A Simple Implementation of the Size-Change Termination Principle (Revised)

Carl Christian Frederiksen

February 28, 2001

Contents

1 Abstract

Based on the "size-change termination" principle [3] for first order functional languages, an algorithm for deciding whether a given subject program terminates for all input, is developed. The algorithm is fully automatic, terminates for all subject programs and is sufficiently strong to handle nontrivial examples.

A comparison is made between the results obtainable by the algorithm and other termination analyses. Terminating programs, for which the program fails to decide termination, are examined in order to find possible improvements to the algorithm. The goal is to illustrate the strengths and weaknesses of the algorithm.

2 Introduction

The work in this paper is based on David Wahlstedts Master's thesis [7]. In the thesis, a program for deciding size-change termination is presented, based on the size-change termination principle of [3]. The language accepted is a small first order language with Base-1 numbers as the only values and case expressions for deconstruction. In order to make the extraction of size-change graphs simple, a number of restrictions are made to the syntax of the programs accepted. This paper aims to implement the size-change analysis for a more realistic first order functional programming language with constructors of arbitrary arity. This extension will allow a more direct comparison to previous work.

2.1 The size-change termination principle

The size-change termination *principle* of [3] consists of two distinct phases. First a set of *size-change graphs* is extracted from the subject program. For a given call site[1], the corresponding size-change graph safely describes the changes over the size of the parameters from the calling function to the called function. The size relation is assumed to be a well founded ordering on the domain of values in the program such that the size-change termination criterion applies [3]:

> If every infinite computation trace gives rise to a infinitely strictly decreasing value tread among the size-change graphs, then no infinite computation is possible (since the ordering is well founded).

In the second phase, the above criterion is applied. All the possible finite and non-repetitive compositions of the size-change graphs are found. The composed graphs describe all possible call sequences from a function to itself. If the program generates an infinite computation trace for some input, then at least one function `f` must appear an infinite number of times in the computation trace. If there exists a composed size-change graph from `f` to itself with a strict decrease in some variable, then the corresponding computation trace has an infinitely

[1] Call site: Each function call appearing in the program text is called a *call site*.

decreasing thread. Thus the computation trace will never occur in an actual computation, because of the well foundedness of the ordering the the domain of values. If there, for all functions in the program, exists a (composed graph) from the function to itself with decrease in one parameter, then the program terminates.

If a program can be shown to terminate by this principle (for any safe size-change graph extraction function and any well founded ordering on the values of the language), then the program is said to be *size-change terminating*.

2.2 The size-change graph extraction

One goal of the paper is the implementation of a size-change termination analysis. The implementation is based on the work of Wahlstedt [7]. Since the language is insufficiently strong to handle more complex examples (examples using lists or trees, for instance) without heavy encoding, a new language has been developed. The implementation in this paper reuses the second phase of the size-change termination principle from Wahlstedts algorithm (as well as some auxiliary functions). The first phase of the algorithm - the size-change graph extraction - has been written for the new language.

Another goal of this paper is to investigate to what extent the ability to extract precise size-change graphs influences the ability to decide size-change termination. In particular two different (rather simple) extraction methods are presented, both based on an analysis of the sizes of the values computed in the program relative to the function parameters. The first algorithm completely ignores function calls, and has thus no information about the return values of functions. The second algorithm performs an analysis of the return values of functions, and is therefore able to extract more precise information about the program.

The implemented algorithms are run on a collection of test suites [2, 7, 3], in order to compare the power of the size-change termination principle to other termination analyses. The interesting cases of the results of the analysis are examined in greater detail, in order to show what can and can not be done with the simple size-change graph extraction methods.

2.3 The remainder of this paper

The language for the subject programs is introduced in Section 3. Sections 4 and 5 discusses the size-change termination analysis and the size-change termination algorithms in this paper. Size-change graph extraction is easily done by a small extension of the size analysis algorithms. The implementation and the results of the test suites are discussed in Section 6. The interesting cases are studied in more detail, exposing the shortcomings of the current implementation. Section 7 concludes with a summary and open problems. The appendix contains the test suites and test results.

3 The language

In this section we present the language L used in the experiments. The presentation follows the presentation in [3]. One of the goals of the design of the language was to extend it to include a binary constructor, which would make the translation of test suites in to the language easier. In the final design we have settled for a language with constructors of arbitrary (but fixed) arity and names. In this way symbols can be represented directly as a 0-ary constructor, without the hassles of encoding symbols and de-constructing them for equality testing as would be necessary if the only data type in the language were lists. At the same time Base-1 numbers can be written directly, allowing for a direct representation of the examples in [7]. But most importantly the constructors allow the representation of trees, allowing Lisp-like examples to be represented more directly.

In the paper [7] a pattern matching notation was used for de-constructing values. We have settled for explicit deconstruction functions[2] since the more liberal notation allows for greater flexibility when translating from various languages. Inspired by [4] the names `1st`, `2nd`, `3rd`, etc. have been chosen as destructor names. The "case" matching is done by the primitive function `eq`, that tests whether the two arguments have the same top-level constructor.

The syntax for the language L.

$\mathtt{p} \in \mathit{Prog}$	::=	$\mathtt{def}_1 \ldots \mathtt{def}_n$
$\mathtt{def} \in \mathit{Def}$	::=	$\mathtt{fname}(\mathtt{x}_1, \ldots, \mathtt{x}_n) = \mathtt{e}^{\mathtt{f}}$
$\mathtt{e} \in \mathit{Expr}$	::=	$\mathtt{x}$
	|	$\mathtt{con}$
	|	$\mathtt{con}(\mathtt{e}_1, \ldots, \mathtt{e}_n)$
	|	$\mathtt{des}(\mathtt{e})$
	|	$\mathtt{if}\ \mathtt{e}_1\ \mathtt{then}\ \mathtt{e}_2\ \mathtt{else}\ \mathtt{e}_3$
	|	$\mathtt{let}\ \mathtt{x}_1 = \mathtt{e}_1 \ldots \mathtt{x}_n = \mathtt{e}_n\ \mathtt{in}\ \mathtt{e}_0$
	|	$\mathtt{f}(\mathtt{e}_1, \ldots, \mathtt{e}_n)$
	|	$\mathtt{op}(\mathtt{e}_1, \ldots, \mathtt{e}_n)$
$\mathtt{x} \in \mathit{Variable}$	::=	identifier beginning with lower case
$\mathtt{f} \in \mathit{FcnName}$	::=	identifier beginning with lower case, not in *Op*
$\mathtt{con} \in \mathit{Constructor}$	::=	capitalized identifier
$\mathtt{des} \in \mathit{Destructor}$	::=	$\{\mathtt{1st}, \mathtt{2nd}, \mathtt{3rd}, \ldots\}$
$\mathtt{op} \in \mathit{Op}$	::=	primitive operator

The domain of values in the language is denoted *Value*, and is defined by

$$\mathit{Value} ::= \mathtt{con} \mid \mathtt{con}(\mathit{Value}_1, \ldots, \mathit{Value}_n).$$

Note that the same names are used for the constructor operators and the constructed values.

[2] As in Scheme.

$v \in \mathit{Value}$ (a flat domain).

$u, w \in \mathit{Value}^\sharp = \mathit{Value} \cup \{\mathit{Error}, \bot\}$.

$$
\begin{array}{lcll}
[\![\cdot]\!]\cdot & : & \mathit{Prog} \to \mathit{Value}^* \to \mathit{Value}^\sharp & \\
\mathcal{E} & : & \mathit{Expr} \to (\mathit{Variable} \to \mathit{Value}) \to \mathit{Value}^\sharp & \\
\mathcal{B} & : & \mathit{Expr} \to (\mathit{Variable} \to \mathit{Value}), \mathit{Variable}^* \to \mathit{Value}^* \to \mathit{Value}^\sharp & \\
\mathcal{O} & : & \mathit{Op} \to \mathit{Value}^* \to \mathit{Value} & \\
\mathcal{C} & : & \mathit{Constructor} \to \mathit{Value}^* \to \mathit{Value}^\sharp & \\
\mathcal{D} & : & \mathit{Destructor} \to \mathit{Value} \to \mathit{Value}^\sharp & \\
\mathit{lift} & : & \mathit{Value} \to \mathit{Value}^\sharp & \text{(the natural injection)} \\
\mathit{strictapply} & : & (\mathit{Value}^* \to \mathit{Value}^\sharp) \to (\mathit{Value}^\sharp)^* \to \mathit{Value}^\sharp &
\end{array}
$$

$$[\![\mathtt{prog}]\!]\vec{v} = \mathcal{E}[\![\mathtt{e}_{\mathtt{f}_{initial}}]\!][\mathtt{f}_{initial}{}^{(i)} \mapsto v_i^{\,\forall i \in \{1,\ldots,arity(\mathtt{f}_{initial})\}}]$$

$$
\begin{array}{ll}
\mathcal{E}[\![\mathtt{x}]\!]\sigma & = \sigma(\mathtt{x}) \\
\mathcal{E}[\![\mathtt{con}]\!]\sigma & = \mathit{lift}(\mathtt{con}) \\
\mathcal{E}[\![\mathtt{con}(\mathtt{e}_1, \ldots, \mathtt{e}_n)]\!]\sigma & = \mathit{strictapply}(\mathcal{C}[\![\mathtt{con}]\!])(\mathcal{E}[\![\mathtt{e}_1]\!]\sigma, \ldots, \mathcal{E}[\![\mathtt{e}_n]\!]\sigma) \\
\mathcal{E}[\![\mathtt{des}(\mathtt{e})]\!]\sigma & = \mathit{strictapply}(\mathcal{D}[\![\mathtt{des}]\!])(\mathcal{E}[\![\mathtt{e}]\!]\sigma) \\
\mathcal{E}[\![\mathtt{if\ e}_1\ \mathtt{then\ e}_2\ \mathtt{else\ e}_3]\!]\sigma & = \mathcal{E}[\![\mathtt{e}_1]\!] \to (\mathcal{E}[\![\mathtt{e}_2]\!]\sigma, \mathcal{E}[\![\mathtt{e}_3]\!]\sigma)
\end{array}
$$

$$
\begin{array}{l}
\mathcal{E}[\![\mathtt{let\ x}_1 = \mathtt{e}_1 \ldots \mathtt{x}_n = \mathtt{e}_n\ \mathtt{in\ e}_0]\!]\sigma\ = \\
\qquad \mathit{strictapply}(\mathcal{B}(\mathtt{e}_0, \sigma)[\![(\mathtt{x}_1, \ldots, \mathtt{x}_n)]\!])(\mathcal{E}[\![\mathtt{e}_1]\!]\sigma, \ldots, \mathcal{E}[\![\mathtt{e}_n]\!]\sigma) \\
\mathcal{E}[\![\mathtt{f}(\mathtt{e}_1, \ldots, \mathtt{e}_n)]\!]\sigma = \\
\qquad \mathit{strictapply}(\mathcal{B}(\mathtt{e}^{\mathtt{f}}, [])[\![(\mathtt{f}^{(1)}, \ldots, \mathtt{f}^{(n)})]\!])(\mathcal{E}[\![\mathtt{e}_1]\!]\sigma, \ldots, \mathcal{E}[\![\mathtt{e}_n]\!]\sigma) \\
\mathcal{E}[\![\mathtt{op}(\mathtt{e}_1, \ldots, \mathtt{e}_n)]\!]\sigma = \\
\qquad \mathit{strictapply}(\mathcal{O}[\![\mathtt{op}]\!])(\mathcal{E}[\![\mathtt{e}_1]\!]\sigma, \ldots, \mathcal{E}[\![\mathtt{e}_1]\!]\sigma)
\end{array}
$$

$$\mathcal{C}[\![\mathtt{con}]\!](v_1, \ldots, v_n) = \mathtt{con}(v_1, \ldots, v_n)))$$

$$\mathcal{D}[\![\mathtt{des}]\!]v = \begin{cases} v_{index(\mathtt{des})} & \text{if } v = \mathtt{con}(v_1, \ldots, v_n) \wedge \mathit{arity}(\mathtt{des}) \leq n \\ \mathtt{Error} & \text{otherwise.} \end{cases}$$

$$u \to (w, w') = \begin{cases} u & \text{if } u \in \{\mathtt{Error}, \bot\}, \\ w & \text{if } u = \mathtt{True}, \\ w' & \text{otherwise.} \end{cases}$$

$$\mathcal{B}(\mathtt{e}, \sigma)[\![(\mathtt{x}_1, \ldots, \mathtt{x}_n)]\!](v_1, \ldots, v_n) = \mathcal{E}[\![\mathtt{e}]\!]\sigma[\mathtt{x}_i \mapsto v_i^{\,\forall i \in \{1,\ldots,n\}}]$$

$$\mathcal{O}[\![\mathtt{eq}]\!](v_1, v_2) = \begin{cases} \mathtt{True} & \text{if } v_1 = \mathtt{con}(v'_1, \ldots, v'_n) \wedge v_2 = \mathtt{con}(v''_1, \ldots, v''_n), \\ \mathtt{False} & \text{otherwise.} \end{cases}$$

$$\mathit{strictapply}\ \psi\ w = \begin{cases} \psi(v_1, \ldots, v_n) & \text{if } w_i \notin \{\mathtt{Error}, \bot\} \text{ for } i \in \{1, \ldots, n\}, \\ & \text{where } w_i = \mathit{lift}\ v_i \text{ for } i \in \{1, \ldots, n\}. \\ w_i & \text{otherwise, where } i = \text{least index} \\ & \text{such that } w_i \in \{\mathtt{Error}, \bot\}. \end{cases}$$

Figure 3.1: Semantics of L programs. `True` *and* `Error` *are distinguished elements of Value.*

Program. A program definition is a sequence of function definitions, and has the form $\mathtt{def}_1 \ldots \mathtt{def}_k$ for some $k \geq 1$. The *entry function* is the function defined in the first function definition of a program, and is denoted $\mathtt{f}_{initial}$. A function definition has the form $\mathtt{f}(\mathtt{x}_1, \ldots, \mathtt{x}_n) = \mathtt{e}^{\mathtt{f}}$, where $\mathtt{e}^{\mathtt{f}}$ is called the *function body* of f. The number of parameters n is called the *arity* of the function, denoted *arity*(f). The *parameters* of a function are written: $Param(\mathtt{f}) = \{\mathtt{f}^{(1)}, \ldots, \mathtt{f}^{(arity(\mathtt{f}))}\}$. Variables are assumed to be in scope when evaluated.

Call Site. For ease of reference each function call will be associated with a unique number, denoting its *call site*. The Call sites numbered by the order they are encountered in the program text by scanning the program left-right, top-down. The set of call sites in a program tt p is denoted $C_{\mathtt{p}}$. A given call from function f to function g occurring in a program p at call site c, is denoted $\mathtt{f} \xrightarrow{c} \mathtt{g}$.

Call Sequence. A *call sequence* is a finite or infinite sequence $cs = c_1 c_2 \ldots \in C_{\mathtt{p}}^{*\omega}$ of call sites in a given program p. A call sequence is said to be *well formed* if there exists a sequence of functions $\mathtt{f}_1\mathtt{f}_2 \ldots$ in p such that $\mathtt{f}_1 \xrightarrow{c_1} \mathtt{f}_2 \xrightarrow{c_2} \mathtt{f}_3 \ldots$ is a sequence of function calls.

Semantics. The semantics of the language is a standard call-by-value semantics, and is given in Figure 3.1. Note that the notation x_i is used to select the i'th element in a tupple x. Also note that in a program "con" is an application of a constructor operator, so Cons(Nil, Nil) is an application of a constructor operator that evaluates to a constant, and not a constant. The number of subterms in a constant c is called the *a*rity of the constant, denoted *arity*(c). It should be clear from the context whether *arity* refers to the arity of a constant or a function. Constructor names are unique and must be used with the same fixed arity throughout the program. This can easily be verified by a syntactic check.

For a destructor operator des the *i*ndex of the destructor indicates the number of the subtree to be extracted from the top-level constructor. Example: *index*(3rd) = 3. Run time errors are modeled by the constant Error. If a subcomputation evaluates to Error, then the result of the entire computation is Error. Note that the 0-ary constructor Error evaluates to the constant Error.

The language contains a single primitive operator eq, that tests whether the topmost constructors of each of the two arguments has the same constructor name. Since a given constructor name can only be used with a fixed arity for the entire program, name equality implies arity equality. Other primitive functions can be added by extending the $\mathcal{O}$ function. The only requirement is that a primitive function never fails, ie. always terminates and does not return Error.

State Transition. A *state* is a pair in $State = FcnName \times Value^*$. A *state transition* is the relation $(\mathtt{f}, \vec{v}) \xrightarrow{c} (\mathtt{g}, \vec{w})$ where the call $\mathtt{g}(\mathtt{e}_1, \ldots, \mathtt{e}_n)$ occurs in

the body of $\mathtt{f}$ at call site c such that $\vec{w} = (w_1, \ldots, w_n)$ and $\mathcal{E}[\![\mathtt{e}_i]\!]\vec{v} = w_i$ for all $i \in \{1, \ldots, n\}$.

State transition sequence. A *State transition sequence* is a sequence of state transitions $(\mathtt{f}_i, \vec{v_i} \xrightarrow{c_i} (\mathtt{f}_{i+1}, \vec{v_{i+1}})$. Such a sequence is denoted

$$(\mathtt{f}_1, \vec{v_1}) \xrightarrow{c_1} (\mathtt{f}_2, \vec{v_2}) \xrightarrow{c_1} (\mathtt{f}_3, \vec{v_3}) \ldots$$

The calls occurring in the state transition sequence cs are denoted $\mathit{calls}(sts)$.

4 Termination analysis

The size-change termination analysis consists of two phases: Size-change graph extraction and detection of call sequences with no infinite descent.

For a given call site, the size-change analysis describes the size-changes of a given parameter relative to the input. The size-change graph for a call site is simply given by the size-changes of the arguments relative to the parameters.

In order to determine termination by the size-change principle, a *size-change graph* must be computed for each call site. A size-change graph for a call site safely describes the size changes over some size-measure from the parameters of the calling function to the parameters of the called function.

For simplicity it will be assumed that each call site in program has been annotated with its call site number (see page 3).

In order to describe size-changes a size ordering is needed on the values in the language. The approach taken in this paper is to decide on a *size-measure* on the values. The size-measure maps values to a natural number describing its size. The usual ordering on the natural numbers will induce a well-founded ordering on the *Value* domain. With the size-measure defined, the size-relations between values in the program should be more intuitively clear.

4.1 The size measure and the size relation

Any value in the language can be viewed as a tree. To keep things simple the size-measure of a value is mapped to the length of the longest branch in the value.

Size-measure of a value. Let $v \in \mathit{Value}$. The size-measure on *Values* is defined recursively as

$$\begin{array}{ll} \|\mathtt{con}\| & = 1 \\ \|\mathtt{con}(v^1, \ldots, v^n)\| & = 1 + \max_{i \in \{1,\ldots,n\}} \|v^i\|. \end{array}$$

With the size-measure and the usual ordering on natural numbers, we can now define the size-relation in *Value*:

Size relation. Let $v_1, v_2 \in \mathit{Value}$, then $v_1 \preceq v_2$ iff $\|v_1\| \leq \|v_2\|$. The induced equality relation will be denoted $v_1 \asymp v_2$.

Safety. The size-change analysis must be safe, in the sense that the size-relations found by the analysis must hold for *any* actual computation trace. The size-change analysis is based on a partial order - the size-relation - on the Value domain. In order for a size relation to be *safe* for language L, it is required that an application of a destructor decreases the size wrt. the size-relation, and similarly an application of a constructor strictly increases the size:

1. The size relation is safe for the destructors: $\mathcal{D}[\![\texttt{des}]\!]v \prec v$, for each $\texttt{des} \in \mathit{Destructor}$, $v \in \mathit{Value}$, provided that $\mathcal{D}[\![\texttt{des}]\!]v \neq \texttt{Error}$.
2. The size relation is safe for the constructors: $\mathcal{C}[\![\texttt{con}]\!]\vec{v} \not\preceq v_i \ \forall \texttt{con} \in \mathit{Constructor}, \vec{v} \in \mathit{Value}^{|A|}, i \in A = \{1, \ldots, \mathit{arity}(\texttt{con})\}$, provided that $v_i \neq \texttt{Error}$.

It is easily seen that the size relation defined above is safe for L .

Before proceeding towards the size-change analysis, a value size-change analysis is presented based on abstract interpretation of the subject program.

4.2 Size change relative to one parameter

We first look at size-changes relative to just one parameter. In order to keep the analysis simple, the abstract values will be represented by the lattice formed by the power-set of $\{\downarrow_s, =_s, \uparrow_s\}$ where the symbols describe "decrease", "no change" and "increase" respectively. Arcs with these annotations will be called *simple arcs*, and will be annotated with a subscript "s" for clarity. The elements in the power-set describe all possible size-changes relative to one parameter and will in short form be written:

$$\begin{array}{lcl}
\emptyset & = & \bot \\
\{\downarrow_s\} & = & \downarrow \\
\{=_s\} & = & = \\
\{\uparrow_s\} & = & \uparrow \\
\{\downarrow_s, =_s\} & = & \mp \\
\{\downarrow_s, \uparrow_s\} & = & \updownarrow \\
\{=_s, \uparrow_s\} & = & \pm \\
\{\downarrow_s, =_s, \uparrow_s\} & = & \top.
\end{array}$$

The set of all arcs is defined to be $\mathit{Arc} = \mathcal{P}(\{\downarrow_s, =_s, \uparrow_s\})$.

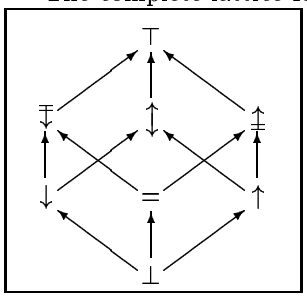
The $\downarrow$-arcs model the effect of a destructor operation, and similarly the $\uparrow$-arcs model the effect of a constructor operation. The abstract values involving the simple $\uparrow_s$-arc are not needed in order to decide termination, and could just as well be represented by $\top$. In fact since the algorithm builds upon Wahlsteds implementation which only include the $\downarrow, \mp, \top$[3]-arcs, the extra information must be discarded before the results of the analysis are passed on the second phase of the size-change termination algorithm. But by including them, the abstract domain can be represented by a power-set, giving the "precision" ordering ($\sqsubseteq_s$), meet ($\sqcap_s$) and join ($\sqcup_s$) operations directly as the usual set relation, intersection and union:

$$\begin{array}{lcl}
a_1 \sqcup_s a_2 & = & a_1 \cup a_2 \\
a_1 \sqcap_s a_2 & = & a_1 \cap a_2 \\
a_1 \sqsubseteq_s a_2 & = & a_1 \subseteq a_2
\end{array}$$

[3]The $\top$ simply means "don't know" and are thus not used in the actual size-change graphs.

By using a power set, the abstraction can be fairly easily extended. Another advantage is that the basic concepts are already covered, when the possible extensions are discussed later in the paper.

The complete lattice for the arc is:



4.3 Size-change graphs

This section (4.3) is adapted from "The Size-Change Principle for Program Termination" by N. D. Jones, C. S. Lee and A. M. Ben-Amram [3]. Let f, g be function names in program p. A *size-change graph* from f to g, written $G : \mathtt{f} \to \mathtt{g}$ is a bipartite graph from f parameters to g parameters, with labeled-arc set E:

$$\begin{array}{rcl} G & = & (Param(\mathtt{f}), Param(\mathtt{g}), E) \\ E & \subseteq & Param(\mathtt{f}) \times Arc \setminus \{\bot, \top\} \times Param(\mathtt{g}), \end{array}$$

where E does not contain both $\mathtt{f}^{(i)} \xrightarrow{a} \mathtt{g}^{(j)}$ and $\mathtt{f}^{(i)} \xrightarrow{b} \mathtt{g}^{(j)}$. for $a, b \in Arc$: $a \neq b$. The last requirement ensures that the graph does not contain redundant information of the size relation between any two variables.

The size-change graph is used to capture "definite" information about a function call. An $\mathtt{f}^{(i)} \xrightarrow{a} \mathtt{g}^{(j)}$ arc indicates that a data value *must* behave as described by the a wrt. the ordering $\prec$ on the *Value* domain in this call:

$\top$	:	"don't know" this means that any change in the size of the data value in the call is admissible.
$\updownarrow$	:	*Must* either increase or remain the same.
$\updownarrow$	:	*Must* either decrease or increase.
$\overline{\downarrow}$	:	*Must* either decrease or remain the same.
$\uparrow$	:	*Must* increase.
$=$	:	*Must* remain the same.
$\downarrow$	:	*Must* decrease.
$\bot$	:	"uninitialized" this means that the value never is used in an actual computation. This abstract value is describes the size-change of the return value of the function: `f(x) = f(x)`.

For example: An $\mathtt{f}^{(i)} \xrightarrow{\downarrow} \mathtt{g}^{(j)}$ arc indicates that a data value *must* decrease in the call, wrt. the $\prec$ ordering on data values, while an $\mathtt{f}^{(i)} \xrightarrow{\overline{\downarrow}} \mathtt{g}^{(j)}$ arc indicates

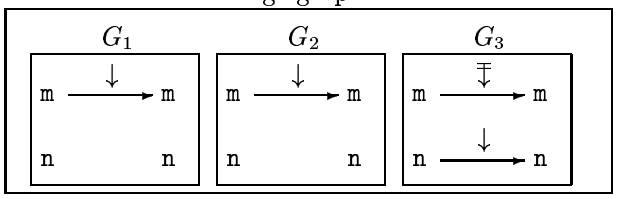
that a value must either decrease or remain the same. The absence of an arc between a pair of parameters means that none of these relations are asserted to be true for them. This is captured by the $\top$ element in the *Arc* lattice.
Note: For given `f`, `g` in program `p` there are only finitely many possible size-change graphs $G : \mathtt{f} \to \mathtt{g}$.

Set of size-change graphs. Henceforth $\mathcal{G} = \{G_c \mid c \in C_{(}\mathtt{p})\}$ denotes a set of size-change graphs associated with subject program `p`, one for each of `p`'s calls.

Example of size-change graphs. As an example, consider the Ackerman function:

```
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)))
```

This has the size-change graphs:



In the call at call site 3 the function calls itself with `n` bound to `2nd(n)` giving rise to a decrease in the size of the value of `n`. This is indicated in the graph G_3 by the $\downarrow$-arc from `n` to `n`. The parameter `m` is simply passed on with no change in the size. This is indicated by the $\mp$-arc from `m` to `m` in G_3.

4.3.1 Safety of size-change graphs

Let $\mathcal{G} = \{G_c \mid c \in C_{(}\mathtt{p})\}$ be a set of size-change graphs for the program `p`.

1. Let `f`'s definition contain call $c : \mathtt{g}(\mathtt{e}_1, \ldots, \mathtt{e}_n)$. The phrase "arc $\mathtt{f}^{(i)} \xrightarrow{r} \mathtt{g}^{(j)}$ safely describes the $\mathtt{f}^{(i)}$-$\mathtt{g}^{(j)}$ size relation in call c" means: For every $v \in \mathit{Value}$ and $\sigma \in \mathit{Env}$ such that $\mathit{vars}(\mathtt{e}_j) \subseteq \mathit{dom}(\phi)$ and $\mathcal{E}[\![\mathtt{e}_j]\!]\sigma = \mathit{liftv}$: $\mathit{rel}(r)(v, v_i)$ where $\mathit{rel} : \mathit{Arc} \to (\mathit{Value} \times \mathit{Value})$ is a function that takes an element of *Arc* and returns the corresponding relation on *Value*:
$$\begin{array}{lll} \mathit{rel}(r)(x,y) & \mathit{iff} & \downarrow_s \in r \wedge x \prec y \vee \\ & & =_s \in r \wedge x \asymp y \vee \\ & & \uparrow_s \in r \wedge x \succ y. \end{array}$$
2. Size-change graph G_c is *safe for call* $c : \mathtt{f} \to \mathtt{g}$ if every arc in G_c is a safe description as just defined.

3. Set $\mathcal{G}$ of size-change graphs is a *safe description of program* p if graph G_c is safe for every call c.

It is easy to see that all the size-change graphs given in the Ackermann example is safe for the call. Section 5 describes the size-change graph extraction algorithms used in this paper.

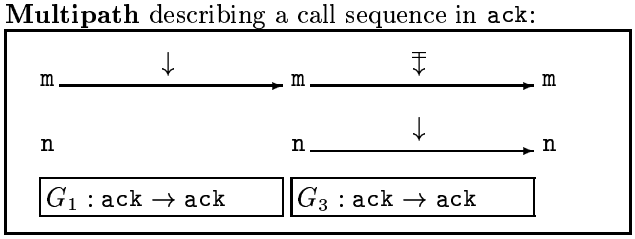
Note. In the following sections (4.3.2 to 4.5), only size-change graphs with arcs in $\{\downarrow, \overline{\mp}\}$ will be considered. All other arcs are considered as $\top$-arcs and are not represented in the size-change graph. Clearly the second phase in the size-change principle could be extended to also include $\uparrow$-arcs, and in this way account for call sequences with increasing values as well[4].

Also note. The remainder of this section (4) is a highly condensed presentation based on excerpts from the paper "The Size-Change Principle for Program Termination" by N. D. Jones, C. S. Lee and A. M. Ben-Amram [3].

4.3.2 Multipaths

A *multipath* $\mathcal{M}$ is a finite or infinite sequence $G_{c_1}, G_{c_2}, \ldots$ of size-change graphs, where $c_1 c_2 \ldots$ must be a wellformed callsequence. This sequence may be viewed as a concatenated (possibly infinite) graph, as illustrated by:

Multipath describing a call sequence in `ack`:



1. A *thread* th in multipath $\mathcal{M} = G_{c_1}, G_{c_2}, \ldots$ is a connected path of arcs:

   $$th \;=\; \mathtt{f}_t^{(i_t)} \xrightarrow{r_{t+1}} \mathtt{f}_{t+1}^{(i_{t+1})} \xrightarrow{r_{t+2}} \ldots$$

   An example is marked by heavy lines in the example. *Remarks:* a thread need not start at $t = 0$. An instance is the thread starting in n in G_3. A thread need not be infinite even if $\mathcal{M}$ is infinite.

   A thread is *maximal* if the connected path of arcs is maximal in the multipath.

2. Thread th is *descending* if the sequence $r_{t+1}, r_{t+2}, \ldots$ has at least one $\downarrow$. The thread is *infinitely descending* if it contains infinitely many occurrences of $\downarrow$.

[4] But where the overall size-change over the entire call sequence still is a decrease.

4.3.3 Multipaths of a state transition sequence and of a call sequence

A size-change graph can be used to describe the parameter size changes in *one concrete state transition sequence*, or it may be used abstractly, to depict size changes following a call sequence *cs*.

Definition. Consider state transition sequence

$$sts = (\mathtt{f}_0, \vec{v}_0) \xrightarrow{c_1} (\mathtt{f}_1, \vec{v}_1) \xrightarrow{c_2} (\mathtt{f}_2, \vec{v}_2) \xrightarrow{c_3} \ldots,$$

Define $\mathcal{M}(sts)$ to be the multipath $G_1, G_2, \ldots$, such that for each t, G_{t+1} is a size-change graph from $\mathtt{f}_t$ to $\mathtt{f}_{t+1}$, with arcs $\mathtt{f}_t^{(i)} \xrightarrow{r} \mathtt{f}_{t+1}^{(j)}$ satisfying $r = \downarrow$ if $u_j \prec v_i$, and $r = \overset{=}{\downarrow}$ if $u_j = v_i$, where $\vec{v}_t = (v_1, \ldots, v_m), \vec{v}_{t+1} = (u_1, \ldots, u_n)$.

Definition. Suppose $\mathcal{G} = \{G_c \mid c \in C_{\mathtt{p}}\}$ is a set of size-change graphs for the program p. Given a call sequence $cs = c_1 c_2 c_3 \ldots$, the *$\mathcal{G}$-multipath for cs* is defined by $\mathcal{M}^{\mathcal{G}}(cs) = G_{c_1}, G_{c_2}, G_{c_3}, \ldots$.
Note that $\mathcal{M}(sts)$ displays the *actual size relations* among parameter values along a state transition sequence, while $\mathcal{M}^{\mathcal{G}}(cs)$ displays the information provided by the size-change graphs in $\mathcal{G}$.

Safety of Multipaths. If $\mathcal{G}$ is a safe set of size-change graphs and *sts* is a state transition sequence, then $\mathcal{M}(sts)$ is safely described by the $\mathcal{G}$-multipath $\mathcal{M}^{\mathcal{G}}(cs)$ that follows the calls *cs* in *sts*:

Corollary 1. *If $\mathcal{M}^{\mathcal{G}}(cs)$ has an infinite thread th, and $cs = calls(sts)$, then $\mathcal{M}(sts)$ also has an infinite thread th'. Furthermore, thread th' has at least as many $\downarrow$-labeled arcs as th.*

4.4 The termination analysis

This section (4.3) is adapted from "The Size-Change Principle for Program Termination" by N. D. Jones, C. S. Lee and A. M. Ben-Amram [3].

Termination criterion. Define for a given program p the following sets of infinite call sequences:

$$FLOW^{\omega} = \{cs = c_1 c_2 \ldots \in C_{\mathtt{p}}^{\omega} \mid cs \text{ is well-formed and } \mathtt{f}_{initial} \xrightarrow{c_1} \mathtt{f}_1\}$$
$$DESC^{\omega} = \{cs \in FLOW^{\omega} \mid \text{ some thread } th \in \mathcal{M}^{\mathcal{G}}(cs) \text{ has infinitely many } \downarrow\text{-arcs}\}$$

The set $FLOW^{\omega}$ describes all the possible well formed call sequences in the program and $DECS^{\omega}$ describes all the infinite call sequences that causes infinite descent. The following theorem characterizes the size-change terminating programs:

Theorem 1. *If $FLOW^{\omega} = DESC^{\omega}$ for some program p then the program terminates for all inputs.*

4.4.1 A graph-based algorithm

The second phase of the size-change termination analysis in this paper, is an implementation of the graph-based algorithm for detecting size-change termination.

Composition of size-change graphs. Let $G : \mathtt{f} \to \mathtt{g}$ and $G' : \mathtt{g} \to \mathtt{h}$ be size-change graphs, then the composition $G;G' : \mathtt{f} \to \mathtt{h}$ with arc set E is defined below. Notation: we write $\mathtt{x} \xrightarrow{r} \mathtt{y} \xrightarrow{r'} \mathtt{z}$ if $\mathtt{x} \xrightarrow{r} \mathtt{y}$ and $\mathtt{y} \xrightarrow{r'} \mathtt{z}$ are respectively arcs of G and G'.

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

Safety. The size-change graph composition preserves the "descending thread" property: Let $cs = c_1 \ldots c_n$ be a call sequence. If the multipath $\mathcal{M}^{\mathcal{G}}(cs)$ has a descending thread, then the size-change graph $G_{c_1};\ldots;G_{c_n}$ has an arc of the form $\mathtt{x} \xrightarrow{\downarrow} \mathtt{x}$.

For a call sequence $cs = c_1 \ldots c_n$ the size-change graph G_{cs} for cs is defined as:

$$G_{cs} = G_{c_1};\ldots;G_{c_n}$$

With this definition a multi graph describing the size-change in a call sequence can be "collapsed" to a single size-change graph describing the size-change. Define the set $\mathcal{S}$ as:

$$\mathcal{S} = \{G_{cs} \mid \exists cs_0 : \; cs, cs_0 \text{ are well-formed and } \mathtt{f}_{initial} \xrightarrow{cs_0} \mathtt{f} \xrightarrow{cs} \mathtt{g}\}$$

The set $\mathcal{S}$ is finite since there are finitely many possible graphs.

Termination analysis using composition. The central idea in the graph based algorithm is:

Theorem 2. *Program* p *is* not *size-change terminating iff S contains* $G : \mathtt{f} \to \mathtt{f}$ *such that* $G = G;G$ *and* G *has no arc of form* $\mathtt{x} \xrightarrow{\downarrow} \mathtt{x}$.

4.5 Termination algorithm

This section (4.5) is adapted from "The Size-Change Principle for Program Termination" by N. D. Jones, C. S. Lee and A. M. Ben-Amram [3].

In order to decide size-change termination for some program p, the following algorithm is used:

1. **Phase 1**: Extract the set of size-change graphs $\mathcal{G}$ from p.
2. **Phase 2a**: Build the set $\mathcal{S}$ by a transitive closure procedure:
   - Include every $G_c : \mathtt{f} \to \mathtt{g}$ where $c : \mathtt{f} \to \mathtt{g}$ is a call in program p, and f is reachable by some well-formed $cs_0 : \mathtt{f}_{initial} \to \mathtt{f}$.
   - For any $G : \mathtt{f} \to \mathtt{g}$ and $H : \mathtt{g} \to \mathtt{h}$ in $\mathcal{S}$, include also $G;H$ in $\mathcal{S}$.
3. **Phase2b**: For each $G : \mathtt{f} \to \mathtt{f}$ in $\mathcal{S}$, test whether $G = G;G$ and $\mathtt{x} \overset{\downarrow}{\to} \mathtt{x} \notin G$ for each $\mathtt{x} \in \mathit{Param}(\mathtt{f})$.

5 Extracting size-change graphs

The size-change graph extraction algorithm extracts safe size-change graphs from the subject program. Conceptually this is done by first performing a size-change analysis: given a program the analysis computes relations between the size of the values computed in the program relative to all possible input. Once the size-changes have been computed, the size-change graphs can be extracted. As it will be evident later, the size-change graph extraction can be combined with the analysis since all the necessary information is available during the analysis. It is then just a matter of accumulating the size-change graphs.

The size-change analysis is computed by an abstract interpretation of the subject program. A cache is maintained during the interpretation to avoid infinite computation. The resulting size-change graphs describe decreases as well as increases, but the second phase only considers decreases. The extra information in the size-change graphs is therefore discarded before the second phases of the termination analysis begins.

The analysis proceeds by analyzing each function in the program in turn, and analyzes the program for size-changes relative to that function's parameters.

Subject function. Let for the remainder of this section (5) $\mathtt{f}_{an}(\mathtt{f}_{an}^{(1)}, \ldots, \mathtt{f}_{an}^{(arity(\mathtt{f}_{an}))})$ denote the function being analyzed.

5.1 Size change relative to all parameters

The size-change of an expression relative to the parameters of $\mathtt{f}_{an}$ will be the abstract values used in the abstract interpretation computing the size-change analysis. An abstract value describes the size of the value computed at a subexpression in the program (or "control point") relative to the parameters of $\mathtt{f}_{an}$. Intuitively an abstract value is a graph[5] describing the size changes from the parameters of $\mathtt{f}_{an}$ to the expression that the abstract value describes. The graph describes all possible relations between the parameters and the possible values that the expression can evaluate to.

[5] *Not* a size-change graph as introduced in section 4.3.

5.1.1 Abstract values

The abstract values are formed from *Arc* (section 4.2) by the independent attribute method [5]. An abstract value consists of a tuple of values from the lattice described on page 4.2, one for each parameter of $\mathtt{f}_{an}$. The set abstract values used in the analysis are thus defined as:

$$Avalue = Arc^{arity(\mathtt{f}_{an})}.$$

An abstract value $v \in Avalue$ is written using the notations [6]:

$$\begin{array}{rcl} v & = & (v_1, \ldots, v_{arity(\mathtt{f}_{an})}) \\ & = & (v_i^{\forall i \in \{1,\ldots,arity(\mathtt{f}_{an})\}}) \end{array}$$

The i'th element (denoted v_i) in the tuple describes the size-change relative to the i'th parameter of $\mathtt{f}_{an}$. For the *Avalue* domain the "precision relation" ($\sqsubseteq$), meet ($\sqcap$) and join ($\sqcup$) are defined by applying the corresponding operations on *Arc* (section 4.2) pairwise to the elements in the tuples.

$$\begin{array}{lcl} v \sqcap w & = & ((v_i \sqcap_s w_i)^{\forall i \in \{1,\ldots,arity(\mathtt{f}_{an})\}}) \\ v \sqcup w & = & ((v_i \sqcup_s w_i)^{\forall i \in \{1,\ldots,arity(\mathtt{f}_{an})\}}) \\ v \sqsubseteq w & = & \bigwedge_{i=1}^{arity(\mathtt{f}_{an})} (v_i \sqsubseteq_s w_i) \end{array}$$

By forming the independent attribute method all connections between the size-changes over different variables are lost, so the abstraction does not capture "global size-change" for instance.

5.1.2 Abstraction

Abstraction function. The abstraction function $\alpha : Expr \to Avalue$ is defined for all possible inputs by:

$$\begin{array}{lcl} \alpha(\mathtt{e}) & = & (\alpha_i(e)^{\forall i \in \{1,\ldots,arity(\mathtt{f}_{an})\}}) \\ \alpha_i(\mathtt{e}) & = & \{arc_s(\mathcal{E}[\![\mathtt{e}]\!]\sigma, \vec{v}_i) \mid \sigma \in (Variable \to Value)\} \end{array}$$

$$arc_s(v,w) = \begin{cases} \downarrow_s & \text{if } v \prec w \\ =_s & \text{if } v \asymp w \\ \uparrow_s & \text{otherwise} \end{cases}$$

The program for which the expression is evaluated is implicit as in the semantics.

We now proceed towards the abstract semantics:

5.1.3 Abstract operations

Safe abstract operation. If $\mathtt{op} : Value^{arity(\mathtt{op})} \to Value$ is an operator then the abstract operator $\alpha_{\mathtt{op}} : Avalue^{arity(\mathtt{op})} \to Avalue$ is *safe* for op iff:

[6] No difference is implied. The two notations are simply two different ways of describing the same value.

$$\forall v_i \in \mathit{Value}, i \in \{1, \ldots, \mathit{arity}(\mathtt{op})\} : \ \alpha(\mathtt{op}(v_1, \ldots, v_n)) \sqsubseteq \alpha_{\mathtt{op}}(\alpha(v_1), \ldots, \alpha(v_n))$$

Abstract operations for the following must be defined:

- Constructor application
- Destructor application
- Conditionals
- Primitive functions
- Function calls

5.1.4 Operations on abstract values

If $v \in \mathit{Avalue}, a \in \mathit{Arc}$ then define $\mathit{dec}, \mathit{inc} : \mathit{Avalue} \to \mathit{Avalue}$:

$$\begin{array}{ll} \mathit{dec}(v) & = (\mathit{dec}_{arc}(v_i)^{\forall i \in \{1,\ldots,\mathit{arity}(\mathtt{f}_{an})\}}) \\ \mathit{inc}(v) & = (\mathit{inc}_{arc}(v_i)^{\forall i \in \{1,\ldots,\mathit{arity}(\mathtt{f}_{an})\}}) \end{array}$$

$$\mathit{dec}_{arc}(a) = \left\{ \begin{array}{ll} \downarrow & \text{if } (\downarrow_s \in a) \vee (=_s \in a) \\ \bot & \text{otherwise} \end{array} \right\} \sqcup_s \left\{ \begin{array}{ll} \pm & \text{if } \uparrow_s \in a \\ \bot & \text{otherwise} \end{array} \right\}$$

$$\mathit{inc}_{arc}(a) = \left\{ \begin{array}{ll} \uparrow & \text{if } (\uparrow_s \in a) \vee (=_s \in a) \\ \bot & \text{otherwise} \end{array} \right\} \sqcup_s \left\{ \begin{array}{ll} \mp & \text{if } \downarrow_s \in a \\ \bot & \text{otherwise} \end{array} \right\}$$

where $\mathit{dec}_{arc}, \mathit{inc}_{arc} : \mathit{Arc} \to \mathit{Arc}$. The *dec* operation describes the effect of applying a destructor to an abstract value, and the *inc* operation describes constructor application. Since the abstract domain was constructed from the power-set of simple arcs, all operations can be described by their effect on those arcs. The definitions are safe for unary constructors and destructors:

$$\begin{array}{lll} \alpha(\mathtt{con}(v)) & \sqsubseteq & \mathit{inc}(v) \\ \alpha(\mathtt{des}(v)) & \sqsubseteq & \mathit{des}(v) \end{array}$$

for all $\mathtt{con} \in \mathit{Constructor}, \mathtt{des} \in \mathit{Destructor} : \mathit{arity}(\mathtt{con}) = \mathit{arity}(\mathtt{des}) = 1$.

5.1.5 Constructors

Constructor applications are split up into two cases depending on whether the arity of the constructor is zero or not.

Let $\mathtt{con}(v^1, \ldots, v^n)$ be a constructor application. The size of the expression is:

$$\|\mathtt{con}(v^1, \ldots, v^n)\| = 1 + \alpha_{max}\{\|v^1\|, \ldots, \|v^n\|\}$$

The problem is to find an abstract value that safely describes $max\{\|v^1\|, \ldots, \|v^n\|\}$.

$$\alpha(\mathtt{con}(v^1, \ldots, v^n)) \sqsubseteq \mathit{inc}(\bigsqcup_{i=1}^{n} v^i).$$

Defining $\alpha_{\texttt{con}}(v^1,\ldots,v^n)) = inc(\bigsqcup_{i=1}^n v^i)$ is safe, but could be more precise. Example: In `cons(1st(x), x)` the first argument is abstracted to $\downarrow$ and the second to $=$[7]:

$$inc(\downarrow \sqcup =) = inc(\updownarrow) = \top.$$

But $\uparrow$ is also safe, since $=$ safely describes the maximal element of the two. This is not a problem for the termination analysis, since it does not depend on precise $\uparrow$-arcs[8], but for the possible extensions it could become a problem, since information is discarded. Thus $\alpha_{\texttt{con}}$ is defined as:

$$\alpha_{\texttt{con}}(v^1,\ldots,v^n) = inc(\alpha_{max}\{v^i \mid i \in \{1,\ldots,n\}\}),$$

where we take $\alpha_{max} = \sqcup$.

By "continuity" one could be tempted to use $inc(\bot) = \bot$ for 0-ary constructors, since $\bot$ is the neutral element for $\sqcup_s$. But the values in L are well founded with the 0-ary constructors as the smallest elements according to the size-relation, so we would have that:

$$\forall v \in \mathit{Value}, \texttt{con} \in \mathit{Constructors} : \mathcal{E}[\![\texttt{con}]\!][] \preceq v,$$

which is well-defined, thus

$$\bot \sqsubset \alpha(\texttt{con}) \sqsubseteq (\updownarrow^{\forall i \in \{1,\ldots,arity(\texttt{f}_{an})\}})$$

violating the operator safety property. Therefore $\alpha_{\texttt{con}}$ is defined as

$$\alpha_{\texttt{con}}(v^1,\ldots,v^n) = \begin{cases} (\updownarrow^{\forall i \in \{1,\ldots,arity(\texttt{f}_{an})\}}) & \text{if } arity(\texttt{con}) = 0, \\ inc(\bigsqcup_{i=1}^n v^i) & \text{otherwise.} \end{cases}$$

5.1.6 Destructors

A destructor in L has precisely one argument, so in analogy with the constructors, the destructor is defined as: $\alpha_{\texttt{des}}(v) = dec(v)$, for $v \in \mathit{Avalue}$. If multi-argument destructors were introduced, the development would be similar to the development of the constructors. The safety of the $\alpha_{\texttt{des}}$ follows directly from the safety of *dec*.

5.1.7 Conditionals

According to the semantics the value of an if statement can either be the value of the then or the else branch. Thus the abstract value must describe both cases, that is the most precise value safely describing the two:

$$\alpha_{\texttt{if}}(\texttt{if } u \texttt{ then } v \texttt{ else } w) = v \sqcup w.$$

Clearly $\alpha_{\texttt{if}}$ is safe for the if expressions.

[7] Through out the paper elements of a set A^1 are denoted a rather than (a)

[8] This will clearly not affect the precision of the $\downarrow$-arcs.

5.1.8 Primitive functions

For the size-change analysis we assume the program is run with an existing library of supporting functions defining logical and arithmetic operations, all of which are easily implemented. In order not to shift the attention away from the program being analyzed, these operations will be treated as primitive functions, instead of linking the subject programs to a library prior to analysis.

Since the primitive functions are not defined in the program, they must be handled separately. The abstract values describing the size of the output of the primitive function relative to the *input of the primitive function* are stored in a list, and retrieved, when a primitive function is called. The size-change graphs are then composed with the size-change graphs for the arguments, giving the size-change graph for the return value relative to the *parameters of* $\mathtt{f}_{an}$.

5.1.9 Composition of abstract values

Let $op \in Arc^{arity(\mathtt{op})}$ be an abstract value describing the size-change behavior of op, and let $arg_k \in Avalue,\ k \in K = \{1, \ldots, arity(op)\}$ be the size-changes of the argument to op the call relative to $\mathtt{f}_{an}$'s parameters. Let $L = \{1, \ldots, arity(\mathtt{f}_{an})\}$. The composition $op \oplus (arg^k)_{k \in K}$, is defined as:

$$op \oplus (arg^k)_{k \in K} = ((\bigsqcup_{k \in K} (op_k \otimes arg_l^k))^{\forall l \in L}), \text{ where}$$

$$a \otimes a' = \begin{array}{l} \left\{ \begin{array}{ll} dec(a') & \text{if } \downarrow_s \in a \\ \top & \text{otherwise} \end{array} \right\} \ \sqcap_s \\ \left\{ \begin{array}{ll} a' & \text{if } =_s \in a \\ \top & \text{otherwise} \end{array} \right\} \ \sqcap_s \\ \left\{ \begin{array}{ll} inc(a') & \text{if } \uparrow_s \in a \\ \top & \text{otherwise} \end{array} \right\} \end{array}$$

$$\begin{array}{ll} \cdot \oplus \cdot : & Arc^{arity(\mathtt{op})} \to (Avalue)^{arity(\mathtt{op})} \to Avalue \\ \cdot \otimes \cdot : & Arc \to Arc \to Arc \end{array}$$

The $\otimes$ operation describes the composition of two arcs. The operation $\oplus$ describes composition for size-change graphs. The size-change over $\mathtt{f}_{an}^{(i)}$ is the worst case size-change of the size-changes computed by composing the i'th arcs from the arguments composed with the arc in op corresponding to the argument. Note that a $\downarrow$-arc in the abstract value of op means a decrease of precisely one, corresponding to *one* deconstruction. If the return value of the primitive operator is a decrease with more than one deconstruction, then $\top$ is used in the abstract description of the operator.

5.1.10 Function calls

If the program is simply abstractly interpreted it may not terminate, since we are computing with abstract values over a smaller domain than in the original program. To ensure termination, a call stack and a cache of previously computed

return values are maintained. The call stack contains function names, and their abstract arguments of all calls that has occurred in the current call sequence. When a new function is called, the interpreter checks whether the call previously has occurred, and if so the call is looked up in the cache[9]. If the call was not on the call stack, the call is performed as normal, and the return value is cached. The abstract interpretation is iterated until a fixpoint has been reached. This is ensured by the monotonicity of the framework[5].

5.2 Abstract interpretation

With the defined abstract operators, the function return size-change can now be computed by abstract interpretation. The variables and `let` expressions are handled according to the usual semantics (Figure 3.1) and function calls are handled as described in section 5.1.10. The other language constructs are evaluated by evaluating the subexpressions and then applying the abstract operator (as described above) corresponding to the construct. Note that the analysis ignores the outcome of the test expression in the conditionals. It is therefore not necessary to evaluate the test expression in order to perform a size-change analysis. The abstract interpretation is started by evaluating the body of $\mathtt{f}^{an}$ in the environment:

$$[\mathtt{f}_{an}^{(i)} \mapsto \alpha_{param}(\mathtt{f}_{an}^{(i)})^{\forall i \in \{1,\ldots,arity(\mathtt{f}_{an})\}}],$$

$$\alpha_{param}(\mathtt{f}_{an}^{(i)}) = \{\underbrace{\top,\ldots,\top}_{i-1},=,\underbrace{\top,\ldots,\top}_{arity(\mathtt{f}_{an})-i}\}.$$

All variables in the initial environment describe exactly one parameter. Therefore all variables have an $=$-arc to their values in the arc from the corresponding parameter. Nothing is assumed of the size-change relative to other parameters, giving a $\top$-arc from all other parameters to the value of the variable. The initial environment is clearly a safe description of the environment in which the body of $\mathtt{f}_{an}$ is evaluated in, for a call to $\mathtt{f}_{an}$.

5.3 Size-change graph extraction

The size-change analysis computes the size-changes of values relative to $\mathtt{f}_{an}$'s parameters by abstract interpretation. It is noted that the size-change graphs for the call sites occurring in $\mathtt{f}_{an}$ can be almost directly extracted during the value size-change analysis. The abstract value describing some argument in a call, describes the size-change from the parameters of $\mathtt{f}_{an}$ to the parameters of the called function. Thus the size-change graph for the call is directly given by the abstract values for the arguments in the call. Since a size-change graph describes how the parameters of the *calling function* relates to the parameters of the *called function*, only graphs collected from call sites in the body of $\mathtt{f}_{an}$ are

[9] The cache maps uncomputed calls to $\top$.

actually size-change graphs[10]. By this method, the size-change graphs from $\mathtt{f}_{an}$ to the functions $\mathtt{f}_{an}$ calls, can be extracted during the analysis. The only catch is that when evaluating a conditional as described above, the test expression is not considered, so no size-change graph for call sites occurring in the expression will be collected. The solution is simply to evaluate the test expression, causing the collection of the size-change graphs, and throw away the resulting abstract value describing the test expression. Only the size-change graphs collected from the last fixpoint iteration are valid. So each time a size-change graph for some call site is collected it replaces any previous size-change graphs for that call site.

By this method the size-change graphs can be extracted by running the analysis/extraction algorithm as described above, on each function in the program. The simple version of the analysis (which will be written SCT_0 in short form, as opposed to SCT [11]), that does not consider the return values of function calls, can easily be obtained from the above algorithm. This is done by always returning "don't know"[12] when evaluating a function call. Since the simple version does not require fixpoint computation, it is easily seen that the simple size-change graph extraction runs in time proportional to the size of the subject program.

6 Experiments

This section presents the results obtained by the implementation of the size-change analysis with the size-change graph extraction as described in section 5. First the different test suites are presented in section 6.1. The test suites are run with different termination analyses which are described in section 6.2. An overview of the results obtained is given in section 6.3 and the results are treated in more detail in section 6.4. Finally section 6.5 gives a summary of the results.

6.1 The test suites

The subject programs (section A) are mainly collected from [7, 2, 3]. The programs have been translated into the L language as well as Haskell in order to run the different termination analyses on them.

Wahlstedt test suite. The test suite from Wahlstedt [7], section A.2 is a collection of first order programs with Base-1 numbers as the only data type. The numbers are encoded using the `S` and `Z` constructors, representing the successor and zero respectively. The programs have been rewritten by hand to conform to the language used in this paper.

[10] Other such graphs describes size-changes over several call rather than just one, and are not considered in this paper.

[11] The subscript refers to whether the size-change analysis functions are analyzed for return values

[12] ie. $\top$ in the product abstraction: $[\mathtt{f}_{an}^{(i)} \mapsto \top^{\forall i \in \{1,\ldots,arity(\mathtt{f}_{an})\}}]$.

Glenstrup test suite. The test suite from Glenstrup [2], (section A.3 to A.7.3is a large and diverse test suite, that was collected for Glenstrup's Master's thesis [2]. The test suite was machine translated from Scheme to the L language with a small compiler written for this purpose. Some minor modifications were made in order to make the simple compiler accept the programs. Strings and character constants were rewritten as Scheme symbols, which are translated into 0-ary constructors with the same name[13]. The changes made have no influence on the outcome of the termination analysis. Some of the examples compute with numbers encoded as Base-1 numbers . Since the test suite is adapted from Scheme source code, the numbers are encoded with list length using the `Cons` and `Nil` constructors, rather than the constructors used in the Wahlstedt test suite.

Jones test suite. The test suite from [3], (section A.1) is a direct implementation of the 6 small examples used to illustrate the power of the size-change termination principle. The test suite was written by hand in L and Haskell.

Other examples. Finally a small collection (section A.8) of programs was written to illustrate various points of this paper. Most of the programs are examples from the above test suites with minor modifications.

6.2 Termination analyses

This section gives an overview of the different termination analyses, that are compared in this paper.

6.2.1 This analysis

The size-change termination algorithms presented in this paper are implemented in Haskell. The focus of the implementation is to display the capabilities of the size-change termination wrt. showing termination of programs, rather than efficiency.

Output. The results of the analysis of the test suites using the implementation of this paper can be found in B. The format of the analysis results is the same as used in the Wahlstedt termination analysis. The analysis either decides that the program always terminates for all inputs and answers "ok", or finds possible termination problems. The termination problems are listed as call sequences from a function back to itself, where the analysis is unable to find any strict decrease in any parameters over the call sequence. Such problematic call sequences are called *critical call sequences*. If the analysis finds that the program might not terminate, the critical call sequences are displayed as lists of the call site numbers.

[13] Non alpha numerical characters are stripped and the name is capitalized in order to conform to the constructor naming convention.

6.2.2 Wahlstedt termination analysis

Due to the restricted language used in Wahlstedts Master's thesis [7], all test suites but the Wahlstedt test suite could not be run using the Wahlstedt termination analysis. Since the algorithms in this paper build on the same "second phase" and since the size-change graph extraction is at least as strong, results equal to or better are expected.

6.2.3 Glenstrup termination analysis

The Glenstrup termination analysis takes its input in form of a Haskell program (The Scheme test suite on which the Glenstrup test suite in this paper was based on, was translated into Haskell prior to analysis by the Glenstrup algorithm). The other test suites were translated by hand into Haskell, in order to try out the Glenstrup termination analysis on these suites. The analysis accepts programs with built-in primitive numbers, and Base-1 numbers encoded with Haskell constructors. The termination analysis is sensitive to the representation of numbers. Size-changes over the built-in numbers are not considered. For this reason, numbers in the test suites have been encoded in the examples where numbers are critical for the termination analysis. The termination analysis requires that the entry function is non-recursive. This is circumvented by defining new entry functions, that call the old ones. If this is not done, the analysis may give weaker results.

Output. The Termination analysis is able to safely decide must terminate, may quasiterminate and may not terminate. Since the analysis is able to decide quasitermination, the results are potentially stronger than the results of this paper, since more types of termination can be decided [14]. The analysis lists the possibly offending program calls, when quasi- or non-termination is decided.

6.3 Results

In this section the results of running the different termination analyses on the different test suites are presented. The symbols used in the tables describing the results are explained in 1.

6.3.1 The Jones examples

The results of the Jones test suite are listed in Table 2. From the listed results it is seen that the size-change graph extraction methods in this paper is sufficiently strong to extract the size-change graphs in [3]. The Glenstrup termination analysis correctly finds that all programs are terminating in the cases where the analysis succeeds.

Symbol	Description
T	Terminating
QT	Quasiterminating
NT	Nonterminating
-	Not analyzed
nr. g	Number of generated size-change graphs
nr. c	Number of compositions

Table 1: Legend for the test suite results

Prog	Opt	Glenstrup	Wahlstedt	SCT_0	SCT	nr.g	nr. c
ex1	T	T	-	T	T	2	3
ex2	T	-	-	T	T	2	7
ex3	T	T	-	T	T	3	2
ex4	T	T	-	T	T	2	22
ex5	T	T	-	T	T	2	6
ex6	T	T	-	T	T	3	6

Table 2: Jones test suite results

Prog	Opt	Glenstrup	Wahlstedt	SCT_0	SCT	nr.g	nr. c
ack	T	T	T	T	T	3	2
add	T	T	T	T	T	0	0
fgh	QT	NT	NT	NT	NT	8	22
boolprog	QT	NT	NT	NT	NT	9	565
ex6	T	T	T	T	T	3	6
permut	T	T	T	T	T	1	3
eq	T	T	T	T	T	1	1
odd_even	T	T	T	T	T	2	4
div2	T	T	T	T	T	1	1

Table 3: Wahlstedt test suite results

6.3.2 The Wahlstedt examples

Results of the Wahlstedt test suite are listed in Table 3. All results from the Wahlstedt test suite match the results found in [7]. Since both analyses use the same second phase algorithm, the question is whether the size-change graph extraction methods are strong enough to extract size-change graphs that are at least as precise as in the Wahlstedt termination analysis. The results indicate that this is the case. An inspection of the generated size-change graphs show that precisely the same graphs are generated. The Wahlstedt paper has a number of limitations on the programs accepted:

- Only variables can be deconstructed.
- Deconstruction can only be done "at top level" in an expression.
- Deconstruction is handled by a case expression directly naming the subterm (since the language only handles Base-1 numbers) in the variable being deconstructed. The name of the subterm must be the name of the variable being deconstructed and must be annotated with an extra prime. In the way the number of primes directly count the number of times a variable has been deconstructed.

By using `let`'s in the translation of the Wahlstedt examples, the deconstructed variables can be named in a similar scheme, as described above. The case expressions can be translated ($\mathcal{T}$) as follows:

$$\mathcal{T}(\ \texttt{case}\ x_1\ \texttt{of}\quad \begin{array}{ll} 0 & \rightarrow\ e_1 \\ \texttt{S}(x_1) & \rightarrow\ e_2 \end{array}\ ) = \begin{array}{lll} \texttt{if eq}(x_1, \texttt{Z}) & \texttt{then} & \mathcal{T}(e_1) \\ & \texttt{else} & \texttt{let}\ x_1' = \texttt{1st}(x_1)\ . \\ & & \texttt{in}\ \mathcal{T}(e_2) \end{array}$$

Clearly the size-change extraction methods will find a decrease in the value from x_1 to x_1'. Since the case construction is the only expression giving rise to size-change graphs in Wahlstedts termination analysis, all the size-change relations found in the Wahlstedt implementation, will also be found with the size-change graph extraction methods. It is therefore not surprising to see that all Wahlstedt examples gives the same results with the size-change analysis.

The Glenstrup termination analysis successfully decides termination for the terminating examples.

6.3.3 The Glenstrup examples

Tables 4 and 5 list the results of the Glenstrup test suite. The results are reviewed on basis of the "optimal" outcome.

All nonterminating examples are classified as such by all analyses, as they should since the analyses are safe. The quasiterminating examples will for the purpose of evaluating the implementation of the size-change termination analysis be treated as if they were nonterminating. The terminating examples for

[14] The analysis in this paper will not decide quasitermination.

Prog	Opt	Glenstrup	Wahlstedt	SCT_0	SCT	nr.g	nr. c
add	T	T	-	T	T	2	3
addlists	T	T	-	T	T	2	3
anchored	T	T	-	T	T	2	3
append	T	T	-	T	T	2	3
assrewrite	T	NT	-	NT	NT	7	5
badd	QT	NT	-	NT	NT	3	5
contrived-1	T	T	-	T	T	9	58
contrived-2	QT	QT	-	NT	NT	10	88
decrease	T	T	-	T	T	3	7
deeprev	T	T	-	T	T	6	29
disjconj	T	T	-	T	T	9	15
duplicate	T	T	-	T	T	2	3
equal	QT	QT	-	NT	NT	3	4
evenodd	T	T	-	T	T	3	7
fold	T	T	-	T	T	6	12
game	T	T	-	T	T	3	15
increase	NT	NT	-	NT	NT	3	7
intlookup	QT	QT	-	NT	NT	4	12
letexp	NT	NT	-	NT	NT	2	3
list	T	T	-	T	T	2	3
lte	T	T	-	T	T	2	3
map	T	T	-	T	T	3	5
member	T	T	-	T	T	2	3
mergelists	T	T	-	T	T	3	7
mul	T	T	-	T	T	4	8
naiverec	T	T	-	T	T	4	6
nestdec	T	T	-	NT	T	5	12
nesteql	QT	QT	-	NT	NT	5	7
nestimeql	QT	NT	-	NT	NT	5	14
nestinc	NT	NT	-	NT	NT	5	14
nolexicord	T	T	-	T	T	4	13
ordered	T	T	-	T	T	2	3
overlap	T	T	-	T	T	4	8
permute	T	NT	-	NT	NT	8	33
revapp	T	T	-	T	T	2	3
select	T	T	-	T	T	4	12
shuffle	T	NT	-	NT	NT	6	12
sp1	QT	QT	-	NT	NT	7	16
subsets	T	T	-	T	T	4	6
thetrick	T	NT	-	NT	NT	11	24
vangelder	QT	QT	-	NT	NT	15	75

Table 4: Glenstrup test suite results - basic

Prog	Opt	Glenstrup	Wahlstedt	SCT_0	SCT	nr.g	nr. c
graphcolour-1	T	QT	-	NT	NT	15	69
graphcolour-2	T	QT	-	NT	NT	17	96
graphcolour-3	T	T	-	T	T	15	51
match	T	T	-	T	T	3	7
reach	QT	QT	-	NT	NT	7	25
rematch	T	-	-	NT	NT	41	463
strmatch	T	T	-	T	T	5	10
typeinf	T	-	-	T	T	20	84
int-dynscope	NT	NT	-	NT	NT	19	25
int-loop	T	T	-	T	T	19	26
int-while	NT	NT	-	NT	NT	25	49
int	NT	NT	-	NT	NT	19	25
lambdaint	NT	NT	-	NT	NT	25	54
parsexp	QT	QT	-	NT	NT	11	37
turing	NT	NT	-	NT	NT	12	225
ack	T	T	-	T	T	4	5
binom	T	T	-	T	T	3	5
gcd-1	T	T	-	NT	T	10	44
gcd-2	T	QT	-	NT	T	10	54
power	T	T	-	T	T	6	10
mergesort	T	NT	-	NT	NT	8	30
minsort	T	NT	-	NT	NT	7	23
quicksort	T	NT	-	NT	NT	9	30

Table 5: Glenstrup test suite results - algorithms, interpreters, simple, sorting

Prog	Opt	Glenstrup	Wahlstedt	SCT_0	SCT	nr.g	nr. c
assrewriteSize	T	-	-	NT	NT	8	17
deadcodeSize	T	-	-	NT	NT	1	1
fghSize	T	-	-	NT	NT	8	22
graphcolour2Size	T	-	-	T	T	17	79
quicksortSize	T	-	-	NT	NT	12	29
thetrickSize	T	-	-	T	T	11	26

Table 6: Size-change termination test suite results

which the size-change termination analysis fails to show termination, are all discussed in detail in section 6.4, as well as other interesting examples.

With the exception of `nestdec` and the `gcd-1`, `gcd-2` examples, all termination analyses find the same results, (disregarding the quasitermination results). The simple size-change termination analysis SCT_0 that does not account for the return values of functions, fails in some instances to decide termination where the other two analyses are able to. The Glenstrup termination analysis fail in one instance (the `gcd-2` example) to decide termination, where the SCT size-change analysis is able to.

6.3.4 Other examples

The remaining examples can be found in Table 6.

6.4 Detailed examination

In this section the examples are reviewed in greater detail. First the terminating examples (sections 6.4.1 to 6.4.7) for which all termination analysis were able to show termination will be reviewed, to illustrate why they size-change terminate. Then the terminating examples (sections 6.4.8, 6.4.9) where some, but not all termination analyses will be examined. Finally the terminating examples (sections 6.4.10 to 6.4.20) for which size-change termination could not be shown, will be examined in detail. The nonterminating examples are not reviewed. It only noted that none of the termination analyses incorrectly classified a nonterminating (or quasiterminating) program as a terminating program.

6.4.1 Simple termination

Most of the programs for which size-change termination can be shown by the algorithms in this paper, terminate for a simple reason. Every function in the program is defined by nonlinear recursion from nonlinear recursive functions, and are stratified. More specifically the programs are constructed from functions with recursion (NSR) on the form:

$$\begin{aligned} f(0,\vec{y}) &= g_0(\vec{y}) \\ f(x,\vec{y}) &= h(x, f(\mathtt{des}_1(x), g_1(\vec{y})), \ldots, f(\mathtt{des}_n(x), g_n(\vec{y}))), \end{aligned}$$

essentially stating that each function has a parameter x, that strictly decreases in each recursive call. The stratification requirement states that there exists an ordering on the functions such that a given function does not call a function "above" the function itself in the ordering.

The decrease in the parameter controlling the recursion is easily detected by the size-change graph extraction, and thus a graph with a decrease in that parameter is generated. The function is called recursively witch some parameter bound to a deconstructor applied to itself. Decreases in recursive calls where some parameter is bound to a deconstructor of some other parameter is easily detected by both size-change graph extraction methods. Since mutual recursion is not possible due to the stratification, any infinite call sequence will end in an infinite number of recursive calls from a function to itself. For this reason the program will size-change terminate.

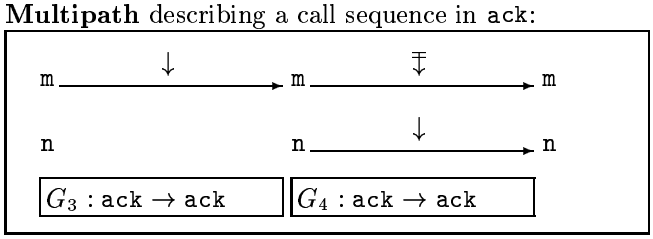
6.4.2 Permuted variables

The programs `nolexicord` (page 59) , `add` (page 48) and `permute` (page 60) each define a simple recursive function that calls itself with a permutation of their parameters (no discarded parameters). In each call at least one parameter is deconstructed. The decrease is easily detected by the size-change graph extraction methods, and a size-change graph with a decrease is generated. Since the function only permutes the parameters, composing the graph with itself repetitively will give a decrease in a parameter over the corresponding call sequence. Because of this the programs are shown to size-change terminate.

6.4.3 Decrease or copy/decrease

The Ackerman programs `ack` (page 78) , (page 48) , `match` (page 67) and `mergelists` (page 57) all contain a single recursive function definition. The defined function can call itself in two ways: either some parameter `x` decreases or the parameter `x` is copied and some other parameter is decreased.

Multipath describing a call sequence in `ack`:



Clearly if a call sequence consists of calls of the first type or second type only, the program has a decrease in one parameter infinitely many times and thus size-change terminates. Otherwise the only possible infinite call sequence must have both call types infinitely many times. So the `x` parameter is decreased and copied infinitely many times giving size-change termination. Since no other possible infinite call sequences exist, the programs must size-change terminate.

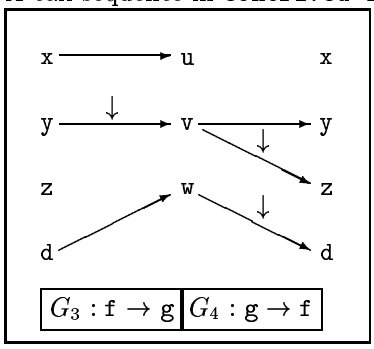
6.4.4 Simple mutual recursion

The programs `evenodd` (page 54) , `odd_even` (page 50) and `disjconj` (page 53) all defines two simple mutually recursive functions. Both functions in the programs take one parameter only, which decreases in each call. Because of this any infinite call sequence will lead to an infinite descent in the size of the parameter, causing size-change termination. The program `disjconj` calls an additional non-recursive function, but clearly this does not influence the detection of size-change termination, as the function can't appear in an infinite call sequence.

6.4.5 contrived-1

The program `contrived-1` (page 52) defines six functions where only three (`f`, `g`, `h`) can appear in an infinite call sequence. Function `h` is of "type" decrease-or-copy/decrease so by the argument in section 6.4.3 it must size-change terminate, so no infinite sequence can have infinitely many calls to `h`. Since `h` can't call back to `f` or `g`, it can't appear in an infinite call sequence at all. The functions `f` and `g` contain a single call to each other and only `f` has a call to itself. An infinite call sequence can't contain calls from `f` to itself since the call has decrease in a parameter value, leading to size-change termination. The only possible infinite call sequence is thus the call sequence `f-g-f`:

A call sequence in `contrived-1`:



Clearly the call sequence above has descent in both parameter `y` and `d`, which leads to termination. Since all infinite call sequences are accounted for then the program must size-change terminate.

6.4.6 type-inf

Considering the functions in the program `type-inf` (page 70) , it is clear that the only non-NSR functions are `unify`, `etype` and `typeinf` (only because they call unify). Because of this the only possible infinite call sequence must contain only calls from `typeinf` to itself since it does not call the other two non-NSR functions. The `unify` function has three parameters `venv`, `t1`, `t2`. Examining the recursive calls of `unify` it is seen the function calls with a decrease in the

value either of `t1` or `t2` in the `t1` and `t2` positions. Thus the call sequence has infinite descent over both parameters, so the function must terminate. If the call to `unify` is disregarded (because it terminates) in functions `etype` and `typeinf`, then they are simply terminating (section 6.4.1) and therefore the program must size-change terminate.

6.4.7 int-loop

The program `int-loop` (page 72) implements a small interpreter for a imperative program with `for`-loops. The only functions that are non-NSR are `eeval` and `run` (because `eeval` is not). The `eeval` has - among others - two parameters: `e` is the expression being evaluated, and `l` is a counter, that counts the number of times the body of a `for`-loop is evaluated. Examining the calls is is clear that either `l` decreases or `e` decreases and `l` is copied in the call. Thus by the argument in section 6.4.3 the function must size-change terminate. Since the function is non-recursive and all the functions it calls size-change terminate, then the function itself must size-change terminate, implying program termination.

6.4.8 nestdec

The program `nestdec` (page 58) consists of a simple recursive function that decreases its single argument until it reaches zero, and then returns a constant. The decrease is computed by the function `dec`, that implements the predecessor function. The simple size-change analysis SCT_0 fails to decide termination for this example since, it would require the analysis to show that the return value of `dec` is strictly smaller than the input. The analysis SCT detects the decrease over `dec` and is thus able to decide size-change termination.

6.4.9 gcd-1, gcd-2

The gcd-1 and gcd-2 examples (pages 78 and 79) both compute the greatest common divisor of the two arguments. The algorithm works by repetitively subtracting the smaller of the two from the larger. The only difference between the two examples is the order of the arguments in the recursive call. Int gcd-2 the arguments are swapped and in gcd-1 they are not. The subtraction is handled by a separate function and showing size-change amounts to showing that the value returned from the subtraction function is strictly less than its input, just as in the `nestdec` example 6.4.8. For this reason the SCT_0 analysis fails to decide size-change termination, whereas the SCT succeeds.

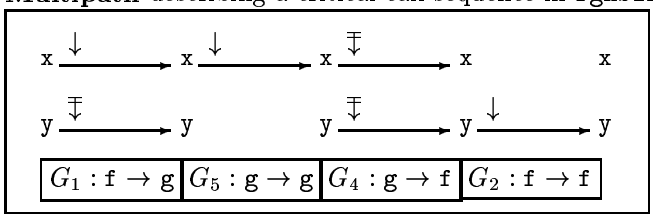
6.4.10 fgh

The program `fghSize` (pages 82, 85) is a slight modification of the `fgh` (pages 48, 85) example from the Wahlstedt test suite (a "2" has been changed to a "1"). The original program was not terminating, but with the modification

it is. The size-change termination algorithms both fail to detect size-change termination.

The program consists of three functions. The function `h` . The call sequence $\{1, 5, 4, 2\}$ results in a size-change graph that gives no information on the relation between the variables, causing possible non-termination.

Multipath describing a critical call sequence in `fghSize`:



An inspection of the program shows that the operations giving rise to the size-changes in the call sequence are simple constructor and destructor applications. The function return analysis can be extended to handle the `fghSize` program.

One possible direction for extension could be to define an abstract domain where increases and decreases in the size-measure can be described more accurately: where decreases such as "decrease by k deconstructions" exist. Unfortunately if we take $\mathcal{P}(\{\ldots, \downarrow 2, \downarrow 1, =, \uparrow 1, \uparrow 2, \ldots\})$ as the abstract domain, the fix-point iteration won't converge in general. The abstract lattice must have finite height in order for the analysis to reach a fix-point.

k-width-cutoff. First the *k-width-cutoff* abstraction is introduced.

$$\mathcal{P}\{\downarrow, \downarrow k, \ldots, \downarrow 1, =, \uparrow 1, \ldots, \uparrow k, \uparrow\}$$

The extended lattice allows for reasoning about size-change sequences that increases and decreases a value, up to some bound (ie. k). given the existing implementation, the k-width-cutoff abstraction is easy to implement. The bound however, will always be somewhat arbitrary so generally the effect of the extension is limited. The lattice is a generalization of the 0-width-cutoff lattice used in the current implementation.

k-height-cutoff. A different approach is to cut the height of the lattice, $K = \mathcal{P}(\{\ldots, \downarrow 2, \downarrow 1, =, \uparrow 1, \uparrow 2, \ldots\})$. The resulting lattice is called the *k-height-cutoff* lattice. This can be done by defining:

$$a_1 \sqcup a_2 = \begin{cases} a_1 \cup a_2 & \text{iff } |a_1 \cup a_2| \leq k \\ \top & \text{otherwise} \end{cases}$$

While the arbitrary bound in width is removed, the abstraction can give worse results than the k-width-cutoff abstraction: $\sqcup_{i \in \{1,\ldots,k+1\}} \downarrow i = \top$ Where the k-width-cutoff abstraction would give $\sqcup_{i \in \{1,\ldots,k+1\}} \downarrow i = \downarrow$, Since $\downarrow k + 1 = \downarrow$.

Thus we define:

$$a_1 \sqcup a_2 = \begin{cases} a_1 \cup a_2 & \text{if } a_1 \in K \wedge a_2 \in K \wedge \neg \mathit{cutoff}(a_1, a_2) \\ \mp & \text{if } \mathit{cutoff}(a_2, a_2) \wedge a_1, a_2 \in \mathit{DownEq} \\ \downarrow & \text{if } \mathit{cutoff}(a_1, a_2) \wedge a_1, a_2 \in \mathit{Down} \\ \updownarrow & \text{if } \mathit{cutoff}(a_1, a_2) \wedge a_1, a_2 \in \mathit{DownUp} \\ \uparrow & \text{if } \mathit{cutoff}(a_1, a_2) \wedge a_1, a_2 \in \mathit{Up} \\ \pm & \text{if } \mathit{cutoff}(a_1, a_2) \wedge a_1, a_2 \in \mathit{UpEq} \\ \top & \text{otherwise} \end{cases},$$

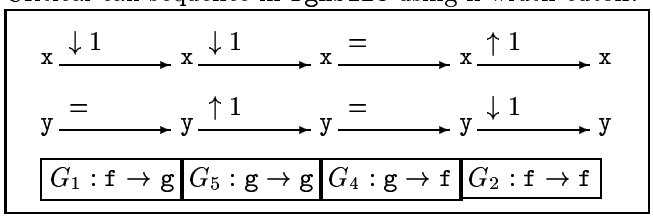
where

$$\begin{array}{ll} \mathit{Down} & = \{\downarrow\} \cup \mathcal{P}(\{\downarrow 1, \downarrow 2, \ldots\}), \\ \mathit{Up} & = \{\uparrow\} \cup \mathcal{P}(\{\uparrow 1, \uparrow 2, \ldots\}). \\ \mathit{DownEq} & = \{\downarrow, \mp\} \cup \mathcal{P}(\{=, \downarrow 1, \downarrow 2, \ldots\}), \\ \mathit{UpEq} & = \{\uparrow, \pm\} \cup \mathcal{P}(\{=, \uparrow 1, \uparrow 2, \ldots\}). \\ \mathit{DownUp} & = \{\overset{\uparrow}{\downarrow}, \downarrow, \uparrow\} \cup \mathcal{P}(\{\ldots, \downarrow 2, \downarrow 1, \uparrow 1, \uparrow 2, \ldots\}). \\ \mathit{cutoff}(a_1, a_2) & = (a_1 \in K \wedge a_2 \in K) \Rightarrow (|a_1 \cup a_2| > k) \end{array}$$

The lattice is the same as before except that $\{\mp, \downarrow, \overset{\uparrow}{\downarrow}, \uparrow, \pm\}$ have been added, correcting the flaw. The k-height-cutoff lattice is strictly more precise than than the k-width-cutoff lattice.

The fgh example with k-width cutoff. If the algorithm was extended to handle k-width-cutoff, and the second phase of the size-change termination algorithm was changed to handle the elements in the lattice as well, then size-change termination can be shown for the `fgh` example. With this analysis (where $k \geq 1$) we find the call graph:

Critical call sequence in `fghSize` using k-width-cutoff:



The call graph has a decrease in the value of x, thus the sequence will lead to termination. Since the sequence was the only one that caused to termination problems previously, the extended algorithm will show termination in the case.

6.4.11 Mergesort

The program `mergesort` (pages 80, 85) performs a merge sort on a list. The termination problems are caused by the recursive calls to mergesort. Since the analysis does not take control flow into consideration, the analysis does not discover that the arguments `xs1` and `xs2` are strictly smaller in size than the list

when `splitmerge` calls `mergesort` on them. Because of this both size-change graph extraction methods (SCT , SCT_0) fail to decide size-change termination.

6.4.12 Minsort

The program `minsort` (pages 80, 85) is an implementation of the minsort algorithm. The termination problems are caused by absence of arcs at call site 5, the call from `appmin` to `minsort`. The analyses fails to find a sufficiently precise size-change graph for the return value of `remove`. To properly handle this example the following could be exploited:

- When `remove` is called then `xs` has a `Cons` constructor at top-level. This can be done by propagating positive information through the program to the functions being called.
- An analysis will show that `min` always is in `xs`, so when `remove` is called, `x` is in `xs`. Further analysis will show that the `then` branch will be taken at least once, thus the size of the returned value will be strictly smaller than `xs`.

6.4.13 Quicksort

The program `quicksort` (pages 81, 85) is an implementation of the quick sort algorithm. The two recursive calls on the divided lists causes possible size-change nontermination. In the output from the analysis this is listed as the call sequences $\{2, 7\}$ and $\{2, 8\}$, where call sites 7 and 8 are the recursive calls and call site 2 is the quicksort function.

The function `part` splits up the list into two new, given a list and an element in the list. One with all the elements that are smaller than or equal to x, and one with all the elements greater that x. The resulting lists are accumulating parameters to the `part` function. When the list has been split, the `quicksort` function is called recursively as usual. This approach causes several problems for the size-change graph analysis.

- The lack of reasoning about the control flow of the program, forces the analysis to think that the recursive calls can be made immediately after the call to `part`. This can be circumvented by propagating the fact that `xs` has a `Cons` constructor on top level when calling `part`. But more work is required to show that `quicksort` terminates.
- The fact that the element used to split up the list is inserted into the list containing all elements less than or equal to it, causes problems. If the splitting element is the largest in the list, then all elements will be inserted into the less-than-or-equal list, so *no size-changes* will occur in that call. Thus the most precise safe description of the call sequence `quicksort`, `part`, `quicksort` (where the last call will be with the less-than-or-equal list) will be no change in the size of the list being sorted. To show size-change termination, the algorithm must show that the largest element can

only be selected a finite number of times (one), over the call-chain, which is anything but trivial.

- Assume that the splitting element is not inserted in one of the lists in the recursive calls to `quicksort`, but afterwards when appending the resulting lists. The task now is to show that the size of the argument in the recursive calls is strictly less than the list being sorted. This can be done by observing that all elements in the accumulating lists are filtered from `xs` (ie. inserted no more than once).

The program `quicksortSize` (pages 83, 85) is a version of `quicksort` where the splitting element is not inserted into the accumulating lists. Showing size-change termination for that program should be somewhat easier.

Semi-filter analysis. An "semi-filter" analysis could be developed to detect functions which return a value with a size that are always less than or equal to the value of some parameter of the function. This is done by detecting functions that return values that are composed purely of subterms of some parameter `x` and where each subterm of the parameter appears no more than once in the return value of the function. Clearly if a function has the "semi-filter" property for some variable `x`, then the return value is always less than or equal to the value of parameter `x` for all input. This gives rise to an abstract value with an $\mp$-arc from the parameter `x`. This can be combined with the size-change analysis to give more precise results. The method is more general than *homeomorphic embedding* [6] of the return value in the parameter (no requirement on the ordering of the subterms, etc.).

Although it may seem like a patch rather than a solution, it does expand the set of programs for which size-change termination can be shown, especially when coupled with size-change analysis of parameter tuples.

quicksortSize with semi-filters. With the semi-filter analysis it should be possible to show that the size of the argument in the recursive calls is strictly less than the list being sorted. All elements in the accumulating lists are filtered from `xs`. The semi-filter analysis should be able to detect this, giving a size-change graph with a decrease in the size of the list being sorted. With such a size-change graph size-change termination can be shown.

6.4.14 assrewrite

The program `assrewrite` (pages 51, 85) takes an expression in the form of a Lisp-style tree. The expression is rewritten such that all occurrences of the associative operator `op` are "pushed to the right", such that the left most term will not have an application of the `op` operator on the level; ie.

$$\texttt{(op (op a b) c)} \rightarrow \texttt{(op a (op b c))}.$$

The expression is contained in a single variable. The problem with this example is that no size-change occurs over the parameter to the function `rewrite`.

The parameter is simply transformed to have a different structure, so any size-measure measuring the amount of storage consumed by the different variables, will fail to decide size-change termination (unless the program is rewritten prior to the analysis).

6.4.15 assrewriteSize

This program `assrewriteSize` (pages 81, 85) is similar to the `assrewrite` (pages 51, 85) in that the same operation is performed, but the immediate subterms to the `op` operator has been split out into different variables when rewriting (function `rw`), instead of reconstructing the expression and then rewrite it. In this way the left most term will decrease in size in each rewrite step until the term can be rewritten no more. Unfortunately the return size analysis is not strong enough to show that rewriting the left most term does not change its size. Because of this, the analysis can not show a decrease in the size of the rewritten left most term, and thus size-change termination can't be shown. To show that the function `rewrite` does not change the size of the argument, a is sufficient (see page 39 for details), in order to handle the swapping of children in a constructor. The semi filter analysis detects functions that returns values that are composed purely of subterms of one variable, and each subterm appears only once. Clearly if a function has the "semi-filter" property for some variable `x`, then the return value is less than or equal to `x`. This gives rise to an abstract value with an $\mp$-arc from the parameter `x`. Using the semifilter analysis and extending the size-change analysis would allow the algorithm to show that `rw` gives rise to increase of at most 1 over the input tuple: $((\texttt{opab}, \texttt{c}), \uparrow 1)$, where the elements in the tuple are the arguments to the `op` application being rewritten. This requires that the semifilter analysis is extended to be able to substitute 0-ary constants for subterms of the parameters, for example: When deconstructing some parameter `x`:

$$\texttt{x} \mapsto \texttt{Cons}(\texttt{a}, \texttt{b}),$$

the analysis should be able to conclude that:

$$\|\texttt{x}\| \leq \|\texttt{Cons}(\texttt{a}, \texttt{Nil})\|$$

Where `b` has been replaced by the 0-ary constructor `Nil`. This is safe since

$$\forall v \in \mathit{Value}, \texttt{con} \in \mathit{Constructors} : \ \mathit{arity}(\texttt{con}) = 0 \Rightarrow \mathit{Nil} \preceq v$$

implying

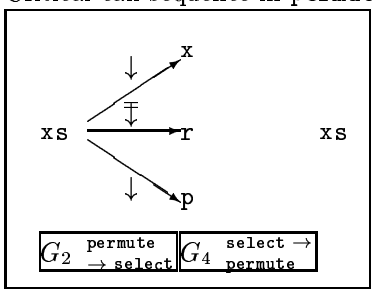
$$\texttt{Cons}(\texttt{a}, \texttt{Nil}) \preceq \texttt{Cons}(\texttt{a}, \texttt{b}).$$

In this way the semifilter analysis would conclude that the result of `rewrite` has a size less than or equal to the input, as desired.

6.4.16 permute

The program `permute` (pages 60, 85) computes all permutations of a list. The program can not be shown to size-change terminate. The problem is caused by the call sequence:

Critical call sequence in `permute` (r = `revprefix`, p = `postfix`):

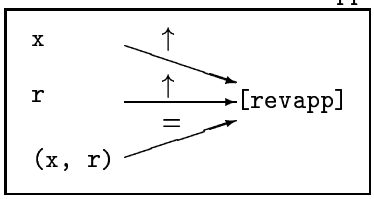


The absence of arcs in the call 4, from `select` to `permute`, is caused by the function `revapp`. The return size analysis is not able to show that:

$$\|(\texttt{xs}, \texttt{rest})\| = \|\texttt{revapp}(\texttt{xs}, \texttt{rest})\| - 1$$

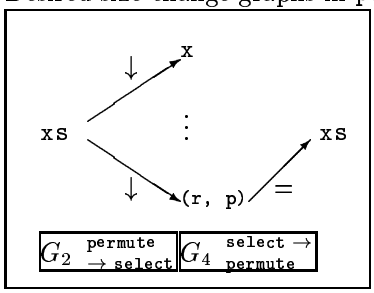
. A possible solution could be to use the semifilter analysis coupled with size-change analysis of tuples, the k-width-cutoff lattice (k¿1) and propagation of positive information. This should give a =-arc in the size of the tuple (`revprefix`, `postfix`) to the returned value in the call to `revappend` in `select`:

Desired return value of `revappend` (x = `xs`, r = `rest`):



The above graph will give an =-arc from `(r, p)` to `xs` at call site 4.

Desired size-change graphs in `permute` (r = `revprefix`, p = `postfix`):



6.4.17 shuffle

The program shuffle (pages 61, 85) takes a list as the single argument and computes a new list where the elements are "shuffled". The new list is computed by alternatingly extracting the first and last element of the list, and inserting them into the result. Example:

```
shuffle([0, 1, 2, 3, 4]) = [0, 4, 1, 3, 2]
```

The shuffling is done by extracting the first element of the list and then calling the `shuffle` function recursively with the list reversed. The size-change analysis *SCT* fails to show that the size the a reversed list is equal to size of the list itself. Thus the size-change analysis does not find the decrease in the recursive call which causes possible nontermination for the program. The program is implemented using a reverse function that calls an append function (pages 61, 85) . Using the semifilter analysis it can be shown that:

$$\|\texttt{append(xs, ys)}\| = \|\texttt{(xs, ys)}\|$$

This information can be used in the call to `append` in `reverse`. In the base case in `reverse`: `x = Nil`, the size of the returned value is equal to the value in `xs`. Proceeding per induction in the non-base case:

```
append(reverse(2nd(xs)), Cons(1st(xs), Nil))
```

So the size is given by:

$$\begin{array}{lcl}
\|\texttt{append(reverse(2nd(xs)), Cons(1st(xs), Nil))}\| & = & \\
\|\texttt{append(2nd(xs), Cons(1st(xs), Nil))}\| & = & \\
\|\texttt{(2nd(xs), Cons(1st(xs), Nil))}\| & = & \\
max\{\|\texttt{(2nd(xs)}\|, \|\texttt{Cons(1st(xs), Nil))}\|\} & = & \\
max\{\|\texttt{(2nd(xs)}\|, 1 + max\{\|\texttt{1st(xs)}\|, \|\texttt{Nil}\|\}\} & = & \\
max\{\|\texttt{xs}\| - 1, 1 + max\{\|\texttt{xs}\| - 1, \|\texttt{Nil}\|\}\} & = & \\
max\{\|\texttt{xs}\| - 1, \|\texttt{xs}\|\} & = & \|\texttt{xs}\|
\end{array}$$

Per induction we now conclude that `reverse` does not increase the size of its input. This example could be generalized to obtain a more powerful algorithm for detecting size-changes.

6.4.18 thetrick

In the program `thetrick` (pages 62, 85) , contains a recursive call (call site 4) where the arguments are two conditionals with the same test expression. As the result indicates, call sequence $[4]^*$ causes possible nontermination, because according to the analysis the arguments might be nondecreasing. This is caused by the way the conditionals are handled; No attempt is made to find out which branch is taken, the result is always the worst-case result of the two branches.

The program `thetrickSize` (pages 84, 85) is the same as `thetrick`, except that the conditional has been lifted out of the context. The this case, no problems arises when deciding termination.

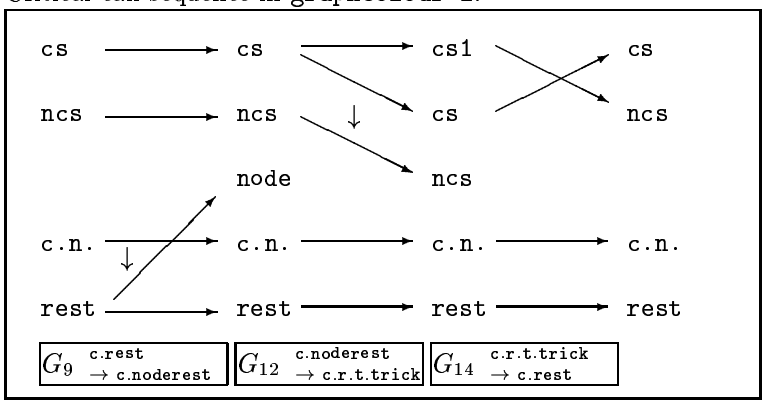
6.4.19 graphcolour-1, 2, 3

The programs `graphcolour-1` (pages 63, 85) , `graphcolour-2` (pages 63, 85) , `graphcolour-3` (pages 63, 85) performs coloring of the nodes in a graph.

The programs are similar, but differ in the way they call themselves recursively. The program `graphcolour-3` size-change terminates for all input, which the algorithms are able to decide.

graphcolour-2. In the program `graphcolour-2` the function `colourresetthetrick` is called instead of the recursive call to the function `colorrest` and one of the parameters `cs` is duplicated in the call as `cs` and `cs1`. The function repetitively applies a destructor to `cs1` in each recursive call, so the function must eventually terminate. When it does it calls `colorrest`, but the `cs1` is used instead of the parameter `ncs` which was used in the call in `graphcolour-1`. Because of this the analysis looses track of the decrease in `ncs`, and can thus not decide size-change termination.

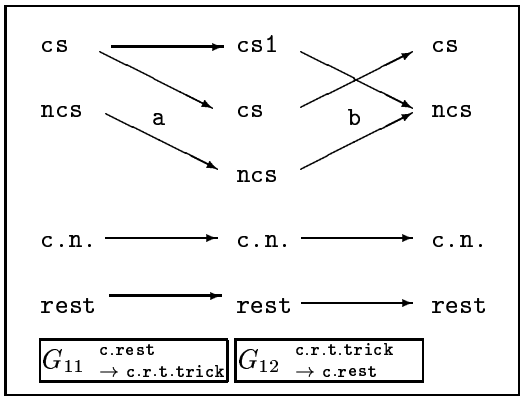
Critical call sequence in `graphcolour-2`:



By inspecting the program it is seen that the call to `colorrest` is made only when `equal(cs1, ncs)` evaluates to `True`, implying that `cs1` and `ncs` maps to the same value in the environment. This provides the missing size-change in the size-change graph, a $\downarrow$-arc from `ncs` to `cs` in G_{14}. The algorithm can obviously be improved by propagating positive information when performing the size-change analysis. This is illustrated in the terminating example `graphcolour2Size` (pages 82, 85) . The program is the identical to `graphcolour2` except that `ncs` is used the recursive call instead of `cs1`.

graphcolour-1. The size-change termination analysis fails to show size-change termination for `graphcolor-1`. The possible nontermination is caused by the missing arcs label `a` and `b` in the multigraph below.

Critical call sequence in `graphcolour-1`:



Examining the program it is evident that the only decrease in the call sequence arises from the argument of the parameter `ncs` in the call from `colorrest` to `colorrestthetrick`, call site 11. The argument involves a call to `colornode`. This function either returns `Nil`, returns `Cons(cs, node)` or call itself recursively with a decrease in `cs`. Clearly the function constructs a list where at least the first element is a subterm of the original `cs` argument (if cs is not `Nil`). The function is called with `ncs` as the argument in the `cs` parameter. In the call at call site 11, the value is deconstructed first with a `1st` and then a `2nd` deconstructor. The resulting value must be a subterm of `ncs` giving rise to the $\downarrow$-arc as the `a`-arc. (if the deconstruction does not fail). The `b`-arc is easily found by propagating positive information as in the `graphcolour-2` example. This gives a $\overline{\downarrow}$-arc as the `b`-arc. All in all, the call sequence contains a decrease in a parameter and must therefore size-change terminate. The analysis is unfortunately somewhat tricky to implement.

6.4.20 rematch

The program `rematch` (pages 67, 85) takes a regular expression and a string and checks whether the string can be generated by the regular expression. This is done by first parsing the string containing the regular expression, and then perform the checking with the parse tree. A problem arises when matching a pattern that can be repeated zero or more times. The pattern is matched against the string. If it matches the result the *-pattern matcher, `domatchstar`, is called again. The main matching function `domatch` returns a value with a pair `Cons`(a, b) iff the matching succeeds. The a is the part of the string that has been matched, and the b part is the remaining part of the string. The size-change analysis is unable to show that the result of the `domatch` function is composed of subtrees from the `cs` parameter, the string to be matched. This causes a overly general size-change graph to be generated for the call from `domatchstar` to itself, giving a possible size-change termination problem.

6.5 Summary

The results of the different analyses are almost identical (again ignoring quasitermination, and failed experiments). The only examples where the analyses differ is `gcd-1`, `gcd-2` and `nestdec`. The SCT_0 analysis fails to show termination in the three examples. The Glenstrup analysis finds quasitermination in example gcd-2, which terminates and SCT show size-change termination for all three. The Glenstrup algorithm runs in PTIME [2] where as the second phase of size-change termination analyses is PSPACE hard [3]. The extra examples handled may not justify the increased complexity of the size-change analyses, given that better results only was obtained in one example in a fairly diverse test suite. However size-change graph extraction phase does allow for extension, and should (perhaps with some supporting analyses) be extendable to handle more examples, including some sorting examples. Given that the size-change graph extraction methods used in this paper are a the simple first steps towards a strong extraction method, the results seem to be fairly good. Clearly there are many directions for investigating how to improve the size-change termination analysis presented in this paper.

7 Future work

Some interesting topics to be investigated:

1. **k-width-cutoff, k-height-cutoff**: Extending the lattice describing the size-changes occurring in the size-change graphs, is a simple modification of the existing algorithm, that generalizes the lattice.

2. **Size-change over parameter tuples**: Extending the analysis clearly increases the power of the algorithm. Such an extension would allow for detection of more complex examples.

3. **Semifilter analysis**: The "semifilter" analysis (section 6.4.13) seems like it does allow detection of size-change termination in many "typical" cases that otherwise would not classified as terminating and a simple version is relatively easily implemented. However a more general approach would be desired.

4. **Propagation of positive information**: Some of the cases of failed termination detection could simply be avoided by propagating positive (and negative as well) information. A more radical approach would be to perform speculative unfolding of the program, in order to get more information about the parameter sizes, but the overall strategy is not clear.

5. **Program transformation**: Another possibility is the perform various program transformation on the program in the hope that the transformations will result in program for which size-change termination is detected more easily.

6. **improving the second phase**: The current implementation uses only $\downarrow$ and $\overline{\downarrow}$ arcs when computing the set of composed size-change graphs. By extending the phase to handle the size-changes from the size-change graph extraction phase, descent can be shown for some call sequence with *increases* over some of the calls. Clearly the current lattice is insufficient, but the k-width-cutoff lattice should be able to handle some of these examples. to Arc

7. **Control flow analysis**: As the results showed, controlflow analysis can help expand the set of program for which size-change termination can be shown.

8. **Removal of dead code**: As the example `deadcodeSize` (section A.8.2) showed, dead code can cause terminating programs to be classified as nonterminating. By adding a dead code analysis, the functions which never gets called can be marked and ignored in th e subsequent termination analysis.

9. **Stronger size measure**: in some cases the "maximal subtree" size-measure on which the algorithm is built upon, is insufficient to show size-change termination. A more precise size-measure could account for the sizes of the individual subtrees as well, as described in the section on the `shuffle` example (section 6.4.17).

10. **A more formal treatment of the analysis**: The arguments in this paper are some what "loose". A more rigorous treatment proving the claims of this paper is needed.

11. **Quasitermination**: An algorithm that detects size-change quasitermination has applications in binding time analysis for partial evaluation. Since many of the examples in the Glenstrup test suite are quasiterminating, the test suite is well suited for testing implementations of size-change quasitermination analyses.

8 Acknowledgements

The author would like to thank the following people without whom the present paper probably never would have been written: Professor Neil D. Jones for his guidance and for keeping the author's ambitions on a reasonable level. Chin Soon Lee for giving pointers to what *not* to do when writing a termination analysis as well as insight and background information. David Wahlstedt kindly provided the source code for a size-change termination analysis that the analysis in this paper builds upon, and explained some of the technicalities of the implementation. Arne J. Glenstrup explained the basics of his termination analysis and provided the program text for the examples in test suite from his Master's thesis.

A Examples

A.1 Jones Examples

A.1.1 ex1

```
1  --
2  -- Example 1: Reverse function, with accumulating parameter
3  --
4  rev(ls)    = r1(ls, Nil)
5  r1(ls, a) = if eq(ls, Nil) then a
6                 else r1(2nd(ls), Cons(1st(ls), a))
```

A.1.2 ex2

```
1  --
2  -- Example 2: Function with indirect recursion
3  --
4  f(i, x)     = if eq(i, Nil) then x else g(2nd(i), x, i)
5  g(a, b, c) = f(a, Cons(b, c))
```

A.1.3 ex3

```
1  --
2  -- Example 3: Function with lexically ordered parameters
3  --
4  a(m, n) = if eq(m, Z) then S(n) else
5            if eq(n, Z) then a(1st(m), S(Z))
6                        else a(1st(m), a(m, 1st(n)))
```

A.1.4 ex4

```
1  --
2  -- Example 4: Program with permuted parameters
3  --
4  p(m, n, r) = if gt(r, Z) then p(m, 1st(r), n) else
5               if gt(n, Z) then p(r, 1st(n), m)
6                           else m
```

A.1.5 ex5

```
1  --
2  -- Example 5: Program with promuted and possibly discarded paramters
3  --
4  f(x, y) = if eq(y, Nil) then x else
5            if eq(x, Nil) then f(y, 2nd(y))
6                          else f(y, 2nd(x))
```

A.1.6 ex6

```
1  --
2  -- Example 6:
3  -- Program with late starting sequence of descending parameter values
4  --
5  f(a, b) = if eq(b, Nil) then g(a, Nil)
6                          else f(Cons(1st(b), a), 2nd(b))
7  g(c, d) = if eq(c, Nil) then d
8                          else g(2nd(c), Cons(1st(c), d))
```

A.2 Wahlstedt Examples

A.2.1 ack

```
ack(x1,x2) = if eq(x1, Z)
               then S(x2)
               else if eq(x2, Z)
                      then ack(1st(x1), S(Z))
                      else ack(1st(x1), ack(x1, 1st(x2)))
```

A.2.2 add

```
add(x1,x2) = if eq(x1, Z)
               then x2
               else S(add(x2, 1st(x1)))
```

A.2.3 fgh

```
-- example from Andreas Abel 3.12:
f(x1,x2) = if eq(x1, Z)
             then x1
             else if eq(x2, Z)
                    then x2
                    else h(g(1st(x1), x2), f(S(S(x1)), 1st(x2)))

g(x1,x2) = if eq(x1, Z)
             then x1
             else if eq(x2, Z)
                     then x2
                     else h(f(x1,x2), g(1st(x1), S(x2)))

h(x1,x2) = if eq(x1, Z)
             then if eq(x2, Z)
                    then x2
                    else h(x1, 1st(x2))
             else h(1st(x1), x2)
```

A.2.4 boolprog

```
{-- Neil Jones et al. example of hard time and space complexity:

 This example was used in [Jones00] to show PSPACE-hardness of the algorithm.
 The hard part is to generate all the finite compositions of
 size-change graphs. the more swappings the harder.

Should be hard to compute and does not size-change terminate.

1: X := not X
2: if Y goto 5 else 3
3: Y := not Y
4: if X goto 2 else 3
5: X := not X
6: Y := not Y

Each line is a function that calls another function(line). The
parameters are all the variables in the program, and their boolean
inverses, plus one extra argument z. We always have a call from the
last line to the first.  From each line there is one call to the next
line in the program, except for if-statements, where there are two
calls, one for each branch to corresponding line number.

So now we get nine graphs.

Below we construct a program that give the same set of size-change
graphs and the same call graph as the example above would give:
```

```
27
28  --}
29
30  f0(x1,x2,x3,x4,x5)=if eq(x1, Z)
31                        then Z
32                        else if eq(x4,Z)
33                               then Z
34                               else f1(1st(x2),x2,1st(x4),x4,S(S(x5)))
35
36  f1(x1,x2,x3,x4,x5)=if eq(x5, Z)
37                        then Z
38                        else f2(x2,x1,x3,x4,1st(x5))
39
40  f2(x1,x2,x3,x4,x5)=
41    if eq(x3, Z)
42      then Z
43      else if eq(x4, Z)
44             then if eq(x5, Z)
45                    then Z
46                    else f3(x1,x2,1st(x3),x4,1st(x5))
47             else if eq(x5, Z)
48                    then Z
49                    else f5(x1,x2,x3,1st(x4),1st(x5))
50
51  f3(x1,x2,x3,x4,x5)=if eq(x5, Z)
52                        then Z
53                        else f4(x1,x2,x4,x3,1st(x5))
54
55  f4(x1,x2,x3,x4,x5)=
56    if eq(x1, Z)
57      then Z
58      else if eq(x2, Z)
59             then if eq(x5, Z)
60                    then Z
61                    else f3(1st(x1),x2,x3,x4,1st(x5))
62             else if eq(x5, Z)
63                    then Z
64                    else f2(x1,1st(x2),x3,x4,1st(x5))
65
66  f5(x1,x2,x3,x4,x5)=if eq(x5, Z)
67                        then Z
68                        else f6(x2,x1,x3,x4,1st(x5))
69
70  f6(x1,x2,x3,x4,x5)=if eq(x5, Z)
71                        then Z
72                        else f0(x1,x2,x4,x3,1st(x5))
```

A.2.5 ex6

```
1  {--
2
3  Ex 6 from [Jones 00]:
4  Program with late-starting sequence of decreasing parameter values:
5
6  f(a,b) = if b = [] then g(a,[]) else f(hd b:a, tl b)
7  g(c,d) = if c = [] then d else g(tl c, hd c:d)
8
9  --}
10
11 -- in this language:
12 f(x1,x2) = if eq(x2, Z)
13              then g(x1, Z)
14              else f(S(x1), 1st(x2))
15 g(x1,x2) = if eq(x1, Z)
16              then x2
17              else g(1st(x1), S(x2))
```

A.2.6 permut

```
-- simple example of swapped parameters
f(x1,x2)=if eq(x1, Z)
            then Z
            else f(x2, 1st(x1))
```

A.2.7 eq

```
-- equality test
eq0(x1,x2) = if eq(x1, Z)
               then if eq(x2, Z)
                      then S(Z)
                      else Z
               else if eq(x2, Z)
                      then Z
                      else eq0(1st(x1), 1st(x2))
```

A.2.8 oddeven

```
-- odd / even function
even(x1) = if eq(x1, Z)
             then S(Z)
             else odd(1st(x1))
odd(x1)  = if eq(x1, Z)
             then Z
             else even(1st(x1))
```

A.2.9 div2

```
-- example of nested case on one variable:
div2(x1) = if eq(x1, Z)
             then Z
             else if eq(1st(x1), Z)
                    then Z
                    else S(div2(1st(1st(x1))))
```

A.3 Glenstrup Examples - basic

A.3.1 add

```
--
-- Source file: basic\add.scm
--
-- Add two numbers unarily represented as '(s s s ... s)
goal(x, y) = add0(x, y)
add0(x, y) =
  if equal(y, Nil)
    then x
    else add0(Cons(Cons(Nil, Nil), x), 2nd(y))
```

A.3.2 addlists

```
--
-- Source file: basic\addlists.scm
--
--- Add two lists elementwise
goal(xs, ys) = addlist(xs, ys)
addlist(xs, ys) =
  if eq(xs, Cons)
    then Cons(add(1st(xs), 1st(ys)), addlist(2nd(xs), 2nd(ys)))
    else Nil
```

A.3.3 anchored

```
1  --
2  -- Source file: basic\anchored.scm
3  --
4  -- Parameter y anchored in parameter x
5  goal(x, y) = anchored(x, y)
6  anchored(x, y) =
7    if equal(x, Nil)
8      then y
9      else anchored(2nd(x), Cons(Cons(Nil, Nil), y))
```

A.3.4 append

```
1  --
2  -- Source file: basic\append.scm
3  --
4  goal(x, y) = append(x, y)
5  append(xs, ys) =
6    if equal(xs, Nil)
7      then ys
8      else Cons(1st(xs), append(2nd(xs), ys))
```

A.3.5 assrewrite

```
1  --
2  -- Source file: basic\assrewrite.scm
3  --
4  --- Rewrite expression with associative operator `op'
5  ---     a -> a1     b -> b1     c -> c1
6  --- ------------------------------------
7  --- '(op (op a b) c) -> '(op a1 (op b1 c1))
8  ---a != 'op   a -> a1   b -> b1              a != '(op ...)
9  --- -------------------------                ------------
10 ---  '(op a b) -> '(op a1 b1)                  a -> a
11 assrewrite(exp) = rewrite(exp)
12 rewrite(exp) =
13   if and(eq(exp, Cons), equal(Op, 1st(exp)))
14     then
15       let
16         opab = 1st(2nd(exp))
17       in
18
19         if and(eq(opab, Cons), equal(Op, 1st(opab)))
20           then
21             let
22               a1 = rewrite(1st(2nd(opab)))
23               b1 = rewrite(1st(2nd(2nd(opab))))
24               c1 = rewrite(1st(2nd(2nd(exp))))
25             in
26               rewrite(Cons(1st(exp), Cons(a1, Cons(Cons(1st(opab), Cons(b1,
27                       Cons(c1, 2nd(2nd(2nd(opab)))))), 2nd(2nd(2nd(exp)))))))
28           else Cons(1st(exp), Cons(rewrite(1st(2nd(exp))),
29                Cons(rewrite(1st(2nd(2nd(exp)))), 2nd(2nd(2nd(exp))))))
30     else exp
```

A.3.6 badd

```
1  --
2  -- Source file: basic\badd.scm
3  --
4  goal(x, y) = badd(x, y)
5  badd(x, y) =
6    if equal(y, Nil)
7      then x
8      else badd(Cons(Nil, Nil), badd(x, 2nd(y)))
```

A.3.7 contrived1

```
--
-- Source file: basic\contrived1.scm
--
-- A contrived example from Arne Glenstrup's Master's Thesis
-- Numbers represented by list length
contrived1(a, b) = f(a, Cons(Cons(Nil, Nil), Cons(Cons(Nil, Nil), a)), a, b)
f(x, y, z, d) =
  if and(gt(z, Zero), gt(d, Zero))
    then f(Cons(Cons(Nil, Nil), x), z, 2nd(z), 2nd(d))
    else
      if gt(y, Zero)
        then g(x, 2nd(y), d)
        else x
g(u, v, w) =
  if gt(w, Zero)
    then f(Cons(Cons(Nil, Nil), u),
        if equal(h(v, Zero), Zero)
          then v
          else dec(v), 2nd(v), 2nd(w))
    else u
h(r, s) =
  if gt(r, Zero)
    then h(2nd(r), number42(Nil))
    else
      if gt(s, Zero)
        then h(r, 2nd(s))
        else r
dec(n) = 2nd(n)
number42(n) = Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(
Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons
(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Con
s(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Co
ns(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, C
ons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Nil))))))))))))))))))))))))))))))))))))
)))))
```

A.3.8 contrived2

```
--
-- Source file: basic\contrived2.scm
--
-- A contrived example from Arne Glenstrup's Master's Thesis
-- Numbers represented by list length
contrived2(a, b) = f(a, Cons(Cons(Nil, Nil), Cons(Cons(Nil, Nil), a)), a, b)
f(x, y, z, d) =
  if and(gt(z, Zero), gt(d, Zero))
    then f(Cons(Cons(Nil, Nil), x), z, 2nd(z), 2nd(d))
    else
      if gt(y, Zero)
        then g(x, 2nd(y), d)
        else x
g(u, v, w) =
  if gt(w, Zero)
    then f(Cons(Cons(Nil, Nil), u),
        if equal(h(v, Zero), Zero)
          then v
          else dec(v), 2nd(v), 2nd(w))
    else u
h(r, s) =
  if gt(r, Zero)
    then h(2nd(r), number42(Nil))
    else
      if gt(s, Zero)
        then h(number17(Nil), 2nd(s))
        else r
dec(n) = 2nd(n)
```

```
number42(n) = Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(
Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons
(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Con
s(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Co
ns(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, C
ons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Nil))))))))))))))))))))))))))))))))))))
)))))
number17(n) = Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(
Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons
(Nil, Cons(Nil, Cons(Nil, Nil)))))))))))))))))
```

A.3.9 decrease

```
--
-- Source file: basic\decrease.scm
--
goal(x) = decrease(x)
decrease(x) =
  if equal(x, Nil)
    then number42(Nil)
    else decrease(2nd(x))
number42(n) = Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(
Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons
(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Con
s(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Co
ns(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, C
ons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Nil))))))))))))))))))))))))))))))))))))
)))))
```

A.3.10 deeprev

```
--
-- Source file: basic\deeprev.scm
--
--- Recursively reverse all list elements in a data structure
--- Example: (deeprev '((1 2 3) 4 5 6 (8 (9 10 11)) . 12))
---      ===>((3 2 1) 4 5 6 ((11 10 9) 8) . 12)
goal(x) = deeprev(x)
deeprev(x) =
  if eq(x, Cons)
    then deeprevapp(x, Nil)
    else x
deeprevapp(xs, rest) =
  if eq(xs, Cons)
    then deeprevapp(2nd(xs), Cons(deeprev(1st(xs)), rest))
    else
      if equal(xs, Nil)
        then rest
        else revconsapp(rest, xs)
revconsapp(xs, r) =
  if eq(xs, Cons)
    then revconsapp(2nd(xs), Cons(1st(xs), r))
    else r
```

A.3.11 disjconj

```
--
-- Source file: basic\disjconj.scm
--
--- Predicates for disjunctive and conjunctive terms p
disjconj(p) = disj(p)
disj(p) =
  if eq(p, Cons)
    then
      if equal(Or, 1st(p))
```

```
          then and(conj(1st(2nd(p))), disj(2nd(2nd(p))))
          else conj(p)
      else conj(p)
conj(p) =
  if eq(p, Cons)
    then
      if equal(And, 1st(p))
        then and(disj(1st(2nd(p))), conj(2nd(2nd(p))))
        else bool(p)
    else bool(p)
bool(p) = or(equal(F, p), equal(T, p))
```

A.3.12 duplicate

```
--
-- Source file: basic\duplicate.scm
--
--- Compute a list where each element is duplicated
goal(x) = duplicate(x)
duplicate(xs) =
  if equal(xs, Nil)
    then Nil
    else Cons(1st(xs), Cons(1st(xs), duplicate(2nd(xs))))
```

A.3.13 equal

```
--
-- Source file: basic\equal.scm
--
goal(x) = equal0(x)
equal0(x) =
  if equal(x, Nil)
    then number42(Nil)
    else equal0(x)
number42(n) = Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(
Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons
(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Con
s(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Co
ns(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, C
ons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Nil)))))))))))))))))))))))))))))))))))
)))))
```

A.3.14 evenodd

```
--
-- Source file: basic\evenodd.scm
--
--- Predicate: is x, unarily represented as '(s s s ... s), even/odd
evenodd(x) = even(x)
even(x) =
  if eq(x, Nil)
    then True
    else odd(2nd(x))
odd(x) =
  if eq(x, Cons)
    then even(2nd(x))
    else False
```

A.3.15 fold

```
--
-- Source file: basic\fold.scm
--
-- The fold operators, using a fixed operator, op
fold(a, xs) = Cons(foldl(a, xs), Cons(foldr(a, xs), Nil))
```

```
foldl(a, xs) =
  if eq(xs, Cons)
    then foldl(op(a, 1st(xs)), 2nd(xs))
    else a
foldr(a, xs) =
  if eq(xs, Cons)
    then op(1st(xs), foldr(a, 2nd(xs)))
    else a
op(x1, x2) = add(x1, x2)
```

A.3.16 game

```
--
-- Source file: basic\game.scm
--
-- The game function from Manuvir Das' PhD Thesis (p. 137)
goal(p1, p2, moves) = game(p1, p2, moves)
game(p1, p2, moves) =
  if equal(moves, Nil)
    then Cons(p1, p2)
    else
      if equal(1st(moves), Swap)
        then game(p2, p1, 2nd(moves))
        else
          if equal(1st(moves), Capture)
            then game(Cons(1st(p2), p1), 2nd(p2), 2nd(moves))
            else Error
```

A.3.17 increase

```
--
-- Source file: basic\increase.scm
--
goal(x) = increase(x)
increase(x) =
  if equal(x, Nil)
    then number42(Nil)
    else increase(Cons(Cons(Nil, Nil), x))
number42(n) = Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(
Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons
(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Con
s(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Co
ns(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, C
ons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Nil)))))))))))))))))))))))))))))))))))
)))))
```

A.3.18 intlookup

```
--
-- Source file: basic\intlookup.scm
--
-- The function call case of an interpreter
-- Function number represented as list length
run(e, p) = intlookup(e, p)
intlookup(e, p) = intlookup(lookup(e, p), p)
lookup(fnum, p) =
  if equal(fnum, Nil)
    then 1st(p)
    else lookup(2nd(fnum), 2nd(p))
```

A.3.19 letexp

```
1  --
2  -- Source file: basic\letexp.scm
3  --
4  -- Testing the let construction
5  goal(x, y) = letexp(x, y)
6  letexp(x, y) =
7    let
8      z = Cons(Cons(Nil, Nil), x)
9    in
10     letexp(z, y)
```

A.3.20 list

```
1  --
2  -- Source file: basic\list.scm
3  --
4  -- The predicate for checking whether the argument is a list
5  goal(x) = list(x)
6  list(xs) =
7    if eq(xs, Cons)
8      then list(2nd(xs))
9      else eq(xs, Nil)
```

A.3.21 lte

```
1  --
2  -- Source file: basic\lte.scm
3  --
4  -- Less than or equal
5  goal(x, y) = and(lte(x, y), even(x))
6  lte(x, y) =
7    if eq(x, Nil)
8      then True
9      else
10       if and(eq(x, Cons), eq(y, Cons))
11         then lte(2nd(x), 2nd(y))
12         else False
13 even(x) =
14   if eq(x, Nil)
15     then True
16     else
17       if eq(2nd(x), Nil)
18         then False
19         else even(2nd(2nd(x)))
```

A.3.22 map0

```
1  --
2  -- Source file: basic\map0.scm
3  --
4  -- The map function with fixed function f
5  goal(xs) = map(xs)
6  map(xs) =
7    if equal(xs, Nil)
8      then Nil
9      else Cons(f(1st(xs)), map(2nd(xs)))
10 f(x) = mul(x, x)
```

A.3.23 member

```
1  --
2  -- Source file: basic\member.scm
3  --
4  -- The member function
5  goal(x, xs) = member(x, xs)
6  member(x, xs) =
7    if equal(xs, Nil)
8      then
9        if equal(x, 1st(xs))
10         then True
11         else member(x, 2nd(xs))
12     else False
```

A.3.24 mergelists

```
1  --
2  -- Source file: basic\mergelists.scm
3  --
4  --- Merge two lists
5  goal(xs, ys) = merge(xs, ys)
6  merge(xs, ys) =
7    if equal(xs, Nil)
8      then ys
9      else
10       if equal(ys, Nil)
11         then xs
12         else
13           if lte(1st(xs), 1st(ys))
14             then Cons(1st(xs), merge(2nd(xs), ys))
15             else Cons(1st(ys), merge(xs, 2nd(ys)))
```

A.3.25 mul

```
1  --
2  -- Source file: basic\mul.scm
3  --
4  --- Unary multiplication and addition, e.g. (mul '(s s z) '(s s s z))
5  goal(x, y) = mul0(x, y)
6  mul0(x, y) =
7    if equal(x, Nil)
8      then Nil
9      else add0(mul0(2nd(x), y), y)
10 add0(x, y) =
11   if equal(x, Nil)
12     then y
13     else add0(2nd(x), Cons(S, y))
```

A.3.26 naiverev

```
1  --
2  -- Source file: basic\naiverev.scm
3  --
4  -- Naive reverse function
5  goal(xs) = naiverev(xs)
6  naiverev(xs) =
7    if equal(xs, Nil)
8      then xs
9      else app(naiverev(2nd(xs)), Cons(1st(xs), Nil))
10 app(xs, ys) =
11   if equal(xs, Nil)
12     then ys
13     else Cons(1st(xs), app(2nd(xs), ys))
```

A.3.27 nestdec

```
--
-- Source file: basic\nestdec.scm
--
-- Parameter decrease by nested call in recursion
goal(x) = nestdec(x)
nestdec(x) =
  if equal(x, Nil)
    then number17(Nil)
    else nestdec(dec(x))
dec(x) =
  if equal(x, Cons(Nil, Nil))
    then 2nd(x)
    else dec(2nd(x))
number17(n) = Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(
Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons
(Nil, Cons(Nil, Cons(Nil, Nil)))))))))))))))))
```

A.3.28 nesteql

```
--
-- Source file: basic\nesteql.scm
--
-- Parameter equality by nested call in recursion
goal(x) = nesteql(x)
nesteql(x) =
  if equal(x, Nil)
    then number17(Nil)
    else nesteql(eql(x))
eql(x) =
  if equal(x, Nil)
    then x
    else eql(x)
number17(n) = Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(
Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons
(Nil, Cons(Nil, Cons(Nil, Nil)))))))))))))))))
```

A.3.29 nestimeql

```
--
-- Source file: basic\nestimeql.scm
--
-- Using an immaterial copy as recursive argument
goal(x) = nestimeql(x)
nestimeql(x) =
  if equal(x, Nil)
    then number42(Nil)
    else nestimeql(immatcopy(x))
immatcopy(x) =
  if equal(x, Nil)
    then Nil
    else Cons(Nil, immatcopy(2nd(x)))
number42(n) = Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(
Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons
(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Con
s(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Co
ns(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, C
ons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Nil)))))))))))))))))))))))))))))))))))
)))))
```

A.3.30 nestinc

```
--
-- Source file: basic\nestinc.scm
--
```

```
-- Parameter increase by nested call in recursion
goal(x) = nestinc(x)
nestinc(x) =
  if equal(x, Nil)
    then number17(Nil)
    else nestinc(inc(x))
inc(x) =
  if equal(x, Nil)
    then Cons(Nil, Nil)
    else Cons(Cons(Nil, Nil), inc(2nd(x)))
number17(n) = Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(
Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons
(Nil, Cons(Nil, Cons(Nil, Nil)))))))))))))))))
```

A.3.31 nolexicord

```
--
-- Source file: basic\nolexicord.scm
--
-- Example not termination-provable by simple lexicographical ordering
goal(a1, b1, a2, b2, a3, b3) = nolexicord(a1, b1, a2, b2, a3, b3)
nolexicord(a1, b1, a2, b2, a3, b3) =
  if equal(a1, Nil)
    then number42(Nil)
    else
      if equal(a1, b1)
        then nolexicord(2nd(b1), 2nd(a1), 2nd(a2), 2nd(b2), 2nd(b3), 2nd(a3))
        else nolexicord(2nd(b1), 2nd(a1), 2nd(b2), 2nd(a2), 2nd(a3), 2nd(b3))
number42(n) = Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(
Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons
(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Con
s(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Co
ns(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, C
ons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Nil)))))))))))))))))))))))))))))))))))
)))))
```

A.3.32 ordered

```
--
-- Source file: basic\ordered.scm
--
-- Predicate that checks whether a list is ordered
goal(xs) = ordered(xs)
ordered(xs) =
  if eq(xs, Cons)
    then
      if eq(2nd(xs), Cons)
        then
          if lte(1st(xs), 1st(2nd(xs)))
            then ordered(2nd(2nd(xs)))
            else False
        else True
    else True
```

A.3.33 overlap

```
--
-- Source file: basic\overlap.scm
--
-- Predicate for checking whether there is an overlap of two sets
goal(xs, ys) = overlap(xs, ys)
overlap(xs, ys) =
  if eq(xs, Cons)
    then
      if member(1st(xs), ys)
        then True
```

```
          else overlap(2nd(xs), ys)
      else False
member(x, xs) =
  if eq(xs, Cons)
    then
      if equal(1st(xs), x)
        then True
        else member(x, 2nd(xs))
    else False
```

A.3.34 permute

```
--
-- Source file: basic\permute.scm
--
-- Compute all the permutations of a list
-- Select x as the first element and cons it onto
-- permutations of the remaining list represented by
-- the list of elements before x (reversed) and the
-- list of elements after x. Finally, recurse by moving
-- on to the next element in postfix
-- Map '(cons x' onto the list of lists xss and append the rest
-- Reverse xs and append the rest
goal(xs) = permute(xs)
permute(xs) =
  if equal(xs, Nil)
    then Cons(Nil, Nil)
    else select(1st(xs), Nil, 2nd(xs))
select(x, revprefix, postfix) =
  mapconsapp(x, permute(revapp(revprefix, postfix)),
    if equal(postfix, Nil)
      then Nil
      else select(1st(postfix), Cons(x, revprefix), 2nd(postfix)))
mapconsapp(x, xss, rest) =
  if equal(xss, Nil)
    then rest
    else Cons(Cons(x, 1st(xss)), mapconsapp(x, 2nd(xss), rest))
revapp(xs, rest) =
  if equal(xs, Nil)
    then rest
    else revapp(2nd(xs), Cons(1st(xs), rest))
```

A.3.35 revapp

```
--
-- Source file: basic\revapp.scm
--
--- Reverse list and append to rest
goal(x, y) = revapp(x, y)
revapp(xs, rest) =
  if equal(xs, Nil)
    then rest
    else revapp(2nd(xs), Cons(1st(xs), rest))
```

A.3.36 select

```
--
-- Source file: basic\select.scm
--
-- Compute a list of lists. Each list is computed by picking out an
-- element of the original list and consing it onto the rest of the list
-- Reverse xs and append to rest
select(xs) =
  if equal(xs, Nil)
    then Nil
```

```
10      else selects(1st(xs), Nil, 2nd(xs))
11  selects(x, revprefix, postfix) = Cons(Cons(x, revapp(revprefix, postfix)),
12      if equal(postfix, Nil)
13        then Nil
14        else selects(1st(postfix), Cons(x, revprefix), 2nd(postfix)))
15  revapp(xs, rest) =
16    if equal(xs, Nil)
17      then rest
18      else revapp(2nd(xs), Cons(1st(xs), rest))
```

A.3.37 shuffle

```
1  --
2  -- Source file: basic\shuffle.scm
3  --
4  -- Shuffle List
5  goal(xs) = shuffle(xs)
6  shuffle(xs) =
7    if equal(xs, Nil)
8      then Nil
9      else Cons(1st(xs), shuffle(reverse(2nd(xs))))
10 reverse(xs) =
11   if equal(xs, Nil)
12     then xs
13     else append(reverse(2nd(xs)), Cons(1st(xs), Nil))
14 append(xs, ys) =
15   if equal(xs, Nil)
16     then ys
17     else Cons(1st(xs), append(2nd(xs), ys))
```

A.3.38 sp1

```
1  --
2  -- Source file: basic\sp1.scm
3  --
4  -- Mutual recursion requiring specialisation points
5  sp1(x, y) = f(x, y)
6  f(x, y) =
7    if equal(x, Nil)
8      then g(x, y)
9      else h(x, y)
10 g(x, y) =
11   if equal(x, Nil)
12     then h(x, y)
13     else r(x, y)
14 h(x, y) =
15   if equal(x, Nil)
16     then h(x, y)
17     else f(x, y)
18 r(x, y) = x
```

A.3.39 subsets

```
1  --
2  -- Source file: basic\subsets.scm
3  --
4  -- Compute all subsets
5  -- map `(cons x' ont the list of lists xss, and append rest
6  goal(xs) = subsets(xs)
7  subsets(xs) =
8    if eq(xs, Cons)
9      then
10       let
11         subs = subsets(2nd(xs))
12       in
13         mapconsapp(1st(xs), subs, subs)
```

```
      else Cons(Nil, Nil)
mapconsapp(x, xss, rest) =
  if eq(xss, Cons)
    then Cons(Cons(x, 1st(xss)), mapconsapp(x, 2nd(xss), rest))
    else rest
```

A.3.40 thetrick

```
--
-- Source file: basic\thetrick.scm
--
-- The trick: pulling out the conditional into the context
goal(x, y) = Cons(f(x, y), Cons(g(x, y), Nil))
f(x, y) =
  if equal(y, Nil)
    then number42(Nil)
    else f(
        if lt0(x, Cons(Nil, Nil))
          then x
          else 2nd(x),
        if lt0(x, Cons(Nil, Nil))
          then 2nd(y)
          else Cons(Cons(Nil, Nil), y))
g(x, y) =
  if equal(y, Nil)
    then number42(Nil)
    else
      if lt0(x, Cons(Nil, Nil))
        then g(x, 2nd(y))
        else g(2nd(x), Cons(Cons(Nil, Nil), y))
lt0(x, y) =
  if equal(y, Nil)
    then False
    else
      if equal(x, Nil)
        then True
        else lt0(2nd(x), 2nd(y))
number42(n) = Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(
Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons
(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Con
s(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Co
ns(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, C
ons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Nil)))))))))))))))))))))))))))))))))))
)))))
```

A.3.41 vangelder

```
--
-- Source file: basic\vangelder.scm
--
--- Following is an example due to Allen Van Gelder.
--- Note that in the following example there is a
--- cycle involving q, p, r, t, and q again, such that
--- nothing gets smaller along that cycle.
--- e(a,b).
--- q(X,Y)        :- e(X,Y).
--- q(X,f(f(X))) :- p(X,f(f(X))), q(X,f(X)).
--- q(X,f(f(Y))) :- p(X,f(Y)).
---
--- p(X,Y)        :- e(X,Y).
--- p(X,f(Y))     :- r(X,f(Y)), p(X,Y).
---
--- r(X,Y)        :- e(X,Y).
--- r(X,f(Y))     :- q(X,Y), r(X,Y).
--- r(f(X),f(X)) :- t(f(X),f(X)).
---
```

```
--- t(X,Y)        :- e(X,Y).
--- t(f(X),f(Y)) :- q(f(X),f(Y)), t(X,Y).
goal(x, y) = q(x, y)
e(a, b) = and(equal(a, A), equal(b, B))
q(x, y) =
  if e(x, y)
    then True
    else
      if and(eq(y, Cons), and(equal(1st(y), F), and(eq(2nd(y), Cons),
             equal(1st(2nd(y)), F))))
        then
          if and(p(x, y), q(x, 2nd(y)))
            then True
            else p(x, 2nd(y))
        else False
p(x, y) =
  if e(x, y)
    then True
    else
      if equal(F, 1st(y))
        then and(r(x, y), p(x, 2nd(y)))
        else False
r(x, y) =
  if e(x, y)
    then True
    else
      if and(eq(y, Cons), equal(1st(y), F))
        then
          if and(q(x, 2nd(y)), r(x, 2nd(y)))
            then True
            else
              if and(eq(x, Cons), equal(1st(x), F))
                then t(x, y)
                else False
        else False
t(x, y) =
  if e(x, y)
    then True
    else
      if and(eq(x, Cons), and(equal(1st(x), F), and(eq(y, Cons),
             equal(1st(y), F))))
        then and(q(x, y), t(2nd(x), 2nd(y)))
        else False
```

A.4 Glenstrup Examples - Algorithms

A.4.1 graphcolour1

```
--
-- Source file: algorithms\graphcolour1.scm
--
--- Colour graph G with colours cs so that neighbors have different colours
--- The graph is represented as a list of nodes with adjacency lists
--- Example:
--- '((a . (b c d)) (b . (a c e)) (c . (a b d e f)) (d . (a c f))
---   (e . (b c f)) (f . (c d e)))
--- Colour a node by appending a colour list to the node. The head of
--- the list is the chosen colour, the tail are the yet untried
--- colours. If impossible, return nil.
--- Example of coloured node: '((red blue yellow) . (a . (b c d)))
--- Can we use color with these adjacent nodes and current coloured nodes
--- Return colour of node. If no colour yet, return nil.
--- Colour the first node of rest with colours from ncs, and
--- colour remaining nodes. If impossible, return nil.
graphcolour(g, cs) =
  let
    ns = g
```

```
  in
    reverse(colorrest(cs, cs, Cons(colornode(cs, 1st(ns), Nil), Nil), 2nd(ns)))
colornode(cs, node, colorednodes) =
  if eq(cs, Cons)
    then
      if possible(1st(cs), 2nd(node), colorednodes)
        then Cons(cs, node)
        else colornode(2nd(cs), node, colorednodes)
    else Nil
possible(color, adjs, colorednodes) =
  if eq(adjs, Cons)
    then
      if equal(color, colorof(1st(adjs), colorednodes))
        then False
        else possible(color, 2nd(adjs), colorednodes)
    else True
colorof(node, colorednodes) =
  if eq(colorednodes, Cons)
    then
      if equal(1st(2nd(1st(colorednodes))), node)
        then 1st(1st(1st(colorednodes)))
        else colorof(node, 2nd(colorednodes))
    else Nil
colorrest(cs, ncs, colorednodes, rest) =
  if eq(rest, Cons)
    then
      let
        colorednode = colornode(ncs, 1st(rest), colorednodes)
      in

        if eq(colorednode, Cons)
          then
            let
              colored = colorrest(cs, cs, Cons(colorednode, colorednodes),
                                  2nd(rest))
            in

              if eq(colored, Cons)
                then colored
                else
                  if eq(1st(colorednode), Cons)
                    then colorrestthetrick(cs, cs, 2nd(1st(colorednode)),
                                           colorednodes, rest)
                    else Nil
          else Nil
    else colorednodes
colorrestthetrick(cs1, cs, ncs, colorednodes, rest) =
  if equal(cs1, ncs)
    then colorrest(cs, cs1, colorednodes, rest)
    else colorrestthetrick(2nd(cs1), cs, ncs, colorednodes, rest)
reverse(xs) = revapp(xs, Nil)
revapp(xs, rest) =
  if eq(xs, Cons)
    then revapp(2nd(xs), Cons(1st(xs), rest))
    else rest
```

A.4.2 graphcolour2

```
--
-- Source file: algorithms\graphcolour2.scm
--
--- Colour graph G with colours cs so that neighbors have different
--- colours (slightly tail recursive version)
--- The graph is represented as a list of nodes with adjacency lists
--- Example:
---  '((a . (b c d)) (b . (a c e)) (c . (a b d e f)) (d . (a c f))
---    (e . (b c f)) (f . (c d e)))
```

```
--- Colour a node by appending a colour list to the node. The head of
--- the list is the chosen colour, the tail are the yet untried
--- colours. If impossible, return nil.
--- Example of coloured node: '((red blue yellow) . (a . (b c d)))
--- Can we use color with these adjacent nodes and current coloured nodes
--- Return colour of node. If no colour yet, return nil.
--- Colour the first node of rest with colours from ncs, and
--- colour remaining nodes. If impossible, return nil.
--- Like colornode, only continue with colouring the rest
graphcolour(g, cs) =
  let
    ns = g
  in
    reverse(colorrest(cs, cs, Cons(colornode(cs, 1st(ns), Nil), Nil), 2nd(ns)))
colornode(cs, node, colorednodes) =
  if eq(cs, Cons)
    then
      if possible(1st(cs), 2nd(node), colorednodes)
        then Cons(cs, node)
        else colornode(2nd(cs), node, colorednodes)
    else Nil
possible(color, adjs, colorednodes) =
  if eq(adjs, Cons)
    then
      if equal(color, colorof(1st(adjs), colorednodes))
        then False
        else possible(color, 2nd(adjs), colorednodes)
    else True
colorof(node, colorednodes) =
  if eq(colorednodes, Cons)
    then
      if equal(1st(2nd(1st(colorednodes))), node)
        then 1st(1st(1st(colorednodes)))
        else colorof(node, 2nd(colorednodes))
    else Nil
colorrest(cs, ncs, colorednodes, rest) =
  if eq(rest, Cons)
    then colornoderest(cs, ncs, 1st(rest), colorednodes, rest)
    else colorednodes
colornoderest(cs, ncs, node, colorednodes, rest) =
  if eq(ncs, Cons)
    then
      if possible(1st(ncs), 2nd(node), colorednodes)
        then
          let
            colored = colorrest(cs, cs, Cons(Cons(ncs, node), colorednodes),
                                2nd(rest))
          in

            if eq(colored, Cons)
              then colored
              else
                if eq(ncs, Cons)
                  then colorrestthetrick(cs, cs, 2nd(ncs), colorednodes, rest)
                  else Nil
        else colornoderest(cs, 2nd(ncs), node, colorednodes, rest)
    else Nil
colorrestthetrick(cs1, cs, ncs, colorednodes, rest) =
  if equal(cs1, ncs)
    then colorrest(cs, cs1, colorednodes, rest)
    else colorrestthetrick(2nd(cs1), cs, ncs, colorednodes, rest)
reverse(xs) = revapp(xs, Nil)
revapp(xs, rest) =
  if eq(xs, Cons)
    then revapp(2nd(xs), Cons(1st(xs), rest))
    else rest
```

A.4.3 graphcolour3

```
--
-- Source file: algorithms\graphcolour3.scm
--
--- Colour graph G with colours cs so that neighbors have different
--- colours (slightly tail recursive version)
--- The graph is represented as a list of nodes with adjacency lists
--- Example:
--- '((a . (b c d)) (b . (a c e)) (c . (a b d e f)) (d . (a c f))
---   (e . (b c f)) (f . (c d e)))
--- Colour a node by appending a colour list to the node. The head of
--- the list is the chosen colour, the tail are the yet untried
--- colours. If impossible, return nil.
--- Example of coloured node: '((red blue yellow) . (a . (b c d)))
--- Can we use color with adjacent nodes and current coloured nodes
--- Return colour of node. If no colour yet, return nil.
--- Colour the first node of rest with colours from ncs, and
--- colour remaining nodes. If impossible, return nil.
--- Like colornode, only continue with colouring the rest
graphcolour(g, cs) =
  let
    ns = g
  in
    reverse(colorrest(cs, cs, Cons(colornode(cs, 1st(ns), Nil), Nil), 2nd(ns)))
colornode(cs, node, colorednodes) =
  if eq(cs, Cons)
    then
      if possible(1st(cs), 2nd(node), colorednodes)
        then Cons(cs, node)
        else colornode(2nd(cs), node, colorednodes)
    else Nil
possible(color, adjs, colorednodes) =
  if eq(adjs, Cons)
    then
      if equal(color, colorof(1st(adjs), colorednodes))
        then False
        else possible(color, 2nd(adjs), colorednodes)
    else True
colorof(node, colorednodes) =
  if eq(colorednodes, Cons)
    then
      if equal(1st(2nd(1st(colorednodes))), node)
        then 1st(1st(1st(colorednodes)))
        else colorof(node, 2nd(colorednodes))
    else Nil
colorrest(cs, ncs, colorednodes, rest) =
  if eq(rest, Cons)
    then colornoderest(cs, ncs, 1st(rest), colorednodes, rest)
    else colorednodes
colornoderest(cs, ncs, node, colorednodes, rest) =
  if eq(ncs, Cons)
    then
      if possible(1st(ncs), 2nd(node), colorednodes)
        then
          let
            colored = colorrest(cs, cs, Cons(Cons(ncs, node), colorednodes),
                                2nd(rest))
          in

            if eq(colored, Cons)
              then colored
              else
                if eq(ncs, Cons)
                  then colorrest(cs, 2nd(ncs), colorednodes, rest)
                  else Nil
        else colornoderest(cs, 2nd(ncs), node, colorednodes, rest)
    else Nil
```

```
reverse(xs) = revapp(xs, Nil)
revapp(xs, rest) =
  if eq(xs, Cons)
    then revapp(2nd(xs), Cons(1st(xs), rest))
    else rest
```

A.4.4 match

```
--
-- Source file: algorithms\match.scm
--
-- Simple pattern matcher
match(p, s) = loop(p, s, p, s)
loop(p, s, pp, ss) =
  if equal(p, Nil)
    then True
    else
      if equal(s, Nil)
        then False
        else
          if equal(1st(p), 1st(s))
            then loop(2nd(p), 2nd(s), pp, ss)
            else loop(pp, 2nd(ss), pp, 2nd(ss))
```

A.4.5 reach

```
--
-- Source file: algorithms\reach.scm
--
--- How can node v be reached from node u in a directed graph.
--- Graph example: '((a . b) (a . d) (b . d) (c . a))
goal(u, v, edges) = reach(u, v, edges)
reach(u, v, edges) =
  if member(Cons(u, v), edges)
    then Cons(Cons(u, v), Nil)
    else via(u, v, edges, edges)
via(u, v, rest, edges) =
  if equal(rest, Nil)
    then Nil
    else
      if equal(u, 1st(1st(rest)))
        then
          let
            path = reach(2nd(1st(rest)), v, edges)
          in

            if equal(path, Nil)
              then via(u, v, 2nd(rest), edges)
              else Cons(1st(rest), path)
        else via(u, v, 2nd(rest), edges)
member(x, xs) =
  if equal(xs, Nil)
    then False
    else
      if equal(x, 1st(xs))
        then True
        else member(x, 2nd(xs))
```

A.4.6 rematch

```
--
-- Source file: algorithms\rematch.scm
--
--- Regular expression pattern matcher
---
--- pat ::= . | character | \ character
```

```
---       |  pat* | (pat) | pat ... pat
--- When parsed, this is represented by:
--- pat ::= ('dot) | ('char c) | ('star pat) | ('seq pat ... pat)
--
-- Modification: char -> symbols, string ->symbols
rematch(patstr, str) =
  let
    matchrest = domatch(parsepat(patstr), stringlist(str))
  in

    if eq(matchrest, Cons)
      then Cons(liststring(reverse(1st(matchrest))),
                liststring(2nd(matchrest)))
      else matchrest
parsepat(patstr) = parsep(stringlist(patstr), Nil, Nil)
parsep(patchars, seq, stack) =
  if eq(patchars, Cons)
    then
      if equal(Cdot, 1st(patchars))
        then parsepdot(patchars, seq, stack)
        else
          if equal(Cstar, 1st(patchars))
            then parsepstar(patchars, seq, stack)
            else
              if equal(Clpar, 1st(patchars))
                then parsepopenb(patchars, seq, stack)
                else
                  if equal(Crpar, 1st(patchars))
                    then parsepcloseb(patchars, seq, stack)
                    else
                      if equal(Cslash, 1st(patchars))
                        then parsepchar(2nd(patchars), seq, stack)
                        else parsepchar(patchars, seq, stack)
    else
      if eq(stack, Cons)
        then error(Unmatchedlparinpattern)
        else Cons(Seq, reverse(seq))
parsepdot(patchars, seq, stack) =
  parsep(2nd(patchars), Cons(Cons(Cdot, Nil), seq), stack)
parsepstar(patchars, seq, stack) =
  if eq(seq, Cons)
    then parsep(2nd(patchars), Cons(Cons(Cstar, Cons(1st(seq), Nil)),
                                    2nd(seq)), stack)
    else parsep(2nd(patchars), Cons(Cons(Char, Cons(Cstar, Nil)), Nil), stack)
parsepopenb(patchars, seq, stack) =
  parsep(2nd(patchars), Nil, Cons(seq, stack))
parsepcloseb(patchars, seq, stack) =
  if eq(stack, Cons)
    then parsep(2nd(patchars), Cons(Cons(Seq, reverse(seq)), 1st(stack)),
                2nd(stack))
    else error(Unmatchedrparinpattern)
parsepchar(patchars, seq, stack) =
  if eq(patchars, Cons)
    then parsep(2nd(patchars), Cons(Cons(Char, Cons(1st(patchars), Nil)), seq),
                stack)
    else parsep(patchars, Cons(Cons(Char, Cons(Cslash, Nil)), seq), stack)
domatch(pat, cs) =
  if eq(pat, Cons)
    then
      if equal(1st(pat), Dot)
        then domatchdot(cs)
        else
          if equal(1st(pat), Char)
            then domatchchar(cs, 1st(2nd(pat)))
            else
              if equal(1st(pat), Star)
                then domatchstar(cs, 1st(2nd(pat)), Nil)
                else
```

```
                    if equal(1st(pat), Seq)
                      then domatchseq(cs, Nil, 2nd(pat))
                      else error(Unknownpatterndata)
    else Cons(Nil, cs)
domatchdot(cs) =
  if eq(cs, Cons)
    then Cons(Cons(1st(cs), Nil), 2nd(cs))
    else Nomatch
domatchchar(cs, c) =
  if eq(cs, Cons)
    then
      if equal(1st(cs), c)
        then Cons(Cons(1st(cs), Nil), 2nd(cs))
        else Nomatch
    else Nomatch
domatchstar(cs, pat, init) =
  if eq(cs, Cons)
    then
      let
        first = domatch(pat, cs)
      in

        if eq(first, Cons)
          then domatchstar(2nd(first), pat, append(1st(first), init))
          else Cons(init, cs)
    else Cons(init, cs)
domatchseq(cs, rest, pats) =
  if eq(pats, Cons)
    then
      let
        first = domatch(1st(pats), cs)
      in

        if eq(first, Cons)
          then
            let
              next = domatchseq(append(2nd(first), rest), Nil, 2nd(pats))
            in

              if eq(next, Cons)
                then Cons(append(1st(next), 1st(first)), 2nd(next))
                else
                  if eq(1st(first), Cons)
                    then domatchseq(reverse(2nd(1st(first))), Cons(1st(1st(
                                                first)), append(2nd(first), rest)), pats)
                    else Nomatch
          else Nomatch
    else Cons(Nil, append(cs, rest))
liststring(x) = dummy(x)
stringlist(x) = dummy(x)
reverse(x) = x
append(x, y) = Cons(x, y)
dummy(x) =
  if True
    then x
    else Cons(Nil, stringlist(x))
```

A.4.7 strmatch

```
--
-- Source file: algorithms\strmatch.scm
--
--- Naive pattern string matcher
--
-- Modified to not call scm conversion functions
strmatch(patstr, str) = domatch(patstr, str, Nil)
domatch(patcs, cs, n) =
  if eq(cs, Cons)
```

```
    then
      if prefix(patcs, cs)
        then Cons(n, domatch(patcs, 2nd(cs), add(n, Cons(Nil, Nil))))
        else domatch(patcs, 2nd(cs), add(n, Cons(Nil, Nil)))
    else
      if equal(patcs, cs)
        then Cons(n, Nil)
        else Nil
prefix(precs, cs) =
  if eq(precs, Cons)
    then
      if eq(cs, Cons)
        then and(equal(1st(precs), 1st(cs)), prefix(2nd(precs), 2nd(cs)))
        else False
    else True
```

A.4.8 typeinf

```
--
-- Source file: algorithms\typeinf.scm
--
--- Type inference for the Typed Lambda Calculus
--- e ::= ('var . x)              variable x
---     | ('apply . (e1 . e2))    apply abstraction e1 to expression e2
---     | ('lambda . (x . e1))    make lambda abstraction
--- t ::= ('tyvar . a)
---     | ('arrow . (t1 . t2))
-----------------------------------------
--- (define inittenv 1) - tenv simply holds the next fresh type variables
--  fixed a call to error arity 2 -> 1
typeinf(inittenv, e) =
  let
    atenv = freshtvar(inittenv)
  in
    1st(etype(Nil, 2nd(atenv), e, 1st(atenv)))
freshtvar(tenv) = Cons(Cons(Tvar, tenv), add(tenv, Cons(Nil, Nil)))
vtype(venv, x) =
  if equal(x, 1st(1st(venv)))
    then 2nd(1st(venv))
    else vtype(2nd(venv), x)
tsubst(a, t, t1) =
  if equal(Tvar, 1st(t1))
    then
      if equal(a, 2nd(t1))
        then t
        else t1
    else
      if equal(Arr, 1st(t1))
        then Cons(Arr, Cons(tsubst(a, t, 1st(2nd(t1))),
                            tsubst(a, t, 2nd(2nd(t1)))))
        else error(Tsubstt1)
subst(venv, a, t) =
  if eq(venv, Cons)
    then Cons(Cons(1st(1st(venv)), tsubst(a, t, 2nd(1st(venv)))),
                                   subst(2nd(venv), a, t))
    else Nil
unify(venv, t1, t2) =
  if equal(Tvar, 1st(t1))
    then Cons(subst(venv, 2nd(t1), t2), t2)
    else
      if equal(Arr, 1st(t1))
        then
          if equal(Tvar, 1st(t2))
            then Cons(subst(venv, 2nd(t2), t1), t1)
            else
              if equal(Arr, 1st(t2))
                then
```

```
                    let
                      venv1tx1 = unify(venv, 1st(2nd(t1)), 2nd(2nd(t1)))
                      venv2tx2 = unify(1st(venv1tx1), 1st(2nd(t2)), 2nd(2nd(t2)))
                    in
                      Cons(1st(venv2tx2), Cons(Arr, Cons(2nd(venv1tx1),
                                                         2nd(venv2tx2))))
                  else error(Unifyt2)
          else error(Unifyt1)
etype(venv, tenv, e, t) =
  if equal(Var, 1st(e))
    then
      let
        venv1t1 = unify(venv, vtype(venv, 2nd(e)), t)
      in
        Cons(2nd(venv1t1), Cons(1st(venv1t1), tenv))
    else
      if equal(App, 1st(e))
        then
          let
            atenv1 = freshtvar(tenv)
            t2venv2tenv2 = etype(venv, 2nd(atenv1), 1st(2nd(e)), 1st(atenv1))
            t1venv3tenv3 = etype(1st(2nd(t2venv2tenv2)), 2nd(2nd(t2venv2tenv2))
                                 , 2nd(2nd(e)), Cons(Arr, Cons(1st(t2venv2tenv2), t)))
            t1 = 1st(t1venv3tenv3)
          in
            Cons(2nd(2nd(t1)), 2nd(t1venv3tenv3))
        else
          if equal(Lam, 1st(e))
            then
              let
                atenv1 = freshtvar(tenv)
                t1venv2tenv2 = etype(Cons(Cons(1st(2nd(e)), 1st(atenv1)), venv)
                                     , 2nd(atenv1), 2nd(2nd(e)), 1st(atenv1))
                venv3t3 = unify(1st(2nd(t1venv2tenv2)), Cons(Arr, Cons(vtype(
                  2nd(2nd(t1venv2tenv2)), 1st(2nd(e))), 1st(t1venv2tenv2))), t)
              in
                Cons(2nd(venv3t3), Cons(2nd(1st(venv3t3)),
                                        2nd(2nd(t1venv2tenv2))))
            else error(Errorinlambdaexpression)
```

A.5 Glenstrup Examples - Interpreters

A.5.1 intdynscope

```
--
-- Source file: interpreters\intdynscope.scm
--
-- Small 1st order interpreter with dynamic scoping
run(p, input) =
  let
    f0 = 1st(1st(p))
    ef = lookbody(f0, p)
    nf = lookname(f0, p)
  in
    eeval(ef, Cons(nf, Nil), Cons(input, Nil), p)
eeval(e, ns, vs, p) =
  if equal(1st(e), Cons(Nil, Nil))
    then 2nd(e)
    else
      if equal(1st(e), Cons(Nil, Cons(Nil, Nil)))
        then lookvar(2nd(e), ns, vs)
        else
          if equal(1st(e), Cons(Nil, Cons(Nil, Cons(Nil, Nil))))
            then
              let
                v1 = eeval(1st(2nd(2nd(e))), ns, vs, p)
                v2 = eeval(1st(2nd(2nd(2nd(e)))), ns, vs, p)
```

```
                in
                  apply(1st(2nd(e)), v1, v2)
              else
                if equal(1st(e), Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Nil)))))
                  then
                    if equal(eeval(1st(2nd(e)), ns, vs, p), T)
                      then eeval(1st(2nd(2nd(e))), ns, vs, p)
                      else eeval(1st(2nd(2nd(2nd(e)))), ns, vs, p)
                  else
                    if equal(1st(e), Cons(Nil, Cons(Nil, Cons(Nil,
                                                 Cons(Nil, Cons(Nil, Nil))))))
                      then
                        if equal(eeval(1st(2nd(e)), ns, vs, p),
                                 eeval(1st(2nd(2nd(e))), ns, vs, p))
                          then T
                          else F
                        else
                          let
                            ef = lookbody(1st(2nd(e)), p)
                            nf = lookname(1st(2nd(e)), p)
                            v = eeval(1st(2nd(2nd(e))), ns, vs, p)
                          in
                            eeval(ef, Cons(nf, ns), Cons(v, vs), p)
lookvar(x, ns, vs) =
  if equal(x, 1st(ns))
    then 1st(vs)
    else lookvar(x, 2nd(ns), 2nd(vs))
lookbody(f, p) =
  if equal(1st(1st(p)), f)
    then 1st(2nd(2nd(1st(p))))
    else lookbody(f, 2nd(p))
lookname(f, p) =
  if equal(1st(1st(p)), f)
    then 1st(2nd(1st(p)))
    else lookname(f, 2nd(p))
apply(op, v1, v2) =
  if equal(op, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Nil))))))
    then
      if equal(v1, v2)
        then T
        else F
    else Cons(v1, v2)
```

A.5.2 intloop

```
--
-- Source file: interpreters\intloop.scm
--
-- Small 1st order interpreter for LOOP programs
run(p, l, input) =
  let
    f0 = 1st(1st(p))
    ef = lookbody(f0, p)
    nf = lookname(f0, p)
  in
    eeval(ef, Cons(nf, Nil), Cons(input, Nil), l, p)
eeval(e, ns, vs, l, p) =
  if equal(1st(e), Cons(Nil, Nil))
    then 2nd(e)
    else
      if equal(1st(e), Cons(Nil, Cons(Nil, Nil)))
        then lookvar(2nd(e), ns, vs)
        else
          if equal(1st(e), Cons(Nil, Cons(Nil, Cons(Nil, Nil))))
            then
              let
                v1 = eeval(1st(2nd(2nd(e))), ns, vs, l, p)
```

```
                  v2 = eeval(1st(2nd(2nd(2nd(e)))), ns, vs, l, p)
                in
                  apply(1st(2nd(e)), v1, v2)
              else
                if equal(1st(e), Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Nil)))))
                  then
                    if equal(eeval(1st(2nd(e)), ns, vs, l, p), T)
                      then eeval(1st(2nd(2nd(e))), ns, vs, l, p)
                      else eeval(1st(2nd(2nd(2nd(e)))), ns, vs, l, p)
                  else
                    if equal(1st(e), Cons(Nil, Cons(Nil, Cons(Nil,
                                                    Cons(Nil, Cons(Nil, Nil))))))
                      then
                        if equal(eeval(1st(2nd(e)), ns, vs, l, p),
                                 eeval(1st(2nd(2nd(e))), ns, vs, l, p))
                          then T
                          else F
                      else
                        let
                          ef = lookbody(1st(2nd(e)), p)
                          nf = lookname(1st(2nd(e)), p)
                          v = eeval(1st(2nd(2nd(e))), ns, vs, l, p)
                        in

                          if equal(l, Nil)
                            then Nil
                            else eeval(ef, Cons(nf, Nil), Cons(v, Nil),
                                       2nd(l), p)
lookvar(x, ns, vs) =
  if equal(x, 1st(ns))
    then 1st(vs)
    else lookvar(x, 2nd(ns), 2nd(vs))
lookbody(f, p) =
  if equal(1st(1st(p)), f)
    then 1st(2nd(2nd(1st(p))))
    else lookbody(f, 2nd(p))
lookname(f, p) =
  if equal(1st(1st(p)), f)
    then 1st(2nd(1st(p)))
    else lookname(f, 2nd(p))
apply(op, v1, v2) =
  if equal(op, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Nil))))))
    then
      if equal(v1, v2)
        then T
        else F
    else Cons(v1, v2)
```

A.5.3 intwhile

```
--
-- Source file: interpreters\intwhile.scm
--
-- Small 1st order interpreter with static scoping
run(p, input) =
  let
    f0 = 1st(1st(p))
    ef = lookbody(f0, p)
    nf = lookname(f0, p)
  in
    eeval(ef, Cons(nf, Nil), Cons(input, Nil), p)
eeval(e, ns, vs, p) =
  if equal(1st(e), number1(Nil))
    then 2nd(e)
    else
      if equal(1st(e), number2(Nil))
        then lookvar(2nd(e), ns, vs)
        else
```

```
            if equal(1st(e), number3(Nil))
              then
                let
                  v1 = eeval(1st(2nd(2nd(e))), ns, vs, p)
                  v2 = eeval(1st(2nd(2nd(2nd(e)))), ns, vs, p)
                in
                  apply(1st(2nd(e)), v1, v2)
              else
                if equal(1st(e), number4(Nil))
                  then
                    if equal(eeval(1st(2nd(e)), ns, vs, p), T)
                      then eeval(1st(2nd(2nd(e))), ns, vs, p)
                      else eeval(1st(2nd(2nd(2nd(e)))), ns, vs, p)
                  else
                    if equal(1st(e), number5(Nil))
                      then
                        if equal(eeval(1st(2nd(e)), ns, vs, p),
                                 eeval(1st(2nd(2nd(e))), ns, vs, p))
                          then T
                          else F
                      else
                        let
                          ef = lookbody(1st(2nd(e)), p)
                          nf = lookname(1st(2nd(e)), p)
                          v = eeval(1st(2nd(2nd(e))), ns, vs, p)
                        in
                          eeval(ef, Cons(nf, Nil), Cons(v, Nil), p)
lookvar(x, ns, vs) =
  if equal(x, 1st(ns))
    then 1st(vs)
    else lookvar(x, 2nd(ns), 2nd(vs))
lookbody(f, p) =
  if equal(1st(1st(p)), f)
    then 1st(2nd(2nd(1st(p))))
    else lookbody(f, 2nd(p))
lookname(f, p) =
  if equal(1st(1st(p)), f)
    then 1st(2nd(1st(p)))
    else lookname(f, 2nd(p))
apply(op, v1, v2) =
  if equal(op, number5(Nil))
    then
      if equal(v1, v2)
        then T
        else F
    else Cons(v1, v2)
number1(n) = Cons(Nil, Nil)
number2(n) = Cons(Nil, Cons(Nil, Nil))
number3(n) = Cons(Nil, Cons(Nil, Cons(Nil, Nil)))
number4(n) = Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Nil))))
number5(n) = Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Nil)))))
```

A.5.4 int

```
--
-- Source file: interpreters\int.scm
--
run(p, input) =
  let
    f0 = 1st(1st(p))
    ef = lookbody(f0, p)
    nf = lookname(f0, p)
  in
    eeval(ef, Cons(nf, Nil), Cons(input, Nil), p)
eeval(e, ns, vs, p) =
  if equal(1st(e), Cst)
    then 2nd(e)
    else
```

```
      if equal(1st(e), Var)
        then lookvar(2nd(e), ns, vs)
        else
          if equal(1st(e), Bsf)
            then
              let
                v1 = eeval(1st(2nd(2nd(e))), ns, vs, p)
                v2 = eeval(1st(2nd(2nd(2nd(e)))), ns, vs, p)
              in
                apply(1st(2nd(e)), v1, v2)
            else
              if equal(1st(e), If)
                then
                  if equal(eeval(1st(2nd(e)), ns, vs, p), T)
                    then eeval(1st(2nd(2nd(e))), ns, vs, p)
                    else eeval(1st(2nd(2nd(2nd(e)))), ns, vs, p)
                else
                  if equal(1st(e), Eq)
                    then
                      if equal(eeval(1st(2nd(e)), ns, vs, p),
                               eeval(1st(2nd(2nd(e))), ns, vs, p))
                        then T
                        else F
                    else
                      let
                        ef = lookbody(1st(2nd(e)), p)
                        nf = lookname(1st(2nd(e)), p)
                        v = eeval(1st(2nd(2nd(e))), ns, vs, p)
                      in
                        eeval(ef, Cons(nf, Nil), Cons(v, Nil), p)
lookvar(x, ns, vs) =
  if equal(x, 1st(ns))
    then 1st(vs)
    else lookvar(x, 2nd(ns), 2nd(vs))
lookbody(f, p) =
  if equal(1st(1st(p)), f)
    then 1st(2nd(2nd(1st(p))))
    else lookbody(f, 2nd(p))
lookname(f, p) =
  if equal(1st(1st(p)), f)
    then 1st(2nd(1st(p)))
    else lookname(f, 2nd(p))
apply(op, v1, v2) =
  if equal(op, Eqop)
    then
      if equal(v1, v2)
        then T
        else F
    else Cons(v1, v2)
```

A.5.5 lambdaint

```
--
-- Source file: interpreters\lambdaint.scm
--
-- Reducer for the lambda calculus
--   Representation:
--     R [[n]]     = (1 0 ... 0)  n zeros
--     R [[\n.e]] = (2 R [[n]] R [[e]])
--     R [[e e']] = (3 R [[e]] R [[e']])
lambdaint(e) = red(e)
red(e) =
  if isvar(e)
    then e
    else
      if islam(e)
        then e
        else
```

```
            let
              f = red(appe1(e))
              a = red(appe2(e))
            in

              if islam(f)
                then red(subst(lamvar(f), a, lambody(f)))
                else mkapp(f, a)
subst(x, a, e) =
  if isvar(e)
    then
      if equal(x, e)
        then a
        else e
    else
      if islam(e)
        then
          if equal(x, lamvar(e))
            then e
            else mklam(lamvar(e), subst(x, a, lambody(e)))
        else mkapp(subst(x, a, appe1(e)), subst(x, a, appe2(e)))
isvar(e) = equal(1st(e), Cons(Nil, Nil))
islam(e) = equal(1st(e), Cons(Nil, Cons(Nil, Nil)))
mklam(n, e) = Cons(Cons(Nil, Cons(Nil, Nil)), Cons(n, Cons(e, Nil)))
lamvar(e) = 1st(2nd(e))
lambody(e) = 1st(2nd(2nd(e)))
mkapp(e1, e2) =
  Cons(Cons(Nil, Cons(Nil, Cons(Nil, Nil))), Cons(e1, Cons(e2, Nil)))
appe1(e) = 1st(2nd(e))
appe2(e) = 1st(2nd(2nd(e)))
```

A.5.6 parsexp

```
--
-- Source file: interpreters\parsexp.scm
--
--- Parse a list of atoms as an expression. Return remaining list.
--- e.g. '(5 * ( 3 + 2 * 4 ))
--
-- Modified to use symbols instead of strings
parsexp(xs) = expr(xs)
expr(xs) =
  let
    rs1 = term(xs)
  in

    if equal(Nil, rs1)
      then Nil
      else
        if member(1st(rs1), Cons(Plus, Cons(Minus, Nil)))
          then
            let
              rs2 = expr(2nd(rs1))
            in

              if equal(Nil, rs2)
                then rs1
                else rs2
          else rs1
term(xs) =
  let
    rs1 = factor(xs)
  in

    if equal(rs1, Nil)
      then Nil
      else
```

```
35          if member(1st(rs1), Cons(Mul, Cons(Div, Nil)))
36            then
37              let
38                rs2 = term(2nd(rs1))
39              in
40
41                if equal(Nil, rs2)
42                  then rs1
43                  else rs2
44            else rs1
45  factor(xs) =
46    if equal(Lpar, 1st(xs))
47      then
48        let
49          rs1 = expr(xs)
50        in
51
52          if and(not(equal(rs1, Nil)), equal(Rpar, 1st(rs1)))
53            then 2nd(rs1)
54            else atom(xs)
55      else atom(xs)
56  member(x, xs) =
57    if eq(xs, Cons)
58      then
59        if equal(x, 1st(xs))
60          then True
61          else member(x, 2nd(xs))
62      else False
63  atom(xs) =
64    if eq(xs, Cons)
65      then 2nd(xs)
66      else Nil
```

A.5.7 turing

```
1  --
2  -- Source file: interpreters\turing.scm
3  --
4  --- Turing machine interpreter
5  --- instrs ::= '(instr . instrs)
6  ---          |  '()
7  --- instr ::= '(Halt)          - Stop interpretation
8  ---         |  '(Write . x)    - Write x onto the tape at current pos
9  ---         |  '(Left)         - Move pos left, extend tape if needed
10 ---         |  '(Right)        - Move pos right, extend tape if needed
11 ---         |  '(Goto . i)     - Continue at instruction i
12 ---         |  '(IfGoto x . i) - If current pos contains x, goto i
13 run(prog, tapeinput) = turing(prog, Nil, tapeinput, prog)
14 turing(instrs, revltape, rtape, prog) =
15   if eq(instrs, Cons)
16     then
17       if equal(Halt, 1st(1st(instrs)))
18         then rtape
19         else
20           if equal(Write, 1st(1st(instrs)))
21             then turing(2nd(instrs), revltape, Cons(2nd(1st(instrs)),
22                         2nd(rtape)), prog)
23             else
24               if equal(Left, 1st(1st(instrs)))
25                 then
26                   if eq(revltape, Cons)
27                     then turing(2nd(instrs), 2nd(revltape), Cons(1st(revltape),
28                                 rtape), prog)
29                     else turing(2nd(instrs), Nil, Cons(Blank, rtape), prog)
30                 else
31                   if equal(Right, 1st(1st(instrs)))
32                     then
```

```
                        if eq(rtape, Cons)
                          then turing(2nd(instrs), Cons(1st(rtape), revltape),
                                      2nd(rtape), prog)
                          else turing(2nd(instrs), Cons(Blank, revltape),
                                      Nil, prog)
                      else
                        if equal(Goto, 1st(1st(instrs)))
                          then turing(lookup(2nd(1st(instrs)), prog), revltape,
                                      rtape, prog)
                          else
                            if equal(Ifgoto, 1st(1st(instrs)))
                              then
                                if equal(1st(rtape), 1st(2nd(1st(instrs))))
                                  then turing(lookup(2nd(2nd(1st(instrs))), prog)
                                              , revltape, rtape, prog)
                                  else turing(2nd(instrs), revltape, rtape, prog)
                              else rtape
    else rtape
lookup(i, instrs) =
  if eq(i, Cons(Nil, Nil))
    then instrs
    else lookup(sub(i, Cons(Nil, Nil)), 2nd(instrs))
```

A.6 Glenstrup Examples - Simple

A.6.1 ack

```
--
-- Source file: simple\ack.scm
--
--- Ackermann's function, numbers represented by list length
goal(m, n) = ack(m, n)
ack(m, n) =
  if equal(Nil, m)
    then Cons(Cons(Nil, Nil), n)
    else
      if equal(Nil, n)
        then ack(2nd(m), Cons(Nil, Nil))
        else ack(2nd(m), ack(m, 2nd(n)))
```

A.6.2 binom

```
--
-- Source file: simple\binom.scm
--
--- Binomial function, numbers represented by list length
goal(n, k) = binom(n, k)
binom(n, k) =
  if equal(Nil, n)
    then Cons(Nil, Nil)
    else
      if equal(Nil, k)
        then Cons(Nil, Nil)
        else add(binom(2nd(n), 2nd(k)), binom(2nd(n), k))
```

A.6.3 gcd1

```
--
-- Source file: simple\gcd1.scm
--
-- Greatest common divisor, numbers represented by list length
goal(x, y) = gcd(x, y)
gcd(x, y) =
  if or(equal(x, Nil), equal(y, Nil))
    then Error
```

```
      else
        if equal(x, y)
          then x
          else
            if gt0(x, y)
              then gcd(monus(x, y), y)
              else gcd(x, monus(y, x))
gt0(x, y) =
  if equal(x, Nil)
    then False
    else
      if equal(y, Nil)
        then True
        else gt0(2nd(x), 2nd(y))
monus(x, y) =
  if equal(lgth(y), Cons(Nil, Nil))
    then 2nd(x)
    else monus(2nd(x), 2nd(y))
lgth(x) =
  if equal(x, Nil)
    then Nil
    else add(Cons(Nil, Nil), lgth(2nd(x)))
```

A.6.4 gcd2

```
--
-- Source file: simple\gcd2.scm
--
-- Greatest common divisor, numbers represented by list length
goal(x, y) = gcd(x, y)
gcd(x, y) =
  if or(equal(x, Nil), equal(y, Nil))
    then Error
    else
      if equal(x, y)
        then x
        else
          if gt0(x, y)
            then gcd(y, monus(x, y))
            else gcd(monus(y, x), x)
gt0(x, y) =
  if equal(x, Nil)
    then False
    else
      if equal(y, Nil)
        then True
        else gt0(2nd(x), 2nd(y))
monus(x, y) =
  if equal(lgth(y), Cons(Nil, Nil))
    then 2nd(x)
    else monus(2nd(x), 2nd(y))
lgth(x) =
  if equal(x, Nil)
    then Nil
    else add(Cons(Nil, Nil), lgth(2nd(x)))
```

A.6.5 power

```
--
-- Source file: simple\power.scm
--
-- Power function: x to the nth power
-- (numbers represented by list length)
goal(x, n) = power(x, n)
power(x, n) =
  if equal(n, Nil)
```

```
9       then Cons(Nil, Nil)
10      else mult(x, power(x, 2nd(n)))
11  mult(x, y) =
12    if equal(y, Nil)
13      then Nil
14      else add0(x, mult(x, 2nd(y)))
15  add0(x, y) =
16    if equal(y, Nil)
17      then x
18      else Cons(Cons(Nil, Nil), add0(x, 2nd(y)))
```

A.7 Glenstrup Examples - Sorting

A.7.1 mergesort

```
1   --
2   -- Source file: sorting\mergesort.scm
3   --
4   -- Mergesort
5   goal(xs) = mergesort(xs)
6   mergesort(xs) =
7     if eq(xs, Cons)
8       then
9         if eq(2nd(xs), Cons)
10          then splitmerge(xs, Nil, Nil)
11          else xs
12      else xs
13  splitmerge(xs, xs1, xs2) =
14    if eq(xs, Cons)
15      then splitmerge(2nd(xs), Cons(1st(xs), xs2), xs1)
16      else merge(mergesort(xs1), mergesort(xs2))
17  merge(xs1, xs2) =
18    if eq(xs1, Cons)
19      then
20        if eq(xs2, Cons)
21          then
22            if lte(1st(xs1), 1st(xs2))
23              then Cons(1st(xs1), merge(2nd(xs1), xs2))
24              else Cons(1st(xs2), merge(xs1, 2nd(xs2)))
25          else xs1
26      else xs2
```

A.7.2 minsort

```
1   --
2   -- Source file: sorting\minsort.scm
3   --
4   --- Minimum sort: remove minimum and cons it onto the rest, sorted.
5   goal(xs) = minsort(xs)
6   minsort(xs) =
7     if eq(xs, Cons)
8       then appmin(1st(xs), 2nd(xs), xs)
9       else Nil
10  appmin(min, rest, xs) =
11    if eq(rest, Cons)
12      then
13        if lt(1st(rest), min)
14          then appmin(1st(rest), 2nd(rest), xs)
15          else appmin(min, 2nd(rest), xs)
16      else Cons(min, minsort(remove(min, xs)))
17  remove(x, xs) =
18    if eq(xs, Cons)
19      then
20        if equal(x, 1st(xs))
21          then 2nd(xs)
22          else Cons(1st(xs), remove(x, 2nd(xs)))
23      else Nil
```

A.7.3 quicksort

```
--
-- Source file: sorting\quicksort.scm
--
-- Quicksort
goal(xs) = quicksort(xs)
quicksort(xs) =
  if eq(xs, Cons)
    then
      if eq(2nd(xs), Cons)
        then part(1st(xs), xs, Cons(1st(xs), Nil), Nil)
        else xs
    else xs
part(x, xs, xs1, xs2) =
  if eq(xs, Cons)
    then
      if gt(x, 1st(xs))
        then part(x, 2nd(xs), Cons(1st(xs), xs1), xs2)
        else
          if lt(x, 1st(xs))
            then part(x, 2nd(xs), xs1, Cons(1st(xs), xs2))
            else part(x, 2nd(xs), xs1, xs2)
    else app(quicksort(xs1), quicksort(xs2))
app(xs, ys) =
  if eq(xs, Cons)
    then Cons(1st(xs), app(2nd(xs), ys))
    else ys
```

A.8 Other Examples

A.8.1 assrewriteSize

```
--
-- Source file: basic\assrewrite.scm
-- Modified to call with multiple arguments
--
--- Rewrite expression with associative operator `op'
---     a -> a1     b -> b1     c -> c1
--- ------------------------------------
--- '(op (op a b) c) -> '(op a1 (op b1 c1))
---a != 'op  a -> a1  b -> b1              a != '(op ...)
--- -------------------------              -------------
---  '(op a b) -> '(op a1 b1)                 a -> a
assrewrite(exp) = rewrite(exp)
rewrite(exp) =
  if and(eq(exp, Cons), eq(Op, 1st(exp)))
    then rw(1st(2nd(exp)), 1st(2nd(2nd(exp))))
    else exp
rw(opab, c) =
         if and(eq(opab, Cons), eq(Op, 1st(opab)))
           then
             let
               a1 = rewrite(1st(2nd(opab)))
               b1 = rewrite(1st(2nd(2nd(opab))))
               c1 = rewrite(c)
             in
               rw(a1, Cons(Op, Cons(b1, Cons(c1, Nil))))
           else Cons(Op, Cons(rewrite(opab), Cons(rewrite(c), Nil)))
```

A.8.2 deadcodeSize

```
--
-- Dead code example
--
f(x) =
  if eq(x, Z)
```

```
      then x
      else 1st(x)
g(x) = g(x)
```

A.8.3 fghSize

```
-- example from Andreas Abel 3.12:
-- Modified to size-change terminate.
f(x1,x2) = if eq(x1, Z)
              then x1
              else if eq(x2, Z)
                     then x2
                     else h(g(1st(x1), x2), f(S(x1), 1st(x2)))

g(x1,x2) = if eq(x1, Z)
              then x1
              else if eq(x2, Z)
                     then x2
                     else h(f(x1,x2), g(1st(x1), S(x2)))

h(x1,x2) = if eq(x1, Z)
              then if eq(x2, Z)
                     then x2
                     else h(x1, 1st(x2))
              else h(1st(x1), x2)
```

A.8.4 graphcolour2Size

```
--
-- Source file: algorithms\graphcolour2.scm
--
--- Colour graph G with colours cs so that neighbors have different
--- colours (slightly tail recursive version)
--- The graph is represented as a list of nodes with adjacency lists
--- Example:
--- '((a . (b c d)) (b . (a c e)) (c . (a b d e f)) (d . (a c f))
---   (e . (b c f)) (f . (c d e)))
--- Colour a node by appending a colour list to the node. The head of
--- the list is the chosen colour, the tail are the yet untried
--- colours. If impossible, return nil.
--- Example of coloured node: '((red blue yellow) . (a . (b c d)))
--- Can we use color with these adjacent nodes and current coloured nodes
--- Return colour of node. If no colour yet, return nil.
--- Colour the first node of rest with colours from ncs, and
--- colour remaining nodes. If impossible, return nil.
--- Like colornode, only continue with colouring the rest
--
-- Modified to call with ncs instead of cs1 in colorrestthetrick.
--
graphcolour(g, cs) =
  let
    ns = g
  in
    reverse(colorrest(cs, cs, Cons(colornode(cs, 1st(ns), Nil), Nil), 2nd(ns)))
colornode(cs, node, colorednodes) =
  if eq(cs, Cons)
    then
      if possible(1st(cs), 2nd(node), colorednodes)
        then Cons(cs, node)
        else colornode(2nd(cs), node, colorednodes)
    else Nil
possible(color, adjs, colorednodes) =
  if eq(adjs, Cons)
    then
      if equal(color, colorof(1st(adjs), colorednodes))
        then False
```

```
39          else possible(color, 2nd(adjs), colorednodes)
40      else True
41  colorof(node, colorednodes) =
42    if eq(colorednodes, Cons)
43      then
44        if equal(1st(2nd(1st(colorednodes))), node)
45          then 1st(1st(1st(colorednodes)))
46          else colorof(node, 2nd(colorednodes))
47      else Nil
48  colorrest(cs, ncs, colorednodes, rest) =
49    if eq(rest, Cons)
50      then colornoderest(cs, ncs, 1st(rest), colorednodes, rest)
51      else colorednodes
52  colornoderest(cs, ncs, node, colorednodes, rest) =
53    if eq(ncs, Cons)
54      then
55        if possible(1st(ncs), 2nd(node), colorednodes)
56          then
57            let
58              colored = colorrest(cs, cs, Cons(Cons(ncs, node), colorednodes),
59                                  2nd(rest))
60            in
61              if eq(colored, Cons)
62                then colored
63                else
64                  if eq(ncs, Cons)
65                    then colorrestthetrick(cs, cs, 2nd(ncs), colorednodes, rest)
66                    else Nil
67          else colornoderest(cs, 2nd(ncs), node, colorednodes, rest)
68      else Nil
69  colorrestthetrick(cs1, cs, ncs, colorednodes, rest) =
70    if equal(cs1, ncs)
71      then colorrest(cs, ncs, colorednodes, rest)
72      else colorrestthetrick(2nd(cs1), cs, ncs, colorednodes, rest)
73  reverse(xs) = revapp(xs, Nil)
74  revapp(xs, rest) =
75    if eq(xs, Cons)
76      then revapp(2nd(xs), Cons(1st(xs), rest))
77      else rest
```

A.8.5 quicksortSize

```
1  --
2  -- Source file: sorting\quicksort.scm
3  --
4  -- Quicksort
5  -- Modified from Glenstrup, note that duplets are thrown away
6  -- as in the original implementation.
7  goal(xs) = quicksort(xs)
8  quicksort(xs) =
9    if eq(xs, Cons)
10     then
11       if eq(2nd(xs), Cons)
12         then part(1st(xs), 2nd(xs))
13         else xs
14     else xs
15 part(x, xs) =
16     app(quicksort(partLt(x, xs)), Cons(x, quicksort(partGt(x, xs))))
17 partLt(x, xs) =
18   if eq(xs, Cons)
19     then
20       if lt(1st(xs), x)
21         then Cons(1st(xs), partLt(x, 2nd(xs)))
22         else partLt(x, 2nd(xs))
23     else Nil
24 partGt(x, xs) =
25   if eq(xs, Cons)
```

```
26      then
27        if gt(1st(xs), x)
28          then Cons(1st(xs), partGt(x, 2nd(xs)))
29          else partGt(x, 2nd(xs))
30      else Nil
31  app(xs, ys) =
32    if eq(xs, Cons)
33      then Cons(1st(xs), app(2nd(xs), ys))
34      else ys
```

A.8.6 thetrickSize

```
1   --
2   -- Source file: basic\thetrick.scm
3   --
4   -- The trick: pulling out the conditional into the context
5   -- Modified: conditional pulled out of the context, 42 -> 4
6   goal(x, y) = Cons(f(x, y), Cons(g(x, y), Nil))
7   f(x, y) =
8     if eq(y, Nil)
9       then number4(Nil)
10      else if lt0(x, Cons(Nil, Nil))
11             then f(x, 2nd(y))
12             else f(2nd(x), Cons(Cons(Nil, Nil), y))
13  g(x, y) =
14    if eq(y, Nil)
15      then number4(Nil)
16      else
17        if lt0(x, Cons(Nil, Nil))
18          then g(x, 2nd(y))
19          else g(2nd(x), Cons(Cons(Nil, Nil), y))
20  lt0(x, y) =
21    if eq(y, Nil)
22      then False
23      else
24        if eq(x, Nil)
25          then True
26          else lt0(2nd(x), 2nd(y))
27  number4(n) = Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Nil))))
```

B Results

In the following sections only the output from the terminating examples for which the analysis failed to show size-change termination are listed. This done for the sake of brewity, the output is meant as a reference for the more interesting examples. The results of the other examples are listed in the tables in section 6.

B.1 Results of *SCT* analysis

B.1.1 Glenstrup test suite

```
1  ==========================================
2  Program graphcolour1...
3
4  Number of size-change graphs: 15
5  Number of compositions      : 69
6
7  Possible termination problems:
8
9  [" colorrest-colorrest:[(1,>,2),(1,>=,1),(3,>=,3),(4,>=,4)]*[11,13,12]*"," colo
10 rrestthetrick-colorrestthetrick:[(2,>,1),(2,>=,2),(4,>=,4),(5,>=,5)]*[12,11,13]
11 *"," colorrest-colorrest:[(1,>=,1),(1,>=,2),(3,>=,3),(4,>=,4)]*[11,12]*"," colo
12 rrestthetrick-colorrestthetrick:[(2,>=,1),(2,>=,2),(4,>=,4),(5,>=,5)]*[12,11]*"
13 ]
14
15 ==========================================
16 Program graphcolour2...
17
18 Number of size-change graphs: 17
19 Number of compositions      : 96
20
21 Possible termination problems:
22
23 [" colorrestthetrick-colorrestthetrick:[(2,>,1),(2,>,3),(2,>=,2),(4,>=,4),(5,>=
24 ,5)]*[14,9,12,14,9,12,15]*"," colorrestthetrick-colorrestthetrick:[(2,>,3),(2,>
25 =,1),(2,>=,2),(4,>=,4),(5,>=,5)]*[14,9,12,14,9,12]*"," colorrest-colorrest:[(1,
26 >,2),(1,>=,1),(3,>=,3),(4,>=,4)]*[9,12,15,14]*"," colornoderest-colornoderest:[
27 (1,>,2),(1,>=,1),(4,>=,4),(5,>,3),(5,>=,5)]*[12,14,9,13]*"," colorrest-colorres
28 t:[(1,>=,1),(1,>=,2),(3,>=,3),(4,>=,4)]*[9,12,14]*"," colornoderest-colornodere
29 st:[(1,>=,1),(1,>=,2),(4,>=,4),(5,>,3),(5,>=,5)]*[12,14,9]*"]
30
31 ==========================================
32 Program reach...
33
34 Number of size-change graphs: 7
35 Number of compositions      : 25
36
37 Possible termination problems:
38
39 [" via-via:[(2,>=,2),(4,>,1),(4,>,3),(4,>=,4)]*[4,3,4,3,5]*"," via-via:[(2,>=,2
40 ),(4,>,1),(4,>=,3),(4,>=,4)]*[4,3,4,3]*"," reach-reach:[(2,>=,2),(3,>,1),(3,>=,
41 3)]*[3,4]*"]
42
43 ==========================================
44 Program rematch...
45
46 Number of size-change graphs: 41
47 Number of compositions      : 463
48
49 Possible termination problems:
50
51 [" parsep-parsep:[(1,>=,1),(3,>=,3)]*[14,23]*"," parsepchar-parsepchar:[(1,>=,1
52 ),(3,>=,3)]*[23,14]*"," stringlist-stringlist:[(1,>=,1)]*[40,41]*"," dummy-dumm
```

```
y:[(1,>=,1)]*[41,40]*"," domatchstar-domatchstar:[(2,>=,2)]*[29]*"," domatchseq
-domatchseq:[(3,>=,3)]*[35]*"]

============================================
Program assrewrite...

Number of size-change graphs: 7
Number of compositions      : 5

Possible termination problems:

[" rewrite-rewrite:[]*[5]*"]

============================================
Program permute...

Number of size-change graphs: 8
Number of compositions      : 33

Possible termination problems:

[" permute-permute:[]*[2,4]*"," select-select:[]*[4,2]*"]

============================================
Program shuffle...

Number of size-change graphs: 6
Number of compositions      : 12

Possible termination problems:

[" shuffle-shuffle:[]*[2]*"]

============================================
Program thetrick...

Number of size-change graphs: 11
Number of compositions      : 24

Possible termination problems:

[" f-f:[(1,>=,1)]*[4]*"]

============================================
Program mergesort...

Number of size-change graphs: 8
Number of compositions      : 30

Possible termination problems:

[" mergesort-mergesort:[]*[2,3,5]*"," splitmerge-splitmerge:[]*[3,5,2]*"," spli
tmerge-splitmerge:[(3,>,1),(3,>=,3)]*[6,2,3]*"," mergesort-mergesort:[(1,>=,1)]
*[2,5]*"," splitmerge-splitmerge:[(2,>=,1),(2,>=,2),(2,>=,3)]*[5,2]*"," splitme
rge-splitmerge:[(3,>=,1),(3,>=,2),(3,>=,3)]*[6,2]*"]

============================================
Program minsort...

Number of size-change graphs: 7
Number of compositions      : 23

Possible termination problems:

[" minsort-minsort:[]*[2,5]*"," appmin-appmin:[]*[5,2]*"]

============================================
Program quicksort...
```

```
121
122 Number of size-change graphs: 9
123 Number of compositions       : 30
124
125 Possible termination problems:
126
127 [" part-part:[]*[3,7,2]*"," part-part:[(4,>,1),(4,>,2),(4,>=,4)]*[8,2,3]*"," qu
128 icksort-quicksort:[]*[2,7]*"," quicksort-quicksort:[(1,>=,1)]*[2,8]*"," part-pa
129 rt:[(4,>,1),(4,>=,2),(4,>=,4)]*[8,2]*"]
```

B.1.2 Other examples

```
1  ==========================================
2  Program assrewriteSize...
3
4  Number of size-change graphs: 8
5  Number of compositions       : 17
6
7  Possible termination problems:
8
9  [" rewrite-rewrite:[]*[2,6,3]*"," rw-rw:[]*[6]*"]
10
11 ==========================================
12 Program deadcodeSize...
13
14 Number of size-change graphs: 1
15 Number of compositions       : 1
16
17 Possible termination problems:
18
19 [" g-g:[(1,>=,1)]*[1]*"]
20
21 ==========================================
22 Program fghSize...
23
24 Number of size-change graphs: 8
25 Number of compositions       : 22
26
27 Possible termination problems:
28
29 [" f-f:[]*[2,6,5,3]*"," g-g:[]*[5,3,2,6]*"]
30
31 ==========================================
32 Program graphcolour2Size...
33
34 Number of size-change graphs: 17
35 Number of compositions       : 79
36
37 Program terminates.
38
39
40 ==========================================
41 Program quicksortSize...
42
43 Number of size-change graphs: 12
44 Number of compositions       : 29
45
46 Possible termination problems:
47
48 [" quicksort-quicksort:[]*[2,4]*"," part-part:[]*[4,2]*"]
49
50 ==========================================
51 Program thetrickSize...
52
53 Number of size-change graphs: 11
54 Number of compositions       : 26
55
56 Program terminates.
```

B.2 Results of SCT_0 analysis

B.2.1 Glenstrup test suite

```
1  ==========================================
2  Program graphcolour1...
3
4  Number of size-change graphs: 15
5  Number of compositions      : 69
6
7  Possible termination problems:
8
9  [" colorrest-colorrest:[(1,>,2),(1,>=,1),(3,>=,3),(4,>=,4)]*[11,13,12]*"," colo
10 rrestthetrick-colorrestthetrick:[(2,>,1),(2,>=,2),(4,>=,4),(5,>=,5)]*[12,11,13]
11 *"," colorrest-colorrest:[(1,>=,1),(1,>=,2),(3,>=,3),(4,>=,4)]*[11,12]*"," colo
12 rrestthetrick-colorrestthetrick:[(2,>=,1),(2,>=,2),(4,>=,4),(5,>=,5)]*[12,11]*"
13 ]
14
15 ==========================================
16 Program graphcolour2...
17
18 Number of size-change graphs: 17
19 Number of compositions      : 96
20
21 Possible termination problems:
22
23 [" colorrestthetrick-colorrestthetrick:[(2,>,1),(2,>,3),(2,>=,2),(4,>=,4),(5,>=
24 ,5)]*[14,9,12,14,9,12,15]*"," colorrestthetrick-colorrestthetrick:[(2,>,3),(2,>
25 =,1),(2,>=,2),(4,>=,4),(5,>=,5)]*[14,9,12,14,9,12]*"," colorrest-colorrest:[(1,
26 >,2),(1,>=,1),(3,>=,3),(4,>=,4)]*[9,12,15,14]*"," colornoderest-colornoderest:[
27 (1,>,2),(1,>=,1),(4,>=,4),(5,>,3),(5,>=,5)]*[12,14,9,13]*"," colorrest-colorres
28 t:[(1,>=,1),(1,>=,2),(3,>=,3),(4,>=,4)]*[9,12,14]*"," colornoderest-colornodere
29 st:[(1,>=,1),(1,>=,2),(4,>=,4),(5,>,3),(5,>=,5)]*[12,14,9]*"]
30
31 ==========================================
32 Program rematch...
33
34 Number of size-change graphs: 41
35 Number of compositions      : 463
36
37 Possible termination problems:
38
39 [" parsep-parsep:[(1,>=,1),(3,>=,3)]*[14,23]*"," parsepchar-parsepchar:[(1,>=,1
40 ),(3,>=,3)]*[23,14]*"," stringlist-stringlist:[(1,>=,1)]*[40,41]*"," dummy-dumm
41 y:[(1,>=,1)]*[41,40]*"," domatchstar-domatchstar:[(2,>=,2)]*[29]*"," domatchseq
42 -domatchseq:[(3,>=,3)]*[35]*"]
43
44 ==========================================
45 Program assrewrite...
46
47 Number of size-change graphs: 7
48 Number of compositions      : 5
49
50 Possible termination problems:
51
52 [" rewrite-rewrite:[]*[5]*"]
53
54 ==========================================
55 Program nestdec...
56
57 Number of size-change graphs: 5
58 Number of compositions      : 14
59
60 Possible termination problems:
61
62 [" nestdec-nestdec:[]*[3]*"]
63
64 ==========================================
```

```
Program permute...

Number of size-change graphs: 8
Number of compositions      : 33

Possible termination problems:

[" permute-permute:[]*[2,4]*"," select-select:[]*[4,2]*"]

=============================================
Program shuffle...

Number of size-change graphs: 6
Number of compositions      : 12

Possible termination problems:

[" shuffle-shuffle:[]*[2]*"]

=============================================
Program thetrick...

Number of size-change graphs: 11
Number of compositions      : 24

Possible termination problems:

[" f-f:[(1,>=,1)]*[4]*"]

=============================================
Program gcd1...

Number of size-change graphs: 10
Number of compositions      : 62

Possible termination problems:

[" gcd-gcd:[]*[3,5]*"," gcd-gcd:[(2,>=,2)]*[3]*"," gcd-gcd:[(1,>=,1)]*[5]*"]

=============================================
Program gcd2...

Number of size-change graphs: 10
Number of compositions      : 74

Possible termination problems:

[" gcd-gcd:[]*[3,3]*"," gcd-gcd:[(2,>=,2)]*[3,5]*"," gcd-gcd:[(1,>=,1)]*[5,3]*"
]

=============================================
Program mergesort...

Number of size-change graphs: 8
Number of compositions      : 30

Possible termination problems:

[" mergesort-mergesort:[]*[2,3,5]*"," splitmerge-splitmerge:[]*[3,5,2]*"," spli
tmerge-splitmerge:[(3,>,1),(3,>=,3)]*[6,2,3]*"," mergesort-mergesort:[(1,>=,1)]
*[2,5]*"," splitmerge-splitmerge:[(2,>=,1),(2,>=,2),(2,>=,3)]*[5,2]*"," splitme
rge-splitmerge:[(3,>=,1),(3,>=,2),(3,>=,3)]*[6,2]*"]

=============================================
Program minsort...

Number of size-change graphs: 7
Number of compositions      : 23
```

```
Possible termination problems:

[" minsort-minsort:[]*[2,5]*"," appmin-appmin:[]*[5,2]*"]

=============================================
Program quicksort...

Number of size-change graphs: 9
Number of compositions      : 30

Possible termination problems:

[" part-part:[]*[3,7,2]*"," part-part:[(4,>,1),(4,>,2),(4,>=,4)]*[8,2,3]*"," qu
icksort-quicksort:[]*[2,7]*"," quicksort-quicksort:[(1,>=,1)]*[2,8]*"," part-pa
rt:[(4,>,1),(4,>=,2),(4,>=,4)]*[8,2]*"]
```

B.2.2 Other examples

```
=============================================
Program assrewriteSize...

Number of size-change graphs: 8
Number of compositions      : 17

Possible termination problems:

[" rewrite-rewrite:[]*[2,6,3]*"," rw-rw:[]*[6]*"]

=============================================
Program deadcodeSize...

Number of size-change graphs: 1
Number of compositions      : 1

Possible termination problems:

[" g-g:[(1,>=,1)]*[1]*"]

=============================================
Program fghSize...

Number of size-change graphs: 8
Number of compositions      : 22

Possible termination problems:

[" f-f:[]*[2,6,5,3]*"," g-g:[]*[5,3,2,6]*"]

=============================================
Program graphcolour2Size...

Number of size-change graphs: 17
Number of compositions      : 79

Program terminates.


=============================================
Program quicksortSize...

Number of size-change graphs: 12
Number of compositions      : 29

Possible termination problems:

[" quicksort-quicksort:[]*[2,4]*"," part-part:[]*[4,2]*"]
```

```
49
50 ==========================================
51 Program thetrickSize...
52
53 Number of size-change graphs: 11
54 Number of compositions      : 26
55
56 Program terminates.
```

References

[1] Peter Clote. "Computation Models and Function Algebras" Preliminary condensed version, to appear in the *Handbook of Recursive Theory*, ed. E. Griffor.

[2] Arne J. Glenstrup. "Terminator II: Stopping Partial Evaluation of Fully Recursive Programs" Master's Thesis. DIKU, University of Copenhagen. Universitetsparken 1, DK-2100 Copenhagen Ø. June, 1999

[3] Chin Soon Lee, Neil D. Jones and Amir M. Ben-Amram "The Size-Change Principle for Program Termination" POPL 2001: Proceedings 28^{th} ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, January 2001.

[4] Yanhong A. Liu, Scott D. Stoller. "Dynamic Programming via Static Incrementalization" Computer Science Department, Indiana University, Bloomington, ESOP 2000.

[5] Flemming Nielson, Hanne Riis Nielson, Chris Hankin. "Principles of Program Analysis" Preliminary copy, Dagstuhl 1998.

[6] Jens Peter Secher "Perfect Supercompilation" DIKU-TR-99/1. Department of Computer Science (DIKU), University of Copenhagen. February, 1999.

[7] David Wahlstedt. "Detecting termination using size-change in parameter values" Master's thesis, Chalmers University of Technology, 2000. http://www.cs.chalmers.se/ davidw/