



The Size-Change Termination Principle

Seminar Report

Christian Sternagel

christian.sternagel@uibk.ac.at

17th February 2006

Supervisor: Dr. Georg Moser

Abstract

This report is an introduction to the size-change termination principle (SCT) and its intrinsic complexity, which is surprisingly high. Deciding size-change termination is PSPACE-complete. Moreover some approximations for deciding size-change termination of a given functional program are discussed.

The first approximation (SCP₁) is an $\mathcal{O}(N^2)$ -time algorithm but it only is applicable to a restricted class of programs. With some modifications and at the cost of performance, the approximation (SCP₂) is applicable to general programs within $\mathcal{O}(N^3)$ -time.

Finally the usage of an existing implementation is discussed and a possible generalisation of the analysis is given.

Contents

1	Introduction	3
2	The Size–Change Principle	5
2.1	Basic Notions	5
2.2	Deciding SCT and its Complexity	8
3	P_{TIME}–Approximations	10
3.1	A Graph–Based Algorithm	10
3.2	The General Idea of the Approximation	13
3.3	Deducing Anchors for Fan–In Free Graphs	14
3.4	Deducing Anchors for General Graphs	16
4	Summary	18
A	Applied SCT and Extensions	19
A.1	An Implementation of SCT	19
A.2	Affine–Based Size–Change Termination	21
A.3	An Example for Affine–Based Size–Change Termination	24
B	Theoretical Background	26
B.1	A First Order Functional Language \mathcal{L}_1^f	26
B.2	*–Notation and ω –Notation	27
B.3	ω –Automata	27
B.4	Infinite Ramsey’s Theorem	27
B.5	BOOLEANPROGRAM	28
B.6	Some Graph–Theory	30
	Bibliography	32

Chapter 1

Introduction

The size-change termination principle is used to prove termination of functional programs over well-founded data. It is suitable for automated analysis, although its complexity is very high (i.e. PSPACE-complete). Thus approximations are needed in order to gain efficient algorithms. Two of them are subject of this report.

Until now a number of papers dealing with size-change termination where published ([1, 2, 3, 4]). This is an attempt to summarise most important concepts and ideas and also to unify different notations that where used in the literature. An overall picture of size-change termination and related topics should be given in a compact way.

Unlike most other publications about this topic, the semantics of the functional programs to be analysed is not discussed in detail. Therefore some proofs needed to be reformulated, but as a result it should be easier to understand size-change termination.

Termination analysis using the size-change termination principle (SCT for short) is done in two phases of which only the second one is contained in this report:

1. Extract a *safe* (see Definition 2.1.5) set of size-change graphs.
2. Apply the SCT (see Theorem 2.1.9) criterion.

The first step is in general undecidable but good choices that are intuitively correct can be made (see [2]). The second step deals with the following problem:

instance: A first order functional program p (see Section B.1)
and a *safe* set of size-change graphs \mathcal{G} (see Definitions 2.1.2 and 2.1.5) for p .

question: Is p size-change terminating?

In the following the set of all size-change terminating programs p , as well as the set of all \mathcal{G} s that satisfy the size-change termination criterion, is denoted by SCT whereas SCT denotes the condition for a program p to be *size-change terminating* or for a set of size-change graphs \mathcal{G} to fulfil the size-change termination criterion respectively.

Hence

$$\text{SCT}(p) \iff p \in \text{SCT}$$

reads “A program p is size-change terminating if and only if it is in the set of all size-change terminating programs”, which is no surprise but should clarify how the different notions are used.

There are several formulations of **SCT** that differ only slightly in their diction and level of abstraction. Let the first formulation of **SCT** be:

If *every infinite* computation would cause an infinite descent in some *well-founded* size measure, then no infinite computation is possible.

I.e. it is not possible to infinitely decrease values of a well-founded domain, because at some point the smallest value is reached. Another formulation (using already notions that will be defined later) is:

If *every infinite call sequence* gives rise to an infinitely *strictly decreasing thread* among the size-change graphs, then a program always terminates.

An overview of the topics discussed in each section follows:

In Section 2.1 the principle is introduced and some necessary preliminaries are done, such as the definition of size-change graphs, multipaths and threads.

In Section 2.2 it is shown how to prove size-change termination using Büchi-automata techniques. This is done by constructing two ω -automata and checking whether their accepted languages are equal. Unfortunately equivalence of Büchi-automata is known to be PSPACE-complete. Therefore a look at the complexity of **SCT** is taken, resulting in the PSPACE-completeness proof of size-change termination.

Due to the PSPACE-completeness result for the given problem, PTIME-approximations are discussed in Section 3 starting from a precise (i.e. non-approximating) graph-based algorithm for proving size-change termination. The first approximation is only applicable to a restricted class of programs whereas some modifications result in applicability to general programs.

In Appendix A firstly the web-interface of an existing implementation of **SCT** is described. Afterwards a possible generalisation of size-change graphs into *affine graphs* is shortly discussed and an example is given.

In Appendix B all the necessary definitions that are not directly linked to size-change termination are given to outsource them from the floating text but still make arguments of the former sections precise. Every time a definition from the Appendix is needed, a reference is given.

Chapter 2

The Size–Change Principle

2.1 Basic Notions

There are some notions that have to be defined. First of all it is supposed that the program p is written in a first order functional language with well–founded data, like \mathcal{L}_1^f from Section B.1.

DEFINITION 2.1.1 (Call Sequences). A *call sequence* is a possibly infinite sequence of *call labels* (see Definition B.1.2) occurring in program p ,

$$c_1, c_2, c_3, \dots \in \mathcal{C}_p^{*\omega}.$$

Where $\mathcal{C}_p^{*\omega}$ denotes the set of all finite *or* infinite sequences over \mathcal{C}_p , the set of call labels occurring in p .

It is *well–formed* if it is possible with respect to p 's control flow, i.e. c_{i+1} has to occur in the *body* of the function called at label c_i for all $i = 1, 2, 3, \dots$

DEFINITION 2.1.2 (Size–Change Graphs). A *size–change graph* $G : f \rightarrow g$ is a bipartite graph $G = (\mathcal{P}\text{ar}(f), \mathcal{P}\text{ar}(g), E)$ with set of labelled edges

$$E \subset \mathcal{P}\text{ar}(f) \times \mathcal{L} \times \mathcal{P}\text{ar}(g)$$

such that $\mathcal{L} = \{>, =\}$ and not both $x \xrightarrow{>} y$ and $x \xrightarrow{=} y$ are elements of E .

In other words a size–change graph describes relations between parameters of some function f and parameters of some function g that is called within f . The only relations of interest at this point are equality of two parameters and strict decrease between a call of f and a call of g .

```
ack(m, n) = if m = 0 then n + 1 else
             if n = 0 then 1 : ack(m - 1, 1)
             else 2 : ack(m - 1, 3 : ack(m, n - 1))
```

Listing 2.1: **ack**

EXAMPLE 2.1.3. In Figure 2.1, three possible size–change graphs for the program **ack** from Listing 2.1 can be seen. One for each function call in the body of the function *ack*, where function calls are labelled by consecutive natural numbers.

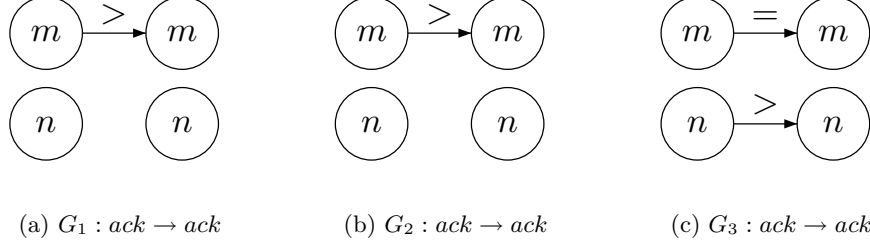


Figure 2.1: Size-change graphs for `ack`

DEFINITION 2.1.4 (Multipaths and Threads). A *multipath* \mathcal{M} is a possibly infinite sequence G_1, G_2, G_3, \dots of size-change graphs s.t. for each $G_i : f_i \rightarrow g_i$ and $G_{i+1} : f_{i+1} \rightarrow g_{i+1}$ the function symbol g_i equals f_{i+1} . This is drawn as a possibly infinite graph like the one in Figure 2.2.

For a well-formed call sequence $cs = c_1, c_2, c_3, \dots$ and a set of size-change graphs \mathcal{G} the \mathcal{G} -multipath of cs (denoted by $\mathcal{M}^{\mathcal{G}}(cs)$) is defined as the sequence $G_{c_1}, G_{c_2}, G_{c_3}, \dots$.

A *thread* within a multipath $\mathcal{M} = G_1, G_2, G_3, \dots$ is a connected possibly infinite path of edges $x_j \xrightarrow{l_j} x_{j+1} \xrightarrow{l_{j+1}} x_{j+2} \xrightarrow{l_{j+2}} \dots$ for some $j \geq 1$ such that $x_i \xrightarrow{l_i} x_{i+1} \in G_i$ for each $i \geq j$.

Note that a thread can start at any vertex within a multipath. Intuitively speaking a multipath represents a possible sequence of function calls during the execution of a program, whereas a thread follows the size-changes (if decreasing or staying equal) of a certain parameter within such a multipath.

DEFINITION 2.1.5 (Safe Sets of Size-Change Graphs). Let p be a first order functional program. A set \mathcal{G} of size-change graphs for p is called *safe* if for every $c \in \mathcal{C}_p$ (i.e. for every call label occurring in p) there is exactly one $G_c : f \rightarrow g \in \mathcal{G}$ and its edges safely describe the size relations between $\text{Par}(f)$ and $\text{Par}(g)$, i.e. G_c encodes only correct but not necessarily all available information about those size relations.

As mentioned above, extracting such a safe set of size-change graphs is in general not decidable. Of course the accuracy of a size-change termination algorithm depends on the quality of this given set.

EXAMPLE 2.1.6. In Figure 2.2 there is an example for a multipath, a finite thread and an infinite thread for the program `ack` respectively.

DEFINITION 2.1.7 (*Flow* and *Desc*). The set of all infinite call sequences that are possible with respect to the control flow of a given first order functional program p (denoted by $\mathcal{F}\text{low}_p$) is defined by:

$$\{cs \mid cs = c_1, c_2, c_3 \dots \in \mathcal{C}_p^\omega, cs \text{ is well-formed, } c_1 \text{ is a call in the body of } f_0\}$$

Where f_0 denotes the initial function (i.e. the entry point) for program p .

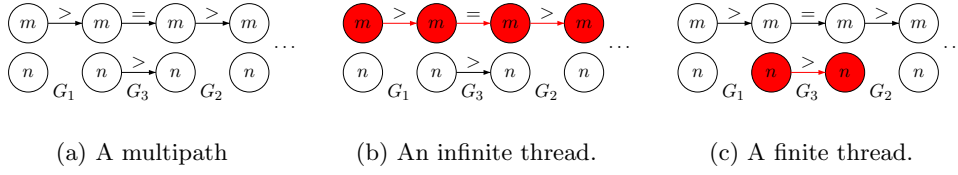


Figure 2.2: Example 2.1.6

Given a program p and a safe set of size-change graphs \mathcal{G} , the set of all infinite call sequences that result in termination (denoted by $\mathcal{D}esc_p$), is defined by:

$$\{cs \mid cs \in \mathcal{F}low_p \text{ and there exists a thread } t \in \mathcal{M}^{\mathcal{G}}(cs) \text{ s.t. } |t|_{>} = \infty\}$$

Note that by definition $\mathcal{D}esc_p \subseteq \mathcal{F}low_p$. Hence for proving equivalence it is sufficient to show $\mathcal{F}low_p \subseteq \mathcal{D}esc_p$.

EXAMPLE 2.1.8. Here is an example of the sets defined above for the program `ack`:

- $\mathcal{F}low_{\text{ack}} = (1 + 2 + 3)^\omega$ which should be obvious.
- And $\mathcal{D}esc_{\text{ack}} = \mathcal{F}low_{\text{ack}}$ which needs some arguing.

Proof. Let $cs \in \mathcal{F}low_{\text{ack}}$. If cs ends with infinitely many 3s then it can be seen from G_3 that there is an infinitely strictly decrease of n , so $cs \in \mathcal{D}esc_{\text{ack}}$ follows. If that is not the case, then there are infinitely many 1s and 2s at the end of cs . As in all three size-change graphs of `ack` there is an edge from m to m it follows that there is an infinite thread with infinitely many descents in it. Hence also in this case $cs \in \mathcal{D}esc_{\text{ack}}$ holds. \square

With these definitions at hand, SCT can be reformulated as follows:

THEOREM 2.1.9 (SCT). *If for given program p and safe set of corresponding size-change graphs \mathcal{G} the following holds,*

$$\mathcal{F}low_p = \mathcal{D}esc_p,$$

then p terminates on all inputs.

Proof. The proof shows that if p does not terminate then there exists a call sequence $cs \in \mathcal{C}_p^\omega$ such that $cs \in \mathcal{F}low_p$ but $cs \notin \mathcal{D}esc_p$ (i.e. $\mathcal{F}low_p \neq \mathcal{D}esc_p$). This is the contraposition of the implication to be shown.

Because p does not terminate, there exists an *infinite* well-formed call sequence $cs = c_1, c_2, c_3, \dots \in \mathcal{C}_p^\omega$, starting in the body of the initial function f_0 . Clearly cs is in $\mathcal{F}low_p$.

Assume $cs \in \mathcal{D}esc_p$, then there exists a thread $t \in \mathcal{M}^{\mathcal{G}}(cs)$ with $|t|_{>} = \infty$. Since \mathcal{G} is *safe* for p , that would give rise to an infinite descent in some well-founded size-measure. As this is impossible cs cannot be an element of $\mathcal{D}esc_p$. \square

For a different proof see [1] Section 2.3.2.

2.2 Deciding SCT and its Complexity

First it is shown how to use Büchi-automata techniques for proving size-change termination. For an introduction to ω -automata see Section B.3. Then a PSPACE-completeness result for SCT is obtained, motivating efficient approximations.

Since there are algorithms for checking equivalence of two ω -regular sets (see Section B.3) it only has to be shown, that $\mathcal{D}esc$ and $\mathcal{F}low$ are ω -regular.

LEMMA 2.2.1. $\mathcal{F}low_p$ and $\mathcal{D}esc_p$ are ω -regular.

Proof. A set is ω -regular if it is accepted by an ω -automaton.

- $\mathcal{F}low_p$ is accepted by the automaton $\mathcal{B}_{\mathcal{F}low_p} = (\mathcal{C}_p, \mathcal{F}_{\mathcal{N}_p}, \{f_0\}, \rho, \mathcal{F}_{\mathcal{N}_p})$ where $\rho = \{(f, c, g) \mid G_c : f \rightarrow g \in \mathcal{G}\}$ and $\mathcal{F}_{\mathcal{N}_p}$ denotes all functions that were defined within program p .
- $\mathcal{D}esc_p$ is accepted by the automaton $\mathcal{B}_{\mathcal{D}esc_p} = (\mathcal{C}_p, S', \{f_0\}, \rho', A')$ s.t.
 - $S' = \mathcal{F}_{\mathcal{N}_p} \cup (\mathcal{V}_p \times \mathcal{L})$
 - $\rho' = \{(x, l), c, (x', l') \mid x \xrightarrow{l'} x' \in G_c, G_c \in \mathcal{G}\} \cup \{(f, c, (x, =)) \mid G_c : f \rightarrow g \in \mathcal{G}, x \in \mathcal{P}ar(g)\} \cup \rho$
 - $A' = \{(x, >) \mid x \in \mathcal{V}_p\}$

□

EXAMPLE 2.2.2. In Figure 2.3 two examples of the automata mentioned in the proof can be seen to clarify the construction. The shown automata are the ones for the Ackerman example.

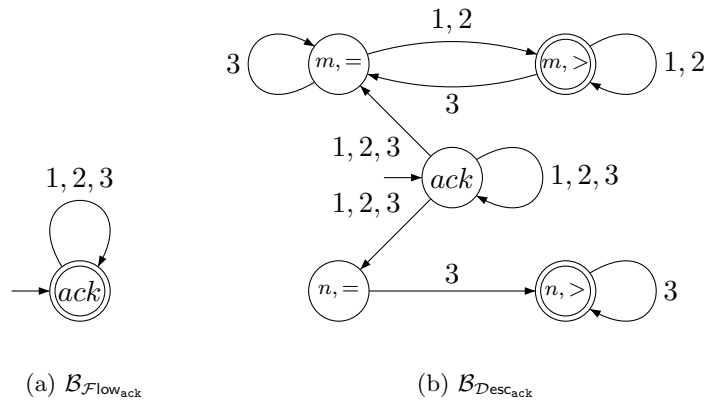


Figure 2.3: $\mathcal{B}_{\mathcal{F}low_{ack}}$ and $\mathcal{B}_{\mathcal{D}esc_{ack}}$

Thus checking **SCT** is reduced to checking equivalence of two ω -regular sets (or languages). Unfortunately this is known to be **PSPACE**-complete. And that is near to the worst case expectable since **PSPACE** is conjectured to be a proper superclass of **NP** and **CoNP** in complexity theory (see [5] for more on that).

So these ω -automata algorithms are not very attractive due to their inefficiency but what about the complexity of **SCT**? Until now it is known that **SCT** is within **PSPACE** (i.e. there is a deterministic Turing machine using at most polynomial space that decides **SCT**). But is this also an intrinsic property of **SCT**?

THEOREM 2.2.3. ***SCT** is **PSPACE**-complete.*

PROOF SKETCH. It has already been argued, that **SCT** is within **PSPACE**. Hence **PSPACE**-hardness of **SCT** remains to be proven.

Let **CoSCT** be the complement of **SCT**. The Theorem is proven by reduction from **BOOLEANPROGRAM** to **CoSCT** (this is possible because **PSPACE** is closed under complement). So given a Boolean program b , a first order functional program p with associated set of size-change graphs \mathcal{G} is constructed. Then it is shown that $b \in \mathbf{BOOLEANPROGRAM}$ if and only if $\mathcal{G} \notin \mathbf{SCT}$. As **BOOLEANPROGRAM** is known to be **PSPACE**-complete (see also Section B.5), this completes the proof. The construction can be found in [1].

Chapter 3

P TIME–Approximations

3.1 A Graph–Based Algorithm

The first algorithm of this section is *not* an approximation, but a precise one, based on graph–theory. But why is it then contained in this section? Simply because the approximations presented afterwards are built on the same idea. Again some preliminaries are needed before the algorithm can be presented.

DEFINITION 3.1.1 (Size–Change Graph Composition). The *composition* of size–change graphs $G : f \rightarrow g$ and $G' : g \rightarrow h$,

$$G \circ G' : f \rightarrow h,$$

is a new size–change graph with edges $E = E_{>} \cup E_{=}$ such that:

$$E_{>} = \{x \xrightarrow{>} z \mid \exists y \in \text{Par}(g), \exists l \in \mathcal{L} \text{ such that } x \xrightarrow{>} y \xrightarrow{l} z \text{ or } x \xrightarrow{l} y \xrightarrow{>} z\}$$

$$E_{=} = \{x \xrightarrow{=} z \mid \exists y \in \text{Par}(g) \text{ such that } x \xrightarrow{=} y \xrightarrow{=} z \text{ and } x \xrightarrow{>} z \notin E_{>}\}$$

The *size–change graph for cs* (denoted by G_{cs}) is defined as $G_{c_1} \circ \dots \circ G_{c_n}$ where cs is a well–formed call sequence c_1, \dots, c_n .

LEMMA 3.1.2. *Multipath* $\mathcal{M} = G_1, \dots, G_n$ has a thread from x to y over its entire length, containing at least one $>$ if and only if $x \xrightarrow{>} y \in G_1 \circ \dots \circ G_n$.

Proof. First the direction from right to left is shown and then the other way round.

\Leftarrow : Suppose the composition $G_1 \circ \dots \circ G_n$ contains an edge $x \xrightarrow{>} y$. By definition of $E_{>}$ this is only possible if in every G_i there *is* an edge from x to y and at least in one of the G_i s this edge is labelled with $>$.

\Rightarrow : Suppose $\mathcal{M} = G_1, \dots, G_n$ has a thread from x to y over its entire length which contains at least one $>$. Because the thread goes over the entire length, there has to be a path $x = x_1 \xrightarrow{l_1} x_2 \xrightarrow{l_2} \dots \xrightarrow{l_{n-1}} x_n = y$ such that at least one of the l_i s equals $>$. This implies $x \xrightarrow{>} y \in G_1 \circ \dots \circ G_n$.

□

DEFINITION 3.1.3 (Composition Closure). The *composition closure* of a set of size-change graphs \mathcal{G} is the smallest set satisfying:

$$\mathcal{G}_{\mathcal{C}} = \mathcal{G} \cup \{G \circ G' \mid G : f \rightarrow g, G' : g \rightarrow h \in \mathcal{G}_{\mathcal{C}}\}$$

```

fib (n) = if n = 0 then 1 else
          if n = 1 then 1 else
            1 : fib (n - 2) + 2 : fib (n - 1)

foo (x) = 3 : foo (x)

```

Listing 3.1: Unused function

REMARK 3.1.4. If the composition closure like above is used, there is a potential problem for proving termination.

In the program of Listing 3.1 for example, we cannot conclude termination because there is the idempotent size-change graph $G : foo \rightarrow foo$ in $\mathcal{G}_{\mathcal{C}}$ but the only edge within G is $x \xrightarrow{=} x$. As the initial function is *fib*, in a real execution of this program we would never reach the body of *foo*. There are two possibilities to circumvent this problem. The first is a different definition of $\mathcal{G}_{\mathcal{C}}$ as in [1] where it is called \mathcal{S} .

As the above formulation of $\mathcal{G}_{\mathcal{C}}$ is an elegant fixpoint computation, it is kept. But then every program p has to be checked for *unused functions* prior to checking size-change termination. Fortunately this can be easily done. Simply check (purely syntactically) for every defined function $f \in \mathcal{F}_{\mathcal{N},p} \setminus \{f_0\}$ whether it is reachable from the initial function f_0 . If a function is not reachable in that way, then it is unused. If an unused function is found it has to be removed from the program. In the sequel it is assumed w.l.o.g. that every program is without unused functions.

-
1. Building $\mathcal{G}_{\mathcal{C}}$:
 - (a) Include every $G \in \mathcal{G}$.
 - (b) For all $G : f \rightarrow g, G' : g \rightarrow h \in \mathcal{G}_{\mathcal{C}}$ include also $G \circ G'$. Repeat 1b as long as $\mathcal{G}_{\mathcal{C}}$ does change.
 2. For each $G : f \rightarrow f \in \mathcal{G}_{\mathcal{C}}$ test whether $G = G \circ G$ and $x \xrightarrow{>} x \notin G$ for each $x \in \mathcal{Par}(f)$. If that is the case answer **no**. Otherwise answer **yes**.
-

Table 3.1: The graph-based algorithm

ALGORITHM 3.1.5. In Table 3.1 the graph-based algorithm corresponding to the proof of Theorem 3.1.6 is shown. So if in step 2 one size-change graph G is found such that $\forall x \in \mathcal{Par}(f) : x \xrightarrow{>} x \notin G$, then the algorithm terminates with

no (i.e. the initial set of size-change graphs \mathcal{G} is not in SCT). Otherwise, after testing all *idempotent* G s, the algorithm terminates with **yes**.

Another formulation of SCT is:

THEOREM 3.1.6 (SCT). *Program p is size-change terminating if and only if for all $G : f \rightarrow f \in \mathcal{G}_{\mathcal{C}}$ such that $G = G \circ G$ (i.e. G is idempotent) there exists $x \xrightarrow{>} x \in G$ for some $x \in \mathcal{P}\text{ar}(f)$.*

Proof. First the above theorem is reformulated in a way, that the structure of the proof below becomes clearer. Let $\neg\text{SCT}(p)$ denote that program p is not size-change terminating. Then the following is equivalent to Theorem 3.1.6 (see also Theorem 4 in [1]):

$$\neg\text{SCT}(p) \\ \iff$$

$$\exists G : f \rightarrow f \in \mathcal{G}_{\mathcal{C}} \text{ with } G = G \circ G \text{ such that } \forall x \in \mathcal{P}\text{ar}(f) : x \xrightarrow{>} x \notin G$$

\Rightarrow : Suppose p is not size-change terminating. Then there exists an infinite call sequence $cs = c_1, c_2, c_3, \dots$ such that there is *no* thread $t \in \mathcal{M}^{\mathcal{G}}(cs)$ with $|t|_{>} = \infty$. Now define a class

$$P_G = \{(i, j) \mid G = G_{cs'} \text{ s.t. } cs' = c_i, c_{i+1}, \dots, c_{j-1} \text{ is a subsequence of } cs\}$$

for every $G \in \mathcal{G}_{\mathcal{C}}$. This partitions $\mathbb{N}_{<}^2$ (i.e. the set of all ordered pairs $(m, n) \in \mathbb{N}$ such that $m < n$) where \mathbb{N} is used as set of indices for cs , into a finite number of classes. Then by Ramsey's Theorem (see page 28), there exists an infinite subset I of \mathbb{N} such that all pairs $(i, j) \in I_{<}^2$ are in the same class P_{G^∞} .

Take $k, m, n \in I$ such that $k < m < n$. Then following holds:

$$\begin{aligned} G^\infty &= G_{c_k} \circ \dots \circ G_{c_{n-1}} \\ &= (G_{c_k} \circ \dots \circ G_{c_{m-1}}) \circ (G_{c_m} \circ \dots \circ G_{c_{n-1}}) \\ &= G^\infty \circ G^\infty \end{aligned}$$

Assume $x \xrightarrow{>} x \in G^\infty$. Then by Lemma 3.1.2 each multipath section $G_{c_i}, \dots, G_{c_{j-1}}$ with $i, j \in I$ such that j is the next bigger natural number within I after i , has a descending thread from x to x . Hence $\mathcal{M}^{\mathcal{G}}(cs)$ would have an infinitely descending thread which leads to a contradiction. Therefore G^∞ can not have an edge $x \xrightarrow{>} x$.

\Leftarrow : Let $G^\infty : f \rightarrow f \in \mathcal{G}_{\mathcal{C}}$ such that $G^\infty = G^\infty \circ G^\infty$ and $x \xrightarrow{>} x \notin G^\infty$. By definition of $\mathcal{G}_{\mathcal{C}}$ there exist well-formed call sequences cs and cs' such that $cs(cs')^\omega \in \mathcal{F}\text{low}_p$ and $G_{cs'} = G^\infty$.

Assume that p is size-change terminating. Then $(cs')^\omega$ has an infinitely descending thread such that some $x \in \mathcal{P}\text{ar}(f)$ is visited infinitely often. That implies that there is a number $n \in \mathbb{N}$ such that $\mathcal{M}^{\mathcal{G}}((cs')^n)$ has a descending thread from x to x . By Lemma 3.1.2, the edge $x \xrightarrow{>} x$ is in $G_{(cs')^n} = (G_{cs'})^n = (G^\infty)^n = G^\infty$ which yields a contradiction. Thus p is not size-change terminating.

□

3.2 The General Idea of the Approximation

As seen in Section 2.2, the problem SCT is PSPACE-complete. Thus approximations are needed in order to obtain efficient algorithms for checking size-change termination. After introducing some preliminaries, the general idea of the approximation(s) follows.

DEFINITION 3.2.1 (Size Relation Graphs). For any set of size-change graphs \mathcal{G} define the following sets:

$$\text{Vert}(\mathcal{G}) = \{x \mid G : f \rightarrow g \in \mathcal{G} \text{ and } x \in \mathcal{P}\text{ar}(f) \cup \mathcal{P}\text{ar}(g)\},$$

the set of all parameters that occur in any size-change graph of \mathcal{G} , and

$$\text{Edge}(\mathcal{G}) = \{x \xrightarrow{l} y \mid G \in \mathcal{G} \text{ and } x \xrightarrow{l} y \in G \text{ with } l \in \mathcal{L}\},$$

the set of all labelled edges that occur in some size-change graph.

The *size relation graph* of \mathcal{G} (denoted by $\text{SRG}_{\mathcal{G}}$) is defined as:

$$\text{SRG}_{\mathcal{G}} = (\text{Vert}(\mathcal{G}), \text{Edge}(\mathcal{G}))$$

DEFINITION 3.2.2 (Call Graphs). The *call graph* of \mathcal{G} (denoted by $\text{CG}_{\mathcal{G}}$) is defined as:

$$\text{CG}_{\mathcal{G}} = (\mathcal{F}_{\mathcal{N},p}, \{(f, g) \mid G : f \rightarrow g \in \mathcal{G}\})$$

A set of size-change graphs \mathcal{G} is *strongly-connected* (see Definition B.6.1) if $\text{CG}_{\mathcal{G}}$ is strongly-connected.

For \mathcal{G} define $\text{Scc}(\mathcal{G})$ to be the set of *strongly connected components* of $\text{CG}_{\mathcal{G}}$ (i.e. $\text{Scc}(\mathcal{G}) = \text{Scc}(\text{CG}_{\mathcal{G}})$).

DEFINITION 3.2.3 (Infinity Sets). The *infinity set* of an infinite sequence $\sigma = s_1, s_2, s_3, \dots$ of elements from the finite set S is

$$S_{\infty}^{\sigma} = \{s \in S \mid |s_1, s_2, s_3, \dots|_s = \infty\}.$$

See Section B.2 for the notation.

DEFINITION 3.2.4 (Anchors). A size-change graph G is called an *anchor* for a set of size-change graphs \mathcal{H} if every infinite \mathcal{H} -multipath \mathcal{M} whose infinity set contains G has infinite descent (i.e. $\forall \mathcal{M} : G \in \mathcal{H}_{\infty}^{\mathcal{M}} \implies |\mathcal{M}|_{>} = \infty$).

LEMMA 3.2.5. *If every non-empty strongly-connected $\mathcal{H} \subseteq \mathcal{G}$ has an anchor, then \mathcal{G} satisfies SCT.*

Proof. Flow_p can also be seen as the set of all infinite paths in $\text{CG}_{\mathcal{G}}$ that are starting in f_0 , where \mathcal{G} is a safe set of size-change graphs for p . Each edge of $\text{CG}_{\mathcal{G}}$ corresponds to exactly one $c \in \mathcal{C}_p$. Because $\text{CG}_{\mathcal{G}}$ is a *finite* graph, every infinite path within it has to ‘settle down’ within some strongly-connected subgraph of $\text{CG}_{\mathcal{G}}$. Because *all* strongly-connected subgraphs $\text{CG}_{\mathcal{H}}$ (that are not empty) have an anchor (also minimal ones consisting of only one node and one edge), every infinite path gives rise to an infinite strict decrease, i.e. $\text{Flow}_p = \text{Desc}_p$. □

LEMMA 3.2.6. *Suppose the set of size-change graphs \mathcal{H} is strongly-connected and $\mathcal{A} \subseteq \mathcal{H}$ is a non-empty set of anchors for \mathcal{H} . If some non-empty strongly-connected $\mathcal{H}' \subseteq \mathcal{H}$ is without anchors, then \mathcal{H}' is a subset of a member of $\text{Scc}(\mathcal{H} \setminus \mathcal{A})$.*

Proof. In the call graph $\text{CG}_{\mathcal{H}}$ an anchor of \mathcal{H} (i.e. an element of \mathcal{A}) corresponds to a *single* edge. So the call graph $\text{CG}_{\mathcal{H} \setminus \mathcal{A}}$ is a subgraph of $\text{CG}_{\mathcal{H}}$ where all edges corresponding to an anchor in \mathcal{A} where deleted. The strongly connected components $\text{Scc}(\text{CG}_{\mathcal{H} \setminus \mathcal{A}})$ of this new call graph cannot contain edges corresponding to graphs within \mathcal{A} , as they were just deleted.

If $\mathcal{H}' \subseteq \mathcal{H}$ and \mathcal{H}' is without anchors, then it is especially not possible for \mathcal{H}' to contain a graph of \mathcal{A} . Because \mathcal{H}' is strongly-connected it is either a part of a strongly-connected component of $\mathcal{H} \setminus \mathcal{A}$ or itself one of those strongly-connected components. \square

ALGORITHM 3.2.7 (SCP). Lemma 3.2.5 leads to following procedure to approximate SCT for a given set of size-change graphs \mathcal{G} , provided there is a way to determine anchors for strongly-connected \mathcal{H} s. Termination is guaranteed by Lemma 3.2.6. The procedure, denoted SCP (size-change polytime), is called with each $\mathcal{H} \in \text{Scc}(\mathcal{G})$. It is shown in Table 3.2.

-
1. Determine anchors \mathcal{A} for \mathcal{H} .
 2. If $\mathcal{A} = \emptyset$ then fail.
 3. For each $\mathcal{H}' \in \text{Scc}(\mathcal{H} \setminus \mathcal{A})$ with $\mathcal{H}' \neq \emptyset$ call $\text{SCP}(\mathcal{H}')$.
-

Table 3.2: The approximation framework

Either the procedure fails, indicating that \mathcal{G} is not size-change terminating, or it terminates without failure, indicating $\text{SCT}(\mathcal{G})$.

In the following two sections, first a way to determine anchors for fan-in free graphs, and then an approach for general graphs, is shown.

3.3 Deducing Anchors for Fan-In Free Graphs

DEFINITION 3.3.1 (Fan-In Free Size-Change Graphs). A size-change graph $G : f \rightarrow g$ is called *fan-in free* if for all $x, y \in \text{Par}(f)$, $z \in \text{Par}(g)$ and $l, l' \in \mathcal{L}$ from $x \xrightarrow{l} z \in G$ and $y \xrightarrow{l'} z \in G$ follows, that $x = y$.

A set \mathcal{H} of size-change graphs is called *fan-in free* if all graphs G within \mathcal{H} are fan-in free.

DEFINITION 3.3.2 (Thread Preservers). Let V be a set of parameters (i.e. $V \subseteq \mathcal{V}_p$) and \mathcal{H} be a set of size-change graphs.

The set of *forward thread preservers* of \mathcal{H} is the largest set $\overrightarrow{\text{TP}}_V(\mathcal{H})$ such that:

$$\{x \in V \mid \forall G : f \rightarrow g \in \mathcal{H} : x \in \text{Par}(f) \implies \exists y \in \overrightarrow{\text{TP}}_V(\mathcal{H}) : x \xrightarrow{l} y \in G\}$$

i.e. a subset of V such that for each of its members there is a non-empty thread within the size-change graphs of \mathcal{H} starting at it.

The set of *backward thread preservers* of \mathcal{H} is the largest set $\overleftarrow{\text{TP}}_V(\mathcal{H})$ such that:

$$\{y \in V \mid \forall G : f \rightarrow g \in \mathcal{H} : y \in \text{Par}(g) \implies \exists x \in \overleftarrow{\text{TP}}_V(\mathcal{H}) : x \xrightarrow{l} y \in G\}$$

i.e. a subset of V such that for each of its members there is a non-empty thread within the size-change graphs of \mathcal{H} ending in it.

LEMMA 3.3.3. *Let \mathcal{H} be a fan-in free and strongly-connected set of size-change graphs. If for $G \in \mathcal{H}$, there is an edge $x \xrightarrow{\succ} y \in G$ such that $x, y \in \overrightarrow{\text{TP}}_{V_p}(\mathcal{H})$, then G is an anchor for \mathcal{H} .*

Proof. For a proof see [2] Theorem 4. □

ALGORITHM 3.3.4 (SCP₁). The variant of SCP where the first step (determining anchors) is done as indicated by Lemma 3.3.3 is denoted by SCP₁.

THEOREM 3.3.5. *Let N be the space requirement for \mathcal{G} . Then SCP₁ is a $\mathcal{O}(N^2)$ -time procedure to approximate SCT.*

Proof. For a proof see [2] Lemma 3. □

But why can this method of determining anchors not be used for general graphs? Here follows a counter example.

EXAMPLE 3.3.6. For the simple program of Listing 3.2, the size-change graph of Figure 3.1(a) can be obtained, considering also branching behaviour (which is in general not possible automatically), i.e. if the programs control flow reaches the recursive function call to f , then y has to be the list x without its first element.

```
f(x, y) = if y = tl(x) then 1 : f(x, y) else 0
```

Listing 3.2: A problematic program

Hence the set of size-change graphs for the given program is $\mathcal{G} = \{G_1\}$, and the call graph of \mathcal{G} is a trivial graph with one node f and one edge $f \rightarrow f$ as can be seen in Figure 3.1(b). $\text{CG}_{\mathcal{G}}$ is strongly-connected. The forward thread preservers of \mathcal{G} are $\overrightarrow{\text{TP}}_{V_p}(\mathcal{G}) = \{x, y\}$ and $x \xrightarrow{\succ} y \in G_1$. Thus SCP would conclude that G is an anchor, which implies $\text{SCT}(\mathcal{G})$.

But this is not the case as can be seen by the multipath $\mathcal{M} = G_1, G_1, G_1, \dots$, which has no infinite descent.

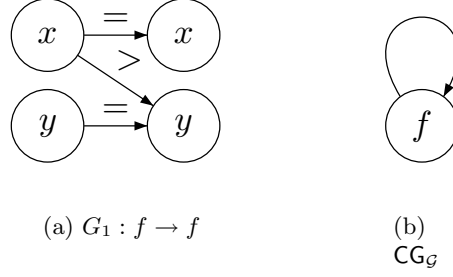


Figure 3.1: Size-change graph and call graph

3.4 Deducing Anchors for General Graphs

Edges that are not in the same strongly-connected component of the size relation graph of \mathcal{G} , cannot give rise to anchors (see [2]). Therefore following *critical parameter sets* are defined, to be inspected independently for descent.

DEFINITION 3.4.1 (Critical Parameter Sets). The *critical parameter sets* for the set of size-change graphs \mathcal{H} is defined as:

$$\text{Crit}(\mathcal{H}) = \{V \mid \overleftarrow{\text{TP}}_V(\mathcal{H}) = V, V \text{ strongly-connected in } \text{SRG}_{\mathcal{H}}, \text{ and } V \text{ maximal}\}$$

ALGORITHM 3.4.2 (Computing $\text{Crit}(\mathcal{H})$). In Table 3.3 it can be seen how to compute the critical parameter sets. See also Definition B.6.3 for the notation.

-
1. Initialise $\text{Crit}(\mathcal{H})$ to $\{\overleftarrow{\text{TP}}_{V_p}(\mathcal{H})\}$.
 2. Replace each $V \in \text{Crit}(\mathcal{H})$ with (possibly several) non-empty $\overleftarrow{\text{TP}}_{V'}(\mathcal{H})$, where $V' \in \text{ScC}(\text{SRG}_{\mathcal{H}}|_V)$. Repeat 2 as long as $\text{Crit}(\mathcal{H})$ changes.
-

Table 3.3: Computing $\text{Crit}(\mathcal{H})$

LEMMA 3.4.3. Let \mathcal{H} be a strongly-connected set of size-change graphs. Then $G \in \mathcal{H}$ is an anchor for \mathcal{H} if there is a set of parameters $V \in \text{Crit}(\mathcal{H})$ such that one of the following holds:

- either:** For each $G' \in \mathcal{H}$ with $x \xrightarrow{l} y, x \xrightarrow{l'} y' \in G'$ such that $x, y, y' \in V$ and $l, l' \in \mathcal{L}$ it follows that $y = y'$ and there exists $x \xrightarrow{>} y \in G$ with $x, y \in V$.
- or:** The graph with edges $\{x \xrightarrow{=} y \in \text{SRG}_{\mathcal{H}} \mid x, y \in V\}$ has no strongly-connected subgraph with an edge $x \xrightarrow{=} y \in G$.

ALGORITHM 3.4.4 (SCP_2). The variant of SCP where anchors are deduced as indicated by Lemma 3.4.3 is denoted by SCP_2 .

THEOREM 3.4.5. *Let N be the space requirement for given set of size-change graphs \mathcal{G} . Then SCP_2 is a $\mathcal{O}(N^3)$ -time algorithm to approximate SCT.*

Chapter 4

Summary

At the beginning the size-change principle was introduced as a sufficient condition to conclude termination of functional programs. Starting at an ω -automata based way to decide SCT the PSPACE-completeness proof for SCT was sketched to demonstrate its very high complexity. This was the motivation for two approximations. Based on graph-theory, this approximations differed only in their manner to determine anchors. The first one was only applicable to fan-in free size-change graphs whereas the second one could be applied to general size-change graphs.

Appendix A

Applied SCT and Extensions

A.1 An Implementation of SCT

An implementation of the size-change termination analysis (SCP₂ to be precise) can be found at <http://www.diku.dk/~panic/sct/>. A short “manual” for this web-tool follows, as the web-page itself does not offer to much insight.

The source language is a first order functional language with expressions as defined in Table A.1.

$\mathcal{E} ::=$	x
	c
	$c(\mathcal{E}, \dots, \mathcal{E})$
	$d(\mathcal{E})$
	if \mathcal{E} then \mathcal{E} else \mathcal{E}
	let $x = \mathcal{E} \dots x = \mathcal{E}$ in \mathcal{E}
	$f(x, \dots, x)$
	$p(x, \dots, x)$

Table A.1: Expressions in the source language

The components are described in the following:

- x : Used for *variables*. It can be any identifier (i.e. starting with a letter followed by letters and digits) that begins with a lower case letter.
- c : Used for *constants* (nullary functions) and *constructors*. It can be any identifier starting with a capitalised letter. The only form of constructors available are constants and tuples (of arbitrary arity). For example the natural numbers \mathbb{N} could be imitated as:

$$\begin{aligned} 0 &\sim \mathbf{Z} \\ 1 &\sim \mathbf{S}(\mathbf{Z}) \\ 2 &\sim \mathbf{S}(\mathbf{S}(\mathbf{Z})) \\ &\vdots \\ 3 - 1 &\sim \mathbf{P}(\mathbf{S}(\mathbf{S}(\mathbf{S}(\mathbf{Z})))) \end{aligned}$$

Where **Z** is used as 0 (zero), **S** as successor function ($n + 1$) and **P** as predecessor function ($n - 1$).

Another example would be lists of Booleans with constants **True**, **False** and **Nil** (for the empty list) and constructor **Cons**.

EXAMPLE A.1.1.

$$\begin{aligned} [] &\sim \text{Nil} \\ [0] &\sim \text{Cons}(\text{False}, \text{Nil}) \\ [1;0] &\sim \text{Cons}(\text{True}, \text{Cons}(\text{False}, \text{Nil})) \end{aligned}$$

d: Used for *destructors*. As the only type of constructors are tuples (and constants which cannot be destructed), there are predefined destructors for accessing elements of those tuples, namely

$$d \in \{\text{1st}, \text{2nd}, \text{3rd}, \text{4th}, \dots\}.$$

For the last list in Example A.1.1 this could be used to imitate the ‘head’ and ‘tail’ operations on lists as follows:

$$\begin{aligned} \text{hd}([1;0]) &\sim \text{1st}(\text{Cons}(\text{True}, \text{Cons}(\text{False}, \text{Nil}))) \\ \text{tl}([1;0]) &\sim \text{2nd}(\text{Cons}(\text{True}, \text{Cons}(\text{False}, \text{Nil}))) \end{aligned}$$

f: Used for *functions*. It can be any identifier starting with a lower case letter that is not used for *p* (i.e. is not a primitive operator).

p: Used for (predefined) *primitive operators*. Following operators could be extracted from the examples found on the web–page of the implementation (but there is no documentation easily available, thus it is not known whether the list is comprehensive).

and: Boolean *and* of two Boolean expressions. E.g. `and(gte(m, Z), not(eq(m, Z)))` is used for expressing $m \geq 0 \wedge m \neq 0$.

add: *Addition*. E.g. `add(S(Z), S(Z)) = S(S(Z))` is equivalent to $1+1 = 2$.

div: *Division*.

eq: This operator tests equality of the outermost constructors of two data values. For example testing whether a natural number **n** equals 0 could be done like: `eq(n, Z)`. Or testing if a list **ls** is empty like: `eq(ls, Nil)`.

equal: Tests whether the two arguments are equal.

error: Takes a constant (i.e. a capitalised identifier) as argument and terminates. It can be used to indicate that an error occurred during runtime. E.g. `if eq(y, Z) then error(DivisionByZero)` could be used within a routine for integer division to avoid an undefined division by zero.

gt: This operator tests whether the first operand is *greater than* the second one. For example testing whether a natural number n is greater than 0 is done like: `gt(n, Z)`.

gte: Greater than or equal.

lt: This operator tests whether the first operand is *less than* the second one.

lte: Less than or equal.

mul: *Multiplication*.

not: The unary Boolean *not* operation.

or: Boolean *or* of two Boolean expressions.

EXAMPLE A.1.2. In Listing A.1 an example for the Ackerman function can be seen. This program can be proven to be size-change terminating by the implementation above.

```
ack(m, n) = if eq(m, Z) then S(n) else
            if eq(n, Z) then ack(1st(m), S(Z))
            else ack(1st(m), ack(m, 1st(n)))
```

Listing A.1: ack

A second example (Listing A.2) demonstrates a program that implements tail-recursive list reversal.

```
rev(ls)      = rev_aux( Nil, ls)
rev_aux(a, ls) = if eq(ls, Nil) then a
                else rev_aux( Cons(1st(ls), a), 2nd(ls) )
```

Listing A.2: rev

For a third example, that cannot be proven to be size-change terminating see Listing A.3.

```
j(m, n) = if eq(add(m, n), Z) then Z else
          if lt(m, n) then j(1st(m), S(n))
          else j(S(m), 1st(n))
```

Listing A.3: Program that cannot be proven to be size-change terminating

There is an extension of SCT that is able to prove the termination of the program in Listing A.3. This extension is discussed in the next section.

A.2 Affine-Based Size-Change Termination

In [4] a translation of size-change graphs to affine-based graphs is proposed. Within an affine-based graph, affine relations among function parameters are expressed by *Presburger* formulæ. An *affine relation* is a property of the form $a_0 + \sum_{i=1}^k a_i x_i \text{ op } 0$ where x_1, \dots, x_k are program variables, a_0, \dots, a_k are integers, and $\text{op} \in \{=, <, \leq, >, \geq\}$.

DEFINITION A.2.1 (Presburger Formulæ). The kind of Presburger formula φ which is used in this section is of the form presented in Table A.2, where x_1, \dots, x_m and y_1, \dots, y_n are variables, χ is a Boolean formula, ϵ is a Boolean expression, and α is an arithmetic expression with $c \in \mathbb{N}$.

$\varphi ::=$	$\{[x_1, \dots, x_m] \rightarrow [y_1, \dots, y_n] \mid \chi\},$
$\chi ::=$	$\epsilon \mid \neg\chi \mid \exists x.\chi \mid \chi \vee \chi \mid \chi \wedge \chi,$
$\epsilon ::=$	$\mathbf{true} \mid \mathbf{false} \mid \alpha = \alpha \mid \alpha < \alpha,$
$\alpha ::=$	$c \mid x \mid c \cdot \alpha \mid \alpha + \alpha \mid -\alpha,$

Table A.2: Presburger formulæ

x_1, \dots, x_m are called *source parameters* and y_1, \dots, y_n *target parameters*. Following abbreviations are used:

$$\begin{aligned} \alpha \leq \beta &\sim \alpha < \beta \vee \alpha = \beta \\ \alpha > \beta &\sim \beta < \alpha \\ \alpha \geq \beta &\sim \beta < \alpha \vee \alpha = \beta \\ \alpha \neq \beta &\sim \neg(\alpha = \beta) \end{aligned}$$

This formulæ can be used to describe affine-based graphs (and also size-change graphs, as they are restricted variants of affine-based graphs).

EXAMPLE A.2.2. Some examples of affine relations for size-change graphs corresponding two Figure 2.1 and Figure 3.1(a) follow. Target parameters are replaced by there primed versions if they have the same name as source parameters (e.g. x' instead of x):

Figure 2.1(a): $\{[m, n] \rightarrow [m', n'] \mid m' < m\}$

Figure 2.1(b): Same as for Figure 2.1(a).

Figure 2.1(c): $\{[m, n] \rightarrow [m', n'] \mid m' = m \wedge n' < n\}$

Figure 3.1(a): $\{[x, y] \rightarrow [x', y'] \mid x' = x \wedge y' < x \wedge y' = y\}$

It is easy to see, that size-change graphs are very restrictive, as they only use:

$$\chi ::= \epsilon \mid \chi \wedge \chi$$

and

$$\epsilon ::= \mathbf{true} \mid y_i = x_j \mid y_i < x_j.$$

DEFINITION A.2.3 (Operations over affine relations). Following operations are defined over affine relations:

composition: The *composition* of affine relations is defined by:

$$\varphi_1 \circ \varphi_2 = \{(x, z) \mid \exists y : (x, y) \in \varphi_1 \wedge (y, z) \in \varphi_2\}$$

union: The *union* of two affine relations is defined by:

$$\varphi_1 \cup \varphi_2 = \{(x, y) \mid (x, y) \in \varphi_1 \cup (x, y) \in \varphi_2\}$$

transitive closure: The *transitive closure* of an affine relation is defined by:

$$\varphi^+ = \bigcup_{i \geq 0} \varphi^i$$

where φ^i means composing φ with itself i times.

DEFINITION A.2.4 (Affine-Based Graphs). An affine-based graph is a size-change graph where the set of labels \mathcal{L} is ignored and each destination parameter is constrained by the source parameters arranged in an affine relation.

EXAMPLE A.2.5. For the program from Listing A.3, there are the two size-change graphs shown in Figure A.1.

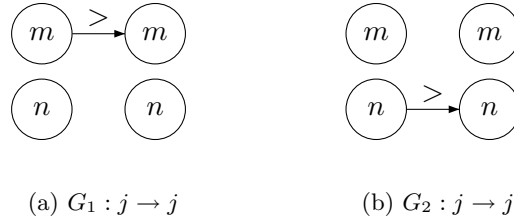


Figure A.1: Size-change graphs for the program from Listing A.3

Corresponding affine-based graphs (represented non-graphically) would be:

$$\Gamma_1 = \{[m, n] \rightarrow [m', n'] \mid m' = m - 1 \wedge n' = n + 1\}$$

and

$$\Gamma_2 = \{[m, n] \rightarrow [m', n'] \mid m' = m + 1 \wedge n' = n - 1\}$$

It can be seen, that with the help of affine-based graphs more accurate information about the size relations among parameters can be captured.

“Presburger formulæ, or affine relations in particular, may be a good candidate for encoding size-change graphs, for the following three reasons:

1. They allow the capturing of constant changes in parameter size. This allows the effect of constant increment and decrement to be cancelled out during the analysis.

2. They can express size change of a destination argument by a linear combination of source parameters. This enables more accurate representation of size change.
3. They can constrain the size change information with information about call context, thus naturally extending the analysis to be context-sensitive. The LJB-analysis method ignores test conditions, but these can be expressed naturally using Presburger constraints.”

The above paragraph has been taken from [4]. With LJB-analysis the analysis presented in [1] is denoted.

Note: There is a translation function from the set of size-change graphs to the set of affine-based graphs and an abstraction in the other direction as can be found in [4].

A.3 An Example for Affine-Based Size-Change Termination

Consider the functional program of Listing A.4.

```
f(m) = if m <= 0 then 1 else 1:g(m + 1)
g(n) = if n <= 0 then 1 else 2:f(n - 2)
```

Listing A.4: Constant change cancellation

The corresponding input for the SCP₂ implementation could be written as shown in Listing A.5.

```
f(m) = if lte(m, Z) then S(Z) else g(S(m))
g(n) = if lte(n, Z) then S(Z) else f(1st(1st(n)))
```

Listing A.5: Input for SCP₂

But it is not possible to prove termination with this approach. As there are only the two size-change graphs to be found in Figure A.2.

The corresponding affine-graphs that can be extracted from the given program are:

$$\Gamma_1 = \{[m] \rightarrow [m'] \mid m' = m + 1\}$$

and

$$\Gamma_2 = \{[n] \rightarrow [n'] \mid n' = n - 2\}$$

The analysis presented in [4] is done by starting from a set of affine-based graphs that were extracted from the given program (i.e. Γ_1 and Γ_2 for the given example). Then some kind of closure for this set is computed. Once this set is stable, every *idempotent* graph is checked for an decreasing parameter. If this condition is satisfied, the given program is size-change terminating. For details on this algorithm see [4].



Figure A.2: Size-change graphs for Listing A.4

In the example the only two idempotent graphs after the closure algorithm are:

$$\Gamma_3 = \{[n] \rightarrow [n'] \mid n' < n\}$$

and

$$\Gamma_4 = \{[m] \rightarrow [m'] \mid m' < m\}.$$

which both have decreasing parameters. So the given program *is* size-change terminating, which could not be proven by the LJB-analysis.

Appendix B

Theoretical Background

B.1 A First Order Functional Language \mathcal{L}_1^f

A first order functional language operates on data structures that have a well-founded size measure (e.g. the natural numbers with their values as sizes, lists of natural numbers with the list-lengths as sizes etc.).

DEFINITION B.1.1. A *signature* is a set \mathcal{F} of *function symbols*. \mathcal{F} is the disjoint union of the two sets $\mathcal{F}_{\mathcal{N}}$ of *newly defined function symbols* and $\mathcal{F}_{\mathcal{P}}$ of *primitive function symbols*. Associated with every $f \in \mathcal{F}$ is a natural number denoting its *arity*, i.e. the number of *parameters* it has.

DEFINITION B.1.2. Let \mathcal{F} be a signature, \mathcal{V} a set of *variables*, and \mathcal{C} a set of *call labels* disjoint from each other. The set $\mathcal{E}(\mathcal{F}, \mathcal{V})$ of *expressions* built from \mathcal{F} and \mathcal{V} is defined inductively as follows:

$$\frac{}{x \in \mathcal{V}}, \quad \frac{e_1 \quad e_2 \quad e_3}{\mathbf{if } e_1 \mathbf{ then } e_2 \mathbf{ else } e_3},$$
$$\frac{e_1 \quad \dots \quad e_n}{p(e_1, \dots, e_n)} p \in \mathcal{F}_{\mathcal{P}}, \quad \frac{e_1 \quad \dots \quad e_n}{c:f(e_1, \dots, e_n)} f \in \mathcal{F}_{\mathcal{N}} \text{ and } c \in \mathcal{C}$$

DEFINITION B.1.3. *New functions* are defined like:

$$f(x_1, \dots, x_n) = e$$

such that $f \in \mathcal{F}_{\mathcal{N}}$ is the function that is defined, $\text{Par}(f) = \{x_1, \dots, x_n\} \subseteq \mathcal{V}$ are the *parameters* of f , and $e \in \mathcal{E}(\mathcal{F}, \text{Par}(f))$ is the *body* of f .

DEFINITION B.1.4. A *program* is a sequence of function definitions such that no newly defined function symbol is used twice as the function that is defined. The *initial function* is the first function defined in a program (denoted by f_0).

For a given program p let $\mathcal{C}_p \subseteq \mathcal{C}$ denote the set of call labels occurring in p . Equivalently define \mathcal{V}_p , \mathcal{F}_p , $\mathcal{F}_{\mathcal{N}p}$, and $\mathcal{F}_{\mathcal{P}p}$ as the variables, function symbols, newly defined function symbols, and primitive function symbols occurring in p respectively.

EXAMPLE B.1.5. Here are two example–programs. One computes the Ackerman function and the other is a tail–recursive version of a list–length function:

```
ack(m, n) = if m = 0 then n + 1 else
            if n = 0 then 1 : ack(m - 1, 1)
            else 2 : ack(m - 1, 3 : ack(m, n - 1))
```

Listing B.1: ack

```
len(ls)      = 1 : len_aux(0, ls)
len_aux(n, ns) = if ns = [] then n
                else 2 : len_aux(n + 1, tl ns)
```

Listing B.2: len

B.2 *–Notation and ω –Notation

DEFINITION B.2.1. For any set S , define S^* to be the set of all *finite* sequences over S and define S^ω to be the set of all *infinite* sequences over S . Define $S^{*\omega} = S^* \cup S^\omega$. For elements of either S^* or S^ω the notation s_1, s_2, s_3, \dots is used, whereas $s_1, s_2, s_3, \dots, s_n$ denotes elements of S^* .

Let $|s_1, s_2, s_3, \dots|_s$ denote the number of occurrences of s in s_1, s_2, s_3, \dots . $|s_1, s_2, s_3, \dots|_s = \infty$ expresses that s occurs infinitely often in the given (necessarily) infinite sequence.

B.3 ω –Automata

DEFINITION B.3.1. A *Büchi automaton* is a tuple $\mathcal{B} = (\Sigma, S, I, \rho, A)$ such that Σ is a finite *input alphabet*, S is a finite set of *states*, $I \subseteq S$ is the set of *initial states*, $A \subseteq S$ is the set of *accepting states*, and $\rho \subseteq S \times \Sigma \times S$ is the *transition relation*. Often $s \xrightarrow{\sigma} s'$ is written instead of $(s, \sigma, s') \in \rho$.

DEFINITION B.3.2. A *run* of a ω –automaton \mathcal{B} on an infinite word $\sigma_1\sigma_2\sigma_3\cdots \in \Sigma^\omega$ (see Definition B.2) is a transition sequence $s_1 \xrightarrow{\sigma_1} s_2, s_2 \xrightarrow{\sigma_2} s_3, \dots \in \rho^\omega$ (also written as $s_1 \xrightarrow{\sigma_1} s_2 \xrightarrow{\sigma_2} s_3 \xrightarrow{\sigma_3} \dots$).

A run π is *accepting* if and only if some $s \in A$ occurs infinitely often in it, i.e. $|\pi|_s = \infty$ (see Definition B.2) for some $s \in A$.

DEFINITION B.3.3. The *language accepted by \mathcal{B}* (denoted by $L_\omega(\mathcal{B})$) is the set of all infinite words w over Σ such that there is an accepting run on w .

Languages are called *ω –regular* if and only if they are accepted by some ω –automaton.

B.4 Infinite Ramsey’s Theorem

In the following a version of Ramsey’s Theorem is presented that is needed in the proof of a central theorem. This version is not as general as the one that is usually referred to. The restriction takes place in the fact that only

ordered pairs of members are considered whereas the general version considers k -member subsets with $k \in \mathbb{N}$.

THEOREM B.4.1 (Ramsey's Theorem, Restricted Version). *Let M be an infinite set and $n \in \mathbb{N}$. Suppose that the set of all ordered pairs $(m, m') \in M \times M$ with $m < m'$ (denoted by $M_{<}^2$) is partitioned into n classes. Then there is an infinite subset of M such that all ordered pairs (like above) built from this subset are in the same class.*

Proof. Let $M_{<}^2 = \{(m, m') \in M \times M \mid m < m'\}$ and the above partition be denoted by $P_{M_{<}^2}$. The n classes of the partition $P_{M_{<}^2}$ are denoted by $P_1, P_2, P_3, \dots, P_n$. Each $(m, m') \in M_{<}^2$ is in exactly one of $P_{M_{<}^2}$'s classes. Select an element m_0 of M and consider all pairs (m_0, m') that are in $M_{<}^2$. There is a class P_{k_0} such that the set

$$M_1 = \{m' \in M \mid (m_0, m') \in P_{k_0}\}$$

is infinite. As next step select an element m_1 of M_1 and consider all pairs $(m_1, m') \in M_{<}^2$. There is a class P_{k_1} such that the set

$$M_2 = \{m' \in M_1 \mid (m_1, m') \in P_{k_1}\}$$

is infinite. Etc.

So three infinite sequences are obtained:

- The sequence of elements of M : m_0, m_1, m_2, \dots
- The sequence of classes: $P_{k_0}, P_{k_1}, P_{k_2}, \dots$
- The sequence of subsets: $M = M_0 \supset M_1 \supset M_2 \supset \dots$

Where m_i is in $M_i \setminus M_{i+1}$ and all pairs (m_i, m') with $m' \in M_{i+1}$ are in the class P_{k_i} . One of the classes (denote it by P_k) occurs infinitely often in the above sequence: $P_k = P_{k_i}$ for $i = i_0, i_1, i_2, \dots$. The set of corresponding elements:

$$H = \{m_i \mid i = i_0, i_1, i_2, \dots\}$$

is an infinite subset of M and all pairs $(m, m') \in H_{<}^2$ are in the same class P_k . Indeed, if $m = m_i$ and $m' = m_j$, where $i = i_k, j = i_m$ and $k < m$, then m is in $M_i \setminus M_{i+1}$ and m' is in M_{i+1} . Hence (m, m') is in the class P_{k_i} , i.e. in P_k . \square

B.5 BOOLEANPROGRAM

First Boolean programs are defined.

DEFINITION B.5.1 (Boolean Program). A *Boolean program* b is an instruction sequence $1 : I_1 \ 2 : I_2 \ 3 : I_3 \ \dots \ m : I_m$ specifying a computation on k Boolean variables X_1, \dots, X_k . There are only two instructions:

1. $X_i := \text{not } X_i$
2. **if** X_i **then goto** l' **else** l''

DEFINITION B.5.2. The *computation* by b is a possibly infinite state sequence $(l_1, \sigma_1), (l_2, \sigma_2), \dots$, where each store σ assigns truth values in $\{\mathbf{true}, \mathbf{false}\}$ to each of b 's variables and l_t is the control point at time t .

Initially $l_1 = 1$ and σ_1 assigns **false** to every variable. The semantics of the two instructions are as follows. During instruction 1 the variable X_i is negated (i.e. **true** goes to **false** and vice versa) and the control point is incremented by one (i.e. the next line of the program is executed). Instruction 2 is a conditional jump depending on the value of the variable X_i . If X_i is **true** then the control flow resumes at control point l' else it resumes at l'' .

A Boolean program b *terminates* if at any point in the computation the control point is 0, where also a jump beyond the last line is considered as 0 (i.e. if its computation is $(l_1, \sigma_1), \dots, (l_t, \sigma_t) = (0, \sigma_t)$ for some t).

The set of all *terminating* Boolean programs is defined by:

$$\text{BOOLEANPROGRAM} = \{b \mid b \text{ is a Boolean program and } b \text{ terminates}\}$$

LEMMA B.5.3. BOOLEANPROGRAM is in PSPACE (see also [6]).

Proof. There is a Turing machine \mathcal{T} that decides for an arbitrary Boolean program b if it is in BOOLEANPROGRAM . The machine \mathcal{T} simulates the program b using at most space proportional to $k + \lceil \log m \rceil$ (i.e. k bits for the values of the variables X_1, \dots, X_k and an $\lceil \log m \rceil$ -bit integer to store the current control point). \mathcal{T} would of course loop infinitely if the interpreted program does so. Thus some additions are needed.

Let the interpreted program b have m labels (i.e. its length is m) and k Boolean variables. Then it can enter at most $m \cdot 2^k$ configurations without repeating one and thus looping.

To ensure termination of \mathcal{T} a binary counter c with $r = k \cdot \lceil \log m \rceil$ bits is used. It is incremented every time an instruction of b has been simulated. If c becomes $2^r - 1$ then \mathcal{T} stops and answers **No**. If b terminates before the counter reaches that value then \mathcal{T} stops and answers **Yes**. \square

It remains to show that BOOLEANPROGRAM is hard for PSPACE (i.e. every computational problem within PSPACE can be reduced to BOOLEANPROGRAM).

LEMMA B.5.4. Let \mathcal{C} be a counter machine running in polynomial space $f(n)$ and let $d = a_1 \dots a_n \in \{0, 1\}^*$ be an input of length n . Then there exists a Boolean program b such that b terminates if and only if $\mathcal{C}(d)$ terminates. Further b can be constructed from \mathcal{C} and d in space $\mathcal{O}(\log n)$.

A counter machine is very similar to a register machine. For a detailed explanation see [6].

Proof. The proof is done in two steps:

1. There is a polynomial π such that for any input d an input-free counter machine \mathcal{C}^d with counters C_1, \dots, C_k can be built in space $\mathcal{O}(\log |d|)$ such that:
 - (a) \mathcal{C}^d terminates if and only if $\mathcal{C}(d)$ terminates; and

(b) if \mathcal{C}^d terminates, then its counter values are bounded by $\pi(n)$.

For this construction see [6] chapter 28.

- The next step is, to construct from $\mathcal{C}^d = \mathbb{I}_1 \dots \mathbb{I}_m$ a Boolean program b such that b terminates if and only if \mathcal{C} terminates on input d . It has the form:

$$b = 1 : \mathbb{I}'_1 \dots m : \mathbb{I}'_m$$

and variables C_i^j for $i \in [1, k], j \in [0, \pi(n)]$ with the intended interpretation: variable C_i^j is true exactly when the j -th bit of counter C_i is a 1. Instructions \mathbb{I}'_l to simulate instruction \mathbb{I}_l of \mathcal{C}^d are defined as in Table B.1.

Counter machine instruction \mathbb{I}_l	BOOLEANPROGRAM instructions \mathbb{I}'_l
$C_i := C_i + 1$	if $\neg C_i^0$ then $C_i^0 := \text{true}$ else $\{C_i^0 := \text{false}; \text{if } \neg C_i^1 \text{ then } C_i^1 := \text{true} \text{ else}$ $\{C_i^1 := \text{false}; \dots \text{ else}$ $\{C_i^{\pi(n)-1} := \text{false}; C_i^{\pi(n)} := \text{true}; \} \dots \}$
$C_i := C_i - 1$	if C_i^0 then $C_i^0 := \text{false}$ else $\{C_i^0 := \text{true}; \text{if } C_i^1 \text{ then } C_i^1 := \text{false} \text{ else}$ $\{C_i^1 := \text{true}; \dots \text{ else if } C_i^{\pi(n)} \text{ then}$ $C_i^{\pi(n)} := \text{false}; \} \dots \}$
if $C_i = 0$ goto l' else l''	if $C_i^0 \vee \dots \vee C_i^{\pi(n)}$ goto l' else l''

Table B.1: Simulating a counter machine as Boolean program

□

THEOREM B.5.5. BOOLEANPROGRAM is complete for PSPACE.

Proof. If a problem P is in PSPACE then it is decidable by some polynomially space-bounded counter machine program \mathcal{C} . This program can be modified in a way, that it only terminates on acceptance. Thus the construction above can be used to reduce P to BOOLEANPROGRAM. □

B.6 Some Graph-Theory

DEFINITION B.6.1 (Strongly-Connected Graphs). A graph $G = (V, E)$ is *strongly-connected* if for all $u, v \in V$, v is reachable from u (i.e. there is a path from u to v).

DEFINITION B.6.2 (Strongly-Connected Components). For any graph G define $\text{Scc}(G)$ to be the set of maximal strongly-connected subgraphs of G (also called strongly-connected components of G).

DEFINITION B.6.3 (Restricted Graphs). For a graph $G = (V, E)$ and $V' \subseteq V$ let $G|_{V'}$ denote the *restriction* of G to the vertices in V' , defined by:

$$G|_{V'} = (V', \{(v, w) \in E \mid v, w \in V'\})$$

Bibliography

- [1] Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amran. The size-change principle for program termination. In *Proceedings of the ACM Symposium on Principles of Programming Languages, January 2001*, volume 28, pages 81–92. ACM press, January 2001.
- [2] Chin Soon Lee. Program termination analysis in polynomial time. Technical report, DIKU, Denmark, 2002.
- [3] Carl Christian Frederiksen. *A simple implementation of the size-change termination principle (revised)*. February 2001.
- [4] Hugh Anderson and Siau-Cheng Khoo. Affine-based size-change termination. In Atsushi Ohori, editor, *Proceedings of the First Asian Symposium on Programming Languages and Systems, APLAS 2003, Beijing, China*, volume 2895 of *Lecture Notes in Computer Science*, pages 122–140. Springer, 2003.
- [5] Christos H. Papadimitriou. *Computational Complexity*. Addison–Wesley, 1994.
- [6] Neil D. Jones. *Computability and Complexity from a Programming Perspective*. Foundations of Computing. MIT Press, 1997.