 to bound it above by $c_2(q(|p| + |\pi_{b,1}|) + k)$. Then, remembering that $|\pi_{b,1}|$ is simply another name for l_b, we use inequality (2), above, to get

$$T_{term} \leq c_2 q \cdot |p| + c_2 c_4 q^2 n_0^{1+\lg q} + c_2 k.$$

Keeping in mind that $n_0 \geq 1$ and $\lg q > 0$, while adding up T_{init}, T_{loop}, and T_{term}, we get

$$T \leq c_6 \cdot |p| + c_6 n_0^{1+\lg q},$$

where $c_6 = c_1 + c_5 + c_2 q + c_2 c_4 q^2 + c_2 k$.

Now, $n_0 = |x| + 1$, and both $|x|$ and $|p|$ are $\leq |p, x|$. Hence, $n_0 \leq 2 \cdot |p, x|$ as soon as $|p, x| > 0$. Thus, excluding the case $|p, x| = 0$ (i.e., $p = x = 0$), we have

$$T \leq c_6(2q + 1) \cdot |p, x|^{1+\lg q}.$$

□ **Theorem 2.2**

2.3 Corollaries

Royer showed that an effective programming system with a polynomial time instance of s-1-1 has a polynomial time instance of any control structure with a trivial predicate [19, Theorem 1.4.3.9]. The following corollary is therefore immediate.

Corollary 2.4 *Every effective programming system with a linear time instance of* composition *has a polynomial time instance of any control structure with a trivial predicate.*

The construction in our proof of Theorem 2.2 is applicable to arbitrarily complex instances of composition, but a truly general result seems hard to express without the definition of an *ad hoc* operation on classes of functions. In the case of a polynomial time instance of composition, however, we have the following corollary. (Remember that for us, "exponential time" is "time $O(2^{p(n)})$ for some polynomial p".)

Corollary 2.5 *Every programming system with a polynomial time instance of* composition *has an exponential time instance of* s-1-1*.*

Proof. Similar to the proof of the theorem. A much coarser analysis suffices. □

Theorem 1.4.3.9 in [19] also states that an effective programming system with an exponential time instance of s-1-1 has an elementary recursive instance of any control structure with a trivial predicate. We now improve this upper bound to exponential time.

Suppose $m > 1$ and ψ is a programming system. An instance of s-m-1 in ψ is an $(m + 1)$-ary function t such that for all $p, x_1, \ldots, x_m$,

$$\psi_{t(p,x_1,\ldots,x_m)} = \lambda z.\psi_p(\langle x_1, \ldots, x_m, z\rangle).$$

In the same way that s-1-1 corresponds intuitively to fixing the first parameter in a program, s-m-1 corresponds to fixing the first m parameters.

A crucial step in Royer's proof of his Theorem 1.4.3.9 is a lemma (Lemma 1.4.3.10) in which he constructs effective instances of s-m-1, for all $m > 1$, in effective programming systems with an effective instance of s-1-1. The upper bound obtained for any control structure with a trivial predicate depends essentially on the complexity of these instances.

Suppose s is an effective instance of s-1-1 in an effective programming system ψ. Royer's way of constructing an instance of s-m-1 in ψ, for any $m > 1$, is to iterate s, m times. Thus, with s exponential time computable, the general upper bound for all m is "elementary recursive". The proof of the next proposition uses a more efficient construction.

Proposition 2.6 *Every effective programming system with an exponential time instance of* s-1-1 *has an exponential time instance of* s-m-1 *for all* $m \geq 1$.

Proof. Suppose $m \geq 1$. Let ψ be an effective programming system, and p_0 a fixed ψ-program for $\lambda\langle\langle p, x_1, \ldots, x_m\rangle, z\rangle.\psi_p(\langle x_1, \ldots, x_m, z\rangle)$. Such a program exists by the effectiveness of ψ. Then, t defined as follows can be verified to be an instance of s-m-1 in ψ.

$$t(p, x_1, \ldots, x_m) \stackrel{\mathrm{d}}{=} s(p_0, \langle p, x_1, \ldots, x_m\rangle)$$

Also, since $\langle\cdot,\cdot\rangle$ is computable in linear time and m and p_0 are fixed, if s is exponential time computable, then so is t. □

Note that the preceding proof can be made to work even if the pairing function used in the definition of an s-1-1 function is *not* our linear time pairing function. Suffice it to replace the *outermost* occurrence of $\langle\cdot,\cdot\rangle$, in the argument of the λ-expression defining the function computed by p_0, with the pairing function used to define an s-1-1 function. It is easily verified that this has no influence on the complexity of t.

By using Proposition 2.6 instead of Royer's Lemma 1.4.3.10 for proving his Theorem 1.4.3.9, we obtain the result that any effective programming system with an exponential time instance of s-1-1 has an exponential time instance of any control structure with a trivial predicate. This last result, together with Corollary 2.5, allows us to immediately deduce the following corollary.

Corollary 2.7 *Every effective programming system with a polynomial time instance of* composition *has an exponential time instance of any control structure with a trivial predicate.*

In general, we can obtain better upper bounds by using the construction in our proof of Proposition 2.6 instead of the one in Royer's proof of his Lemma 1.4.3.10. As an example, for instances of control structures with a

trivial predicate in effective programming systems with a polynomial time instance of s-1-1, we get polynomial time upper bounds of degrees smaller than those of the corresponding polynomial time upper bounds guaranteed by Royer's Theorem 1.4.3.9. A similar improvement is obtained for Corollary 2.4.

Royer's construction of instances of s-m-1 (for all $m > 1$) is the best we know of for arbitrary programming systems with an effective instance of s-1-1 but *without* an effective instance of composition (such programming systems are known to exist [15]). However, if an effective instance of composition is available, then we can sometimes exploit an idea similar to that in the proof of Proposition 2.6. For instance, we can show that any programming system with a polynomial time instance of composition has an exponential time instance of s-m-1 (for all $m > 1$). (*Proof sketch.* On input $(p, x_1, \ldots, x_m)$, compose to the right of p a fixed program that "splits" its first parameter into m different ones. Then, apply the exponential time instance of s-1-1 from Corollary 2.5 to the resulting program and the *pair* $\langle x_1, \ldots, x_m \rangle$. This can be verified to compute an instance of s-m-1 and run in exponential time.)

3 The Lower Bound

We shall now show that the construction of an s-1-1 function given in our proof of Theorem 2.2 is optimal in a wide range of *acceptable* programming systems with a linear time composition function.

Theorem 3.1 *There exists an acceptable programming system with an instance of* composition *computable in time $n + k_0$ for some small k_0, but for which computing any instance of* s-1-1 *requires time $\Omega(n \log n)$ infinitely often. For all rational $q > 1$, there exists an acceptable programming system with an instance of* composition *computable in time $qn + k_0$ for some small fixed k_0, but for which computing any instance of* s-1-1 *requires time $\Omega(n^{1+\lg q})$ infinitely often.*[3]

Proof. We present only the case $q > 1$. The case $q = 1$ is proven similarly.

Let a rational $q > 1$ be given. We shall construct an acceptable programming system ψ that satisfies the conditions of the theorem. The exposition of the proof is easier if we consider ψ-programs to be strings of symbols instead of integers. Thus, let ψ-programs be finite strings over $\{[,\ ],\ *,\ \mathtt{t},\ \mathtt{u},\ \mathtt{v}\}$. With an appropriate encoding of strings and very minor additions to our argument, the proof is applicable directly to integer programs. The k_0 in the statement of the theorem need not be more than 4 times the maximum number of bits used to encode a string symbol.

[3]Our proof is valid for any real $q \geq 1$ such that $\lambda n.\lfloor qn \rfloor + 1$ is fully time-constructible. This includes not only all rationals greater than or equal to 1, but also some irrationals (see [12]).

In our discussion, a variable used to denote a ψ-program is implicitly of type "string". The length of a string a, denoted by $|a|$, is the number of symbol occurrences in a. If a and b are strings, then $|a, b|$ denotes $|a| + |b|$.

Let us now define our programming system ψ. Function g is intended to be a "built-in" composition function; we shall say more about it in a moment.

Definition 3.2

$$\begin{array}{lcl}
\psi(\mathtt{t}) & \stackrel{\mathrm{d}}{=} & \lambda z.\langle 0, z\rangle \\
\psi(\mathtt{u}) & \stackrel{\mathrm{d}}{=} & \lambda\langle y, z\rangle.\langle y+1, z\rangle \\
\psi(\mathtt{v}) & \stackrel{\mathrm{d}}{=} & \lambda\langle p, x\rangle.\phi_p(x) \\
\psi(g(a,b)) & \stackrel{\mathrm{d}}{=} & \psi_a \circ \psi_b \\
\psi(p) & \stackrel{\mathrm{d}}{=} & \lambda z.\uparrow \quad \text{if } p \notin \{\mathtt{t},\ \mathtt{u},\ \mathtt{v}\} \cup \mathbf{range}(g)
\end{array}$$

Assume for the moment that g is recursive and causes no conflict or circularity in Definition 3.2.

Clearly, by effectiveness of ϕ and recursiveness of g, we can effectively "interpret" any given ψ-program. Thus, if ψ turns out to be a programming system, it is going to be an effective one. Also note that g would then be a composition function for ψ.

Now, observe that programs $\mathtt{t}$ and $\mathtt{u}$, together with g, allow us to carry out Machtey and Young's original construction of an s-1-1 function, and that there thus exists a recursive function s such that for all p and x, $\psi_{s(p,x)} = \lambda z.\psi_p(\langle x, z\rangle)$. Then, in particular,

$$\psi_{s(\mathtt{v},x)} = \lambda z.\psi_{\mathtt{v}}(\langle x, z\rangle) = \lambda z.\phi_x(z) = \phi_x.$$

(We can view s as having either one string and one integer argument, or two integer arguments, the first of which is the encoding of a string.) Hence, ψ has at least one program for each partial recursive function and is therefore a programming system. Moreover, it is programmable via $\lambda i.s(\mathtt{v}, i)$. Hence, it is an acceptable programming system.[4]

Let us now turn to g. First of all, we want g to be computable in time $qn + k_0$ for some small k_0, because we want ψ to have a composition function

[4] One interpretation of the fact that ψ is a programming system is that the set $\{\psi_{\mathtt{t}}, \psi_{\mathtt{u}}, \psi_{\mathtt{v}}\}$ forms a "basis" for generating all the partial recursive functions, in that its closure under composition equals *PartRec*. When challenged by the author to prove that no basis of less than three elements could generate *PartRec*, Stuart Kurtz [6] promptly came up with (essentially) the following counterexample. Let $[x]$ stand for the string (over {0,1}) that is the minimum binary representation of integer x, and let $v(\sigma)$ stand for the value of string σ interpreted as a binary integer. Note that $\lambda x, i.v([2x]10^{i+1}11)$ is one-one and strictly increasing in both arguments. Now, define α and β by:

$$\begin{array}{lcll}
\alpha(n) & \stackrel{\mathrm{d}}{=} & 2n & \text{for all } n, \\
\beta(v([2x]10^{i+1}11)) & \stackrel{\mathrm{d}}{=} & \phi_i(x) & \text{for all } (x, i), \\
\beta(n) & \stackrel{\mathrm{d}}{=} & 2n+1 & \text{if } n \text{ is not } v([2x]10^{i+1}11) \text{ for any } (x, i).
\end{array}$$

computable in that much time. However, we also want g to have an output (i.e., a value) of length at least qn. Let us refer to this requirement as the "qn length output" requirement. The idea behind this requirement is to force ψ-programs resulting from many applications of g to be long. We will later show that for any s-1-1 function s for ψ, some programs in $\mathbf{range}(s)$ have to be the result of many applications of g, and if all such programs are long, we will be able to bound below the running time of s.

Of course, we also want g to be recursive and to introduce no conflict or circularity in Definition 3.2. A first try could be the following definition.

$$g(a,b) \stackrel{\mathrm{d}}{=} [a][b]\underbrace{*\cdots\cdots\cdots\cdots\cdots*}_{\lfloor (q-1)|a,b|\rfloor \text{ many}}$$

This g satisfies the "qn length output" requirement by padding its output with enough asterisks. We will see shortly that it is computable in time $\lfloor qn \rfloor + k_0$ for some small k_0. It causes no circularity, since we have $|g(a,b)| > \max(|a|,|b|)$ for all a and b.

However, it is not injective and can cause conflicts in Definition 3.2. Indeed, with $a =$ "[t][t]", $b =$ "t", $a' =$ "[t", $b' =$ "t]][t" and assuming q is very small, we have $g(a,b) = g(a',b') =$ "[[t][t]][t]". To see how this causes a conflict, consider the following correspondences established by Definition 3.2. The ψ-programs a' and b' are certainly not in $\mathbf{range}(g)$, and would therefore correspond to the nowhere defined function. Hence, $g(a',b')$ should correspond to the nowhere defined function. On the other hand, both a and b can be verified to correspond to total functions. Hence, $g(a,b)$ should correspond to a total function. This is a contradiction since $g(a,b) = g(a',b')$.

This problem is easy to solve, however. We say a ψ-program a is *well-formed* iff it has the same number of occurrences of "[" and "]" and no prefix of a has more occurrences of "]" than "[". We say a is *ill-formed* iff it is not well-formed. It is not difficult to show that g, as defined above, is injective when restricted to the set of well-formed programs. Furthermore, verifying that a program is well-formed can be done in real time (i.e., time n). Thus, we could modify an algorithm for g so that it verifies whether both of its arguments are well-formed, as it copies them from input to output, and systematically produces an

---

Obviously, α and β are partial recursive. It can be verified that for all i,

$$\phi_i = \beta^3 \circ \alpha^{i+1} \circ \beta \circ \alpha. \tag{3}$$

Thus, $\{\alpha, \beta\}$ generates $\mathcal{P}art\mathcal{R}ec$. Moreover, (3) suggests a uniform procedure to translate any ϕ-program into (the description of) an equivalent sequence of α's and β's. Note that under any straightforward encoding of sequences, this procedure requires exponential time; however, as a corollary to our upper bound result (Theorem 2.2), there exists one taking $O(n \log n)$ time, and in fact, it is not hard to see that there exists one taking only linear time. This situation is very similar to the fact, pointed out below, that the most efficient s-1-1 algorithm for ψ is not the most obvious one.

It is trivial to show that no one element basis generates $\mathcal{P}art\mathcal{R}ec$.

ill-formed program if it is not the case. This leads us to our final definition of g.

$$g(a,b) \stackrel{\mathrm{d}}{=} \begin{cases} [a][b]\underbrace{*\cdots\cdots\cdots\cdots*}_{\lfloor (q-1)\mid a,b\mid \rfloor \text{ many}} & \text{if } a \text{ and } b \text{ are well-formed,} \\ [a][b[\underbrace{*\cdots\cdots\cdots\cdots*}_{\lfloor (q-1)\mid a,b\mid \rfloor \text{ many}} & \text{otherwise.} \end{cases}$$

Clearly, g still satisfies the "qn length output" requirement, and still causes no circularity. It is still not injective, but it no longer causes conflicts in Definition 3.2, because we now have

$$(\forall a,b,a',b')[g(a,b) = g(a',b') \Longrightarrow \psi_a \circ \psi_b = \psi_{a'} \circ \psi_{b'}].$$

Indeed, it is easy to show that all ill-formed ψ-programs compute the nowhere defined function.

It can also be shown that this g (as well as the original one) is computable everywhere in time $\lfloor qn \rfloor + k_0$ for some small k_0, in fact, in no more time than it takes to write the output. Although this may not immediately be clear, it follows from the fact that $\lambda n.\lfloor qn \rfloor + 1$ is fully time-constructible [3], for $q \geq 1$ rational, and from a technical lemma on fully time-constructible functions, stating that for any such function f, there exists a Turing machine which, on all inputs i, runs for exactly $f(|i|)$ steps, while scanning its input in the first $|i|$ of these steps. We refer the reader to [11] for a proof of this lemma.

Our definition of ψ is now complete.[5] It is an acceptable programming system with an instance of composition computable everywhere in time $\lfloor qn \rfloor + k_0$ for some small k_0.

There remains to show that computing any s-1-1 function for ψ requires time $\Omega(n^{1+\lg q})$ infinitely often. We do this by proving that, for any s-1-1 function s for ψ, there exists a constant c_0 such that there are infinitely[6] many p's for which there are infinitely many x's for which $|s(p,x)| \geq c_0 \cdot |p,x|^{1+\lg q}$. (We shall in fact exhibit a *single* constant c_0, independent of s; i.e., prove the statement with the first two quantifications inverted.)

We say a ψ-program is *atomic* iff it is not in $\mathbf{range}(g)$. Implicit in our earlier discussion of g is the fact that one can "parse" any well-formed ψ-program p, i.e., break it into the unique nonempty sequence of (necessarily well-formed) atomic programs which, composed *in that order* by some application pattern of g yield p. (This parsing can be effective because g is strictly increasing in each argument; however, we do not rely on this fact.) For any well-formed ψ-program p, call the sequence of atomic programs obtained by parsing p, the *atomic sequence* of p, and denote it by $as(p)$. The length of $as(p)$, denoted

[5] ψ is very similar to Royer's *completions of finite bases* [19, Definition 2.4.1].

[6] With some definitions of Ω for multi-argument functions, it would be sufficient to prove this for a single p. What we prove is stronger and might be required for other definitions of Ω.

by $|as(p)|$, is the number of program occurrences in it. By the definition of ψ and by associativity of function composition, the semantics of a well-formed ψ-program is entirely determined by its atomic sequence. In other words, two well-formed ψ-programs with the same atomic sequence are equivalent.

Remember that all ill-formed ψ-programs compute the nowhere defined function. It takes only a moment of reflection to realize that the same is also true of well-formed ψ-programs having atomic programs other than `t, u,` and `v` in their atomic sequence. Thus, any ψ-program not computing the nowhere defined function is well-formed, and has only occurrences of `t, u,` and `v` in its atomic sequence. We say an atomic sequence is *meaningful* iff it has only occurrences of `t, u,` and `v` in it. By extension, a well-formed ψ-program whose atomic sequence is meaningful is said to be meaningful.

It is well known that an acceptable programming system has infinitely many programs for every partial recursive function [18]. Thus, there are infinitely many ψ-programs that compute the first projection function of $\langle \cdot, \cdot \rangle$, i.e., the function $\lambda \langle x, z \rangle . x$. Let p_0 be any of these programs, and s, any s-1-1 function for ψ. Observe that for all k, the ψ-program $s(p_0, k)$ computes $\lambda z.k$, and thus, must be meaningful.

Suppose we compute successively $s(p_0, k)$ for $k = 0,\ 1,\ 2, \ldots$ With these programs, we build two lists, both indexed by k, which we call AS and $MAXLEN$ for, respectively, "atomic sequence", and "maximum length". For each new $s(p_0, k)$ obtained, we fill out position k in each list. In AS, we write the atomic sequence of $s(p_0, k)$, then in $MAXLEN$, we write the maximum length of an atomic sequence on the AS list at this moment.

No two $s(p_0, k)$ programs are equivalent, and hence, no two atomic sequences on the AS list can be equal. On the other hand, they must all be meaningful. Since there are only 3^m meaningful atomic sequences of length m, the following must be true.

$$\text{For all } k, \quad \sum_{i=1}^{MAXLEN(k)} 3^i \geq k. \tag{4}$$

Informally, this says that for any k, there must be at least k different meaningful atomic sequences of length up to $MAXLEN(k)$.

From (4), we get $MAXLEN(k) \geq \log_3(2k/3+1)$ for all k. Now, $\log_3(2k/3+1) \geq |k|/4$ for all sufficiently large k. Obviously, $MAXLEN(k)$ grows to infinity with k, and every time $MAXLEN(k) < MAXLEN(k+1)$, we have $|as(s(p_0, k+1))| = MAXLEN(k+1)$. Thus, $|as(s(p_0, k))| = MAXLEN(k) \geq |k|/4$ for infinitely many k's. The following lemma allows us to get a lower bound on $|s(p_0, k)|$ for any such k.

Lemma 3.3 *Let p be any meaningful ψ-program. Then, $|p| \geq mq^{\lfloor \lg m \rfloor}$, where $m = |as(p)|$.*

Proof. The *parse tree* associated with a given meaningful program p is a labelled tree whose root is labelled p, and in which any vertex either is labelled with an

atomic program and has no child, or is labelled $g(a, b)$ for some programs a and b, and has a left child labelled a and a right child labelled b. It is easy to show that for all meaningful p, the parse tree associated with p is unique, finite, and that its leaves, in order, give $as(p)$, and therefore, consist only of programs `t`, `u`, and `v`.

Pick any meaningful program p. Let $m = |as(p)|$ and let T be the parse tree associated with p. Define w, a function that associates to each vertex v of T a *weight* (a real number), as follows. If v is a leaf, then $w(v) = 1$, otherwise, $w(v) = q \cdot (w(\textit{left-child}(v)) + w(\textit{right-child}(v)))$. Note that, by the "$qn$ length output" requirement satisfied by g, for any vertex of T, the length of the label is greater or equal to the weight. We show that the weight of T (the weight of its root) is at least $mq^{\lfloor \lg m \rfloor}$, thereby establishing the desired lower bound on $|p|$.

Observe that the weight of a parse tree is equal to the summation over all leaves v of $q^{\ell(v)}$, where $\ell(v)$ is the level (distance from the root) at which v is located. Suppose T' is a minimum weight parse tree with m leaves. Obviously, $w(T) \geq w(T')$. Define the depth of a tree as the maximum level over all leaves. Suppose T' has depth d and a leaf at some level $e < d - 1$. Note that, because every vertex in a parse tree has either 0 or 2 children, there are at least 2 leaves at level d. By moving 2 leaves from level d to level $e+1$, we obtain a new tree whose weight differs from that of T' by $-2q^d + q^{d-1} + 2q^{e+1} - q^e = (2 - 1/q)(q^{e+1} - q^d)$, which is strictly negative, because $e + 1 < d$ and $q > 1$. Hence, since T' is a minimum weight parse tree, it must have all its leaves on level d or $d - 1$, i.e., either on level $\lfloor \lg m \rfloor$ or $\lfloor \lg m \rfloor + 1$. Thus, $w(T') \geq mq^{\lfloor \lg m \rfloor}$. □ **Lemma 3.3**

Thus, for the infinity of k's such that $|as(s(p_0, k))| \geq |k|/4$, we get

$$|s(p_0, k)| \geq (|k|/4)q^{\lfloor \lg(|k|/4) \rfloor}.$$

As soon as $k \geq p_0$, we have $|k| \geq |p_0, k|/2$, and we obtain

$$|s(p_0, k)| \geq \frac{1}{8q^4} \cdot |p_0, k|^{1+\lg q}$$

for infinitely many k's. □ **Theorem 3.1**

Note the following peculiar fact about the programming system ψ of the preceding proof. The only obvious s-1-1 function suggested by the definition of ψ is the one obtained by Machtey and Young's construction. A quick analysis shows that the corresponding algorithm takes double-exponential time infinitely often (see Subsection 2.1). However, by Theorem 2.2, there *exists* an algorithm for an s-1-1 function for ψ taking only time $O(n^{1+\lg q})$. The paradoxical situation is that, in order to *constructively* exhibit an efficient algorithm, one seems to have no other choice than to first apply Machtey and Young's construction to obtain (by translation from ϕ-programs) a set of ψ-programs adequate for an efficient construction (such as programs p_0 to p_2 in the proof of Theorem 2.2).

The following is immediate by Theorems 2.2 and 3.1.

Corollary 3.4 *For all rational $q > 1$, there exists an acceptable programming system with an instance of* composition *computable everywhere in time $\lfloor qn \rfloor + k$ for some small k, but with no instance of* composition *computable almost everywhere in time $rn + k'$ for any constants $r < q$ and k'.*

The construction in our proof of Theorem 3.1 is applicable to any fully time-constructible function. Again, however, a truly general result seems hard to express without *ad hoc* definitions. Nevertheless, we have the following corollaries.

Corollary 3.5 *There exists an acceptable programming system with a polynomial time instance of* composition *but for which computing any instance of* s-1-1 *requires time $2^{\Omega(n)}$ infinitely often.*

Proof sketch. Modify the proof of the theorem as follows. Make g "pad" its output so that $|g(a,b)| \geq |a,b|^2$ for all a and b. Also, make sure each string symbol is encoded on at least 2 significant bits. Then, in lieu of Lemma 3.3, observe that the "true" integer length of any meaningful program (i.e., its length as an integer and not as a string) is at least $2^{2^{\lceil \lg m \rceil}}$, where m is the length of its atomic sequence. Indeed, consider a weighted parse tree for the program as in Lemma 3.3, in which all leaves receive weight 0 *except* a single leaf that is farthest from the root, which receives weight 2, and in which the weight of an internal node is given by the squared sum of the weights of its children. Clearly, the weight of this tree is at least $2^{2^{\lceil \lg m \rceil}}$, and gives a lower bound on the "true" integer length of the program. The corollary follows easily. □

Corollary 3.6 *There exists an acceptable programming system with a polynomial time, but no linear time, instance of* composition.

4 An Absolute Lower Bound for Composition

As mentioned in Subsection 1.7, if we use $\max(|i|, |j|)$ as the length of two arguments i and j, we obtain $n^{\lg q}$ $(q > 2)$ or $n \log n$ $(q = 2)$ as a bound in both Theorems 2.2 and 3.1. It follows from the next proposition that the condition $q \geq 2$ does not affect the generality of these results.

Proposition 4.1 *Suppose c is an instance (not necessarily effective) of* composition *in any programming system (not necessarily effective). Then, there is no $r < 2$ such that for some fixed k and for all i and j, $|c(i,j)| \leq r \cdot \max(|i|, |j|) + k$.*

Proof. Suppose there exist $r < 2$ and k such that for all i and j, $|c(i,j)| \leq r \cdot \max(|i|,|j|) + k$. Let p_0 and p_1 be programs (in the programming system for which c is an instance of composition) for $\lambda z.2z$ and $\lambda z.2z+1$ respectively. Define $P_0 = \{p_0, p_1\}$ and recursively, for $n \geq 1$, $P_n = \{c(p,q) \mid p,q \in P_{n-1}\}$. Note that for all n, for all $p,q,p',q' \in P_n$, if either $p \neq p'$ or $q \neq q'$, then $c(p,q)$ and $c(p',q')$ are inequivalent and, hence, different. (The programs in P_n each append a different bit string of length 2^n to the right of their nonzero arguments.) Thus, $\mathrm{card}(P_n) = 2^{2^n}$.

Now, let $\ell_0 = \max(|p_0|,|p_1|)$ and recursively define for $n \geq 1$, $\ell_n = r\ell_{n-1}+k$. By our hypothesis on c, it is clear that for all n, for all $p \in P_n$, $|p| \leq \ell_n$. If $r \neq 1$, $\ell_n \leq (\ell_0 + k/(r-1))r^n$, otherwise, $\ell_n = \ell_0 + kn$. In any case, for sufficiently large values of n, $\ell_n < 2^n$. This is a contradiction because there are 2^{2^n} programs in P_n and only 2^{2^n-1} programs of length less than 2^n. □

Note that we have not ruled out the possibility of an instance of composition c (effective or otherwise) for which $\lim_{n\to\infty}[\inf_{i,j>n}(2\cdot\max(|i|,|j|)-|c(i,j)|)] = \infty$, or even $\lim_{n\to\infty}[\inf_{i+j>n}(2 \cdot \max(|i|,|j|) - |c(i,j)|)] = \infty$. We do not know whether any such instances exist in any programming system.

5 Concluding Remarks and Open Questions

The fact that we get a base 2 logarithm in our bounds in Theorems 2.2 and 3.1 comes from the fact that composition corresponds to composing 2 programs together. If we used another control structure corresponding to composing m programs together for some $m > 2$, we would get a base m logarithm. We would also obtain a lower bound of m in Proposition 4.1.

It is also worthwhile noting that, in our proof of Theorem 3.1, we do not use the fact that the instance of s-1-1 under consideration is recursive. In other words, the length lower bound we obtain applies to the output (i.e., the value) of *any* instance of s-1-1 in the programming system constructed, not just recursive ones. Thus, Theorem 3.1 and Corollary 3.5 could be rephrased to express lower bounds on the length of the output of *arbitrary* instances of s-1-1. With some work, we could similarly extend Corollaries 3.4 and 3.6. In [11], we prove results like Theorem 3.1 and Corollary 3.5 in a way that makes the corresponding output length lower bound apply only to *effective* instances of s-1-1. For example, we construct an acceptable programming system with an instance of composition computable in time linear with constant q, in which computing any effective instance of s-1-1 requires time $\Omega(n^{1+\lg q})$ infinitely often, but in which there exists a *noneffective* instance of s-1-1 whose output length is everywhere linearly related to that of the input.

Consider the class of effective programming systems with a polynomial time s-1-1 function. We argued in Subsection 1.4 that this class corresponds to the programming systems whose programming task is "feasible practically"; it is

therefore of special interest. We established by Theorem 2.2 that any effective programming system with a linear time composition function is in that class. Is the converse true? Does any effective programming system with a polynomial time s-1-1 function necessarily have a linear time composition function? We leave this question open, although we suspect the answer to be negative. Note that it is easy to show that the class of effective programming systems with a polynomial time s-1-1 function coincides with the class *GNPtime* defined by Hartmanis and Baker [2] (but named *GNPtime* by Young [23]), the class of effective programming systems (*G*ödel *N*umberings) into which any other effective programming system can be translated via a polynomial time function.

An informal open question about Theorems 2.2 and 3.1 is whether these results can be generalized elegantly, without resorting to *ad hoc* definitions. In other words, is there a "simple" relation between complexity classes that corresponds satisfactorily to the actual complexity interrelationship between composition and s-1-1 functions in the same programming system?

A possible avenue for future research would be to try to adapt results of the kind we present here to "real-world" semantics, as opposed to the unary partial recursive function semantics of programming systems. To our knowledge, no work has been done yet in this direction.

Acknowledgements

Financial support for this research was provided by FCAR, and by Professor Pierre McKenzie through Canada NSERC grant A9979. We wish to thank Jim Royer for submitting the problem, and for the idea of using constant functions to get a lower bound for s-1-1. We are also indebted to him as well as to Pierre McKenzie, Gilles Brassard, Mark Fulk, Carl Smith, Paul Young and, most especially, John Case, for fruitful and enlightening discussions. We are grateful to Geňa Hahn for the proof of Lemma 3.3 presented here, which is simpler than our original proof. Pierre McKenzie painstakingly read and commented on many different versions of this paper. An anonymous referee made numerous good suggestions for improving the presentation of the material.

References

[1] J. Hartmanis, A note on natural complete sets and Gödel numberings, *Theor. Comput. Sci.* **17** (1982) 75–89.

[2] J. Hartmanis and T. Baker, On simple Gödel numberings and translations, *SIAM J. Comput.*, **4** (1975) 1–11.

[3] J. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages and Computation* (Addison-Wesley, 1979).

[4] N. D. Jones, C. Gomard and P. Sestoft, *Partial Evaluation and the Generation of Program Generators* (Prentice Hall International, 1993).

[5] N. D. Jones, Computer implementation and applications of Kleene's s-m-n and recursion theorems, *in:* Y. N. Moschovakis, *Logic from Computer Science*, Mathematical Sciences Research Institute Publications **21** (Springer-Verlag, Berlin, 1991).

[6] S. Kurtz, personal communication (1989).

[7] I. A. Lavrov, Computable numberings, *in:* R. E. Butts and J. Hintikka, ed., *Logic, Foundations of Mathematics, and Computability Theory* (D. Reidel Publishing Company, Boston, 1977) 195–206.

[8] M. Machtey, K. Winklmann and P. Young, Simple Gödel numberings, isomorphisms, and programming properties, *SIAM J. of Comput.* **7** (1978) 39–60.

[9] M. Machtey and P. Young, *An Introduction to the General Theory of Algorithms* (North-Holland, Amsterdam, 1978).

[10] M. Machtey and P. Young, Remarks on recursion versus diagonalization and exponentially difficult problems, *J. Comput. Systems Sci.* **22** (1981) 442–453.

[11] Y. Marcoux, Complexité des relations sémantiques dans les systèmes de programmation, Ph.D. Thesis, Université de Montréal, 1991. Available as "Document de travail #214", Dép. IRO, Université de Montréal, 1992.

[12] Y. Marcoux, Fully time-constructible real numbers, in preparation.

[13] G. Riccardi, The independence of control structures in abstract programming systems, Ph.D. Thesis, State University of New York at Buffalo, 1980.

[14] G. Riccardi, The independence of control structures in abstract programming systems, *J. Comput. Systems Sci.* **22** (1981) 107–143.

[15] G. Riccardi, The independence of control structures in programmable numberings of the partial recursive functions, *Z. Math. Logik Grundlagen Math.* **48** (1982) 285–296.

[16] H. Rogers, Gödel numberings of the partial recursive functions, *J. Symbolic Logic* **23** (1958) 331–341.

[17] Original edition of [18] (McGraw-Hill, 1967).

[18] H. Rogers, *Theory of Recursive Functions and Effective Computability* (MIT Press, 1987).

[19] J. Royer, *A Connotational Theory of Program Structure*, LNCS 273 (Springer-Verlag, 1987).

[20] J. Royer and J. Case, Progressions of relatively succinct programs in subrecursive hierarchies, Technical Report 86-007, Computer Science Department, University of Chicago, 1986.

[21] J. Royer and J. Case, *Intensional Subrecursion and Complexity Theory*, Research Notes in Theoretical Computer Science (Pitman Press, being revised for publication, 1989).

[22] J. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory* (MIT Press, 1977).

[23] P. Young, Juris Hartmanis: fundamental contributions to isomorphism problems, Technical Report 88-06-02, Computer Science Department, University of Washington, 1988.