The Size-Change Principle of Termination

Peter Møller Neergaard

Termination is intriguing to researchers—not the least because we know that the Halting Problem is undecidable. These notes will describe how to determine termination using the Size-Change Principle developed by Lee et al. We will use a first-order functional language, but the techniques are easily extended to imperative languages. The solution is however not completely satisfactory in the presence of pointers.

1 An Example

Consider the following program:

fun 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))

The running time is extremely high—beyond anything computable with primitive recursion—still the program terminates. Why?

In every recursive call, at least one of the parameters is decreased:

1. In the call $a(m-1, 1)$, the parameter $m$ is decreased ($n$ is increased: from 0 to 1)
2. In the call $a(m-1, a(m, n-1))$, the parameter $m$ is decreased (while $n$ grows enormously to $a(m, n-1)$).
3. In the call $a(m, n-1)$, the parameter $m$ is unchanged, while $n$ is decreased by 1.

From this observation we can conclude that $a(m, n)$ terminates for all inputs: If not there would be some $m$ and $n$ with an infinite sequence of recursive calls. In that case, we would either end up doing with an infinite sequence of calls corresponding to Case 3 or have infinitely many calls corresponding to Case 1 and Case 2. In the former case, we would decrease $n$ infinitely many times; this is not possible since $n$ cannot go below 0. The latter cases is not better since we

---

*These notes are based heavily on the article [LJBA01] from which it borrows a lot of notation and structure. They thus deserve all intellectual credit, while I take the blame for mistakes and typos are mine though.

*It computes Ackermann’s function which is well-known from Homework 1.
The Size-Change Principle of Termination

2 of 16

\[ a(m-1, 1) \quad a(m-1, a(m, n-1)) \quad a(m, n-1) \]

Figure 1: Parameter changes for the different calls in Ackermann’s function.

\[ m \downarrow \rightarrow m \quad m \downarrow \rightarrow m \quad m \uparrow \rightarrow m \]

\[ n \quad n \quad n \quad n \quad n \downarrow \rightarrow n \]

\[ a(1, 2) \quad a(1, 1) \quad a(1, 0) \quad a(0, 1) \quad a(1, 2) \quad a(0, 3) \]

Figure 2: Some of the sequences of recursive calls in computer \( a(2, 2) \).

would keep decreasing \( m \). We must conclude that an infinite sequence of calls is not possible.

This line of reasoning is captured in the size-change principle:

A program terminates on all inputs if every infinite call sequence would cause an infinite descent in some data value.

We can depict the reasoning about the argument changes with the graphs in Fig. 1 (we use \( \downarrow \) to mark a parameter being decreased and \( \uparrow \) for a parameter which does not increase). The parameter changes taking place when doing any computation of \( a(m, n) \) are combinations of these figures. Consider for instance computing

\[
a(1, 2) = a(0, a(1, 1)) = a(0, a(0, a(1, 0))) = \]

\[
a(0, a(0, a(0, 1))) = a(0, a(0, 2)) = a(0, 3) = 4 .
\]

One sequence of recursive calls arises from the nested recursive call in Case 3 above: \( a(1, 2) \rightarrow a(1, 1) \rightarrow a(1, 0) \rightarrow a(0, 1) \). Another, rather short, sequence of recursive calls comes from Case 2: \( a(1, 2) \rightarrow a(0, 3) \). The two sequences are depicted in Fig. 2. Note in the left figure the unbroken chains of arrows corresponding to the parameters \( m \) and \( n \). The \( \downarrow \) annotation marks that the parameter is decreased along the chain.

As all parameter changes that happens when computing the function are captured as combinations of the size-change graphs in Fig. 1, we can do termination analysis by approximating what combinations can occur. One (crude) way to do this is to consider the flow graph of the program which tells which functions call each other. We then assume that a call to a function \( f \) can be followed by any function call occurring in the body of \( f \). In the concrete example of our Ackermann program this means that we assume that any of the calls in Case 1–3 can be followed by any of the Case 1–3. (Notice that this is an approximation: Case 1 will for instance never be followed by another Case 1.) Consequently, we can approximate the set of infinite sequences of recursive
calls by considering arbitrary infinite combinations of the graphs in Fig. 1. If such a combination results in an infinite descent in parameters, we know that the infinite sequence cannot take place. So if all such infinite combinations result in infinite descent, we conclude that the program is terminating.

The above describes the idea behind doing termination analysis using the size-change principle. In the following section we will formalize this and show that our intuitions are true. In Sec. 2 we introduce our first-order functional. In Sec. 3 we formalize the notation of size-changes graphs. In Sec. 4 we formalize the size-change termination principle.

2 The Language

We will use a functional language similar in style to what we have seen in [Jon01, Jon97] as the language of study. For convenience we will explicitly mark every call site with a tag \( c \) and we will not care too much about the basic operations. This gives the following syntax:

\[
p ::= \text{def}_1 \cdots \text{def}_n
\]
\[
\text{def} ::= f(f_1, \ldots, f_n) = e^f
\]
\[
e ::= f^i
\]
\[
\mid \text{if } e_1 \text{ then } e_2 \text{ then } e_3
\]
\[
\mid \text{op}(e_1, \ldots, e_{\text{arity}(\text{op})})
\]
\[
\mid c : f(e_1, \ldots, e_{\text{arity}(f)}).
\]

To simplify our notation later, we have assumed that \( f \)'s parameters, from left to right, are named \( f_1, \ldots, f_n \); we will however in concrete examples continue to use telling variable names. When evaluating the program we start evaluating the first function in the list of definitions; we refer to this function as the entry function and denote it \( f_{\text{init}} \).

We use \( \text{op} \) to represent our primitive operations which for instance can include:

- \( n \) (with \( n \in \mathbb{N} \)), \(+, -, >\), and \( =\) to implement arithmetic.
- \( \text{Nil}, \text{null?}, \text{hd}, \text{tl}, \text{and cons} \) to implement list operations.
- \( \text{fst}, \text{snd}, \text{and } (, ) \) to implement pairs.

As we only concerned about termination, and not exact running time, we can be more lax in our choice of operations. The only requirement we will have is that applying an operator always terminates though it might terminate signaling error. This is stressing that the operators are primitive.

The next thing is to give a semantics of our language. We will use \( \mathcal{V} \) to denote the set of values we can compute. With the operators above \( \mathcal{V} \) will include the integers, the booleans as well as pairs and lists constructed from booleans and integers. Formally, we can state it as follows:
Definition 2.1.

1. The set of values $\mathbb{V}$ is the least set that contains the integers and booleans and is closed under pairing and lists, i.e., the least set such that
   - $\text{Nil} \in \mathbb{V}$
   - $\{T, F\} \subseteq \mathbb{V}$
   - $\mathbb{N} \subseteq \mathbb{V}$
   - $(u, v) \in \mathbb{V}$ when $u \in \mathbb{V}$ and $v \in \mathbb{V}$, and

2. $\mathbb{V}_\bot$ is $\mathbb{V} \cup \{\bot, \text{Err}\}$ where $\bot, \text{Err} \notin \mathbb{V}$ are special elements marking non-termination and error, resp.

3. We use $\prec$ to compare elements of $\mathbb{V}$:
   - $n \prec m$ when $n, m \in \mathbb{N}$ and $n < m$.
   - $(u, v) \prec (u', v')$ when $|u'| + |v'| < |u| + |v|$ where $|\cdot|$ is the length function: $|\text{Nil}| = 0$, $|n| = |T| = |F| = 1$, and $|(u, v)| = 1 + |u| + |v|$.
   - $\text{Nil} \prec v$ for any $v \in \mathbb{V}$.

All other pair of elements in $\mathbb{V}$ are incomparable.

This definition corresponds very much to Scheme; in particular there is no type distinctions so we can simply encode lists as in Scheme: $[] = \text{Nil}$ and $[u_1, \ldots, u_n] = (u_1, (u_2, \cdots (u_n, \text{Nil}) \cdots))$.

It is important to notice that our value domain is what the mathematicians call well-founded: it is not possible to find an infinite chain of descending values, i.e., there is no infinite chain $v_1, v_2, \cdots \in \mathbb{V}$ such that $v_{i+1} \prec v_i$ for all $i$. Note that our choice of less than for pair means that the tail of a list is strictly smaller than the list.

We can then define the semantics of our programs using the functions $\mathcal{P} : p \rightarrow \mathbb{V}^* \rightarrow \mathbb{V}_\bot$ and $\mathcal{E} : e \rightarrow \mathbb{V}^* \rightarrow p \rightarrow \mathbb{V}_\bot$ which assigns meaning to programs expressions. We base it on $\mathcal{O} : op \rightarrow \mathbb{V}^* \rightarrow \mathbb{V}_\bot$ which provides the meaning of
the operators:

\[ \mathcal{P}[p]v = \mathcal{E}[\mathcal{f}_{init}]p \]

\[ \mathcal{E}[\mathcal{f}]v_p = v_i \]

\[ \mathcal{E}[\text{if } e_1 \text{ then } e_2 \text{ then } e_3]v_p = \begin{cases} \bot & \text{if } \mathcal{E}[e_1]v_p = \bot \\ \mathcal{E}[e_2]v_p & \text{if } \mathcal{E}[e_1]v_p = T \\ \mathcal{E}[e_3]v_p & \text{if } \mathcal{E}[e_1]v_p = F \\ \text{Err} & \text{otherwise} \end{cases} \]

\[ \mathcal{E}[\text{op}(e_1, \ldots, e_n)]v_p = \begin{cases} \mathcal{O}(\text{op})(v_1, \ldots, v_n) & \text{if } v_i = \mathcal{E}[e_i]v_p \neq \{\bot, \text{Err}\} \text{ for } i = 1, \ldots, n \\ v_i & \text{where } v_i = \mathcal{E}[e_i]v_p \text{ and } i \text{ is the least index} \\ & \text{such that } v_i \in \{\bot, \text{Err}\} \end{cases} \]

\[ \mathcal{E}[\text{e : f}(e_1, \ldots, e_n)]v_p = \begin{cases} \mathcal{E}[\mathcal{f}(v_1, \ldots, v_n)]p & \text{if } v_i = \mathcal{E}[e_i]v_p \neq \{\bot, \text{Err}\} \text{ for } i = 1, \ldots, n \\ v_i & \text{where } v_i = \mathcal{E}[e_i]v_p \text{ and } i \text{ is the least index} \\ & \text{such that } v_i \in \{\bot, \text{Err}\} \end{cases} \]

We require that \( \mathcal{O}(op)v \neq \bot \) for all values \( v \): an operator can thus return an error, e.g., \( \text{fst(Nil)} \), but not result in non-terminating. The above semantics makes it explicit that we do left-to-right evaluation of the arguments to an operator or a function: by choosing the value of the first argument producing an error or non-termination we effectively ignore the result of anything to the right of this argument.

By having added the value \( \text{Err} \) we can ensure that operations like \( \text{tl} \) and \( \text{pred} \) always decrease their argument by choosing \( \text{tl(Nil)} = \text{Err} \) rather than \( \text{tl(Nil)} = \text{Nil} \) as we have done before. This gives more precision in our analysis. If we really want our old \( \text{tl} \) we can define it as

\[ \text{tl}'(l) = \text{if } \text{null?(l)} \text{ then } \text{nil} \text{ else } \text{tl}(l) \]

### 3 Size Change Graphs

Having established the language of our choice, we can now formalize the ideas we described in Sec. 1. We will first introduce the notion of state transition sequence as an abstract way of describing the execution of the program. Then we will introduce the notion of call sequences which can describe both what takes places and what can take place. Finally, we can formalize the notion of size-change graph.

We have several times seen that all that really matters for keeping a functional program going is the function calls. For instance in the complexity proof of the interpreters in [Jon01] we noted that the remaining constructors were evaluated in linear time. For that reason we cached only the function results.
We can therefore make a broad description of a run of a program by focusing only on the function calls taking place. More formally, we will call the combination of the current function and its parameters the program’s state\(^2\) and talk about state transitions. We get:

**Definition 3.1.**

1. A *state* is pair in \( FcnName \times \forall^* \).
2. A *state transition* is a pair of states \((f, \bar{u})\) and \((g, \bar{v})\) connected by a call \( c : g(e_1, \ldots, e_n) \) in \( f \)'s body such that \( \bar{u} = (E[f_1][\bar{v}p], \ldots, E[f_k][\bar{v}p]) \).
3. A *state transition sequence* is a finite or infinite sequence of the form

   \[(f_0, \bar{v}_0) \xrightarrow{c_1} (f_1, \bar{v}_1) \xrightarrow{c_2} (f_2, \bar{v}_2) \xrightarrow{c_3} \ldots \]

   such that \((f_t, \bar{v}_t) \xrightarrow{c_{t+1}} (f_{t+1}, \bar{v}_{t+1})\) is a state transition for each \( t = 0, 1, \ldots \)

As an example consider the following program:

\[
\text{fib}(n) = \begin{cases} 
  1 & \text{if } n=0 \\
  1 & \text{if } n=1 \\
  1 & \text{else} \\
  \text{fib}(n-1) + \text{fib}(n-2) & \text{else}
\end{cases}
\]

we have three state sequences when evaluating 3: one for each of the paths to the root in the following diagram:

\[
\begin{array}{c}
\text{fib}(3) \\
\text{fib}(1) \\
\text{fib}(0)
\end{array}
\begin{array}{c}
1 \\
1 \\
1
\end{array}
\begin{array}{c}
\text{fib}(1) \\
\text{fib}(2) \\
\text{fib}(1)
\end{array}
\begin{array}{c}
1 \\
1
\end{array}
\begin{array}{c}
\text{fib}, 1
\end{array}
\]

We can abstract things even further and just talk about how functions call each other. This gives us *call sequences*:

**Definition 3.2.**

1. We write \( f \xrightarrow{C} g \) for a call of the function \( g \) at site \( c \) in the function \( f \). We use \( C \) for the set of all call sites in the program \( p \).
2. Recall the notation \( A^* \) and \( A^\omega \) from the theory of finite automatons: \( A^* \) is the set of all finite sequences of elements from \( A \), i.e., \( \{a_1 \cdots a_n \mid n \geq 0 \text{ and } a_i \in A \text{ for all } i = 1, \ldots, n\} \) while \( A^\omega \) is the set of infinite sequences, i.e., \( \{a_1a_2\cdots \mid a_i \in A \text{ for all } i \in \mathbb{N}\} \).

\(^2\)This is partly justified by the fact that a function’s result is completely determined when we know the parameters.
3. A call sequence is a finite or infinite sequence \( cs = c_1c_2\cdots \in C^* \cup C^\omega \)
where there exists a sequence of functions \( f_0, f_1, \ldots \) such that
\( f_0 \xrightarrow{c_1} f_1 \xrightarrow{c_2} \cdots \).

A key feature of call sequences is that they can describe both an actual run of a program and all the potential runs of a program:

**Definition 3.3.**

1. Given a state transition sequence \( sts = (f_0, \bar{v}_0) \xrightarrow{c_1} (f_1, \bar{v}_1) \xrightarrow{c_2} (f_2, \bar{v}_2) \xrightarrow{c_3} \cdots \) the associated call sequence is \( \text{calls}(sts) = c_1c_2c_3\cdots \).

2. Define the set \( \text{Flow} \) of all possible flows as
\[
\{ cs = c_1c_2\cdots \in C^* \cup C^\omega \mid cs \text{ is a call sequence in } p \text{ and } f_{\text{init}} \xrightarrow{c_1} f_1 \}
\]

The call sequence \( \text{calls}(sts) \) describes the function calls that took place when running the program. That it indeed is a call sequence is immediate from the definitions. The set of all possible flows \( \text{Flow} \) on the other hand describes all possible ways to run the program simply by examining the program. Obviously this will include call sequences that never occur, but this is better than excluding some that does occur: If we for instance annotate Ackermann’s function as

\[
\text{fun } a(m,n) = \\
\text{if } m = 0 \text{ then } n+1 \\
\text{else if } n = 0 \text{ then } 1:a(m-1, 1) \\
\text{else } 2:a(m-1, 3:a(m, n-1))
\]
we have \( \text{Flow} = \{1,2,3\}^* \cup \{1,2,3\}^\omega \). This suggest that two of the recursive calls at point 1 can follow each other; this does however never occur in practice.

Finally, we can introduce the crucial notion of size-change graph which we will use to reason about the behavior of a program:

**Definition 3.4.**

1. Let \( f \) and \( g \) be function names in a program. A size-change graph \( f \) to \( g \), written \( G : f \rightarrow g \) is a bipartite graph from \( f \)'s parameters to \( g \)'s parameters. The edges are labelled with either \( \downarrow \) or \( \Uparrow \).

2. A multipath \( M \) is a finite or infinite sequence \( G_1G_2\cdots \) of size-change graphs such that \( G_t : f_{t-1} \rightarrow f_t \) and \( G_{t+1} : f_t \rightarrow f_{t-1} \) for all \( t \).

3. A thread \( th \) in \( M \) is a connected path of edges in \( M \), i.e.,
\[
t : f_t^{i_t} \xrightarrow{\tau_{t+1}} f_{t+1}^{i_{t+1}} \xrightarrow{\tau_{t+2}} f_{t+2}^{i_{t+2}} \xrightarrow{\tau_{t+3}} \cdots
\]
4. Let $G = \{G_c | c \text{ is a call in program } p\}$ be a set of size-change graphs for program $p$. Given a call sequence $cs = c_1c_2 \cdots$, the $G$-multigraph is defined as $M^G(cs) = G_{c_1}G_{c_2} \cdots$.

When discussing a specific graph we will often write $x \Downarrow y$ when there is $\Downarrow$-edge from $x$ to $y$. Similarly, $x \Uparrow y$ means there is $\Uparrow$-edge from $x$ to $y$.

The beauty of size-change graphs is that they, like state transition sequences, can describe both an actual run and an approximation of the possible runs. First an actual run:

**Definition 3.5.** Let $sts = (f_0, \vec{v}^0) \xrightarrow{c_1} (f_1, \vec{v}^1) \xrightarrow{c_2} (f_2, \vec{v}^2) \cdots$ be a state transition sequence. Define the corresponding multipath $M(sts)$ to be the multipath $G_{f_1}G_{f_2} \cdots$ where the edges of the size change graph $G_{f_i} : f_{i-1} \to f_i$ satisfy the following

- there is a $\Downarrow$ from $f_{i-1}^j$ to $f_i^j$ if, and only if, $v_{j}^i < v_{j}^{i-1}$.
- there is a $\Uparrow$ from $f_{i-1}^j$ to $f_i^j$ if, and only if, $v_{j}^i = v_{j}^{i-1}$.

As this is constructed completely based on the parameter values, it can including some “surprising edges”. Take for instance our favorite call of Ackermann $a(2,1)$:

```
\[
\begin{array}{ccc}
  & \top & \\
  m & f & n \\
  \downarrow & \top & \downarrow \\
  a(1,2) & a(1,1) & a(1,0) & a(0,1)
\end{array}
\]
```

where there are edges between $m$ and $n$ you might not have expected from the function definition.

To achieve the second goal, approximating the run-time behavior, we introduce the notion of being safe: a size-change graph is safe for a function call if it is subgraph of any size-change graph that will arise for that function call when running the program. In other words it is a safe approximation of the run-time behavior because it describes what for sure will happen when running the program. As an example we will consider the middle graph in Fig. 1 safe for second recursive call of Ackermann’s function: we know that any of the corresponding size-change graphs in any of the multipaths $M(sts)$ will contain the edge in the graph of Fig. 1. Similar the two other graphs in Fig. 1 are safe for the first and the third recursive call. We define it as follow:

**Definition 3.6.**

1. A size change graph $G_c : f \to g$ is safe for a call $f \xrightarrow{c} g$ if, and only if, $e^f$ contains the call $g(e_1, \ldots, e_n)$ and fulfills the following condition: if there is an edge from $f^i$ to $g^j$ with label $r_{ij}$ then
The Size-Change Principle of Termination

(a) \( r_{ij} = \downarrow \) implies \( E[e_j] v_i < v_i \) for all \( v_i \in \text{arity}_f \).

(b) \( r_{ij} = \uparrow \) implies \( E[e_j] v_i \leq v_i \) for all \( v_i \in \text{arity}_f \).

2. A set \( G \) of size-change graphs for a program \( p \) is safe if graph \( G_c \in G \) is safe for call \( c \).

The above definition is flexible as there are many safe size-change graphs for a program. For instance a size-change graph with no edges is safe for any call. The task is of course to find the best possible approximation in reasonable time.

We can now formally state that a safe set of size-change graphs is a safe description of a program:

**Lemma 3.7.** Let \( G \) be a set of safe size-change graphs for a program \( p \) and let \( \text{sts} \) be a state transition sequence \((f_0, v^0) \xrightarrow{c_1} (f_1, v^1) \xrightarrow{c_2} (f_2, v^2) \xrightarrow{c_3} \cdots \). Consider the multipaths \( M^G(\text{calls(sts)}) = G_1 G_2 \cdots \) and \( M(\text{sts}) = G'_1 G'_2 \cdots \). We have

1. if \( G_t \) has edge \( f_{i-1} \xrightarrow{\downarrow} f_i \) then \( G'_t \) has the same edge; and

2. if \( G_t \) has edge \( f_{i-1} \xrightarrow{\uparrow} f_i \) then \( G'_t \) has an edge \( f_{i-1} \xrightarrow{r} f'_i \) with \( r = \downarrow \) and \( r = \uparrow \).

**Proof.** Immediate from the Definitions 3.5 and 3.6.

**Exercise 3.8.** We will call a safe size-change graph \( G_c \) maximal for a call \( c \) if, and only if, all other safe size-change graphs are a subgraph of \( G_c \), i.e., if \( G \) is a safe size-change graph for call \( c \) then all the edges in \( G \) are in \( G_c \). Prove that finding the maximal size of size-change graph for a call is undecidable.

4 Size-Change Termination

We concluded the previous section with Lemma 3.7 which establishes that the multipath we obtain from a safe set of size-change graphs and a call sequence \( \text{cs} \) approximates the multipath we get when running the program. A key consequence of the lemma is the following: *if the multipath of an infinite call sequence has an infinite path with infinitely many \( \downarrow \) then so does the multipath we obtain when running the program.* Since infinitely many \( \downarrow \) implies an infinite descent in the value, we must conclude that the particular infinite sequence is not possible. This leads to the *size-change principle of termination:* *if every infinite call sequence of a program provably has an infinite descent, the program is terminating.*

We formalize this as follows:

**Definition 4.1.**
1. The set \( \text{Flow}^\omega \) of all infinite call sequences is
\[
\{ cs = c_1c_2 \cdots \in C^\omega \} \cap \text{Flow}
\]

2. The set \( \text{Desc}^\omega \) of call sequences with infinite descent is
\[
\{ cs \in \text{Flow}^\omega \mid \text{some thread } th \text{ in } M^S(cs) \text{ has infinitely many } \}
\]

By definition \( \text{Desc}^\omega \subseteq \text{Flow}^\omega \). The interesting is that when \( \text{Flow}^\omega \subseteq \text{Desc}^\omega \) every infinite call sequence so the program must terminate. This is captured in the following theorem which is the goal of this section:

**Theorem 4.2.** If \( \text{Flow}^\omega = \text{Desc}^\omega \) then program \( p \) terminates for all inputs.

The key to this theorem is to show that a non-terminating program has an infinite call sequence: if we have an infinite call sequence and \( \text{Flow}^\omega = \text{Desc}^\omega \) then we have an infinite descent which is impossible. We first prove the following lemma which shows that we can pinpoint the source of non-terminating within an expression.

**Lemma 4.3.** Let \( e \) be an expression and \( v \in \mathbb{V} \). If \( E[e][v]p = \bot \) then there exists a call \( c : g(e_1, \ldots, e_n) \) in \( e \) such that \( E[g(e_1, \ldots, e_n)][v]p = \bot \) but \( E[e_1][v]p \neq \bot \).

**Proof.** If \( E[e][v]p \neq \bot \) the lemma is trivially true. So we assume \( E[e][v]p = \bot \) and use induction on the structure of \( e \).

1. \( e = f^i \): This case never happens: as \( v_i \in \mathbb{V} \) we cannot have \( E[f^i][v]p = \bot \).

2. \( e = \text{if } e'_1 \text{ then } e'_2 \text{ then } e'_3 \): There are three cases:
   - \( E[e'_1][v]p = \bot \),
   - \( E[e'_2][v]p = \bot \) and \( E[e'_1][v]p = T \), or
   - \( E[e'_3][v]p = \bot \) and \( E[e'_1][v]p = F \).

   We find \( c : g(e_1, \ldots, e_n) \) by applying the induction hypothesis to \( e'_1, e'_2, \) or \( e'_3 \), resp.

3. \( e = op(e_1, \ldots, e_n) \): Since \( O(op)\bar{w} \neq \bot \) for all \( \bar{w} \) we can only have \( E[e][v]p = \bot \) by having \( E[e_i][v]p = \bot \) for some \( i \). We find \( c : g(e_1, \ldots, e_n) \) by applying the induction hypothesis to \( e_i \).

4. \( e = c' : f(e'_1, \ldots, e'_n) \): We have two subcases
   - (a) We have \( E[e_i][v]p = \bot \) for some \( i \). We find \( c : g(e_1, \ldots, e_n) \) by applying the induction hypothesis to \( e_i \).
   - (b) We have \( E[e_i][v]p \in \mathbb{V} \) for all \( i \) but \( E[c'] : f(e'_1, \ldots, e'_n)[v]p = \bot \). We choose \( c : g(e_1, \ldots, e_n) = c' : f(e'_1, \ldots, e'_n) \).
This exhausts all the possibilities and we conclude that if evaluation of an expression is non-terminating we can find a function call causing the non-termination.

From this we can establish that a non-terminating program has a non-terminating state transition sequence.

**Lemma 4.4.** Suppose \( E[e_{\text{init}}][\vec{v}]p = \bot \). Then there exists an infinite state transition sequence \( \text{sts} = (f_{\text{init}}, \vec{v}_0) \xrightarrow{c_1} (f_1, \vec{v}_1) \xrightarrow{c_2} (f_2, \vec{v}_2) \cdots \).

**Proof.** We establish that in any state \((f, \vec{v})\) where \( E[f][\vec{v}]p = \bot \) there exists a state transition into a state \((g, \vec{u})\) where \( E[f][g][\vec{u}]p = \bot \). It follows from this by induction that we can erect an infinite state transition sequence \( (f_{\text{init}}, \vec{v}_0) \xrightarrow{c_1} (f_1, \vec{v}_1) \xrightarrow{c_2} (f_2, \vec{v}_2) \cdots \).

We thus assume given a state \((f, \vec{v})\) where \( E[f][\vec{v}]p = \bot \). By the previous lemma there exists a call \( c : g(e'_1, \ldots, e'_n) \) in \( e^2 \) such that \( E[e][g][\vec{v}]p = \bot \) while \( E[e'_i][\vec{v}]p \neq \bot \) for all \( i \). Taking \( \vec{u} = (E[e'_1][\vec{v}]p, \ldots, E[e'_n][\vec{v}]p) \) it follows from Def 3.1 that \((f, \vec{v}) \xrightarrow{c} (g, \vec{u})\) is a state transition. From (1) we get that \( \| e \|_{\vec{u}}p = \| g(e'_1, \ldots, e'_n) \|_{\vec{v}}p = \bot \). This concludes the proof that any non-terminating state can be taken into another non-terminating state.

We are now in the position to prove Thm. 4.2:

**Theorem 4.2.** We will use contraposition and prove that if \( p \) is not terminating, then there exists a \( cs \in \text{Flow}^\omega \setminus \text{Desc}^\omega \).

We thus assume that \( p \) is not terminating and use Lemma 4.4 to obtain an infinite state transition sequence \( \text{sts} = (f_{\text{init}}, \vec{v}_0) \xrightarrow{c_1} (f_1, \vec{v}_1) \xrightarrow{c_2} (f_2, \vec{v}_2) \cdots \).

The call sequence \( cs = \text{calls} \text{sts} \) is in \( \text{Flow}^\omega \) by definition. Suppose that we also had \( cs \in \text{Desc}^\omega \). Then by definition of \( \text{Desc}^\omega \), there would be a path in \( M^G(cs) \) with infinitely many \( \downarrow \). By Lemma 3.7 there is also a path in \( M(\text{sts}) \) with infinitely many \( \downarrow \). By definition 3.5 this means that some value is decreased infinitely many times. However, this is impossible as we have assumed that \( V \) is well-founded. We must conclude that \( cs \notin \text{Desc}^\omega \).

This concludes the proof and we conclude that if \( \text{Desc}^\omega = \text{Flow}^\omega \) for a program \( p \) then the program is terminating for all inputs.

This justifies the following definition:

**Definition 4.5.** Program \( p \) is size-change terminating (for a given choice of safe set of size-change graphs \( G \)) if, and only if, \( \text{Flow}^\omega = \text{Desc}^\omega \).

## 5 Algorithm

We now have a clearly stated condition for testing termination. It is however not clear how we in practice can test whether every infinite call sequence has a thread with infinite descent. In this section we give an alternative formulation based on composing size-change graphs. This formulation easily yields an...
algorithm for testing termination. The algorithm is rather costly: it is \( \text{PSPACE} \)-complete (proof can be found in [LJBA01]). This fact just stresses that we are dealing with a tough problem. It might however perform far better in practice. The algorithmic builds on the intuition that any non-termination must necessarily involve looping, i.e., some function calling itself, either directly or indirectly. If no parameter is decreased between such recursive calls, we cannot be sure that the program is terminating.

We first introduce the composition of two size-change graphs:

**Definition 5.1.**

1. Given two size-change graphs \( G : f \to g \) and \( G' : g \to h \) the composition, written \( G; G' : f \to h \), is the size-change graph from \( f \)'s parameters to \( h \)'s parameters with edges \( E \) defined as follows:

\[
E = \{ x \xrightarrow{r} z | \exists y. r.x \xrightarrow{r} y \xrightarrow{r} z \text{ or } x \xrightarrow{r} y \xrightarrow{r} z \}
\]

where \( x \xrightarrow{r} y \xrightarrow{r'} z \) means \( x \xrightarrow{r} y \) and \( y \xrightarrow{r'} z \).

2. Given a call sequence \( cs = c_1 \cdots c_n \) the size change graph for \( cs \), denoted \( G_{cs} \), is \( G_{c_1} ; \cdots ; G_{c_n} \).

**Exercise 5.2.** Proof that composition of size-change graphs is associative, i.e., prove that \( G_1 ; (G_2 ; G_3) = (G_1 ; G_2) ; G_3 \).

We define set \( S \) which contains the size-change graphs for any direct or indirect call between pairs of functions in the program. This set will in particular include the size-change graphs for the calls we are interested: a function \( f \) calling itself directly or indirectly.

**Definition 5.3.** The set \( S \) is defined as

\[
S = \{ G_{cs} | cs, cs_0 \text{ are call sequences and } f_{\text{init}} \xrightarrow{cs_0} f \xrightarrow{cs} g \} .
\]

We note that \( S \) is finite as there are only finitely many different size-change graphs. We can now state the central characterization of size-change termination:

**Theorem 5.4.** Program \( p \) is not size-change terminating if, and only if, \( S \) contains \( G : f \to f \) such that \( G = G; G \) and \( G \) has no arc of form of \( x \xrightarrow{r} x \).
The intuition behind the theorem is as follows: if there is a recursive call from \( f \) to \( f \) which can be repeated infinitely \( (G = G; G) \) then it should have some decreasing parameter for the program to be size-change terminating. Intuitively speaking: if there is no guarantee that a parameter is decreased between two recursive calls, we cannot know whether any parameter eventually hits the bottom.

The theorem also gives an algorithmic way to check for size-change termination:

1. Build the set \( S \) by a transitive closure procedure:
   - Include every \( G_c : f \rightarrow g \) where \( f \xrightarrow{c} g \) is a call in program \( p \) and \( f \) is reachable by some call sequence \( cs_0 : f_{\text{init}} \rightarrow f \). This is done for instance by a depth-first search of the flow graph of the program.
   - Repeatedly include \( G; H \) for any \( G : f \rightarrow g \) and \( H : g \rightarrow h \) until there are no more \( G; H \) to be included.

2. For each \( G : f \rightarrow f \in S \), test whether \( G = G; G \) and \( x \xrightarrow{\downarrow} x \notin G \) for any of the parameters \( x \) of \( f \).

The big cost of this algorithm is the transitive closure where the \( S \) can grow exponentially.

We are now ready to throw ourselves at the main theorem of this section. To do this we need the following lemma on how an edge \( x \xrightarrow{r} y \in G_1; \cdots; G_n \) represents a thread from \( x \) and \( y \) in the multipath \( G_1 \cdots G_n \).

**Lemma 5.5.** The multipath \( M = G_1 \cdots G_n \) has a thread from \( x \) to \( y \) over its entire length containing at least one \( \downarrow \)-labeled edge if, and only if, \( x \xrightarrow{\downarrow} y \in G_1; \cdots; G_n \).

**Proof.** For the "if"-part we assume \( x \xrightarrow{\downarrow} y \in G_1; \cdots; G_n \). We use induction over \( n \) to prove something slightly stronger: if there is an edge between \( x \) and \( y \) in \( G_1; \cdots; G_n \) then there is a thread from \( x \) to \( y \) over the entire length of \( M \); if the edge is labeled \( \downarrow \) then the thread contains at least one \( \downarrow \)-edge:

1. The case \( n = 1 \) is immediate because there is only one size-change graph.

2. For \( n > 1 \), we notice that there must be \( r, r' \), and \( z \) such that \( x \xrightarrow{r} z \in G_1; \cdots; G_{n-1} \) and \( z \xrightarrow{r'} y \in G_n \). By induction hypothesis this means that there is a thread from \( x \) to \( z \) in \( G_1; \cdots; G_{n-1} \). Combining this thread with the edge from \( z \) to \( y \) in \( G_n \), we conclude that there is a thread from \( x \) to \( y \) in \( G_1 \cdots G_n \).
   
   If the edge in \( G_1; \cdots; G_n \) is \( \downarrow \)-labelled, then by definition either \( r \) or \( r' \) is labelled \( \downarrow \). If it is \( r \) it follows by induction hypothesis that the thread has a \( \downarrow \)-edge; otherwise it follows because \( r' \) is labeled with \( \downarrow \).
For the "only if"-part we also use induction on $n$. We prove that if $M$ has a thread from $x$ to $y$ over its entire length, then there is an edge from $x$ to $y$ in $G_1; \ldots ; G_n$. Furthermore, if the thread contains at least one edge labelled $\downarrow$, then the edge $G_1; \ldots ; G_n$ is labeled $\downarrow$:

1. The case $n = 1$ is immediate because there is only one size-change graph.

2. For $n > 1$ we notice that there exists a $z$ such that $G_n$ contains an edge from $z$ to $y$ and the multipath $G_1; \ldots ; G_{n-1}$ contains a thread from $x$ to $z$. By induction hypothesis, $G_1; \ldots ; G_{n-1}$ has an edge from $x$ to $z$ and we conclude that $G_1; \ldots ; G_n$ has an edge from $x$ to $y$.

The thread can have at least one $\downarrow$ in two ways: By $z \downarrow y \in G_n$ where it is immediate from the definition that $x \downarrow y \in G_1; \ldots ; G_n$. Otherwise, there is at least one $\downarrow$ in the thread from $x$ to $z$ in $G_1; \ldots ; G_{n-1}$. By induction hypothesis this implies that $x \downarrow z \in G_1; \ldots ; G_{n-1}$ and thus $x \downarrow y \in G_1; \ldots ; G_n$.

This exhaust all the possibilities and we conclude that $M$ has a thread from $x$ to $y$ over its entire length containing at least one $\downarrow$ labeled edge if, and only if, $x \downarrow y \in G_1; \ldots ; G_n$.

Proof of Theorem 5.4. The "only if"-part is rather elaborate as we have to accomplish two things:

1. we need to prove that $S$ contains a $G : \mathcal{F} \rightarrow \mathcal{F}$ such that $G = G; G$, and

2. prove that this $G$ does not have an edge of the form $x \downarrow x$.

The first part is rather tricky and depends on a result from combinatorics called Ramsey’s Theorem. This theorem is a generalization of the pigeon-hole principle. The second point follows easily when we have established the first.

Let us first recall the (infinite) Ramsey’s Theorem: Let $X$ be some countably infinite set and let $X(n)$ be the subsets of $X$ of size $n$, i.e.,

$$X(n) = \{ \{x_1, \ldots, x_n \mid x_i \neq x_j \text{ for all } i, j = 1, \ldots, n \text{ where } i \neq j \} \subseteq X \} .$$

Now color the elements of $X(n)$ in $c$ different colors. Then there exists some infinite subset $Y$ of $X$ such that the size $n$ subsets of $Y$ all have the same color. Intuitively this holds because there are infinitely many size $n$ subsets of $X$. When we color these subsets with a finite number of colors, here $c$, we have to use at least one of the colors infinitely often.

The proof idea is that an infinite call sequence gives rise to an infinite multipath. As there is only finitely many size-change graphs, at least one of the size-change graphs has to occur infinitely many times in the multipath. We shall see that such a graph, $G^p$, has the desired properties when $p$ is not size-change terminating.
We recall that when \( p \) is not size-change terminating there is, by definition, an infinite call-sequence \( cs \) such that \( M^\varnothing(cs) \) has no thread with infinitely many \( ] \). Our first step is to divide all the subsequences \( c_t, \ldots, c_{t'-1} \) of \( cs \) into classes based on which graph in \( S \) they produce. This is done by considering the class \( P_G \) of pairs yielding \( G \) for each \( G \in S \):

\[
P_G = \{(t, t') \mid G = G_{c_t}; \ldots; G_{c_{t'-1}} \text{ and } t < t'\}.
\]

The classes in the set \( C = \{P_G \mid G \in S\} \) are mutually disjoint: \( G_{c_t}; \ldots; G_{c_{t'-1}} \) produces exactly one graph so the pair \((t, t')\) can belong only to one class. Furthermore, \( C \) is finite as \( S \) is finite.

We can now apply Ramsey’s theorem to find an infinite subset \( T \) that produces the same graph \( G^\varnothing \). We choose \( X = N \) and \( n = 2 \) and let the number of colors \( c \) be the number of size-change graphs in \( S \). Then each of classes \( P_G \) corresponds to one color, i.e., we color the 2-element set \( \{t, t'\} \) with the “color” \( P_G \) if \((t, t') \in P_G \). By Ramsey’s Theorem there is an infinite set \( T \) such that all pairs \((t, t')\) with \( t, t' \in T \) and \( t < t' \) belong to the same class \( P_G \). Thus for any \( t, t' \in T \) with \( t < t' \) we have \( G_{c_t}; \ldots; G_{c_{t'-1}} = G^\varnothing \). This has two implications:

- \( G^\varnothing: f \to g \) for some \( f \): Let \( f \) and \( g \) be the source and destination, resp., of \( G^\varnothing \), i.e., \( G^\varnothing: f \to g \). Now choose any \( t, t', t'' \in T \) such that \( t < t' < t'' \).

We have \( G^\varnothing = G_{c_t}; \ldots; G_{c_{t'-1}}: f \to g \) and \( G^\varnothing = G_{c_t}; \ldots; G_{c_{t''-1}}: f \to g \).

As \( cs \) is a call sequence we have: \( f_{t'-2} \overrightarrow{c_{t'-1}} f_{t'-1} \overrightarrow{c_{t'}} f_{t'} \) and thus \( G_{c_{t'-1}}: f_{t'-2} \to f_{t'-1} \) and \( G_{c_{t'}}: f_{t'-1} \to f_{t'} \). By the definition of composition, we have \( f_{t'-1} = g \) and \( f_{t'-1} = f \) and thus \( f = g \).

- \( G^\varnothing = G^\varnothing; G^\varnothing \): Choose any \( t, t', t'' \in T \) such that \( t < t' < t'' \). As composition is associative we have

\[
G^\varnothing = G_{c_t}; \ldots; G_{c_{t'-1}} = \left( (G_{c_t}; \ldots; G_{c_{t'-1}}) : (G_{c_{t'}}; \ldots; G_{c_{t''-1}}) \right) = G^\varnothing; G^\varnothing.
\]

This establishes that we can find a \( G^\varnothing \) with the properties of Case 1 when \( p \) is not size-change terminating.

We can now use contraposition to prove Case 2, i.e., that \( G^\varnothing \) has no edge \( x \xrightarrow{\perp} x \). Assume that \( G^\varnothing \) has an edge \( x \xrightarrow{\perp} x \) and consider any multipath section \( G_{c_t}; \ldots; G_{c_{t'-1}} \) where \( t, t' \in T \) and \( t' \) is the smallest integer in \( T \) bigger than \( t \). As \( x \xrightarrow{\perp} x \in G^\varnothing = G_{c_t}; \ldots; G_{c_{t'-1}} \), the multipath section has a thread from \( x \) to \( x \) with a \( \perp \)-edge by Lemma 5.5. This implies that \( M^\varnothing(cs) \) has a thread with infinitely many \( \perp \)-edges. This violates the assumption about \( cs \). Consequently \( G^\varnothing \) has no edge \( x \xrightarrow{\perp} x \). This concludes the forward “only if”.

The “if”-part is fortunately much simpler: We assume that there is \( G^\varnothing : f \to f \in \mathcal{S} \) such that \( G^\varnothing = G^\varnothing; G^\varnothing \) and \( G^\varnothing \) has no edge of the form \( x \xrightarrow{\perp} x \). By Definition 5.3 there are call sequences \( cs_0 \) and \( cs_1 \) such that \( f_{\text{init}} \xrightarrow{cs_0} f \xrightarrow{cs_1} f \) and \( G^\varnothing = G_{cs_1} \). Using the definitions 3.2 and 4.1, \( cs_0(cs_1)^\omega \) is a call
sequence belonging to Flow\(^\omega\). We use contraposition and assume that \( p \) is size-change terminating, i.e., DESC\(^\omega\) = Flow\(^\omega\). This implies that \( cs_0(cs_1)^\omega \) and more specifically \( (cs_1)^\omega \) has a thread with infinitely many \( \downarrow \).

We now consider the infinite thread at the start of every \( cs_1 \)-section. As each \( cs_1 \)-section starts with the same function \( f \), and \( f \) has only a finite number of parameters, some parameter \( x \) must be visited infinitely many times by the thread. As there are infinitely many \( \downarrow \) on the thread there is a \( \downarrow \) somewhere after the first visit to \( x \); say \( n' \) repeats of \( cs_1 \) after the first visit to \( x \). As there are infinitely many visits to \( x \) there will be another one somewhere after the \( \downarrow \); say after another \( n'' \) repeats of \( cs_1 \). Let \( n = n' + n'' \). We thus now that \( M^G((cs_1)^n) \) has a descending thread from \( x \) to \( x \). Using Lemma 5.5 this means that \( G_{(cs_1)^n} = (G_{cs_1})^n = (G^\omega)^n = G^\omega \) has an edge \( x \downarrow x \). This does however contradict the assumption about \( G^\omega \). We thus conclude that program \( p \) cannot be size-change terminating.

This concludes the proof that lack of size-change termination is the same as the existence of some \( G : f \rightarrow f \in S \) with \( G = G; G \) and no edge of the form \( x \downarrow y \).

This proof establishes that our algorithm for testing size-change termination is correct.

6 Notes

These notes are based on work by Lee, Jones, and Ben-Amram [LJBA01]. There is even an experimentation tool available [Fre01].

References


