Detecting termination using size-change in parameter values

Masters Thesis in Computer Science

David Wahlstedt
Supervisor: Thierry Coquand

30 October 2000

Abstract

We present a method to automatically detect termination in a strict, first order functional language. This is a first step towards an application of the method on Agda [Agda]. The method is based on a paper of Neil Jones et al [Jones 00].

To any program, seen as a set of equations defining recursive functions, we associate a graph of calls, whose arcs are themselves graphs. These graphs describe for each call the size relations between the formal parameters and the actual parameters.

The termination condition can then be stated in terms of these graphs: each infinite path in the graph of calls must contain an infinitely decreasing thread.

What is surprising is that this condition can then be decided by a fully automatic algorithm. The method is quite general, and it is not dependent of auxiliary requirements like for example lexicographically ordered parameters. We have written a small Haskell prototype and tested this method on some examples.

Contents

1 Introduction

1.1 Background

There are several reasons for studying automated termination detection techniques: One general field of application is as an aid for debugging programs, e.g. finding loops that may cause non-termination. More issues motivating automated termination detection – e.g partial evaluation – can be found in [Jones 00]. An algorithm has been introduced for the functional programming paradigm by [Jones 00] which is inspired from [Sagiv 91], a termination detection algorithm for prolog/datalog programs.

1.2 Motivation for this work

In Agda [Agda], notations of functional programming are used for representing proofs: case expressions correspond to case analysis, let expressions represent local definitions/lemmas, and constructors represent introduction rules for inductive definitions. It is then natural to represent inductive proofs by using case expressions and recursive definitions. However for this proof to be correct, the recursion has to be well-founded, i.e. recursive calls have to be decreasing with respect to a well-founded ordering. The algorithm of [Jones 00] provides a uniform method to ensure what may be called "structural well-founded termination", and this can be applied to Agda's functional representation of proofs.

Our contribution is a detailed presentation of the algorithm of [Jones 00], suitably adapted to programs using case notation. We limit ourselves to the data type of unary natural numbers, but we believe that the method can be extended to the general case of data types used in Agda.

1.3 Headlines of the algorithm

Assume we want to define the subtype relation in a Haskell-like language given a subtype function `subA` for some previously defined type `A`, as follows:

Example 1.3.1

```
subA :: A -> A -> Bool

data T = Prim A | Arrow T T

sub :: T -> T -> Bool
sub (Prim a1) (Prim a2) = subA a1 a2
```

```
sub (Arrow t1 u1)(Arrow t2 u2) = sub t2 t1 && sub u1 u2
sub _ _ = False
```

Provided all elements of `T` are well-founded, we know that for instance `Arrow t1 u1` is greater than both `t1` and `u1`, because `t1` and `u1` are subterms of `Arrow t1 u1`. Using this information we can extract size-change relations between the actual and formal parameters in `sub` summarised in what we will call a *"size-change graph"* , a bipartite graph with labelled arcs. In our diagrams of these graphs an arc (or an undirected edge) between nodes i and j denotes that j'th formal parameter is strictly greater (or greater or equal) than the i'th actual parameter. For each function application defined in the body of `sub` we extract a size-change graph in this manner illustrated below:

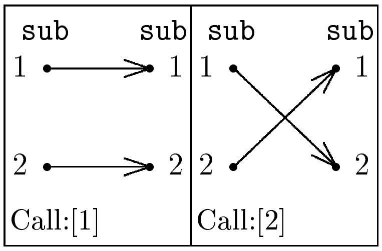

Figure 1: The size-change graphs extracted from `sub`.

From the figure we see that any infinite sequence of these graphs must contain an infinite sequence of strictly decreasing adjacent arcs constituting what we will call an *infinitely descending thread.* The first (and also the second) argument decreases infinitely often. Hence an infinite computation is not possible for any well-founded input, and the program must terminate.

The surprising fact is that the property stating that any infinite composition contains an argument that decreases (formally) infinitely often is decidable. Furthermore the algorithm is relatively simple:

The termination detection method of [Jones 00] consists of two phases. The first is to derive a set of *size-change graphs* from the program we want to check. For each function definition, to each function application in the program, a size-change graph is associated that reflects the known size relations between the formal parameters of the function definition and actual parameter values of the application. The second phase of the analysis then decides if every infinite computation of the program gives an infinitely decreasing sequence with respect to the size-change graphs: We first compute all possible finite compositions, and then we check that every idempotent graph contains a decreasing argument (i.e. an arrow from i to i). In this case there is only one idempotent graph. For that graph both arguments are decreasing.

The concept of size-change graphs will be formally described in Section 3. The termination criterion will be treated in detail in Section 4. Our language and its semantics is defined in Section 2, and in Section 3.3 we connect the programming language with the termination detection criterion.

1.4 Some examples

Earlier techniques for approximating size-change termination have auxiliary requirements on the order of the parameters, etc. In [Abel 98], a lexicographic ordering is required for termination to be detected. The tuple of formal parameters must be strictly greater than any tuple of actual parameters of any call, both direct and indirect recursive. If there is a lexicographic ordering of the components of the tuples such that the above condition holds, the program must terminate. This method is relatively strong and also fast, but there are cases where it fails as we will see below.

The method of [Jones 00] contains the lexicographic ordering termination of [Abel 98], for instance it detects termination of the Ackermann function, that meets these requirements:

Example 1.4.1

$$
\begin{array}{l}
\mathtt{ack}(\mathtt{x}_1, \mathtt{x}_2) = \mathtt{case}\ \mathtt{x}_1\ \mathtt{of} \\
\qquad\qquad \mathtt{0} \to \mathtt{S}(\mathtt{x}_2) \\
\qquad\qquad \mathtt{S}(\mathtt{x}_1{}') \to \mathtt{case}\ \mathtt{x}_2\ \mathtt{of} \\
\qquad\qquad\qquad\qquad \mathtt{0} \to \mathtt{ack}(\mathtt{x}_1{}', \mathtt{S}(\mathtt{0})) \\
\qquad\qquad\qquad\qquad \mathtt{S}(\mathtt{x}_2{}') \to \mathtt{ack}(\mathtt{x}_1{}', \mathtt{ack}(\mathtt{x}_1, \mathtt{x}_2{}'))
\end{array}
$$

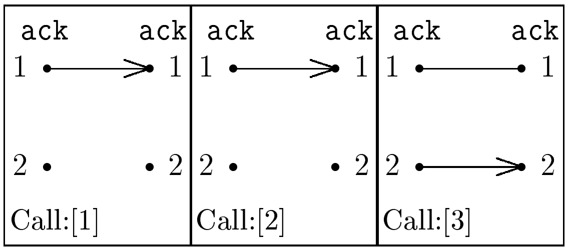

Figure 2: The size-change graphs extracted from `ack`.

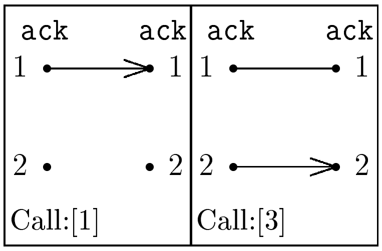

Figure 3: The set of all distinct compositions of graphs of `ack`.

The following example is a translation from a similar program in [Abel 98]. The program outputs a list of elements interleaved from the lists that are the two arguments. In our language the program becomes an addition program adding up ones from every other argument giving their sum as output. The program terminates and does so by the algorithm of [Jones 00], but it does not meet the requirement of lexicographical ordering of parameter tuples, so this example was used in [Abel 98] to show a shortcoming of their algorithm – it cannot detect termination due to swapped parameters.

Example 1.4.2

$$
\begin{array}{lll}
\mathtt{add}(\mathtt{x}_1, \mathtt{x}_2) = & \mathtt{case} & \mathtt{x}_1\ \mathtt{of} \\
& & \mathtt{0} \ \rightarrow\ \mathtt{x}_2 \\
& & \mathtt{S}(\mathtt{x_1}') \ \rightarrow\ \mathtt{S}(\mathtt{add}(\mathtt{x}_2, \mathtt{x_1}'))
\end{array}
$$

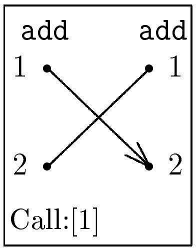

Figure 4: The size-change graphs extracted from `add`.

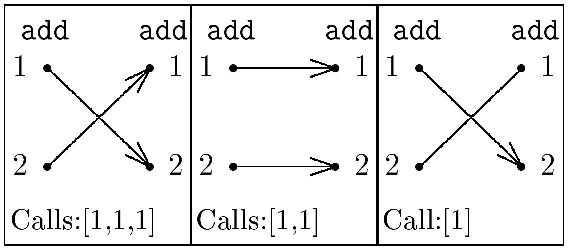

Figure 5: The set of all distinct compositions of graphs of `add`.

A less artificial example of swapped parameters is the subtype function of Example 1.3.1 above, which unfortunately can not yet be implemented in our language due to lack of user-definable type constructors.

The following example from [Abel 98] illustrates how [Jones 00] method applied to a non-terminating program give a useful output that indicates which loops (sequence of calls) may cause non-termination. In Abels example this program shows that the lexicographical ordering requirement is violated by an indirect call `f`, `g`, `g`, `f`.

Example 1.4.3

```
f(x1, x2) = case x1 of
                0 → 0
                S(x1') → case x2 of
                             0 → 0
                             S(x2') → h(g(x1', x2), f(S(S(x1)), x2'))
g(x1, x2) = case x1 of
                0 → 0
                S(x1') → case x2 of
                             0 → 0
                             S(x2') → h(f(x1, x2), g(x1', S(x2)))
h(x1, x2) = case x1 of
                0 → case x2 of
                        0 → 0
                        S(x2') → h(x1, x2')
                S(x1') → h(x1', x2)
```

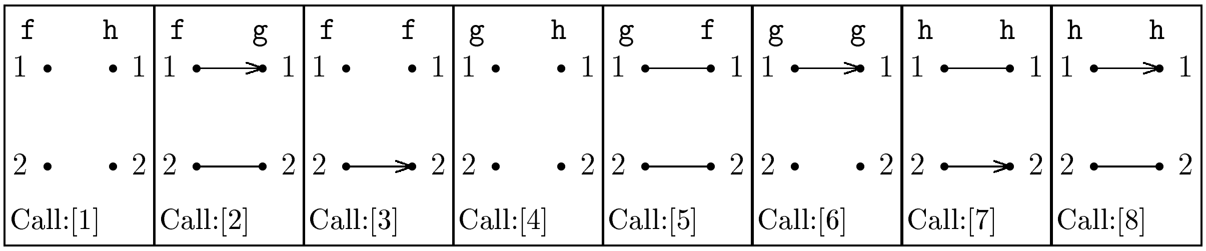

Figure 6: The size-change graphs extracted from f, g, h.

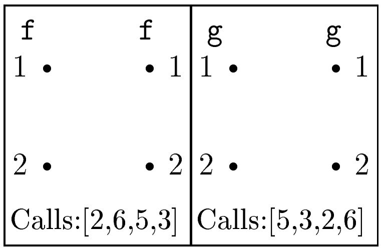

Figure 7: Compositions of graphs of f, g, h identifying a loop that may cause non-termination.

The algorithm of [Jones 00] is PSPACE-hard in program size, which may appear to be a problem. Here follows an example program containing nine graphs. The fixed point iteration giving all finite compositions give 565 distinct graphs. To find out that this program is not terminating by the algorithm, we have to find and idempotent graph that has no decreasing argument.

Example 1.4.4

$$
\begin{aligned}
\mathtt{f_0}(\mathtt{x}_1,\mathtt{x}_2,\mathtt{x}_3,\mathtt{x}_4,\mathtt{x}_5) = \mathtt{case}\ \mathtt{x}_1\ \mathtt{of}\ & \\
& 0 \to 0 \\
& \mathtt{S}(\mathtt{x}_1{}') \to \mathtt{case}\ \mathtt{x}_4\ \mathtt{of} \\
& \qquad 0 \to 0 \\
& \qquad \mathtt{S}(\mathtt{x}_4{}') \to \mathtt{f_1}(\mathtt{x}_2{}',\mathtt{x}_2,\mathtt{x}_4{}',\mathtt{x}_4,\mathtt{S}(\mathtt{S}(\mathtt{x}_5)))
\end{aligned}
$$

$$
\begin{aligned}
\mathtt{f_1}(\mathtt{x}_1,\mathtt{x}_2,\mathtt{x}_3,\mathtt{x}_4,\mathtt{x}_5) = \mathtt{case}\ \mathtt{x}_5\ \mathtt{of}\ & \\
& 0 \to 0 \\
& \mathtt{S}(\mathtt{x}_5{}') \to \mathtt{f_2}(\mathtt{x}_2,\mathtt{x}_1,\mathtt{x}_3,\mathtt{x}_4,\mathtt{x}_5{}')
\end{aligned}
$$

$$
\begin{aligned}
\mathtt{f_2}(\mathtt{x}_1,\mathtt{x}_2,\mathtt{x}_3,\mathtt{x}_4,\mathtt{x}_5) = \mathtt{case}\ \mathtt{x}_3\ \mathtt{of}\ & \\
& 0 \to 0 \\
& \mathtt{S}(\mathtt{x}_3{}') \to \mathtt{case}\ \mathtt{x}_4\ \mathtt{of} \\
& \qquad 0 \to \mathtt{case}\ \mathtt{x}_5\ \mathtt{of} \\
& \qquad\qquad 0 \to 0 \\
& \qquad\qquad \mathtt{S}(\mathtt{x}_5{}') \to \mathtt{f_3}(\mathtt{x}_1,\mathtt{x}_2,\mathtt{x}_3{}',\mathtt{x}_4,\mathtt{x}_5{}') \\
& \qquad \mathtt{S}(\mathtt{x}_4{}') \to \mathtt{case}\ \mathtt{x}_5\ \mathtt{of} \\
& \qquad\qquad\quad 0 \to 0 \\
& \qquad\qquad\quad \mathtt{S}(\mathtt{x}_5{}') \to \mathtt{f_5}(\mathtt{x}_1,\mathtt{x}_2,\mathtt{x}_3,\mathtt{x}_4{}',\mathtt{x}_5{}')
\end{aligned}
$$

$$
\begin{aligned}
\mathtt{f_3}(\mathtt{x}_1,\mathtt{x}_2,\mathtt{x}_3,\mathtt{x}_4,\mathtt{x}_5) = \mathtt{case}\ \mathtt{x}_5\ \mathtt{of}\ & \\
& 0 \to 0 \\
& \mathtt{S}(\mathtt{x}_5{}') \to \mathtt{f_4}(\mathtt{x}_1,\mathtt{x}_2,\mathtt{x}_4,\mathtt{x}_3,\mathtt{x}_5{}')
\end{aligned}
$$

$$
\begin{aligned}
\mathtt{f_4}(\mathtt{x}_1,\mathtt{x}_2,\mathtt{x}_3,\mathtt{x}_4,\mathtt{x}_5) = \mathtt{case}\ \mathtt{x}_1\ \mathtt{of}\ & \\
& 0 \to 0 \\
& \mathtt{S}(\mathtt{x}_1{}') \to \mathtt{case}\ \mathtt{x}_2\ \mathtt{of} \\
& \qquad 0 \to \mathtt{case}\ \mathtt{x}_5\ \mathtt{of} \\
& \qquad\qquad 0 \to 0 \\
& \qquad\qquad \mathtt{S}(\mathtt{x}_5{}') \to \mathtt{f_3}(\mathtt{x}_1{}',\mathtt{x}_2,\mathtt{x}_3,\mathtt{x}_4,\mathtt{x}_5{}') \\
& \qquad \mathtt{S}(\mathtt{x}_2{}') \to \mathtt{case}\ \mathtt{x}_5\ \mathtt{of} \\
& \qquad\qquad\quad 0 \to 0 \\
& \qquad\qquad\quad \mathtt{S}(\mathtt{x}_5{}') \to \mathtt{f_2}(\mathtt{x}_1,\mathtt{x}_2{}',\mathtt{x}_3,\mathtt{x}_4,\mathtt{x}_5{}')
\end{aligned}
$$

$$
\begin{aligned}
\mathtt{f_5}(\mathtt{x}_1,\mathtt{x}_2,\mathtt{x}_3,\mathtt{x}_4,\mathtt{x}_5) = \mathtt{case}\ \mathtt{x}_5\ \mathtt{of}\ & \\
& 0 \to 0 \\
& \mathtt{S}(\mathtt{x}_5{}') \to \mathtt{f_6}(\mathtt{x}_2,\mathtt{x}_1,\mathtt{x}_3,\mathtt{x}_4,\mathtt{x}_5{}')
\end{aligned}
$$

$$
\begin{aligned}
\mathtt{f_6}(\mathtt{x}_1,\mathtt{x}_2,\mathtt{x}_3,\mathtt{x}_4,\mathtt{x}_5) = \mathtt{case}\ \mathtt{x}_5\ \mathtt{of}\ & \\
& 0 \to 0 \\
& \mathtt{S}(\mathtt{x}_5{}') \to \mathtt{f_0}(\mathtt{x}_1,\mathtt{x}_2,\mathtt{x}_4,\mathtt{x}_3,\mathtt{x}_5{}')
\end{aligned}
$$

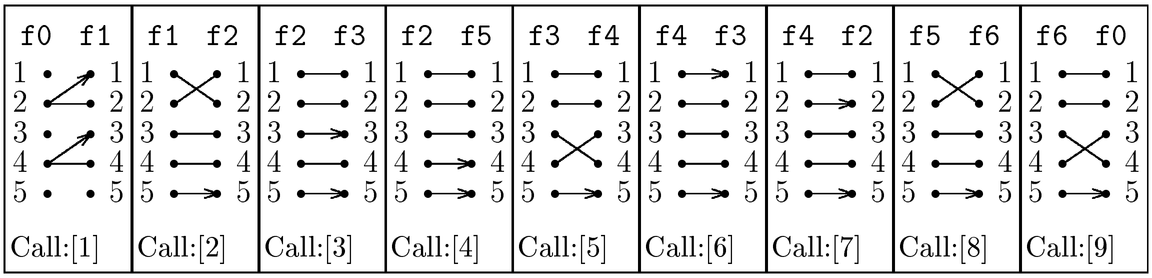

Figure 8: The size-change graphs extracted from example 1.4.4

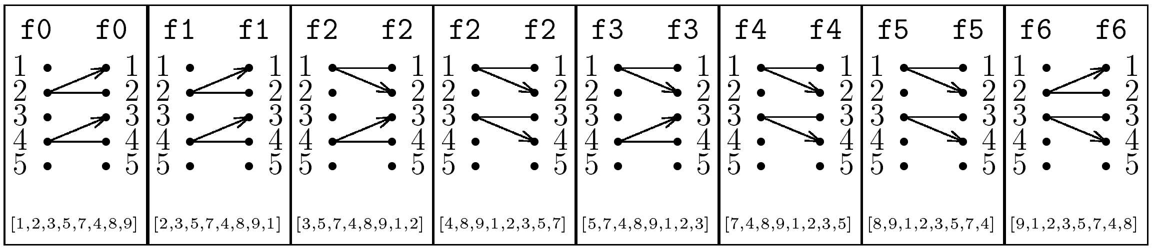

Figure 9: The sources of possible non-termination from example 1.4.4 – cyclic idempotent compositions of size-change graphs. The lists of numbers indicate the sequences of size-change graphs from Figure 8 that produced the corresponding composition.

2 Language, syntax, semantics

2.1 Language

We consider a strict first-order functional language:

$$
\begin{array}{llll}
\mathtt{p} & \in & \mathit{Prog} & \text{, the set of programs} \\
\mathtt{def} & \in & \mathit{Fundef} & \text{, the set of function definitions} \\
\mathtt{f} & \in & \mathit{FunName} & \text{, the set of function names} \\
\mathtt{e} & \in & \mathit{Exp} & \text{, the set of expressions} \\
\mathtt{t} & \in & \mathit{Term} & \text{, the set of non-case expressions} \\
\mathtt{x}, \mathtt{y} & \in & \mathit{Var} & \text{, the set of variable names}
\end{array}
$$

Programs in abstract notation are defined inductively as follows:

- A program $\mathtt{p}$ in this language is a list of definitions $\mathtt{def}_1, \ldots, \mathtt{def}_m$.
- A function definition $\mathtt{def}_i$ is written $\mathtt{f}_i(\mathtt{x}_1, \ldots, \mathtt{x}_{ar(\mathtt{f}_i)}) = \mathtt{e}_{\mathtt{f}_i}$, where all the formal parameters $\mathtt{x}_1, \ldots, \mathtt{x}_{ar(\mathtt{f}_i)}$ are distinct and $ar(\mathtt{f}_i)$ denotes the arity of function $\mathtt{f}_i$. The expression $\mathtt{e}_{\mathtt{f}_i}$ is called the function body of $\mathtt{f}_i$.
- An expression $\mathtt{e}$ is either a term $\mathtt{t}$ or a case-expression $\mathtt{case\ x\ e}_1\ \mathtt{e}_2$. To make it more comprehensible we will write $\mathtt{case\ x\ of\ 0} \to \mathtt{e}_1\ \ \mathtt{S}(\mathtt{x}') \to \mathtt{e}_2$.
- A term $\mathtt{t}$ is either a variable $\mathtt{x}$, $\mathtt{0}$, or the successor of a term $\mathtt{S}(\mathtt{t})$ or a function application $\mathtt{f}_i(\mathtt{t}_1, \ldots, \mathtt{t}_{ar(\mathtt{f}_i)})$, where $\mathtt{t}_j$ are the actual parameters of the application.

We use variable names that are of the form $\mathtt{x}_i$ or $\mathtt{x}'$, where $\mathtt{x}$ is a variable name. We define $\mathtt{x}^{(k)}$ as follows:

$$
\begin{array}{lll}
\mathtt{x} & = & \mathtt{x}^{(0)} \\
\mathtt{x}^{(k)\prime} & = & \mathtt{x}^{(k+1)}
\end{array}
$$

Definition 2.1.1 (Free variables)

$$\frac{(\mathtt{y} = \mathtt{x}) \ \textit{or} \ (\mathtt{y} \ \textit{free in} \ \mathtt{e}_1) \ \textit{or} \ (\mathtt{y} \neq \mathtt{x}' \ \textit{and} \ \mathtt{y} \ \textit{free in} \ \mathtt{e}_2)}{\mathtt{y} \ \textit{free in} \ (\mathtt{case\ x\ of\ 0} \to \mathtt{e}_1 \ \ \mathtt{S(x')} \to \mathtt{e}_2)}$$

$$\frac{}{\mathtt{y} \ \textit{free in} \ \mathtt{y}}$$

$$\frac{\mathtt{y} \ \textit{free in} \ \mathtt{t}}{\mathtt{y} \ \textit{free in} \ \mathtt{S(t)}}$$

$$\frac{\exists j.\mathtt{y} \ \textit{free in} \ \mathtt{t}_j}{\mathtt{y} \ \textit{free in} \ \mathtt{f}_i(\mathtt{t}_1, \ldots, \mathtt{t}_{ar(\mathtt{f}_i)})}$$

Definition 2.1.2 (Call) *The pair* $(\mathtt{f}_i, \mathtt{f}_j(\mathtt{t}_1, \ldots, \mathtt{t}_{ar(\mathtt{f}_j)}))$ *is a call for* $\mathtt{f}_i(\mathtt{x}_1, \ldots, \mathtt{x}_{ar(\mathtt{f}_i)})$*, if* $\mathtt{f}_j(\mathtt{t}_1, \ldots, \mathtt{t}_{ar(\mathtt{f}_j)})$ *is a sub-term of* $\mathtt{e}_{\mathtt{f}_i}$*.*

Definition 2.1.3 (Syntactical size relation)
Define the relations $>$ *and* $\geq$ *in* $Exp \times Exp$ *as follows:*

$\mathtt{x}_i > \mathtt{S}^l(\mathtt{x}_i^{(k)})$ *if* $k > l$
$\mathtt{x}_i \geq \mathtt{S}^l(\mathtt{x}_i^{(k)})$ *if* $k = l$

From now on, we assume given a program $\mathtt{p}$ with the following property: The set of free variables in $\mathtt{e}_{\mathtt{f}_i}$ is contained in $\{\mathtt{x}_1, \ldots, \mathtt{x}_{ar(\mathtt{f}_i)}\}$, the formal parameters of $\mathtt{f}_i$, for each i.

2.2 Semantics

A denotational semantics will be given for the language. D is the usual flat domain of natural numbers. If D_1 and D_2 are domains, let $[D_1 \to D_2]$ be the domain of continuous functions from D_1 to D_2.

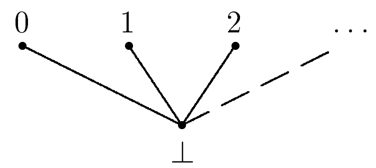

Figure 10: The semantic domain D

Let

$$
\begin{array}{l}
D = \mathbb{N} \cup \{\bot\} \\
VarEnv = Var \to D \\
FunEnv = \prod_{\mathtt{f} \in FunName} [D^{ar(\mathtt{f})} \to D] \\
\mathcal{E} \in Exp \to [FunEnv \to \ [VarEnv \to D]] \\
\mathcal{P} \in Prog \to [D^{ar(\mathtt{f}_1)} \to D]
\end{array}
$$

Syntactical constructors $\mathtt{0}$ and $\mathtt{S}$ are distinguished from the value constructors $\bar{\mathtt{0}}$ and $\bar{\mathtt{S}}$ for elements of $\mathbb{N}$, to avoid confusion.

Definition 2.2.1 (Variable environments)
Given $\rho = (\mathtt{x}_1 = n_1, \dots, \mathtt{x}_k = n_k)$, *then*

$$
\rho(\mathtt{x}_i) = \begin{cases} n_i & , \quad \textit{if } i \in [1..k] \\ \bot & , \quad \textit{otherwise.} \end{cases}
$$

where $\rho \in VarEnv$.

Definition 2.2.2 (Predecessor function)

$$
PRED(d) = \begin{cases} n & , \quad \textit{if } d = \bar{\mathtt{S}}(n), n \in \mathbb{N} \\ \bot & , \quad \textit{otherwise.} \end{cases}
$$

Definition 2.2.3 *The tuple* $(d_1, \dots, d_n) \in D^n$ *is well-defined if and only if all of its components are distinct from* $\bot$.

Definition 2.2.4 *The semantics of a program is defined as follows:*

$$\mathcal{E}_{\phi}[\![\mathtt{0}]\!]_{\rho} = \bar{\mathtt{0}}$$

$$\mathcal{E}_{\phi}[\![\mathtt{S(t)}]\!]_{\rho} = \begin{cases} \perp & , \ \textit{if } \mathcal{E}_{\phi}[\![\mathtt{t}]\!]_{\rho} = \perp \\ \bar{\mathtt{S}}(\mathcal{E}_{\phi}[\![\mathtt{t}]\!]_{\rho}) & , \ \textit{otherwise.} \end{cases}$$

$$\mathcal{E}_{\phi}[\![\mathtt{x}_i]\!]_{\rho} = \rho(\mathtt{x}_i)$$

$$\mathcal{E}_{\phi}[\![\mathtt{x}']\!]_{\rho} = PRED(\mathcal{E}_{\phi}[\![\mathtt{x}]\!]_{\rho})$$

$$\mathcal{E}_{\phi}[\![\mathtt{case\ x\ of\ 0} \to \mathtt{e}_1 \ \ \mathtt{S(x')} \to \mathtt{e}_2]\!]_{\rho} =$$
$$\begin{cases} \perp & , \ \textit{if } \mathcal{E}_{\phi}[\![\mathtt{x}]\!]_{\rho} = \perp \\ \mathcal{E}_{\phi}[\![\mathtt{e}_1]\!]_{\rho} & , \ \textit{if } \mathcal{E}_{\phi}[\![\mathtt{x}]\!]_{\rho} = \bar{\mathtt{0}} \\ \mathcal{E}_{\phi}[\![\mathtt{e}_2]\!]_{\rho} & , \ \textit{if } \mathcal{E}_{\phi}[\![\mathtt{x}]\!]_{\rho} = \bar{\mathtt{S}}(n), \ \textit{for some } n \in \mathbb{N}. \end{cases}$$

$$\mathcal{E}_{\phi}[\![\mathtt{f}_i(\mathtt{t}_1, \ldots, \mathtt{t}_{ar(\mathtt{f}_i)})]\!]_{\rho} =$$
$$\begin{cases} \phi(\mathtt{f}_i)(\vec{d}) & , \ \textit{if } \ \vec{d} \textit{ is well-defined} \\ \perp & , \ \textit{otherwise.} \end{cases}$$
$$\textit{where } \vec{d} = (\mathcal{E}_{\phi}[\![\mathtt{t}_1]\!]_{\rho}, \ldots, \mathcal{E}_{\phi}[\![\mathtt{t}_{ar(\mathtt{f}_i)}]\!]_{\rho}).$$

Let
$\tau_i \in [FunEnv \to [D^{ar(\mathtt{f}_i)} \to D]]$
$\phi \mapsto ((d_1, \ldots, d_{ar(\mathtt{f}_i)}) \mapsto \mathcal{E}_{\phi}[\![\mathtt{e}_{\mathtt{f}_i}]\!]_{(\mathtt{x}_1 = d_1, \ldots, \mathtt{x}_{ar(\mathtt{f}_i)} = d_{ar(\mathtt{f}_i)})})$
$\tau \in [FunEnv \to FunEnv]$
$\phi \mapsto \mathtt{f}_i \mapsto \tau_i(\phi)$
$\Omega \in FunEnv$
$\mathtt{f}_i \mapsto (d_1, \ldots, d_{ar(\mathtt{f}_i)}) \mapsto \perp$

Let ϕ_∞ denote the least fixed point of τ, a solution of $\tau(\phi) = \phi$:

$$\phi_\infty = \bigsqcup_{k=0}^{\infty} \{\tau^k(\Omega)\}$$

Then,

$$\mathcal{P}[\![\mathtt{f}_1(\mathtt{x}_1, \ldots, \mathtt{x}_{ar(\mathtt{f}_1)}) = \mathtt{e}_{\mathtt{f}_1}, \ldots, \mathtt{f}_m(\mathtt{x}_1, \ldots, \mathtt{x}_{ar(\mathtt{f}_m)}) = \mathtt{e}_{\mathtt{f}_m}]\!] = \phi_\infty(\mathtt{f}_1)$$

Thus, the first function in the list of definitions is considered as being the entry function of the program.

Lemma 2.2.1 *If* $\mathtt{x}_i \ R \ \mathtt{t}$*, and* $\mathcal{E}_\phi[\![\mathtt{t}]\!]_\rho \neq \bot$*, where* $R \in \{>, \geq\}$*, then*

$$\rho(\mathtt{x}_i) \ R \ \mathcal{E}_\phi[\![\mathtt{t}]\!]_\rho.$$

Proof: Assume $\mathtt{x}_i \ > \ \mathtt{t}$, and $\mathcal{E}_\phi[\![\mathtt{t}]\!]_\rho \neq \bot$. Then by Definition 2.1.3, we must have $\mathtt{t} = \mathtt{S}^l(\mathtt{x}_i{}^{(k)})$, where $k > l$. By assumption $\mathcal{E}_\phi[\![\mathtt{t}]\!]_\rho \neq \bot$, so $\rho(\mathtt{x}_i^{(k)}) \neq \bot$. So $\bot \neq \rho(\mathtt{x}_i) = m \geq k$. By Definition 2.2.4, $\mathcal{E}_\phi[\![\mathtt{S}^l(\mathtt{x}_i{}^{(k)})]\!]_\rho = \bar{\mathtt{S}}^l(\rho(\mathtt{x}_i) - k) = m - k + l < m$, because $k > l$. The proof for $\geq$ is similar. $\square$

3 Call graph and size-change graphs

3.1 Size-change graphs

Definition 3.1.1 *For a call* $c = (\mathtt{f}_i, \mathtt{f}_j(\mathtt{t}_1, \ldots, \mathtt{t}_{ar(\mathtt{f}_j)}))$ *for* $\mathtt{f}_i(\mathtt{x}_1, \ldots, \mathtt{x}_{ar(\mathtt{f}_i)})$*, we define its size-change graph to be a bipartite graph,*

$$G_c = ([1..ar(\mathtt{f}_i)], [1..ar(\mathtt{f}_j)], E)$$

where E *is a set of labelled arcs:* $E \subseteq [1..ar(\mathtt{f}_i)] \times \{>, \geq\} \times [1..ar(\mathtt{f}_j)]$*, where* $(m, R, n) \in E$ *if and only if* $\mathtt{x}_m \ R \ \mathtt{t}_n$*, according to Definition 2.1.3.*

We will say that a size-change graph (V, V, E) is *decreasing* if it contains an arc of the form $(i, >, i)$ for some $i \in V$.

Definition 3.1.2 *The set of size change graphs* $\mathcal{G}_\mathtt{p}$ *of a program* $\mathtt{p}$ *is defined as follows:*

$$\mathcal{G}_\mathtt{p} = \{G_c \mid c \text{ is a call } (\mathtt{f}_i, \mathtt{f}_j(\mathtt{t}_1, \ldots, \mathtt{t}_{ar(\mathtt{f}_j)})) \text{ in program } \mathtt{p} \ \}$$

Definition 3.1.3 *By juxtaposition the relations* $>, \geq$ *are composed as follows:*

$$R_1 R_2 = \begin{cases} > & , \ \text{if } R_1 \text{ is } > \text{ or } R_2 \text{ is } > \\ \geq & , \ \text{otherwise.} \end{cases}$$

Definition 3.1.4 *Given arc sets*

$$E_1 = \{(i_1, R_2, i_2) | i_1 \in [1..ar(\mathtt{f}_i)], i_2 \in [1..ar(\mathtt{f}_j)], R_2 \in \{>, \geq\}\}$$

and

$$E_2 = \{(i_2, R_3, i_3) | i_2 \in [1..ar(\mathtt{f}_j)], i_3 \in [1..ar(\mathtt{f}_k)], R_3 \in \{>, \geq\}\}$$

their composition $E_1 E_2$ *is defined as follows:*

$$E_1 E_2 = \{(i, RR', k) \mid (i, R, j) \in E_1, \ (j, R', k) \in E_2\}$$

Definition 3.1.5 *Given size-change graphs* $G = (V_1, V_2, E)$ *and* $H = (V_2, V_3, E')$ *their composition is* $GH = (V_1, V_3, EE')$*, where* EE' *is the composition of arc sets* E *and* E'*.*

3.2 Call graph

Definition 3.2.1 *The call graph* $\Gamma_{\mathtt{p}}$*, of program* $\mathtt{p} = \mathtt{def}_1, \ldots, \mathtt{def}_m$ *is the directed graph* (V, E) *whose edges* $E = \mathcal{G}_{\mathtt{p}}$*, and whose vertices* $V = [1..m]$*, the indices of the function names in* p*. Each arc* G_c *of* $\Gamma_{\mathtt{p}}$ *has a source and a target* $(s,t)G_c = (i,j)$ *among the vertices* $i, j \in V$*, such that* $(\mathtt{f}_i, \mathtt{f}_j(\mathtt{t}_1, \ldots, \mathtt{t}_{ar(\mathtt{f}_j)}))$ *is a call in* p*.*

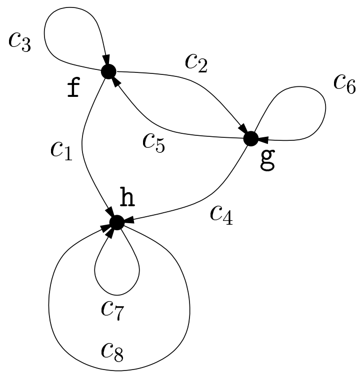

Figure 11: Call graph of Example 1.4.3. The arc c_i is associated to the i'th size-change graph in Figure 6.

Definition 3.2.2 *A path*

$$\gamma = G_{c_1}, G_{c_2}, \ldots$$

in $\Gamma_{\mathtt{p}}$ *is a sequence of arcs of* $\Gamma_{\mathtt{p}}$ *that may be finite or infinite, such that* $t(G_{c_i}) = s(G_{c_{i+1}})$*.*

Definition 3.2.3 *We define a thread of a path* $G_{c_1}, G_{c_2}, \ldots$ *as a sequence of arcs*

$$(m_1, R_2, m_2), (m_2, R_3, m_3), \ldots$$

such that (m_k, R_{k+1}, m_{k+1}) *is an arc of* $G_{c_{k+1}}$*.*

Definition 3.2.4 *A path is said to be infinitely descending if and only if it contains a thread* $\nu = (m_1, R_2, m_2), (m_2, R_3, m_3), \ldots$ *such that there are infinitely many indices* k *where* R_k *is* $>$ *in the thread* ν*.*

Definition 3.2.5 *Given*

$$\gamma = G_1, G_2, \dots, G_k$$

a finite path of $\Gamma_{\mathtt{p}}$, *let* G_γ *denote the composition:*

$$G_1 G_2 \dots G_k$$

Remark:
If (i_0, R, i_k) is an arc of G_γ, then γ has a thread

$$(i_0, R_1, i_1), (i_1, R_2, i_2), \dots, (i_{k-1}, R_k, i_k)$$

such that $R = R_1 R_2 \dots R_k$. See Figure 12 below for an intuition.

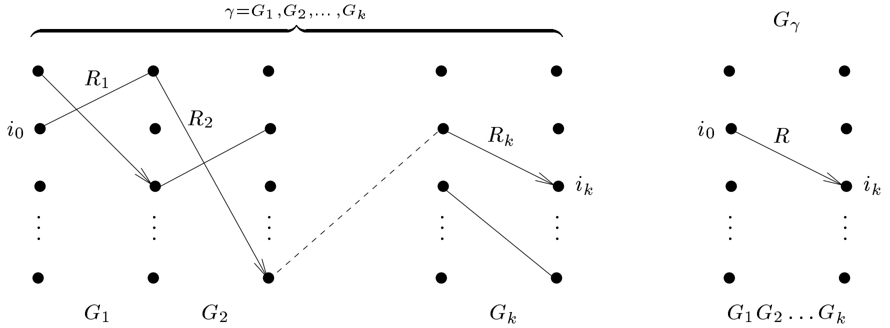

Figure 12: An example of a thread in γ and a corresponding arc in G_γ.

3.3 Properties of non-termination

Lemma 3.3.1 *If* $\mathcal{E}_{\phi}[\![\mathtt{e}]\!]_\rho = \bot$ *and* $\rho(\mathtt{x}) \neq \bot$ *for all* $\mathtt{x}$ *free in* $\mathtt{e}$, *then* $\mathtt{e}$ *has a sub-term* $\mathtt{f}_i(\mathtt{t}_1, \dots, \mathtt{t}_{ar(\mathtt{f}_i)})$ *such that*
$\mathcal{E}_{\phi}[\![\mathtt{f}_i(\mathtt{t}_1, \dots, \mathtt{t}_{ar(\mathtt{f}_i)})]\!]_\rho = \bot$, *and* $\forall j. \mathcal{E}_{\phi}[\![\mathtt{t}_j]\!]_\rho \neq \bot$.

Proof:
By structural induction on $\mathtt{e}$:
Assume $\mathcal{E}_{\phi}[\![\mathtt{e}]\!]_\rho = \bot$ and $\rho(\mathtt{x}) \neq \bot$ for all $\mathtt{x}$ free in $\mathtt{e}$.
Then either:

1. $\mathtt{e}$ is a free variable $\mathtt{x}$. This is not possible because $\rho(\mathtt{x}) \neq \bot$ by assumption.

2. $\mathtt{e}$ is $\mathtt{0}$. This is obviously not possible.

3. $\mathtt{e}$ is $\mathtt{S(t)}$. Then apply Lemma 3.3.1 recursively on $\mathtt{t}$.

4. $\mathtt{e}$ is $\mathtt{case\ x\ of\ 0} \to \mathtt{e}_1\ \ \mathtt{S(x')} \to \mathtt{e}_2$. Then either:

   (a) $\rho(\mathtt{x}) = \bot$. Impossible by assumption.

   (b) $\rho(\mathtt{x}) = \bar{\mathtt{0}}$. Then apply Lemma 3.3.1 recursively on $\mathtt{e}_1$.

   (c) $\rho(\mathtt{x}) = \bar{\mathtt{S}}(d)$. Now, $\mathtt{x}'$ may be free in $\mathtt{e}_2$, but $\rho(\mathtt{x}') = d \neq \bot$. Hence we can apply Lemma 3.3.1 recursively on $\mathtt{e}_2$.

5. $\mathtt{e}$ is $\mathtt{f}_i(\mathtt{t}_1, \ldots, \mathtt{t}_{ar(\mathtt{f}_i)})$. Then either:

   (a) $\exists j \in [1..ar(\mathtt{f}_i)].\mathcal{E}_{\phi}[\![\mathtt{t}_j]\!]_\rho = \bot$. Then apply Lemma 3.3.1 recursively on $\mathtt{t}_j$.

   (b) $\forall j \in [1..ar(\mathtt{f}_i)].\mathcal{E}_{\phi}[\![\mathtt{t}_j]\!]_\rho \neq \bot$. Then Lemma 3.3.1 holds.

Since $\mathtt{e}$ is finite, case 5b is the only valid terminal alternative, which must apply to some sub-term of $\mathtt{e}$. $\square$

Lemma 3.3.2 *If $\phi_\infty(\mathtt{f}_{i_0})(\vec{d_0}) = \bot$, where $\vec{d_0}$ is well-defined, then there exists an infinite sequence*

$$(c_1, \vec{d_1}), (c_2, \vec{d_2}), (c_3, \vec{d_3}), \ldots$$

such that $(m, R, n) \in G_{c_{j+1}} \Rightarrow \vec{d_j}(m)\ R\ \vec{d}_{j+1}(n)$, where $\vec{d_j}$ is well-defined, for all $j \in \mathbb{N}$ and c_1 is a call for $\mathtt{f}_{i_0}$.

Proof: Assume $\phi_\infty(\mathtt{f}_{i_0})(\vec{d_0}) = \bot$ and $\vec{d_0}$ is well-defined. Then, according to Definition 2.2.4, $\mathcal{E}_{\phi_\infty}[\![\mathtt{e}_{\mathtt{f}_{i_0}}]\!]_\rho = \bot$, where $\rho = (\mathtt{x}_1 = \vec{d_0}(1), \ldots, \mathtt{x}_{a_{i_0}} = \vec{d_0}(a_{i_0}))$. Recall that from the hypothesis of Section 2.1 that these are the only possible free variables of $\mathtt{e}_{\mathtt{f}_{i_0}}$. By Lemma 3.3.1, $\mathtt{e}_{\mathtt{f}_i}$ contains a call $c_1 = (\mathtt{f}_{i_0}, \mathtt{f}_{i_1}(\mathtt{t}_1, \ldots, \mathtt{t}_{ar(\mathtt{f}_{i_1})}))$ such that $\mathcal{E}_{\phi_\infty}[\![\mathtt{f}_{i_1}(\mathtt{t}_1, \ldots, \mathtt{t}_{ar(\mathtt{f}_{i_1})})]\!]_\rho = \bot$, where $(\mathcal{E}_{\phi_\infty}[\![\mathtt{t}_1]\!]_\rho, \ldots, \mathcal{E}_{\phi_\infty}[\![\mathtt{t}_{a_{i_1}}]\!]_\rho) = \vec{d_1}$ and $\vec{d_1}$ is well-defined.

Now assume (m, R, n) is an arc of G_{c_1}. Then by Definition 3.1.1, we have $\mathtt{x}_m\ R\ \mathtt{t}_n$. This implies $\rho(\mathtt{x}_m)\ R\ \mathcal{E}_{\phi_\infty}[\![\mathtt{t}_n]\!]_\rho$, by Lemma 2.2.1 because $\mathcal{E}_{\phi_\infty}[\![\mathtt{t}_n]\!]_\rho \neq \bot$. Since $\rho(\mathtt{x}_m) = \vec{d_0}(m)$ and $\mathcal{E}_{\phi_\infty}[\![\mathtt{t}_n]\!]_\rho = \vec{d_1}(n)$, we must have $\vec{d_0}(m)\ R\ \vec{d_1}(n)$.

We have that $\phi_\infty(\mathtt{f}_{i_1})(\vec{d_1}) = \bot$, so we can proceed repeatedly in the same way to build the infinite sequence $(c_1, \vec{d_1}), (c_2, \vec{d_2}), (c_2, \vec{d_3}), \ldots$, with the properties stated above. $\square$

Theorem 3.3.1 *If every infinite path in $\Gamma_{\mathtt{p}}$ is infinitely descending, then* $\mathtt{p}$ *terminates on all well-defined input.*

Proof: Assume $\mathcal{P}[\![\mathtt{p}]\!](\vec{d_0}) = \perp$, where $\vec{d_0}$ is well-defined, and that all the infinite paths of $\Gamma_{\mathtt{p}}$ are infinitely descending. Then $\phi_\infty(\mathtt{f}_1)(\vec{d_0}) = \perp$, so by Lemma 3.3.2 there exists an infinite sequence $s = (c_1, \vec{d_1}), (c_2, \vec{d_2}), (c_3, \vec{d_3}), \ldots$ such that $\vec{d_j}$ is well-defined and $c_{j+1} = (\mathtt{f}_{i_j}, \mathtt{f}_{i_{j+1}}(\mathtt{t}_1, \ldots, \mathtt{t}_{ar(\mathtt{f}_{i_{j+1}})}))$ is a call for $\mathtt{f}_{i_j}(\mathtt{x}_1, \ldots, \mathtt{x}_{ar(\mathtt{f}_{i_j})})$, where $i_0 = 1$, and whenever (m, R, n) is an arc of $G_{c_{j+1}}$, then $\vec{d_j}(m)\ R\ \vec{d}_{j+1}(n)$.

Consider the infinite path $\gamma = G_{c_1}, G_{c_2}, G_{c_3}, \ldots$ By assumption γ is infinitely descending, so there is a thread of γ:

$$(m_l, R_{l+1}, m_{l+1}), (m_{l+1}, R_{l+2}, m_{l+2}), (m_{l+2}, R_{l+3}, m_{l+3}), \ldots$$

starting at some index l in γ, such that $(m_j, R_{j+1}, m_{j+1}) \in G_{c_{j+1}}$, where there are infinitely many j such that R_j is $>$. Let us define $d_k = \vec{d_k}(m_k)$. Recall that by Lemma 3.3.2 we have $d_k\ R_{k+1}\ d_{k+1}$ with $R_{k+1} \in \{>, \geq\}$, for all $k \geq l$, and the set $\{j \mid R_{l+j} \text{ is } >\}$ is infinite. Hence $d_l, d_{l+1}, d_{l+2}, \ldots$ is an infinitely descending sequence in $\mathbb{N}$, which is absurd. $\Box$

4 Termination detection algorithm

4.1 Ramsey's theorem

Definition 4.1.1 *Let $[A]^2$ denote the set $\{B \subseteq A : |B| = 2\}$. By a "finite coloration" of $[A]^2$, we mean a function from $[A]^2$ into a finite set.*

Theorem 4.1.1 Ramsey's theorem*(Binary infinite version [1])*
Given any finite colouring χ of $[\mathbb{N}]^2$, there exists an infinite subset T of $\mathbb{N}$, such that $[T]^2$ is monochromatic under χ (that is χ is constant on $[T]^2$).

Proof: Define the infinite sequence of infinite sets S_i as follows:

i) Let $S_1 = \mathbb{N}$.
ii) Having chosen S_i, choose $x_i \in S_i$ arbitrarily.
iii) Let $T_j = \{u \in S_i \mid \chi(x_i, u) = j\}$.

Then $\{T_j \mid j \in \chi\}$ is a finite partition on $S_i \setminus \{x_i\}$. Choose S_{i+1} to be one of the infinite T_j. The sequence

$$X = x_1, x_2, x_3, \ldots$$

[1] The proof principally follows [Graham 80], page 16.

has the property that $\chi(x_i, x_j) = \chi(x_i, x_k)$ for all $j > i, k > i$, because if $k > j$, then $S_k \subseteq S_j$. Define a colouring χ^* of the singletons x_i

$$\chi^*(x_i) = c \text{ such that } \chi(x_i, x_j) = c \text{ for all } j > i.$$

Then χ^* is a finite partition on the sequence X. So for some colour j, there must be an infinite sequence

$$X' = x_{i_1}, x_{i_2}, x_{i_3}, \ldots$$

where $i_1, i_2, i_3, \ldots$ are indices of elements in X, such that $\chi^*(x_{i_s}) = j$ for all s. But for any $1 \leq s < t$,

$$\chi(x_{i_s}, x_{i_t}) = \chi^*(x_{i_s}) = j.$$

Hence $[X']^2$ is monochromatic under χ. $\square$

4.2 Termination criterion using size-change graphs

Definition 4.2.1 *Define $\mathcal{S}$ to be the set $\{G_\gamma \mid \gamma$ is a finite path of $\Gamma_{\mathtt{p}}\}$.*

Note that $\mathcal{S}$ is a finite set, because for each finite path γ in $\Gamma_{\mathtt{p}}$, where $G_\gamma = (V_1, V_2, E)$ there is only a finite set of possible configurations of E. And there is only a finite number of pairs of vertices in $\Gamma_{\mathtt{p}}$.

Theorem 4.2.1 *All the infinite paths of $\Gamma_{\mathtt{p}}$ are infinitely descending if and only if for every $G \in \mathcal{S}$ such that $G^2 = G$ there is an arc $(n, >, n)$ in G.*

Proof: Suppose for every $G \in \mathcal{S}$ such that $G^2 = G$ there is an arc $(n, >, n)$ in G. Choose an infinite path γ of $\Gamma_{\mathtt{p}}$. We label the edges of γ as follows:

$$\gamma = G_1, G_2, G_3, \ldots$$

Let γ_{ij} denote the finite section $G_i, G_{i+1}, \ldots, G_{j-1}$ of γ. Then we can define the function $\chi : [\mathbb{N}]^2 \to \mathcal{S}$ as: $\chi(i, j) = G_{\gamma_{ij}}$ for $j > i$.

$\mathcal{S}$ is a finite set, since there is only a finite number of possible size-change graphs of fixed arity. So χ is a finite colouring of $[\mathbb{N}]^2$. By Ramsey's Theorem, there is an infinite subset I of $\mathbb{N}$ such that $[I]^2$ is monochromatic under χ. Consider the sorted sequence $i_1, i_2, i_3, \ldots$ of elements of I. For each $j, k \in \mathbb{N}$ we have $\chi(i_j, i_k) = G_{\gamma_{i_j i_k}} = G$ for some G in $\mathcal{S}$.

So γ can be sectioned $\gamma_{1i_1}, \gamma_{i_1 i_2}, \gamma_{i_2 i_3}, \ldots$, where each section $\gamma_{i_j i_k}$ has composition $G_{\gamma_{i_j i_k}}$ which is G. But then also $G^2 = G$, because $G_{\gamma_{i_1 i_2}} G_{\gamma_{i_2 i_3}} = G^2 = G_{\gamma_{i_1 i_2}, \gamma_{i_2 i_3}} = G_{\gamma_{i_1 i_3}} = G$.

By assumption G has an arc $(n,>,n)$ and then by the remark of Definition 3.2.5 each section $\gamma_{i_j i_{j+1}}$ has a thread

$$(m_0, R_1, m_1), (m_1, R_2, m_2), \ldots, (m_{q-1}, R_q, m_q)$$

where $m_0 = n = m_q, q = i_k - i_j$, and at least one of $R_1, \ldots, R_q$ equals $>$, because $R_1 R_2 \ldots R_q = >$. Hence γ is infinitely descending.

For the reverse implication, assume $G = (V, V, E) \in \mathcal{S}$, and $G = G^2$, where there is no arc of the form $(v, >, v)$ in G. Now suppose that every infinite path of $\Gamma_{\mathtt{p}}$ is infinitely descending, and from this deduce a contradiction.

We have $G \in \mathcal{S}$, so G is a composition G_γ of some finite path $\gamma = G_1, G_2, G_3, \ldots, G_q$ in $\Gamma_{\mathtt{p}}$. The left hand side set of vertices in G_1 which is V, is the same as the right hand side set of vertices in G_q, since $G = (V, V, E)$.

Consider the infinite path $\gamma^\omega = \gamma, \gamma, \gamma, \ldots$. By assumption γ^ω is infinitely descending, and by Definition 3.2.4, γ^ω contains an infinitely descending thread $\nu = (m_0, R_1, m_1), (m_1, R_2, m_2), (m_2, R_3, m_3), \ldots$, starting in some arc of γ, then passing through γ infinitely often.

It is clear that ν passes through arcs of G_1 infinitely often. By the pigeon hole principle there must be an infinite subset of these arcs $(m_{i_j}, R_{i_j+1}, m_{i_j+1})$ of G_1, such that $m_{i_j} = n$ for some $n \in V$, because V is a finite set.

Let us choose a finite section ν' of ν as follows: Choose the first arc of ν' to be $(m_{i_0}, R_{i_0+1}, m_{i_0+1})$. Let ν' be long enough to contain an arc $(m_l, >, m_{l+1})$, where $l \geq i_0$. Such an arc must exist because ν is infinitely descending. Choose the last arc of ν' to be an arc $(m_{i_t-1}, R_{i_t}, m_{i_t})$ of G_q (which is the last size-change graph in γ).

Hence ν' is a thread of some power γ^k of the same length kq. Hence $(m_{i_0}, >, m_{i_t}) = (n, >, n)$ must be an arc of G_{γ^k}. But $G_{\gamma^k} = G^k$, and $G = G^k$, because $G = G^2$. So the arc $(n, >, n)$ must belong to G, which contradicts the assumption that G has no such arc. Hence all the infinite paths of $\Gamma_{\mathtt{p}}$ cannot be infinitely descending. $\square$

5 Remark on criterion by Jones et al.

The termination criterion presented by [Jones 00] is equivalent of the criterion presented in this paper. In [Jones 00] the graph G^+ denotes the union of arcs from all distinct powers of a size-change graph G. In our terminology their main theorem is:

All the infinite paths in $\Gamma_{\mathtt{p}}$ are infinitely descending if and only if for all $G \in \mathcal{S}$ such that G is self-composable there is an arc $(i, >, i)$ in G^+.

The latter is equivalent with the assertion: for all $G \in \mathcal{S}$ such that G is self-composable , there is n, such that G^n has an arc $(i, >, i)$. It follows from the definition of G^+, and we have simplified this by the statement that all G such that $G = G^2$ has an arc $(i, >, i)$. This eliminates the need of defining G^+ and its generation in the termination detection algorithm. We give a direct proof of equivalence between these two results:

Theorem 5.0.2 *For all $G \in \mathcal{S}$ such that $G^2 = G$, there is an arc $(i, >, i)$ in G if and only if, for all $G \in \mathcal{S}$, such that G is self-composable, there is n, such that G^n has an arc $(i, >, i)$.*

Proof: Assume $((\forall G \in \mathcal{S}$ such that $G^2 = G).(i, >, i) \in G)$.

Consider $H = (V, V, E) \in \mathcal{S}$. Let $Q = H^m$ such that $H^m = (H^m)^2$, $m \geq 1$. Such an m must exist because of Lemma 5.0.1 below. By assumption, since $Q = Q^2$, $(i, >, i)$ belongs to Q. So we have m such that H^m contains the arc $(i, >, i)$.

Assume now $\forall G \in \mathcal{S}$ such that G is self-composable, there is an $(i, >, i)$ in some power G^n. Consider $H \in \mathcal{S}$, such that $H^2 = H$. We know by assumption that there is m such that H^m has an arc $(i, >, i)$. But $H^m = H$, because $H = H^2$. So H has an arc $(i, >, i)$. $\square$

Lemma 5.0.1 *For any size-change graph G composable with itself, there is a natural number n, such that $(G^n)^2 = G^n$.*

Proof: Since there is a finite number of size-change graphs for fixed arities, there must exist $i < j$ such that $G^i = G^j$.

Then $G^i = G^{i+l}$ for $j - i = l > 0$. We have $G^i = G^{i+kl}$ for any k, m, and hence $G^{i+m} = G^{i+m+kl}$ for any k, m. We can choose k, m such that $i + m + kl = 2(i + m)$. Then $G^{i+m} = (G^{i+m})^2$. $\square$

6 Conclusions

This algorithm appears to be both simpler and more powerful than the algorithm presented in [Abel 98]. One potential problem was efficiency. Although it it PSPACE-hard in the size of the program (see [Jones 00]), the worst cases arise when the size-change graphs contain many arcs with many swapped parameters. It seems that the algorithm is efficient enough for small programs with simple call structure. Functions with a large number of parameter values are rarely used, so the average time and space complexity seem to be quite good.

7 Future work

As termination analysis for Agda is the main motivation for our study of this algorithm, a task to be undertaken is the implementation of this system for Agda. First the algorithm should be generalised to fit a language with general data types. This can be achieved by using disjunct sums to distinguish constructors and Cartesian products for the arities of the constructors.

Furthermore higher order functions would be desirable to cover by the algorithm. We do not yet know how this should be accomplished. One possibility is that some higher order functions, like the “map”-function, for instance, can be transformed to first order form and then be checked for termination.

Another topic could be to use the method described in this paper in the “reverse direction” looking for infinitely increasing sequences, to ensure *productivity* for infinite structures. We don’t know if this is a good idea, but it may be worth investigating. See [Giménez 95] for more reading about these issues.

A A small implementation

This section gives an example showing that the algorithm we present can be easily implemented in a straight-forward manner. In this work focus is not put on this implementation, but on the proof and understanding of the algorithm.

A.1 Main module

```
module Checker where
import Asyntax
import Parser
import Prettylatex
import Scgraph
import Callgraph
import Termcheck
import Examples



-- derive the set S:
finite_compos p = let gs = callgraph$parse p
                      in s gs

-- derive the set of size-change graphs:
graphs = callgraph.parse

-- derive all the self-composable graphs from S:
selfG p = let gs = graphs p
              in [ g | g@(G f f' _ _) <- s gs, f == f']

-- finds loops that may cause non-termination:
check p =  let gs = graphs p
               non_sat = map showPath (find_nonterm gs)
               in if null non_sat then putStr "Ok\n"
                                  else putStr(
                                        "Possible termination problems:\n" ++
                                        show non_sat)
```

A.2 Termination detection module

```
module Termcheck where
```

```
import List
import Scgraph


--                        Some auxiliary functions

-- make a graph comparable
-- if its arcs are sorted and unique,
-- a size-change graph can be compared for equality
normaliseG :: Scgraph -> Scgraph
normaliseG (G f g arcs path) = G f g (nub(sort arcs)) path


-- make all the graphs in a list comparable
normalise :: [Scgraph] -> [Scgraph]
normalise = map normaliseG


-- all possible compositions between graphs in gs1 and gs2
allcomps ::  [Scgraph] -> [Scgraph] -> [Scgraph]
allcomps gs1 gs2 =
         [composG g1 g2 | g1@(G _ g _ _)<-gs1, g2@(G g' _ _ _)<-gs2, g == g']


-- fixedpoint iteration:
-- add elements to a set emerging from s0 with f on the last change.
-- s0 U f(s0) U f(f(s0)) U ... U (f^n)(s0)
-- such that (f^(n-1))(s0) equals (f^n)(s0).
-- To use ++ below is like `union` because new and acc has no common elements
fixedpoint :: Eq a => ([a] -> [a]) -> [a] -> [a]
fixedpoint f s0 = let af acc last = case nub(f last) \\ acc of
                                    [] -> acc
                                    new -> af (new++acc) new
                      in af s0 s0



--                          Compute S
--            i.e. all the distinct finite compositions
--              of size-change graphs in the program.
s :: [Scgraph] -> [Scgraph]
s gs = let gs' = (nub.normalise) gs
           in fixedpoint (normalise.allcomps gs') gs'


--                   Detect size-change termination

-- If all G=(V,V,E) in S where G^2 = G has an arc (n,>,n), then p terminates.
```

```
-- result is True if size-change termination is detected, False otherwise
check_idempot :: [Scgraph] -> Bool
check_idempot gs = let desc_param (n,rel,n') = rel == Gt && n==n'
                       has_desc_param = any desc_param
                       in
                       all has_desc_param
                        [ as | g@(G f f' as _)<-s gs,
                        f==f', g == normaliseG (composG g g) ]
```

```
-- finds the compositions that may cause non-termination
find_nonterm :: [Scgraph] -> [Scgraph]
find_nonterm gs = let desc_param (n,rel,n') = rel == Gt && n==n'
                      has_desc_param = any desc_param
                      in
                      filter (\ (G _ _ as _) -> (not(has_desc_param as)))
                             [ g | g@(G f f' as _)<-s gs,
                               f==f', g == normaliseG (composG g g) ]
```

```
--  Alternative termination criterion of [Jones 00], using G+

--        Compute G+ (Neil Jones way, see [Jones 00] in thesis)

-- create G+ by adding arcs from powers of g until no new arcs are added
-- we don't take the path of calls into account here
gplus :: Scgraph -> Scgraph
gplus (G f f' as path) = G f f' (fixedpoint (composAs as) as) path
```

```
-- result is True if size-change termination is detected, False otherwise
-- If for all G=(V,V,E) in S
--    there exists m such that G^m has an arc (n,>,n)
-- then the program terminates on all input.
checkGplus :: [Scgraph] -> Bool
checkGplus gs = let arcs (G _ _ as _) = as
                    desc_param (n,rel,n') = rel == Gt && n==n'
                    has_desc_param = any desc_param
                    in
                    all has_desc_param
                     [arcs (gplus g) | g@(G f f' as _)<-s gs, f==f' ]
```

A.3 Abstract syntax

```
module Asyntax where
```

```
data Program = Prog FDs
               deriving (Show,Eq)

type FDs = [FD]

type Varname = (String,Int)

type Funname = String

data FD = FD Funname [Varname] Exp
           deriving (Show,Eq)

data Exp = Var Varname
         | Z
         | S Exp
         | App Funname [Exp]
         | Case Exp Exp Exp
            deriving (Show,Eq)


showVar (x,k) = x ++ replicate k '\''


instance Ord Exp where
   (>=)  = compare' (==)
   (>)   = compare'  (>)

compare' rel (Var(xs,ks)) t =
  maybe False (\(l,(xt,kt)) -> xs == xt && (kt - ks) `rel`  l) (getInfo t)

compare' _ _ _ = False

getInfo (S t) = maybe Nothing (\(l,xk) -> Just(1+l,xk)) (getInfo t)

getInfo (Var xk) = Just(0,xk)

getInfo _ = Nothing
```

A.4 Size-change graph data type

```
module Scgraph where

type Param = Int

data R = Gt
       | Gte
         deriving (Eq, Ord)
```

```
instance Show R where {
show r = case r of
              Gt -> ">"
              Gte -> ">="}


type Arc = (Param, R, Param)

type Funid = String

type Arcs = [Arc]

-- calls are given numbers
-- as they appear in the program, left-right, top-down.
--
type Callid = Int

-- A sequence of such numbers will identify
-- loops that may cause non-termination
--
type Path = [Callid] -- to trace the way a graph is composed

data Scgraph = G Funid Funid Arcs Path

-- Two graphs are equal iff
-- they have the same sets of vertices and the same arcs.
-- We assume the lists of arcs to be sorted, thereby comparable with (==).
-- Their paths are not compared.
--
instance Eq Scgraph where {
(==) (G f f' as _) (G g g' as' _) = f == g && f' == g' && as == as' }

instance Show Scgraph where { -- we don't show their paths by default
show (G f f' as _) = ' ':f++'-':f'++':':show as }

showPath g@(G _ _ _ path) = show g ++ "*" ++ show path ++ "*"


--                          Composition of graphs

composG :: Scgraph -> Scgraph -> Scgraph
composG (G f g as cs) (G g' h as' cs') | g == g' =
                                        G f h (composAs as as') (cs ++ cs')
composG (G f g _ _) (G g' h _ _) =
                              error ("incompatible size-chance graphs:\n " ++
                                      f ++ " -> " ++ g ++
                                      " can't be composed with " ++
                                      g' ++ " -> " ++ h ++ "\n")
```

```
-- composes the arcs from two sets of arcs
-- duplicate arcs may be created if some of the arc sets are taken from
-- non-size-change graphs - eg if it is a union of arcs between graphs
-- as in G+.
composAs :: Arcs -> Arcs -> Arcs
composAs xs ys =
         [(s, r `composR` r', t') | (s,r,t)<-xs, (s',r',t')<-ys, t==s' ]


composR :: R -> R -> R
composR Gte Gte = Gte
composR Gte Gt  = Gt
composR Gt  Gte = Gt
composR Gt  Gt  = Gt
```

A.5 Extraction of size-change graphs and building callgraph

```
{--
               Make the callgraph by adding a size-change graph
                                for each call.
               For a call f_j(t_1,...,t_m) for f_i(x_1,...,x_n)
               x > t if  x=x_i and t=S^l(x_i^(k)), where k > l.
     or rather x > t if  x=x_i^(k1) and t=S^l(x_i^(k2)), where k2 - k1 > l.
--}
module Callgraph where

import Scgraph
import Asyntax
import Parser


-- Calls are given numbers when collected
-- as they appear in the program, left-right, top-down.
-- A sequence of such numbers will identify
-- loops that may cause non-termination.


-- collect the size-change graphs of the calls and enumerate them
callgraph p = zipWith enumerate (graphsP p) [1..]
              where enumerate (G f f' as _) n = G f f' as [n]


-- The graphs in a program
graphsP (Prog fds) = concat(map graphsFD fds)


-- The graphs in a function definition
```

```
graphsFD (FD fi xs e_fi) =
         [get_size_changes fi xs fj ts | App fj ts <- calls e_fi ]


-- collecting calls occurring in expressions:

-- calls: given e it gives a list of calls in e
calls :: Exp -> [Exp]

calls (S t) = calls t

calls (Case _ e1 e2) = calls e1 ++ calls e2

calls c@(App _ ts) = c:concat(map calls ts)

calls _ = []

-- we create the graph without giving its path
get_size_changes fi xs fj ts = G fi fj arcs []
                       where enum_form_params = zip (map Var xs) [1..]
                             enum_act_params  = zip ts [1..]
                             arcs =
                               ([(i,Gte,j)|
                                 (x,i) <- enum_form_params,
                                 (t,j) <- enum_act_params,   x >= t ] ++
                                [(i,Gt,j)|
                                 (x,i) <- enum_form_params,
                                 (t,j) <- enum_act_params,   x  > t ])

-- Syntactical size-change x > t defined in Asyntax.hs.
-- See end of section 2.1 in the thesis.
```

A.6 Example programs

```
module Examples where

import Scgraph

--                          Examples

ack = "ack(x1,x2) = case x1 of                                            \
      \                 0 -> S(x2)                                        \
      \                 S(x1') -> case x2 of                              \
      \                               0 -> ack(x1',S(0))                  \
      \                               S(x2') -> ack(x1',ack(x1,x2'))     "
```

```
add = "add(x1,x2) = case x1 of                                          \
      \                 0 -> x2                                          \
      \                 S(x1') -> S(add(x2,x1'))                         "


-- example from Andreas Abel 3.12:
fgh =
 " f(x1,x2) = case x1 of                                                 \
 \                0 -> 0                                                 \
 \                S(x1') -> case x2 of                                   \
 \                              0 -> 0                                   \
 \                              S(x2') -> h(g(x1',x2),f(S(S(x1)),x2'))   \
 \ g(x1,x2) = case x1 of                                                 \
 \                0 -> 0                                                 \
 \                S(x1') -> case x2 of                                   \
 \                              0 -> 0                                   \
 \                              S(x2') -> h(f(x1,x2),g(x1',S(x2)))       \
 \ h(x1,x2) = case x1 of                                                 \
 \                0 ->(case x2 of                                        \
 \                        0 -> 0                                         \
 \                        S(x2') -> h(x1,x2') )                          \
 \                S(x1') -> h(x1',x2)                                    "

{--
Example runs:

The size-change graphs of the program consisting of f, g and h:
Checker> map showPath$graphs fgh [" f-h:[]*[1]*","
f-g:[(2,>=,2),(1,>,1)]*[2]*"," f-f:[(2,>,2)]*[3]*"," g-h:[]*[4]*","
g-f:[(1,>=,1),(2,>=,2)]*[5]*"," g-g:[(1,>,1)]*[6]*","
h-h:[(1,>=,1),(2,>,2)]*[7]*"," h-h:[(2,>=,2),(1,>,1)]*[8]*"]


The set S: Checker> map showPath$finite_compos fgh ["
f-f:[]*[2,6,5,3]*"," g-g:[]*[5,3,2,6]*"," f-f:[(1,>,1)]*[2,6,5]*","
f-g:[]*[3,2,6]*"," g-f:[(1,>,1),(2,>=,2)]*[5,2,5]*","
g-g:[(2,>,2)]*[5,3,2]*"," g-f:[]*[6,5,3]*","
f-f:[(1,>,1),(2,>=,2)]*[2,5]*"," f-g:[(1,>,1)]*[2,6]*","
f-g:[(2,>,2)]*[3,2]*"," g-g:[(1,>,1),(2,>=,2)]*[5,2]*","
g-f:[(2,>,2)]*[5,3]*"," g-f:[(1,>,1)]*[6,5]*","
h-h:[(1,>,1),(2,>,2)]*[7,8]*"," f-h:[]*[1]*","
f-g:[(1,>,1),(2,>=,2)]*[2]*"," f-f:[(2,>,2)]*[3]*"," g-h:[]*[4]*","
g-f:[(1,>=,1),(2,>=,2)]*[5]*"," g-g:[(1,>,1)]*[6]*","
h-h:[(1,>=,1),(2,>,2)]*[7]*"," h-h:[(1,>,1),(2,>=,2)]*[8]*"]

Finding possible non-termination:
Checker> check fgh
Possible termination problems:
[" f-f:[]*[2,6,5,3]*"," g-g:[]*[5,3,2,6]*"]
```

```
--}


{-- 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.


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 as given in [Jones 00] and the same call graph as the example
above would give:

--}

boolprog =
 " f0(x1,x2,x3,x4,x5)=case x1 of                                        \
  \                        0->0                                         \
  \                        S(x1')->case x4 of                           \
  \                                     0 -> 0                          \
  \                                     S(x4')->f1(x2',x2,x4',x4,S(S(x5))) \
  \                                                                     \
  \ f1(x1,x2,x3,x4,x5)=case x5 of                                       \
  \                        0->0                                         \
  \                        S(x5')->f2(x2,x1,x3,x4,x5')                  \
  \                                                                     \
  \ f2(x1,x2,x3,x4,x5)=                                                 \
  \   case x3 of                                                        \
  \        0->0                                                         \
  \        S(x3')->case x4 of                                           \
  \                     0 -> (case x5 of                                \
```

```
 \                                0 -> 0                                          \
 \                                S(x5')-> f3(x1,x2,x3',x4,x5') )                 \
 \                     S(x4')-> case x5 of                                        \
 \                                   0->0                                         \
 \                                   S(x5') -> f5(x1,x2,x3,x4',x5')               \
 \                                                                                \
 \ f3(x1,x2,x3,x4,x5)=case x5 of                                                  \
 \                         0->0                                                   \
 \                         S(x5')->f4(x1,x2,x4,x3,x5')                            \
 \                                                                                \
 \ f4(x1,x2,x3,x4,x5)=                                                            \
 \   case x1 of                                                                   \
 \        0->0                                                                    \
 \        S(x1')->case x2 of                                                      \
 \                     0 -> (case x5 of                                           \
 \                                0 -> 0                                          \
 \                                S(x5')-> f3(x1',x2,x3,x4,x5') )                 \
 \                     S(x2')-> case x5 of                                        \
 \                                   0->0                                         \
 \                                   S(x5') -> f2(x1,x2',x3,x4,x5')               \
 \                                                                                \
 \ f5(x1,x2,x3,x4,x5)=case x5 of                                                  \
 \                         0->0                                                   \
 \                         S(x5')->f6(x2,x1,x3,x4,x5')                            \
 \                                                                                \
 \ f6(x1,x2,x3,x4,x5)=case x5 of                                                  \
 \                         0->0                                                   \
 \                         S(x5')->f0(x1,x2,x4,x3,x5')                            \
 \                                                                                "
```

```
{-- Sample run:

Main>  length$s$graphs boolprog
565

                                             18.6 seconds

Main> map showPath$find_nonterm$graphs boolprog ["
f0-f0:[(2,>,1),(2,>=,2),(4,>,3),(4,>=,4)]*[1,2,3,5,7,4,8,9]*","
f1-f1:[(2,>,1),(2,>=,2),(4,>,3),(4,>=,4)]*[2,3,5,7,4,8,9,1]*","
f2-f2:[(1,>,2),(1,>=,1),(4,>,3),(4,>=,4)]*[3,5,7,4,8,9,1,2]*","
f2-f2:[(1,>,2),(1,>=,1),(3,>,4),(3,>=,3)]*[4,8,9,1,2,3,5,7]*","
f3-f3:[(1,>,2),(1,>=,1),(4,>,3),(4,>=,4)]*[5,7,4,8,9,1,2,3]*","
f4-f4:[(1,>,2),(1,>=,1),(3,>,4),(3,>=,3)]*[7,4,8,9,1,2,3,5]*","
f5-f5:[(1,>,2),(1,>=,1),(3,>,4),(3,>=,3)]*[8,9,1,2,3,5,7,4]*","
```

```
f6-f6:[(2,>,1),(2,>=,2),(3,>,4),(3,>=,3)]*[9,1,2,3,5,7,4,8]*"]

                                             18.8 seconds

Main>  checkGplus$graphs boolprog
False

                                             18.2 seconds

Main>  check_idempot$graphs boolprog

False

                                             18.3 seconds

--}


-- Some more examples

{--

Ex 6 from [Jones 00]:
Program with late-starting sequence of decreasing parameter values:

f(a,b) = if b = [] then g(a,[]) else f(hd b:a, tl b)
g(c,d) = if c = [] then d else g(tl c, hd c:d)

--}

--  in this language:
ex6 = " f(x1,x2) = case x2 of                 \
      \                0 -> g(x1,0)           \
      \                S(x2') -> f(S(x1),x2') \
      \ g(x1,x2) = case x1 of                 \
      \                0 -> x2                \
      \                S(x1') -> g(x1',S(x2)) "

{- Sample run:

Checker> check ex6
Ok

 All finite compositions:

Checker> map showPath$finite_compos ex6
[" f-g:[(1,>,1)]*[1,3]*"," f-g:[]*[2,1]*",
" f-g:[(1,>=,1)]*[1]*"," f-f:[(2,>,2)]*[2]*"," g-g:[(1,>,1)]*[3]*"]
```

```
-}


-- simple example of swapped parameters
permut = "f(x1,x2)=case x1 of 0->0            \
         \                 S(x1) -> f(x2,x1') "

{- Sample run: the program terminates - this can be seen on the graphs:
Checker> graphs permut
[ f-f:[(2,>=,1),(1,>,2)]]
Checker> finite_compos permut
[ f-f:[(1,>,2),(2,>,1)], f-f:[(1,>,1),(2,>,2)], f-f:[(1,>,2),(2,>=,1)]]
Checker> map showPath$finite_compos permut

[" f-f:[(1,>,2),(2,>,1)]*[1,1,1]*"," f-f:[(1,>,1),(2,>,2)]*[1,1]*","
f-f:[(1,>,2),(2,>=,1)]*[1]*"]
-}

-- equality test
eq = "eq(x1,x2) = case x1 of                              \
     \                0 -> case x2 of                     \
     \                          0 -> S(0)                 \
     \                          S(x2') -> 0               \
     \                S(x1') -> case x2 of                \
     \                               0 -> 0               \
     \                               S(x2') -> eq(x1',x2')"

-- odd / even function
odd_even = "  even(x1) = case x1 of                    \
           \                 0 -> S(0)                 \
           \                 S(x1') -> odd(x1')        \
           \  odd(x1) = case x1 of                     \
           \                0 -> 0                     \
           \                S(x1') -> even(x1')        "


-- example of nested case on one variable:
div2 = " div2(x1) = case x1 of                                  \
       \                0 -> 0                                  \
       \                S(x1') -> case x1' of                   \
       \                              0 -> 0                    \
       \                              S(x1'') -> S(div2(x1'')) "

{-
Checker> finite_compos div2
[ div2-div2:[(1,>,1)]]
-}
```

References

[Jones 00] Neil D. Jones, Amir Ben-Amram, Chin Soon Lee. *Termination Analysis by Size Change Graphs.* Draft applet from DIKU Copenhagen, January 2000

[Graham 80] R.R.Graham, B.L.Rotchild, J.H.Spencer. *Ramsey Theory.* John Wiley, New York 1980 ISBN 0-471-05997-8

[Abel 98] Andreas Abel. "`foetus` - *Termination Checker for Simple Functional Programs*", Theoretical Computer Science, Ludwigs-Maximilians-University Munich, July 1998.

[Sagiv 91] Yehoshua Sagiv. *A termination test for logic programs.* In Vijay Saraswat and Kazunori Ueda, editors, Logic programming, Proceedings of the 1991 International Symposium, San Diego, California, USA, Oct 28-Nov 1, 1991, pages 518-532. MIT Press, 1991.

[Agda] Catarina Coquand. *The Agda Home page.* URL: `http://www.cs.chalmers.se/~catarina/agda`, Department of Computer Science, Chalmers University of Technology and Göteborgs Universitet, 2000.

[Giménez 95] Giménez, E. *Codifying guarded definitions with recursive schemes.* In Dybjer, P., Nordström, B., & Smith, J. (editors) *Types for proofs and programs (types '94). Lecture Notes in Computer Science, vol 996. Springer-Verlag. International Workshop, TYPES '94 held in June 1994.*