# General Size-Change Termination and Lexicographic Descent

Amir M. Ben-Amram*

The Academic College of Tel-Aviv Yaffo
Tel-Aviv, Israel
amirben@mta.ac.il

**Abstract.** Size-change termination (SCT) is a general criterion to identify recursive function definitions that are guaranteed to terminate. It extends and subsumes the simpler criterion of lexicographic descent in function calls, which in classical recursion theory is known as *multiple recursion*. Neil Jones has conjectured that the class of functions computable by size-change terminating programs coincides with the multiply-recursive functions. This paper proves so.

## 1 Introduction

Consider the following recursive function definition (over the natural numbers — as are all function definitions in this paper). Is computation of this function, by straight-forward evaluation of the defining expressions, guaranteed to terminate?

```
f(x,y) = if x=0 then y
                else f(x-1, y+1)
```

The answer is clrealy positive. In fact, `f` has been defined by *primitive recursion*, a form of recursion which always terminates. In this paper, we take a programming view of function definitions, rather than an abstract equational view; more precisely, we consider a definition as above to be a program in a first-order pure-functional language, with the natural semantics. Thus, we say that the occurrence of `f` on the right-hand side represents a *call* of `f` to itself. If all uses of recursion in the program conform to the primitive recursion scheme, and the built-in operators of the language are appropriate, the functions computable will span the class of primitive recursive functions.

A well-known example of a recursively-defined function which is total but not primitive recursive is Ackermann's function:

```
A(m,n) = if m=0 then n+1
    else if n=0 then A(m-1, 1)
                else A(m-1, A(m,n-1))
```

* Part of this research was done while the author was visiting DIKU, the Department of Computer Science at the University of Copenhagen, with support from DART, a project under the Danish Research Councils.

The termination of this computation can be proved by observing that the *pair of arguments* descends lexicographically in every recursive call. This *lexicographic descent criterion* is quite useful in proving termination of functional programs; in recursion theory, the set of functions obtained by recursive definitions of this kind is known as the *multiply-recursive functions* (a more precise definition, in programming terms, appears in Section 2; the classical Recursion-Theoretical reference is Péter [3]). Multiply-recursive definitions extend primitive-recursive ones in two essential ways. First, they use lexicographic descent of a tuple of parameter values as termination guarantee, rather than insisting that the same parameter decrease on every call. Secondly, they allow multiple recursive calls within a definition, which can also nest, as is the case with Ackermann's function. Consider now the following examples:

```
p(m,n,r) = if r>0 then p(m, r-1, n)
      else if n>0 then p(r, n-1, m)
                  else m

g(x,y) = if t=0 then x
    else if x=0 then g(y, y-1)
                else g(y, x-1)
```

Both programs do not satisfy the lexicographic descent criterion, but nonetheless, the termination of these functions can be inferred from the obvious relations between parameter values of calling and called functions. In fact, these programs satisfy the SCT (size-change termination) criterion as formulated by Neil Jones [2][1]. Briefly, satisfaction of this criterion means that if an infinite sequence of recursive calls were possible in a computation, some value would decrease infinitely many times, which is impossible. As is the case with the above examples, this value may meander among parameter positions, and not necessarily decrease in *every* call. SCT also handles mutual recursion naturally, as will be seen from the precise definition in Section 2. SCT clearly subsumes primitive recursion, in which the recursion parameter must decrease in every call; it also subsumes multiple recursion (and does so without reference to lexicographic descent).

Thus, programs satisfying SCT form a superset of multiply-recursive function definitions. But is the class of *functions* that can be computed by such programs larger? Neil Jones has conjectured (in private communication) that these classes coincide. This paper proves that his conjecture is correct.

## 2 Definitions

### 2.1 Programs

We consider programs written in a simple, first-order functional language, with a standard call-by-value semantics (a formal definition can be found in [2] but the

[1] There is earlier work by Sagiv and Lindenstrauss, that used essentially the same idea for termination proofs, see [4, 5].

reader can probably skip it). The only fine detail in the semantics definition is the handling of an "erroneous" expression like `x-1` when `x` is zero (recall that we compute over natural numbers). In [2] we consider the result of evaluating this expression to be a special error value, that essentially aborts the computation. This is a programming-oriented definition, that preserves (even for programs with bugs) the correctness of the basic termination argument, namely that a call sequence that repeatedly replaces `x` with `x-1` must lead to termination.

As in classical recursion theory, the natural numbers will be the only data type in our programs. For practical purposes, size-change termination arguments are also applied to programs using other data types, most notably structures like lists and trees. We note, however, that in analysing a program using such data, a mapping into natural numbers (such as size or height) is typically used in order to argue that recursive calls involve descent. Hence, the essence of the termination arguments is captured completely by working with numbers, with the descent in question relating to the natural order on the natural numbers.

Some notations: the The parameters of function `f` are named, for uniqueness, $Param(\mathtt{f}) = \{\mathtt{f}^{(1)}, \mathtt{f}^{(2)}, \ldots\}$ (though in examples, we prefer to use identifiers as in ordinary programming). We thus consider a parameter name to identify its function uniquely. The set of all parameters in program `p` is $Param(\mathtt{p})$. The *initial function*, denoted $\mathtt{f}_{initial}$, is the entry point of the program, i.e., the function whose termination we wish to prove. We assume that every function in our program is reachable from $\mathtt{f}_{initial}$. Call sites in a function body are identified by (unique) labels. If a call $c$ to function `g` occurs in the body of `f`, we write $c : \mathtt{f} \to \mathtt{g}$, or alternatively, $\mathtt{f} \xrightarrow{c} \mathtt{g}$. A finite or infinite sequence of calls $cs = c_1 c_2 c_3 \ldots$ is *well-formed* if the target function of $c_i$ coincides with the source function of $c_{i+1}$, for every $i$.

We do not fix the set of *primitive operators* available to the program; the successor and predecessor operations, as well as comparison to zero, are necessary and sufficient for computing all multiply-recursive functions, but when writing programs it may be convenient to employ other operators, e.g., subtraction or addition. We do make the (quite natural) assumption that all such operators are primitive recursive, so their presence will not affect the class of computable functions.

### 2.2 Size-change Graphs

In basic definitions of primitive and multiple recursion the only operator assumed to guarantee descent is the predecessor; however in practice one often makes use of other operators and of knowledge about them. For instance, we may make use of the fact that $x - y \leq x$ to prove termination of the following program:[2]

```
h(m,n) = if m=0 then n
    else if n=0 then h(m-1, n)
                else h(m-n, n-1)
```

[2] The subtraction operator represents "monus" in this paper, since we deal with non-negative numbers.

The definition of SCT in [2] is made very general by viewing such knowledge about properties of operators or auxiliary functions as part of the problem data. Put otherwise, SCT is defined not as a property of a program *per se* but of a program annotated with size-change information, in the form of *size-change graphs*, defined in the following subsections. Since these definitions are the same as in [2], a reader familiar with the latter may wish to skip to Sec. 2.4.

**Definition 2.1.** *A* state *is a pair* $(\mathtt{f}, \boldsymbol{v})$*, where* $\boldsymbol{v}$ *is a finite sequence of integers corresponding to the parameters of* f.
*A* state transition $(\mathtt{f}, \boldsymbol{v}) \xrightarrow{c} (\mathtt{g}, \boldsymbol{u})$ *is a pair of states connected by a call. It is required that call* $c : \mathtt{g}(\mathtt{e}_1, \ldots, \mathtt{e}_n)$ *actually appear in* f*'s body, and that* $\boldsymbol{u} = (u_1, \ldots, u_n)$ *be the values obtained by expressions* $(\mathtt{e}_1, \ldots, \mathtt{e}_n)$ *when* f *is evaluated with parameter values* $\boldsymbol{v}$.

**Definition 2.2.** *Let* f, g *be function names in program* p. *A* size-change graph *from* f *to* g*, written* $G : \mathtt{f} \to \mathtt{g}$*, is a bipartite graph from* $Param(\mathtt{f})$ *to* $Param(\mathtt{g})$*, with arcs labeled with either* $\downarrow$ *or* $\overline{\downarrow}$*. Formally, the arc set* $E$ *of* $G$ *is a subset of* $Param(\mathtt{f}) \times \{\downarrow, \overline{\downarrow}\} \times Param(\mathtt{g})$*, that does not contain both* $\mathtt{f}^{(i)} \xrightarrow{\downarrow} \mathtt{g}^{(j)}$ *and* $\mathtt{f}^{(i)} \xrightarrow{\overline{\downarrow}} \mathtt{g}^{(j)}$.

The size-change graph is used to describe the given knowledge about relations between parameters of caller and called functions. A *down-arc* $\mathtt{f}^{(i)} \xrightarrow{\downarrow} \mathtt{g}^{(j)}$ indicates that a value *definitely decreases* in this call, while a *regular arc* $\mathtt{f}^{(i)} \xrightarrow{\overline{\downarrow}} \mathtt{g}^{(j)}$ indicates that a value does not increase. The absence of an arc between a pair of parameters means that neither of these relations is asserted. For visual clarity, regular arcs will be drawn in diagrams without the $\overline{\downarrow}$ label.

Henceforth $\mathcal{G} = \{G_c \mid c \in C\}$ denotes a set of size-change graphs associated with subject program p, one for each of p's calls. Correctness of the information expressed by the size-change graphs is captured by the following definition of *safety*.

**Definition 2.3.** *Let* f*'s definition contain call* $c : \mathtt{g}(\mathtt{e}_1, \ldots, \mathtt{e}_n)$.

1. *A labeled arc* $\mathtt{f}^{(i)} \xrightarrow{r} \mathtt{g}^{(j)}$ *is called* safe for call $c$ *if for every state* $(\mathtt{f}, \boldsymbol{v})$*, if* $(\mathtt{f}, \boldsymbol{v}) \xrightarrow{c} (\mathtt{g}, \boldsymbol{u})$*, then* $r = \downarrow$ *implies* $u_j < v_i$*; and* $r = \overline{\downarrow}$ *implies* $u_j \leq v_i$.
2. *Size-change graph* $G_c$ *is* safe for call $c$ *if every arc in* $G_c$ *is safe for the call.*
3. *Set* $\mathcal{G}$ *of size-change graphs is a* safe description of program p *if it contains, for every call* $c$*, a graph* $G_c$ *safe for that call.*

*Example.* Recall the definition of Ackermann's function:

```
A(m,n) = if m=0 then n+1
    else if n=0 then A(m-1, 1)
                else A(m-1, A(m,n-1))
```

Number the call sites in the order of their occurrence (from 1 to 3). Then the diagrams in Fig. 1 show a safe size-change graph set for the program.

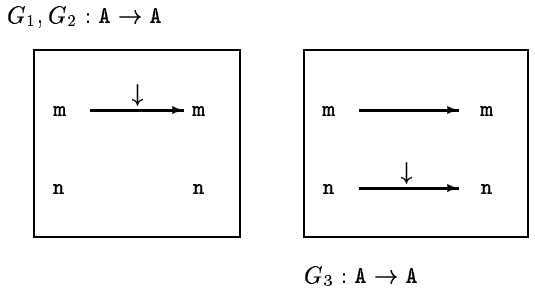



**Fig. 1.** Size-change graphs for the Ackermann function

### 2.3 The SCT Criterion

**Definition 2.4.** *Let $\mathcal{G}$ be a collection of size-change graphs associated with program* p. *Then:*

1. *A* $\mathcal{G}$-multipath *is a finite or infinite sequence $\mathcal{M} = G_1 G_2 \ldots$ of $G_t \in \mathcal{G}$ such that for all $t$, the target function of $G_t$ is the source function of $G_{t+1}$. It is convenient to identify a multipath with the (finite or infinite)* layered directed graph *obtained by identifying the target nodes of $G_t$ with the source nodes of $G_{t+1}$.*
2. *The* source function of $\mathcal{M}$ *is the source function of $G_1$; if $\mathcal{M}$ is finite, its* target function *is the target function of the last size-change graph.*
3. *A* thread *in $\mathcal{M}$ is a (finite or infinite) directed path in $\mathcal{M}$.*
4. *A thread in $\mathcal{M}$ is* complete *if it starts with a source parameter of $G_1$ and is as long as $\mathcal{M}$.*
5. *Thread th is* descending *if it has at least one arc with $\downarrow$. It is* infinitely descending *if the set of such arcs it contains is infinite.*
6. *A multipath $\mathcal{M}$* has infinite descent *if some thread in $\mathcal{M}$ is infinitely descending.*

To illustrate the definitions, consider the following program fragment; one possible corresponding multipath is shown in Fig. 2.

```
f(a,b,c) =    g(a+b, c-1)
g(d,e)   =    ... h(k(e), e, d)...
h(u,v,w) =    g(u, w-1)
```

**Definition 2.5.** *Let $\mathcal{G}$ be a collection of size-change graphs. $\mathcal{G}$ is SCT (or,* satisfies size-change termination*) if every infinite $\mathcal{G}$-multipath has infinite descent.*

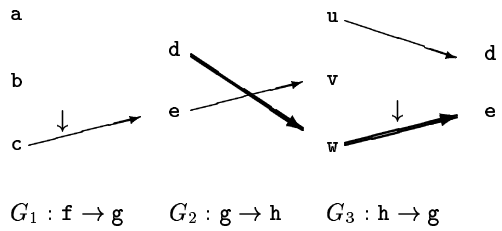



**Fig. 2.** A multipath, with one thread emphasized

This definition captures the essence of proving termination by impossibility of infinite descent. A program that has a safe set of size-change graphs that satisfies the SCT criterion must terminate on every input (for a formal proof see [2]). For a simple example, consider the Ackermann program; it is easy to see that any infinite multipath, i.e., an infinite concatenation of size-change graphs from $\{G_1, G_2, G_3\}$ (shown in Fig. 1) contains an infinitely descending thread. In fact, in this simple case we have *in-situ* descent, where the descending thread is composed entirely of either arcs $\mathtt{m} \to \mathtt{m}$ or of arcs $\mathtt{n} \to \mathtt{n}$.

*Remark:* We adopt the linguistic abuse of relating the size-change termination property to a program, writing "program p is size-change terminating," instead of "p has a safe set of size-change graphs satisfying SCT."

For practical testing of the SCT property, as well as for some developments in this paper, it is useful to paraphrase the SCT condition in finitary terms.

**Definition 2.6.** *The* composition *of two size-change graphs* $G : \mathtt{f} \to \mathtt{g}$ *and* $G' : \mathtt{g} \to \mathtt{h}$ *is* $G;G' : \mathtt{f} \to \mathtt{h}$ *with arc set* $E$ *defined below. Notation: we write* $\mathtt{x} \xrightarrow{r} \mathtt{y} \xrightarrow{r'} \mathtt{z}$ *if* $\mathtt{x} \xrightarrow{r} \mathtt{y}$ *and* $\mathtt{y} \xrightarrow{r'} \mathtt{z}$ *are respectively arcs of* $G$ *and* $G'$.

$$\begin{aligned} E \;=\; & \{\mathtt{x} \xrightarrow{\downarrow} \mathtt{z} \mid \exists \mathtt{y}, r \;.\; \mathtt{x} \xrightarrow{\downarrow} \mathtt{y} \xrightarrow{r} \mathtt{z} \text{ or } \mathtt{x} \xrightarrow{r} \mathtt{y} \xrightarrow{\downarrow} \mathtt{z}\} \\ & \bigcup \{\mathtt{x} \xrightarrow{\mp} \mathtt{z} \mid (\exists \mathtt{y} \;.\; \mathtt{x} \xrightarrow{\mp} \mathtt{y} \xrightarrow{\mp} \mathtt{z}) \text{ and} \\ & \qquad \forall \mathtt{y}, r, r' \;.\; \mathtt{x} \xrightarrow{r} \mathtt{y} \xrightarrow{r'} \mathtt{z} \text{ implies } r = r' = \mp\} \end{aligned}$$

**Definition 2.7.** *For a well-formed nonempty call sequence* $cs = c_1 \ldots c_n$*, define the size-change graph for* $cs$*, denoted* $G_{cs}$*, as* $G_{c_1}; \ldots; G_{c_n}$.

The composition $G_{cs}$ provides a compressed picture of the multipath $\mathcal{M}(cs) = G_{c_1} \ldots G_{c_n}$, in the sense that $G_{cs}$ has an arc $\mathtt{x} \to \mathtt{y}$ if an only if there is a complete thread in $\mathcal{M}$ starting with parameter x and ending with y; and the arc will be labeled with $\downarrow$ if and only if the thread is descending.

**Definition 2.8.** *Let* $\mathcal{G}$ *be a collection of size-change graphs for the subject program. Its* closure set *is*

$$\mathcal{S} = \{G_{cs} \mid cs \text{ is well-formed.}\}$$

Note that the set $\mathcal{S}$ is finite. In fact, its size is bounded by $3^{n^2}$ where $n$ is the maximal length of a parameter list in the program.

**Theorem 2.9 ([2]).** *Program* p *is size-change terminating if and only if for all* $G \in \mathcal{S}$ *such that* $G;G = G$, *there is in* $G$ *an arc of the kind* $\mathtt{x} \stackrel{\downarrow}{\to} \mathtt{x}$.

The essence of the theorem is that in order for a program *not to be* size-change terminating, there must be an infinite multipath without infinite descent. Such a multipath can be decomposed into an infinite concatenation of segments, that are all (but possibly the first) represented by the same graph $G$, and $G$ has no arc of the above kind.

### 2.4 Lexicographic Descent

**Definition 2.10.** *Let a program* p *be given, together with safe size-change graphs.*

1. *We say that function* f *has* semantic lexicographic descent *in parameters* $(\mathtt{x}_1, \ldots, \mathtt{x}_k)$ *if every state-transition sequence from* f *to itself,* $\mathtt{f}(\boldsymbol{v}) \to \cdots \to \mathtt{f}(\boldsymbol{u})$, *that can arise in program execution, satisfies* $\boldsymbol{v} >_{lo} \boldsymbol{u}$, *where* $>_{lo}$ *is the standard lexicographic order on* $\mathbb{N}^k$.
2. *We say that function* f *has* observable lexicographic descent *in parameters* $(\mathtt{x}_1, \ldots, \mathtt{x}_k)$ *if every size-change graph* $G : \mathtt{f} \to \mathtt{f}$ *in the closure set* $\mathcal{S}$ *for the program satisfies the following condition: there is an* $i \leq k$ *such that for all* $j \leq i$, $G$ *contains an arc* $\mathtt{x}_j \stackrel{r_j}{\to} \mathtt{x}_j$, *and moreover* $r_i = \downarrow$.
3. *We say that* f *has* immediate lexicographic descent *in parameters* $(\mathtt{x}_1, \ldots, \mathtt{x}_k)$, *if there are no indirect call paths from* f *to itself; and for each direct call* $c : \mathtt{f} \to \mathtt{f}$, *the size-change graph* $G_c$ *satisfies the lexicographic descent condition as above.*
4. *We say that program* p *has semantic (resp. observable, immediate) lexicographic descent if every function in* p *has semantic (resp. observable, immediate) lexicographic descent. A function is called* multiply-recursive *if it is computable by a program with immediate lexicographic descent.*

As an example, consider the size-change graphs for the Ackermann program (Fig. 1): this program satisfies immediate lexicographic descent.

Clearly, programs with immediate lexicographic descent form a proper subset of those with observable one, and those with observable descent form a proper subset of those with semantic one.

We next show that all three classes of programs yield the same class of computable functions; thus they all characterize the multiply-recursive functions.

**Theorem 2.11.** *Every program with semantic lexicographic descent can be transformed into an equivalent program, with a matching set of size-change graphs, that has immediate lexicographic descent.*

*Proof.* We first show how to remove mutual recursion from a program, maintaining its *semantic* lexicographic descent.

Let $\mathtt{f}_1, \ldots, \mathtt{f}_n$ be all the functions of the given program. Let the parameters of $\mathtt{f}_i$ be $\mathtt{x}_1^{(i)}, \ldots, \mathtt{x}_{k_i}^{(i)}$, where we assume that they are organized in the order that satisfies lexicographic descent. Let $bit(i) = 2^n - 1 - 2^{i-1}$ (a bit map that has only the $i$th bit clear). We construct the following function, to simulate all functions of the given program, using the parameter `which` as selector.

$$\begin{aligned}
&\mathtt{f}(\mathtt{S}, \mathtt{x}_1^{(1)}, \mathtt{x}_2^{(1)}, \ldots, \mathtt{x}_1^{(2)}, \mathtt{x}_2^{(2)}, \ldots, \mathtt{x}_1^{(n)}, \mathtt{x}_2^{(n)}, \ldots, \mathtt{which}) = \\
&\quad \mathtt{case\ which\ of} \\
&\qquad 1 \Rightarrow \mathrm{smash}(\mathtt{f}_1) \\
&\qquad \vdots \\
&\qquad n \Rightarrow \mathrm{smash}(\mathtt{f}_n)
\end{aligned}$$

Where $\mathrm{smash}(\mathtt{f}_i)$ is the body of $\mathtt{f}_i$ with every call $\mathtt{f}_j(\mathtt{e}_1, \ldots, \mathtt{e}_{k_j})$ replaced with a call to `f`, in which:

1. Expressions $\mathtt{e}_1, \ldots, \mathtt{e}_{k_j}$ are passed for $\mathtt{x}_1^{(j)}, \ldots, \mathtt{x}_{k_j}^{(j)}$.
2. The constant $j$ is passed for `which`.
3. `S` is replaced with $(\mathtt{S} \wedge bit(j))$. The *bitwise and* operator $\wedge$, if not present in the language, should be defined as an auxiliary function (it is primitive recursive).
4. All other parameters of `f` (which represent parameters of original functions other than $\mathtt{f}_j$) are passed as received by `f`.

We also add a new initial function (assume w.l.o.g. that the initial function in the source program is $\mathtt{f}_1$):

$$\begin{aligned}
&\mathtt{f}_{initial}(\mathtt{x}_1^{(1)}, \mathtt{x}_2^{(1)}, \ldots, \mathtt{x}_{k_1}^{(1)}) = \\
&\quad \mathtt{f}(bit(1), \mathtt{x}_1^{(1)}, \mathtt{x}_2^{(1)}, \ldots, \mathtt{x}_{k_1}^{(1)}, 0, \ldots, 0, 1)\,.
\end{aligned}$$

*Explanation*: by smashing the program into essentially a single function, all recursion loops become loops of immediate recursive calls. The value of the parameter `S` is always a bit map, where the clear bits identify original functions that have already been simulated; in other words, values that `which` has already taken. The usefulness of this parameter is explained below. Note that in the general case, that is, when simulating a call that is not the first call to a given function, the value of `S` is unchanged.

In fact, when simulating a call to $\mathtt{f}_j$ (except for the first one), only the parameters $\mathtt{x}_1^{(j)}, \ldots, \mathtt{x}_{k_j}^{(j)}$ and `which` may change. We know $\mathtt{f}_j$ to have lexicographic descent (in the unrestricted sense, i.e., over any loop). It follows that the new sequence of values of $\mathtt{x}_1^{(j)}, \ldots, \mathtt{x}_{k_j}^{(j)}$ is lexicographically less than the previous sequence. Since all other parameters retain their values, the parameter list of `f` descends lexicographically.

The above argument does not hold with respect to the first call to $\mathtt{f}_j$ that is simulated; here the previous values of $\mathtt{x}_1^{(j)}, \ldots, \mathtt{x}_{k_j}^{(j)}$ are all zero, and the new

values are arbitrary. For this reason we introduced the S parameter. It strictly decreases whenever an original function is simulated for the first time. Thus, in such a call, lexicographic descent for f is ensured as well. We conclude that in the constructed program, f has lexicographic descent in every recursive call.

Next we show that semantic lexicographic descent can be generally turned into observable one. We assume that mutual recursion has been removed. Let f be any function of the program with parameters $\mathtt{x}_1, \ldots, \mathtt{x}_k$, ordered for lexicographic descent. We replace every call $\mathtt{f}(\mathtt{e}_1, \ldots, \mathtt{e}_k)$ in the body of f with the conditional expression

$$\begin{array}{ll} & \texttt{if } \mathtt{e}_1 < \mathtt{x}_1 \texttt{ then } \mathtt{f}(\mathtt{min}(\mathtt{e}_1, \mathtt{x}_1 - 1), \mathtt{e}_2, \ldots, \mathtt{e}_k) \\ \texttt{else if } & \mathtt{e}_2 < \mathtt{x}_2 \texttt{ then } \mathtt{f}(\mathtt{e}_1, \mathtt{min}(\mathtt{e}_2, \mathtt{x}_2 - 1), \ldots, \mathtt{e}_k) \\ & \vdots \\ \texttt{else if } & \mathtt{e}_k < \mathtt{x}_k \texttt{ then } \mathtt{f}(\mathtt{e}_1, \mathtt{e}_2, \ldots, \mathtt{min}(\mathtt{e}_k, \mathtt{x}_k - 1)) \\ \texttt{else } & 0 \end{array}$$

It is easy to see that this change does not affect the semantics of the program: the expression $\mathtt{min}(\mathtt{e}_i, \mathtt{x}_i - 1)$ will only be used when $\mathtt{e}_i$ evaluates to less then $\mathtt{x}_i$ anyway. However, it is now possible to supply each call with a safe size-change graph so that observable lexicographic descent holds. □

## 3 Stratifiable Programs

In this section we show that for a particular type of programs, called *stratifiable*, size-change termination implies observable lexicographic descent[3]. Since the notion of semantic lexicographic descent will not be used, we omit the adjective "observable" in the sequel.

**Definition 3.1.** *Consider a program with its associated size-change graphs. We call the program* stratifiable *if the parameters of every function can be assigned a partial order $\preceq$ so that the existence of a thread leading from $\mathtt{f}^{(i)}$ to $\mathtt{f}^{(j)}$ (in any valid multipath) implies $\mathtt{f}^{(i)} \prec \mathtt{f}^{(j)}$.*

An example of a stratifiable program is the following:

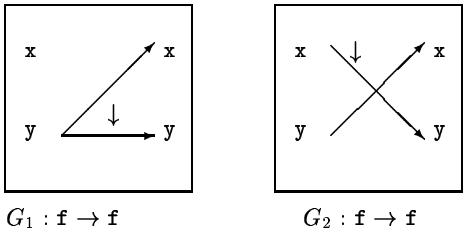
```
h(m,n) = if m=0 then n
    else if n=0 then h(m, n-1)
                else h(m-1, m)
```

With the obvious size-change graphs for the two calls. The order of parameters is $\mathtt{m} \preceq \mathtt{n}$; indeed, there is an arc from m to n (in the graph for the second call) but there is no arc from n to m.

Fig. 3 shows a program with its size-change graphs that are not stratifiable, since the two parameters depend on each other.

[3] The term *stratifiable* is used differently in the context of logic programming. However, there should be no conflict of denotations in our context.

```
f(x,y) = if ... then f(y, y-1)
                else f(y, x-1)
```




$G_1 : \mathtt{f} \to \mathtt{f}$ $G_2 : \mathtt{f} \to \mathtt{f}$

**Fig. 3.** Example program with size-change graphs.

**Theorem 3.2.** *In a stratifiable, size-change terminating program, every function has lexicographic descent. Moreover the lexicographic ordering of the parameters can be effectively (and efficiently) found from the size-change information.*

The theorem follows from the following algorithm to find a lexicographic ordering for a given stratifiable function `f`.

1. Let $\mathcal{S}_\mathtt{f}$ be the set of all size change graphs $G : \mathtt{f} \to \mathtt{f}$ in the closure set $\mathcal{S}$ for the given program (see Sec. 2.3). Delete every arc in the graphs of $\mathcal{S}_\mathtt{f}$ that is not a self-arc ($\mathtt{x} \xrightarrow{r} \mathtt{x}$). These deletions have no effect on the size-change termination properties, because, due to stratifiability, a thread that visits `f` infinitely often can only move a finite number of times between different parameters of `f`.
2. Let $L = \langle\,\rangle$ (the empty list). $L$ will hold the parameters for the lexicographic ordering.
3. Let `x` be a parameter not in $L$ such that an arc $\mathtt{x} \xrightarrow{r} \mathtt{x}$ exists in *every* graph of $\mathcal{S}_\mathtt{f}$, and at least in one of them $r = \downarrow$. Append `x` to $L$.
4. Delete from $\mathcal{S}_\mathtt{f}$ all size-change graphs in which `x` descends. If $\mathcal{S}_\mathtt{f}$ remains non-empty, return to Step 3.

*Correctness*: A parameter `x` as required in Step 3 must always exist, because of size-change termination of `p`; specifically, size-change termination implies that every composition of graphs from $\mathcal{S}_\mathtt{f}$ must contain a descending thread. Note that the parameters already on $L$ will *not* have self-down-arcs in any of the graphs remaining in $\mathcal{S}_\mathtt{f}$.

It is not hard to verify that $L$ provides a lexicographic ordering for `f`. □

## 4 From SCT to Lexicographic Descent

This section gives an algorithm to transform any size-change terminating program into one with lexicographic descent. The construction can be characterized as *instrumentation* of the given program by adding extra function names and parameters, but without affecting its semantics. A simple example of such an

instrumentation is the following. Let p have a function f which calls itself. Replace f with two functions $\mathtt{f}_{\mathtt{odd}}$ and $\mathtt{f}_{\mathtt{even}}$ that call each other; the functions are identical to f otherwise. We obtain a semantically equivalent program where the function name carries some information on the number of recursive calls performed. Similarly, we can add extra parameters to a function to carry around some information that is not directly used by the program, but may be helpful in its analysis. The goal of our construction is to obtain a semantically equivalent program together with a set of size-change graphs which show that the new program is both SCT and stratifiable. Combined with the result of the last section, this implies lexicographic descent.

### 4.1 Preliminaries

Throughout this section, p denotes the subject program (given with its size-change graphs, and known to satisfy SCT). We construct a new program, $\mathtt{p}^*$.

It will be useful to consider a function as receiving an aggregate of parameters, which may be of a different shape than the usual linear list. In fact, in $\mathtt{p}^*$, we have functions whose parameters are arranged in a p-multipath. Such a multipath $\mathcal{M}$ is considered to define the shape of the aggregate; a *valued multipath* $M$ is obtained by assigning an integer value to each of the nodes of $\mathcal{M}$. We refer to $\mathcal{M}$ as the *underlying multipath* of $M$.

We introduce some notation for operating with (valued) multipaths. Recall that the expression $\mathcal{M} : \mathtt{f} \to \mathtt{g}$ indicates that f is the *source function* of multipath $\mathcal{M}$ and g its *target function*. The target parameters of $\mathcal{M}$, the front layer of the layered graph, are therefore g's parameters. Remark that the parameters of each subject-program function are assumed to be distinct, so the source and target function are uniquely determined for a multipath.

Say $M$ has $n$ target parameters. The notation:

$$M \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}$$

can be used to name the values of $M$'s target parameters, or to assign them values, according to context.

The basic operation on multipaths is concatenation, written as juxtaposition $\mathcal{M}_1\mathcal{M}_2$. It identifies the target parameters of $\mathcal{M}_1$ with the source parameters of $\mathcal{M}_2$. To name (or set) the values that these parameters take (in a valued multipath) we write

$$M_1 \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} M_2 \,.$$

All we said about multipaths applies in particular to a single size-change graph, which is a multipath of length one, and to a single list of parameters, which we consider as a zero-length multipath.

Recall the composition operation for size-change graphs, denoted by a semicolon (Definition 2.6). Composing all the size-change graphs that form a multipath $\mathcal{M}$ (or the underlying multipath of a valued multipath $M$) yields a single size-change graph which we denote by $\overline{\mathcal{M}}$ (respectively, $\overline{M}$).

### 4.2 Normal Multipaths

The following observation is central to the size-change analysis of the program we construct.

**Observation 4.1.** Let $A$, $B$, $C$, $D$ be multipaths such that $\overline{B} = \overline{C} = \overline{BC}$ (hence, if we let $G = \overline{B}$, then $G$ satisfies $G;G = G$). Every pair of nodes in $A$, $D$ which are connected by a (descending) thread in multipath $ABCD$ are also connected by a (descending) thread in multipath $ACD$.

Informally, *folding* the multipath by replacing $BC$ by $C$ does not affect connectedness properties for nodes outside the folded part (including nodes on the folded part's boundary).

**Definition 4.2.** *A* p*-multipath $\mathcal{M}$ is* foldable *if it can be broken into three parts, $\mathcal{M} = \mathcal{M}_0\mathcal{M}_1\mathcal{M}_2$, such that $\mathcal{M}_1$ and $\mathcal{M}_2$ are of positive length, and $\overline{\mathcal{M}_2} = \overline{\mathcal{M}_1} = \overline{\mathcal{M}_1\mathcal{M}_2}$.*
*A multipath is* normal *if it does not have a foldable prefix.*

**Lemma 4.3.** *There is a finite bound on the length of a normal multipath.*

*Proof.* Let $N$ be the number of possible size-change graphs over p's parameters ($N$ is bounded by $3^{n^2}$, where $n$ is the maximal length of a function's parameter list). Associate a multipath $\mathcal{M} = G_1G_2\ldots G_t$ with a complete undirected graph over $\{0,1,\ldots,t\}$, where for all $i < j$, edge $(i,j)$ is labeled ("colored") with the size-change graph $G_{i+1};\cdots;G_j$. By Ramsey's theorem [6, p. 23], there is a number $N'$ such that if $t \geq N'$, there is a monochromatic triangle in the graph. Such a triangle $\{i,j,k\}$, with $i < j < k$, implies that $\mathcal{M}_k = G_1G_2\ldots G_k$ is foldable. □

### 4.3 Constructing p*

All functions in the program p* will be called F($M$), with the underlying multipath of $M$ identifing the function referenced. We write F($M : \mathcal{M}$) to name the underlying multipath explicitly.

We define F($M : \mathcal{M}$) for every normal p-multipath $\mathcal{M}$ whose source is p's initial function $\mathtt{f}_{initial}$. The initial function of p* is associated with the zero-length multipath that represents $\mathtt{f}_{initial}$'s parameter list. For each $\mathcal{M} : \mathtt{f}_{initial} \to \mathtt{f}$, the body of F($M : \mathcal{M}$) is the body of f. Occurrences of f's parameters in the

body are replaced with the corresponding target parameters of $M$. Function calls in the body are modified as we next describe. We also describe size-change graphs for the calls.

Consider a call $c : \mathtt{f} \to \mathtt{g}$ with size-change graph $G$. We distinguish two cases.

*Case* 1: $\mathcal{M}G$ is a normal multipath. In this case the call $\mathtt{g}(\mathtt{e}_1, \ldots, \mathtt{e}_n)$ becomes

$$\mathtt{F}(MG \begin{pmatrix} \mathtt{e}_1 \\ \vdots \\ \mathtt{e}_n \end{pmatrix}).$$

The associated size-change graph contains a regular arc from each node of $M$ (on the source side) to its counterpart on the target side (in the prefix $M$ of $MG$). This size-change graph is correct because the data in the connected nodes are identical.

*Case* 2: $\mathcal{M}G$ is not normal, so it must be foldable. Let $MG = ABC$ where $\overline{B} = \overline{C} = \overline{BC}$, $B$ and $C$ of positive length. Let $H = \overline{B}$; note that $H$ must have identical sets of source and target parameters, call them $\mathtt{x}_1, \ldots, \mathtt{x}_n$. Assume that

$$MG = A \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} B \begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix} C,$$

then the call $\mathtt{g}(\mathtt{e}_1, \ldots, \mathtt{e}_n)$ becomes

$$\mathtt{F}(A \begin{pmatrix} v_1 \\ \vdots \\ v_n \end{pmatrix} C \begin{pmatrix} \mathtt{e}_1 \\ \vdots \\ \mathtt{e}_n \end{pmatrix})$$

where

$$v_i = \begin{cases} y_i & \text{if } \mathtt{x}_i \overset{\downarrow}{\to} \mathtt{x}_i \in H \\ x_i & \text{if } \mathtt{x}_i \overset{\downarrow}{\to} \mathtt{x}_i \notin H. \end{cases}$$

The size-change graph for this call contains an arc from every node in $A$ (on the source side) to its counterpart on the target side. For a parameter $\mathtt{x}_i$ in the front layer of $A$, the arc that connects the source node $\mathtt{x}_i$ to the corresponding target node is a down-arc in case that $v_i = y_i$ (note that the choice of $v_i$ is static). All other arcs are regular.

The correctness of this size-change information should be obvious from the construction of the call.

A call that follows Case 2 is called *a folding call.* The length of multipath $A$ is the *folding depth* of this call.

**Lemma 4.4.** *Program* $\mathtt{p}^*$ *is size-change terminating.*

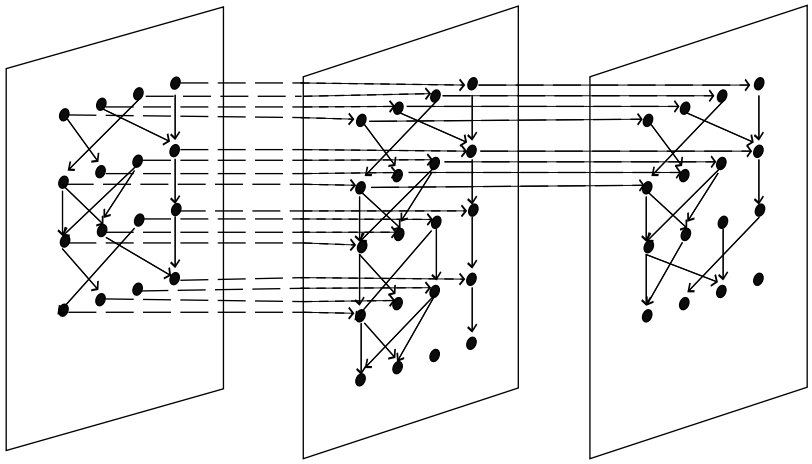


**Fig. 4.** An illustration of a multipath in $\mathtt{p}^*$. The second call is a folding call.

*Proof.* Consider any infinite multipath $\mathcal{M}^*$ of $\mathtt{p}^*$ (note that such a multipath is a layered graph whose layers are finite p-multipaths, connected by size-change arcs for $\mathtt{p}^*$. See Fig. 4).

Because the number of normal p-multipaths is finite, there must be infinitely many folding calls along $\mathcal{M}^*$. Let $d_j$ be the depth of the $j$th folding call, let $d = \liminf_{j\to\infty} d_j$, and consider the suffix of $\mathcal{M}^*$ from the first folding at depth $d$ after which there is no folding at a smaller depth. It suffices to show that this multipath has infinite descent.

Let $A$ be the prefix of length $d$ of the p-multipath that starts $\mathcal{M}^*$. Note that $A$ appears unmodified throughout $\mathcal{M}^*$. Each of its $n$ target parameters $\mathtt{x}_i$ carries an in-situ thread throughout $\mathcal{M}^*$. The arc of this thread is a down-arc whenever the condition $\mathtt{x}_i \stackrel{\downarrow}{\to} \mathtt{x}_i \in H$ is satisfied, where $H$ is the size-change graph representing the folded multipath segment. Now, $H$ is an idempotent size-change graph in the closure set for p and according to Theorem 2.9 there must be some $i$ such that $\mathtt{x}_i \stackrel{\downarrow}{\to} \mathtt{x}_i \in H$. Hence in each call, one of the $n$ in-situ arcs has a down-arrow. This clearly means that at least one of the infinite threads is descending. □

### 4.4 Conclusion

From a given size-change terminating program p we obtained a program $\mathtt{p}^*$ that is size-change terminating and, quite obviously, stratifiable. Therefore, by Theorem 3.2, we obtain

**Theorem 4.5.** *Every size-change terminating program can be effectively transformed into an equivalent program that has lexicographic descent.*

**Corollary 4.6.** *Assume that function $f : \mathbb{N} \to \mathbb{N}$ is computed by a size-change terminating program that uses only primitive recursive operators. Then $f$ is multiply-recursive.*

Suppose we limit the class of multiply-recursive function definitions by disallowing nested recursive calls. This rules out, for instance, the definition of Ackermann's function in Section 1. Péter [3] shows that the functions that can be defined this way are the primitive recursive functions. Since the transformation of p to $\mathtt{p}^*$ never introduces nested calls, we also have

**Corollary 4.7.** *Assume that function $f : \mathbb{N} \to \mathbb{N}$ is computed by a size-change terminating program that uses only primitive recursive operators, and has no nested recursion. Then $f$ is primitive recursive.*

*A comment on complexity.* Our construction to transform a size-change terminating program into one with lexicographic descent has super-exponential complexity. Is this necessary? It is not hard to verify that the construction is in fact a reduction from the set of size-change terminating program descriptions to the set of such program descriptions that are also stratifiable. We know that the former set is PSPACE-complete [2], while the latter belongs to PTIME [1]. Hence, it is unlikely that a construction of *this kind* can be done in sub-exponential time.

## References


[1] Amir M. Ben-Amram. SCT-1-1 $\in$ PTIME (plus some other cases). unpublished note, Copenhagen, 2001.

[2] Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. In *Proceedings of the Twenty-Eigth ACM Symposium on Principles of Programming Languages, January 2001*, pages 81–92. ACM press, 2001.

[3] R. Péter. *Recursive Functions.* Academic Press, 1967.

[4] N. Lindenstrauss, N. and Y. Sagiv. Automatic termination analysis of Prolog programs. In *Proceedings of the Fourteenth International Conference on Logic Programming*, L. Naish, Ed, pages 64—77. MIT Press, Leuven, Belgium, 1977. See also: `http://www.cs.huji.ac.il/~naomil/`.

[5] Y. Sagiv. A termination test for logic programs. In *Logic Programming, Proceedings of the 1991 International Symposium, San Diego, California, USA, Oct 28–Nov 1, 1991*, V. Saraswat and K. Ueda, Eds. MIT Press, 518–532.

[6] J. H. van Lint and R. M. Wilson. *A Course in Combinatorics*, page 23. Cambridge University Press, 1992.