THE SIZE-CHANGE TERMINATION PRINCIPLE FOR CONSTRUCTOR BASED LANGUAGES

PIERRE HYVERNAT

Laboratoire de Mathématiques, CNRS UMR 5126 – Université de Savoie, 73376 Le Bourget-du-Lac Cedex, France
e-mail address: pierre.hyvernat@univ-savoie.fr
URL: `http://lama.univ-savoie.fr/~hyvernat/`

Abstract. This paper describes an automatic *termination checker* for a generic first-order call-by-value language in ML style. We use the fact that values are built from constructors and tuples to keep some information about how arguments of recursive calls evolve during evaluation.

The result is a criterion for termination extending the *size-change termination principle* of Lee, Jones and Ben-Amram that can detect size changes inside subvalues of arguments. Moreover the corresponding algorithm is easy to implement, making it a good candidate for experimentation.

Introduction

Our goal is to automatically check the termination of mutually recursive definitions written in a first-order call-by-value language in ML style. The problem is of course undecidable and we can only hope to capture *some* recursive definitions. Lee, Jones and Ben-Amram's size-change termination principle (SCT) is a simple, yet surprisingly strong sufficient condition for termination of programs [6]. It relies on a notion of *size* of values and a static analysis interpreting a recursive program as a *control-flow graph* with information about how the size of arguments evolves during recursive calls. The procedure checking that such a graph is "terminating" amounts to a (conceptually) simple construction of a graph of paths.

We specialize and extend this principle to an ML-like language where first-order values have a specific shape: they are built with n-tuples and constructors. It is then possible to record more information about arguments of recursive calls than "decreases strictly" or "decreases". The main requirement is that the set of possible informations is finite, which we get by choosing bounds for the *depth* and the *weight* of the terms describing this information. We obtain a parametrized criterion for checking termination of first-order recursive programs. The weakest version of this criterion corresponds to the original SCT where the size of a value is its depth. An important point is that because we know some of the constructors present in the arguments, it is possible to ignore some paths in the

Received by the editors December 23, 2013.
1998 ACM Subject Classification: D.3.1, F.3.2, D.2.4.
Key words and phrases: program analysis, termination analysis, size-change principle, ML.

control-flow graph because they cannot correspond to real evaluation steps. Moreover, it makes it possible to inspect *subvalues* of the arguments and detect a "local" size change. Another important point is that there is a simple syntax directed static analysis that can be done in linear time.

The criterion has been implemented as part of the PML [12] language, where it plays a central role: PML has a notion of *proofs*, which are special programs that need to terminate. As far as usability is concerned, this criterion was a success: it is strong enough for our purpose, its output is usually easy to predict and its implementation was rather straightforward. The core consists of about 600 lines of OCaml code without external dependencies.[1]

The paper is organized as follows: after introducing the ambient programming language and some paradigmatic examples, we first define an abstract interpretation for *calls* and look at their properties. This makes it possible to give an abstract interpretation for sets of recursive definitions as control-flow graphs. A subtle issue arises when we try to make the set of possible interpretations finite, making the notion of composition not associative in general. We then describe and prove the actual criterion. We finish with an appendix giving some technical lemmas, details about the implementation and a simple static analysis.

Comparison with other work. Two aspects of this new criterion appeared in the literature [2, 7] (see Section 2.5), but what seems to be new here is that the algorithm for testing termination is, like for the original SCT, "finitary". Once the static analysis is done —and this can be as simple as a linear-time syntactical analysis of the definitions— one needs only to compute the graph of paths of the control-flow graph and inspect its loops. This makes it particularly easy to implement from scratch as it needs not to rely on external automatic proof-checker [7] or integer linear programming libraries [2].

One advantage of this minimalistic approach is that a formal proof of the criterion is probably easier, making the criterion well-suited for proof assistants based on type theory like Coq [9] or Agda [11]. The closest existing criterion seems to be the termination checker of Agda. It is based on the "`foetus`" termination checker [1]. The implementation incorporates a part of SCT but unfortunately, the exact criterion isn't formally described anywhere.

It should be noted that native datatypes (integers with arithmetic operations for example) are not addressed in this paper. This is not a problem as proof assistants don't directly use native types. Complementing the present approach with such internal datatypes and analysis of higher-order programs [13] is the subject of future research.

Ambient Programming Language. The programming language we are considering is a first-order call-by-value language in ML-style. It has constructors, pattern-matching, tuples and projections. The language is described briefly in Figure 1 and the syntax should be obvious to anyone familiar with an ML-style language. The "`match`" construction allows to do pattern matching, while "π_i" is used for projecting a tuple on one of its components. The only proviso is that all constructors are unary and written as "`C[`u`]`". Note that the `f` in the grammar for expressions can either be one of the functions that are being inductively defined, or any function in the global environment. Other features like `let` expressions,

[1]A standalone version is available from `http://lama.univ-savoie.fr/~hyvernat/research.php`

$$
\begin{array}{rcl}
\textit{program} & ::= & \texttt{val rec}\ \ \textit{def}\ \ (\texttt{and}\ \ \textit{def})^* \\
\textit{def} & ::= & \texttt{f}\ \texttt{x}_1\ \texttt{x}_2\ \ldots\ \texttt{x}_n\ \texttt{=}\ \textit{term} \\
\textit{expr} & ::= & \texttt{x}_k \ \mid\ \texttt{f} \ \mid\ \textit{expr}\ \textit{expr}^+ \ \mid \\
 & & \texttt{C[}\textit{expr}\texttt{]} \ \mid\ \texttt{(}\textit{expr}, \ldots, \textit{expr}\texttt{)} \ \mid \\
 & & \texttt{match}\ \textit{expr}\ \texttt{with}\ \textit{branch}^+ \ \mid\ \pi_i\ \textit{expr} \quad \text{(with } i > 0) \\
\textit{branch} & ::= & \texttt{| C[x}_k\texttt{] ->}\ \textit{expr} \ \mid\ \texttt{| \_ ->}\ \textit{expr}
\end{array}
$$

Figure 1: syntax of the programming language

exceptions, (sub)typing etc. can easily be added as they don't interfere with the criterion. (They might make the static analysis harder though.)

The operational semantics is the usual one and we only consider programs whose semantics is well defined. This can be achieved using traditional Hindley-Milner type checking / type inference [10] or a *constraint checking* algorithm [12] ensuring that

- a constructor is never projected,
- a tuple is never matched,
- an n-tuple is only projected on its i-th component if $1 \leq i \leq n$.

To simplify the presentation, we assume that functions have an arity and are always fully applied. Moreover, we suppose that the arguments of functions are all first-order values. These constraints are relaxed in the actual implementation.

An important property of this language is that non-termination can only be the result of evaluation going through an infinite sequence of calls to recursive functions [12]. A consequence of that is that it is not possible to use the notions described in this paper directly for languages where a fixed point combinator can be defined without recursion. Extensions similar to the work of Jones and Bohr for untyped languages [3] might be possible, at the cost of a greatly increased complexity of implementation.

A *first-order value* is a closed expression built only with constructors and (possibly empty) tuples. Examples include unary natural numbers built with constructors "`Z`" and "`S`" or lists built with constructors "`Nil`" and "`Cons`". The *depth* of a value is

$$
\begin{aligned}
\mathrm{depth}(\texttt{C[}u\texttt{]}) &\overset{\mathsf{def}}{=} 1 + \mathrm{depth}(u) \\
\mathrm{depth}\big((u_1, \ldots, u_n)\big) &\overset{\mathsf{def}}{=} \max_{1 \leqslant i \leqslant n} \big(1 + \mathrm{depth}(u_i)\big) .
\end{aligned}
$$

Note that values are not explicitly typed and that depth counts all constructors. For example, the depth of a list of natural numbers counts the `Nil`, `Cons`, `S` and `Z` constructors, as well as the tuples coming with the `Cons` constructors.

To make examples easier to read, we will deviate from the grammar of Figure 1 and use ML-like deep pattern-matching, including pattern-matching on tuples. Moreover, parenthesis around tuples will be omitted when they are the argument of constructors. For example, here is how we write the usual `map` function:

```
val rec map x = match x with Nil[]  ->  Nil[]
                           | Cons[a,y]  ->  Cons[f a, map y]
```

Without the previous conventions, the definition would look like

```
val rec map x = match x with | Nil[y]  ->  Nil[()]
                             | Cons[y]  ->  Cons[(f π₁y, map π₂y)]
```

Note that because we here restrict to first-order arguments, we cannot formally make `f` an argument of `map`. We thus assume that it is a predefined function. This constraint is relaxed in the actual implementation.

Vocabulary and notation. We use a fixed-width font, possibly with subscripts, for syntactical tokens: "`x`", "`y`" or "$\mathtt{x}_i$" for variables, "`f`" or "`g`" for function names, "`A`" for a constructor, etc. The only exception will be the letter π, used to represent a projection. Meta variables representing terms will be written with italics: "t", "u" or "t_i" etc.

For a set of mutual recursive definitions

```
  val rec f x1 x2 x3 = ... g t1 t2 ...
and       g y1 y2    = ...
```

where $\mathtt{x}_1$, $\mathtt{x}_2$, $\mathtt{x}_3$, $\mathtt{y}_1$ and $\mathtt{y}_2$ are variables and t_1 and t_2 are expressions,

- "$\mathtt{x}_1$", "$\mathtt{x}_2$" and "$\mathtt{x}_3$" are the *parameters of the definition of* `f`,
- "`g` t_1 t_2" is a *call site from* `f` *to* `g`,
- "t_1" and "t_2" are the *arguments of* `g` *at this call site.*

We usually abbreviate those to *parameters*, *call* and *arguments.*

Examples. Here are some examples of recursive (ad-hoc) definitions that are accepted by our criterion.

- All the structurally decreasing inductive functions, like the `map` function given previously are accepted.
- Our criterion generalizes the original SCT (where the size of a value is its depth), and thus, all the original examples [6] pass the test. For example, the Ackermann function is accepted:

```
val rec ack x1 x2 = match (x1,x2) with
                        (Z[],Z[]) -> S[Z[]]
                      | (Z[],S[n]) -> S[S[n]]
                      | (S[m],Z[]) -> ack m S[Z[]]
                      | (S[m],S[n]) -> ack m (ack S[m] n)
```

- In the original SCT, the size information is lost as soon as a value increases. We do support a local bounded increase of size as in

```
val rec f1 x = g1 A[x]
  and g1 x = match x with A[A[x]] -> f1 x
                        | _       -> ()
```

  The call from $\mathtt{f}_1$ to $\mathtt{g}_1$ (that increases the depth by 1) is harmless because it is followed by a call from $\mathtt{g}_1$ to $\mathtt{f}_1$ (that decreases the depth by 2).
- In the definition

```
val rec f2 x = match x with   A[x] -> f2 B[C[x]]
                            | B[x] -> f2 x
                            | C[x] -> f2 x
```

  the size of the argument increases in the first recursive call. This alone would make the definition non size-change terminating for the original SCT. However, the constructors and pattern matching imply that the first recursive call is necessarily followed by the second and third one, where the size decreases at last. This function passes the improved test.
- In the definition

```
val rec push_left x =
  match x with Leaf[] -> Leaf[]
             | Node[t, Leaf[]] -> Node[t, Leaf[]]
             | Node[t₁, Node[t₂,t₃]] -> push_left Node[Node[t₁,t₂],t₃]
```

the depth of the argument does not decrease but the depth of its right subtree does. In the original SCT, the user could choose the ad-hoc notion of size "depth of the right-subtree". Our criterion will see that this is terminating without help.

Idea of the Algorithm. Just like the original SCT, our algorithm works by making an abstract interpretation of the recursive definitions as a control-flow graph. This is done by a static analysis independent of the actual criterion. A simple, syntactical static analysis that allows to deal with the examples of the paper is described in Appendix C. This control-flow graph only represents the evolution of arguments of recursive calls. For example both the `map` function and the `last` function

```
val rec last x = match x with Cons[a,Nil[]] -> a
                            | Cons[_,x] -> last x
```

have the same control-flow graph: when the function is called on a non-empty list, it makes a recursive call to the tail of the list.

Ideally, each argument to a call should be represented by a transformation describing how the argument is obtained from the parameters of the defined function. To make the problem tractable, we restrict to transformations described by a simple term language. For example, the argument of the `map/last` functions is described by "π_2`Cons`$^-$`x`": starting from parameter `x`, we remove a `Cons` and take the second component of the resulting tuple. When a function has more than one parameter, each argument of the called function is described by a term with free variables among the parameter of the calling function.

Checking termination is done by finding a sufficient condition for the following property of the control-flow graph: *no infinite path of the graph may come from an infinite sequence of real calls.* The two main reasons for a path to not come from a sequence of real calls are:

- there is an incompatibility in the path: for example, it is not possible to remove a `Cons` from the `Nil` value,
- it would make the depth of some value negative: for example, it is not possible to remove infinitely many `Cons` from a given list.

In order to do that, we will identify loops that every infinite path must go through, and check that for all these "coherent" loops, there is some part of an argument that decreases strictly. For example, in the definition of **`push_left`** (page 5), the right subtree of the argument is decreasing, which makes the function pass the termination test.

1. Interpreting Calls

1.1. **Terms and Reduction.**

The next definition gives a way to describe how an argument of a recursive calls is obtained from the parameters of the calling function:

Definition 1.1. Representations for arguments are defined by the following grammar

$$t \in \mathcal{T} \quad ::= \quad \mathtt{x}_k \quad | \quad \underbrace{\mathtt{C}t \quad | \quad (t_1, \dots, t_n)}_{\text{constructors}} \quad | \quad \underbrace{\mathtt{C}^{-}t \quad | \quad \pi_i t}_{\text{destructors}} \quad | \quad t_1 + t_2 \quad | \quad \mathbf{0} \quad | \quad \langle w \rangle t$$

where $\mathtt{x}_k$ can be any variable of the ambient language, $n \geqslant 0$, $i \geqslant 1$ and $w \in \mathbf{Z}_\infty = \mathbf{Z} \cup \{\infty\}$. We write $\mathcal{T}(\mathtt{x}_1, \dots, \mathtt{x}_n)$ for the set of terms whose variables are in $\{\mathtt{x}_1, \dots, \mathtt{x}_n\}$.

We enforce linearity (or n-linearity for n-tuples) for all term formation operations with the following equations:

$$\begin{array}{rclrcl}
\mathtt{C}\mathbf{0} & = & \mathbf{0} & \mathtt{C}(t_1 + t_2) & = & \mathtt{C}t_1 + \mathtt{C}t_2 \\
(\dots, \mathbf{0}, \dots) & = & \mathbf{0} & (\dots, t_1 + t_2, \dots) & = & (\dots, t_1, \dots) + (\dots, t_2, \dots) \\
\mathtt{C}^{\text{-}}\mathbf{0} & = & \mathbf{0} & \mathtt{C}^{\text{-}}(t_1 + t_2) & = & \mathtt{C}^{\text{-}}t_1 + \mathtt{C}^{\text{-}}t_2 \\
\pi_i\mathbf{0} & = & \mathbf{0} & \pi_i(t_1 + t_2) & = & \pi_i t_1 + \pi_i t_2 \\
\langle w \rangle \mathbf{0} & = & \mathbf{0} & \langle w \rangle (t_1 + t_2) & = & \langle w \rangle t_1 + \langle w \rangle t_2\ .
\end{array}$$

We also quotient $\mathcal{T}$ by associativity, commutativity, neutrality of $\mathbf{0}$, *and idempotence* of $+$:

$$\begin{array}{rclrcl}
t_1 + (t_2 + t_3) & = & (t_1 + t_2) + t_3 & t + \mathbf{0} & = & t \\
t_1 + t_2 & = & t_2 + t_1 & t + t & = & t\ .
\end{array}$$

The intuition is that:

- $\mathtt{x}_k$ is a parameter of the calling function.
- $\mathtt{C}$ is a constructor and $(\_, \dots, \_)$ is a tuple.
- π_i is a projection. It gives access to the ith component of a tuple.
- $\mathtt{C}^{\text{-}}$ corresponds to a branch of pattern matching. It removes the $\mathtt{C}$ from a value.
- $\mathbf{0}$ is an artifact used to represent an error during evaluation. Since we only look at well defined programs (see remark on page 3), any $\mathbf{0}$ that appears during analysis can be ignored as it cannot come from an actual computation.
- $t_1 + t_2$ acts as a non-deterministic choice. Those sums will play a central role in our analysis of control-flow graphs.
- $\langle w \rangle$ stands for an unknown term that may increase the depth of its argument by at most w. For example, if $w < 0$, then the depth of $\langle w \rangle v$ is strictly less than the depth of v. Those terms will serve as *approximations* of other terms: for example, both $\mathtt{C}t$ and $\mathtt{D}t$ can be approximated by $\langle 1 \rangle t$, but each one contains strictly more information than $\langle 0 \rangle t$.

There is a natural notion of reduction on terms:

Definition 1.2. We define a reduction relation on $\mathcal{T}$:

$$\begin{array}{lrclrcl}
(1) & \mathtt{C}^{\text{-}}\mathtt{C}t & \to & t & \pi_i(t_1, \dots, t_n) & \to & t_i \quad \text{if } 1 \leqslant i \leqslant n \\
(2) & \langle w \rangle \mathtt{C}t & \to & \langle w+1 \rangle t & \langle w \rangle (t_1, \dots, t_n) & \to & \sum_{1 \leqslant i \leqslant n} \langle w+1 \rangle t_i \quad \text{if } n > 0 \\
(2) & \mathtt{C}^{\text{-}}\langle w \rangle t & \to & \langle w-1 \rangle t & \pi_i \langle w \rangle t & \to & \langle w-1 \rangle t \\
(2) & \langle w \rangle \langle v \rangle t & \to & \langle w+v \rangle t & & & \\
(3) & \pi_i \mathtt{C}t & \to & \mathbf{0} & \pi_i(t_1, \dots, t_n) & \to & \mathbf{0} \quad \text{if } i > n \\
(3) & \mathtt{C}^{\text{-}}(t_1, \dots, t_n) & \to & \mathbf{0} & \mathtt{C}^{\text{-}}\mathtt{D}t & \to & \mathbf{0} \quad \text{if } \mathtt{C} \neq \mathtt{D}
\end{array}$$

The symbol "+" for elements of $\mathbf{Z}_\infty$ denotes the obvious addition, with $\infty + \infty = \infty + n = \infty$.

This reduction extends the operational semantics of the ambient language: the two rules from group (1) correspond to the evaluation mechanism and the four rules from group (3) correspond to unreachable states of the evaluation machine. The five rules from group (2) explain how approximations behave. Note in particular that:

- a $\langle w \rangle$ absorbs constructors on its right and destructors on its left,
- a $\langle w \rangle$ may approximate some projections and we don't know which components of a tuple it may access. This is why a sum appears in the reduction.

Lemma 1.3. *The reduction $\to$ is strongly normalizing and confluent. We write* $\mathrm{nf}(t)$ *for the unique normal form of t.*

We write $t \approx u$ when t and u have the same normal form. This lemma implies that $\approx$ is the least equivalence relation containing reduction.

Proof of Lemma 1.3. Strong normalization is easy as the depth of terms decreases strictly during reduction. By Newman's lemma, confluence thus follows from local confluence which follows from examination of the critical pairs:

$$\begin{array}{ccc} \mathtt{D}^{\text{-}}\langle w\rangle \mathtt{C}t & \mathtt{D}^{\text{-}}\langle w\rangle(t_1,\dots,t_n) & \mathtt{C}^{\text{-}}\langle w\rangle\langle v\rangle t \\ \pi_i\langle w\rangle \mathtt{C}t & \pi_i\langle w\rangle(t_1,\dots,t_n) & \pi_1\langle w\rangle\langle v\rangle t \\ \langle w\rangle\langle v\rangle \mathtt{C}t & \langle w\rangle\langle v\rangle(t_1,\dots,t_n) & \langle w\rangle\langle v\rangle\langle u\rangle t\ . \end{array}$$

For example, $\mathtt{D}^{\text{-}}\langle w\rangle\mathtt{C}t$ reduces both to $\langle w-1\rangle\mathtt{C}t$ and $\mathtt{D}^{\text{-}}\langle w+1\rangle t$. Luckily, those two terms reduce to $\langle w\rangle t$. The same holds for the eight remaining critical pairs. □

Call a term $t \in \mathcal{T}$ *simple* if it is in normal form and doesn't contain $+$ or $\mathbf{0}$. We have:

Lemma 1.4. *Every term $t \in \mathcal{T}$ reduces to a (possibly empty) sum of simple terms, where the empty sum is identified with $\mathbf{0}$.*

Proof. This follows from the fact that all term constructions are linear and that the reduction is strongly normalizing. Note that because of confluence, associativity, commutativity and idempotence of $+$, this representation is essentially unique. □

Simple terms have a very constrained form: all the constructors are on the left and all the destructors are on the right. More precisely:

Lemma 1.5. *The simple terms of $\mathcal{T}$ are generated by the grammar*

$$\begin{array}{rcl} t & ::= & \mathtt{C}t \ \mid\ (t_1,\dots,t_n) \ \mid\ \overline{d} \ \mid\ \langle w\rangle\overline{d} \qquad (n>0) \\ \overline{d} & ::= & () \ \mid\ \overline{d_v} \\ \overline{d_v} & ::= & \mathtt{x}_k \ \mid\ \pi_i\overline{d_v} \ \mid\ \mathtt{C}^{\text{-}}\overline{d_v} \end{array}$$

We will sometimes write $\overline{d}\,\mathtt{x} = d_1\cdots d_n\mathtt{x}$ for some $\overline{d_v}$ ending with variable $\mathtt{x}$.
The length $|\overline{d}|$ of $\overline{d}$ is the number of destructors $\mathtt{C}^{\text{-}}/\pi_i$ it contains.

We now introduce a preorder describing approximation.

Definition 1.6. The relation $\preccurlyeq$ is the least preorder on $\mathcal{T}$ satisfying

- $\preccurlyeq$ is contextual: if t is a term, and if $u_1 \preccurlyeq u_2$, then $t[\mathtt{x} := u_1] \preccurlyeq t[\mathtt{x} := u_2]$,
- $\preccurlyeq$ is compatible with $\approx$: if $t \approx u$ then $u \preccurlyeq t$ and $t \preccurlyeq u$,
- $\preccurlyeq$ is compatible with $+$ and $\mathbf{0}$: $\mathbf{0} \preccurlyeq t$ and $t \preccurlyeq t+u$,
- if $v \leqslant w$ in $\mathbf{Z}_\infty$ then $\langle v\rangle t \preccurlyeq \langle w\rangle t$,
- $t \preccurlyeq \langle 0\rangle t$.

When $t \preccurlyeq u$, we say that "t is finer than u" or that "u is an approximation of t".

This definition implies for example that $\mathtt{C}\mathtt{x} \preccurlyeq \langle 0\rangle\mathtt{C}\mathtt{x} \approx \langle 1\rangle\mathtt{x}$, and thus, by contextuality, that $\mathtt{C}t \preccurlyeq \langle 1\rangle t$ for any term t. Appendix A gives a characterization of this preorder that is easier to implement because it doesn't use contextuality. It implies in particular the following lemma:

Lemma 1.7. *We have $\langle w\rangle\overline{d} \preccurlyeq \langle w'\rangle\overline{b}$ if and only $\overline{b}$ is a suffix of $\overline{d}$ and $w + |\overline{b}| \leqslant w' + |\overline{b}|$. In particular, $\langle w\rangle() \preccurlyeq \langle w'\rangle()$ if and only if $w \leqslant w'$.*

An important property is that a finer term has at least as many head constructors as a coarser one:

Lemma 1.8. *If $u \not\approx \mathbf{0}$, we have:*

$$u \preccurlyeq \mathtt{C}v \qquad \Longleftrightarrow \qquad u = \mathtt{C}u' \text{ with } u' \preccurlyeq v$$

and

$$u \preccurlyeq (v_1, \dots, v_n) \qquad \Longleftrightarrow \qquad u = (u_1, \dots, u_n) \text{ with } u_1 \preccurlyeq v_1 \dots u_n \preccurlyeq v_n$$

Proof. There is a simple, direct inductive proof, but this lemma also follows from the characterization of $\preccurlyeq$ in Appendix A (Lemma A.5). □

The next lemma gives some facts about the preorder $(\mathcal{T}, \preccurlyeq)$ that may help getting some intuitions.

Lemma 1.9. *We have*

- $\mathbf{0}$ *is the least element,*
- $\langle\infty\rangle()$ *is the greatest element of $\mathcal{T}()$, the set of* closed terms*,*
- $+$ *is a least-upper bound, i.e., $t_1 + t_2 \preccurlyeq u$ iff $t_1 \preccurlyeq u$ and $t_2 \preccurlyeq u$,*
- *if t and u are simple, then $t \preccurlyeq u$ and $u \preccurlyeq t$ iff $t = u$.*

The last point follows from Lemmas 1.7 and 1.8. The rest is direct.

1.2. **Substitutions and Control-Flow Graphs.**

Just like a term is meant to represent one argument of a recursive call, a substitution $[\,\mathtt{x}_1 := u_1\,;\,\dots\,;\,\mathtt{x}_n := u_n\,]$ is meant to represent *all* the arguments of a recursive call to an n-ary function. In order to follow the evolution of arguments along several recursive calls, we need to *compose* substitutions: given some terms t, u_1, …,u_n in $\mathcal{T}$, we define $t\,[\,\mathtt{x}_1 := u_1\,;\,\dots\,;\,\mathtt{x}_n := u_n\,]$ as the parallel substitution of each $\mathtt{x}_i$ by u_i. The composition $\tau \circ \sigma$ of two substitutions $\tau = [\,\mathtt{x}_1 := u_1\,;\,\dots\,;\,\mathtt{x}_n = u_n\,]$ and σ is simply the substitution $\tau \circ \sigma = [\,\mathtt{x}_1 := u_1\sigma\,;\,\dots\,;\,\mathtt{x}_n := u_n\sigma\,]$.

Lemma 1.10. *Composition of substitutions is associative and monotonic (for the pointwise order) on the right and on the left: if $\tau_1 \preccurlyeq \tau_2$ then $\sigma \circ \tau_1 \preccurlyeq \sigma \circ \tau_2$ and $\tau_1 \circ \sigma \preccurlyeq \tau_2 \circ \sigma$.*

Proof. Associativity is obvious. Monotonicity on the left follows from the fact that $\preccurlyeq$ is contextual. For monotonicity on the right, we show "$t_1 \preccurlyeq t_2$ implies $t_1[\mathtt{x} := u] \preccurlyeq t_2[\mathtt{x} := u]$" by induction on $t_1 \preccurlyeq t_2$. The only interesting case is when $t_1 \preccurlyeq t_2$ because $t_1 = t[\mathtt{y} := v_1]$ and $t_2 = t[\mathtt{y} := v_2]$ and $v_1 \preccurlyeq v_2$. By induction hypothesis, we have $v_1[\mathtt{x} := u] \preccurlyeq v_2[\mathtt{x} := u]$. There are two cases:

- if $\mathtt{x} = \mathtt{y}$, we have $t_i[\mathtt{x} := u] = t[\mathtt{x} := v_i][\mathtt{x} := u] = t\big[\mathtt{x} := v_i[\mathtt{x} := v_i]\big]$ and we get $t_1[\mathtt{x} := u] \preccurlyeq t_2[\mathtt{x} := u]$ by contextuality applied to the induction hypothesis;
- if $\mathtt{x} \neq \mathtt{y}$, we have $t_i[\mathtt{x} := u] = t[\mathtt{y} := v_i][\mathtt{x} := u] = t[\mathtt{x} := u]\big[\mathtt{y} := v_i[\mathtt{x} := u]\big]$, and here again, we get $t_1[\mathtt{x} := u] \preccurlyeq t_2[\mathtt{x} := u]$ by contextuality applied to the induction hypothesis.

□

We can now define what the abstract interpretations for our programs will be:

Definition 1.11. A *control-flow graph* for a set of mutually recursive definitions is a labeled graph where:

- vertices are function names,
- if the parameters of $\mathtt{f}$ are $\mathtt{y}_1, \dots, \mathtt{y}_m$ and the parameters of $\mathtt{g}$ are $\mathtt{x}_1, \dots, \mathtt{x}_n$, the labels of arcs from $\mathtt{f}$ to $\mathtt{g}$ are substitutions $[\,\mathtt{x}_1 := u_1\,;\,\dots\,;\,\mathtt{x}_n := u_n\,]$, where each u_i is a term in $\mathcal{T}(\mathtt{y}_1, \dots, \mathtt{y}_m)$.

That a control-flow graph is *safe* (Definition 1.13) means that it gives approximations of the real evolution of arguments of the recursive calls during evaluation. Since we are in a call-by-value language, those arguments are first-order values of the ambient language (see page 2) which can be embedded in $\mathcal{T}()$.

Definition 1.12. A *value* is a simple term of $\mathcal{T}()$, i.e., a simple closed term. An *exact value* is a value which doesn't contain any $\langle w \rangle$, with $w \in \mathbf{Z}_\infty$.

First-order values of the ambient language correspond precisely to exact values in $\mathcal{T}$. We can now define safety formally:

Definition 1.13. Let G be a control-flow graph for some recursive definitions,

(1) suppose we have a call site from $\mathtt{f}$ to $\mathtt{g}$:

```
val rec f x1 x2 ... xn =
  ... g u1 ... um
  ...
```

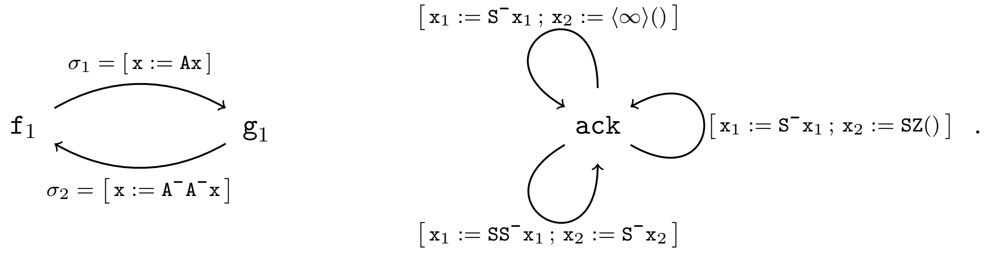
An arc $\mathtt{f} \xrightarrow{\sigma} \mathtt{g}$ in G *safely represents* this particular call site if for every substitution ρ of the parameters by *exact values* $\rho = [\,\mathtt{x}_1 := v_1\,;\,\dots\,;\,\mathtt{x}_n := v_n\,]$, we have

$$[\,\mathtt{y}_1 := [\![u_1]\!]_\rho\,;\,\dots\,;\,\mathtt{y}_m := [\![u_m]\!]_\rho\,] \quad \preccurlyeq \quad \sigma \circ \rho$$

where each $[\![u_i]\!]_\rho$ is the value of u_i given by the operational semantics of the language, in the environment where each variable $\mathtt{x}_i$ has value $\rho(\mathtt{x}_i)$.

(2) A set of mutually recursive definitions is *safely represented* by a control-flow graph if each call site is safely represented by at least an arc in the graph.

For example, the recursive definitions for $\mathtt{f}_1$ and $\mathtt{g}_1$ from page 4 and the Ackermann function are safely represented by the following control-flow graphs:



$$\begin{tikzcd} \mathtt{f}_1 \arrow[r, bend left, "\sigma_1 = [\,\mathtt{x} := \mathtt{Ax}\,]"] & \mathtt{g}_1 \arrow[l, bend left, "\sigma_2 = [\,\mathtt{x} := \mathtt{A}^{-}\mathtt{A}^{-}\mathtt{x}\,]"] \end{tikzcd} \qquad \begin{tikzcd} \mathtt{ack} \arrow[loop above, "{[\,\mathtt{x}_1 := \mathtt{S}^{-}\mathtt{x}_1\,;\,\mathtt{x}_2 := \langle\infty\rangle()\,]}"] \arrow[loop right, "{[\,\mathtt{x}_1 := \mathtt{S}^{-}\mathtt{x}_1\,;\,\mathtt{x}_2 := \mathtt{SZ}()\,]}"] \arrow[loop below, "{[\,\mathtt{x}_1 := \mathtt{SS}^{-}\mathtt{x}_1\,;\,\mathtt{x}_2 := \mathtt{S}^{-}\mathtt{x}_2\,]}"] \end{tikzcd} .$$

The arc σ_2 safely represents the call

```
... match x with A[A[y]] -> f1 y
```

because any value v of `x` reaching the call must be of the form $\mathtt{AA}v'$. We thus have $[\![\mathtt{x}]\!] = v$ and $[\![\mathtt{y}]\!] = v'$ in this environment. We have that

$$\big[\,\mathtt{x} := [\![y]\!]\,\big] = \big[\,\mathtt{x} := v'\,\big] \quad \preccurlyeq \quad \sigma_2 \circ \big[\,\mathtt{x} := [\![x]\!]\,\big] = \big[\,\mathtt{x} := \overline{\mathtt{A}}\,\overline{\mathtt{A}}\,\mathtt{AA}v'\,\big] \approx \big[\,\mathtt{x} := v'\,\big] \ .$$

The "$\mathtt{x}_2 := \langle\infty\rangle()$" in the loop for the Ackermann function is needed because we don't know how to express the second argument at the call site `ack m (ack S[m] n)`. The upper arc safely represents this call site because for all possible values of `m` and `n`, the semantics of `ack S[m] n` (a natural number) is approximated by $\langle\infty\rangle()$.

1.3. **Collapsing.** For combinatorial reasons, we will need the labels of the control-flow graph (substitutions) to live in a finite set. The two main obstructions for the finiteness of $\mathcal{T}$ are that the depth of terms is unbounded and that there are infinitely many possible weights for the approximations $\langle w\rangle$s. Define the *constructor depth* and the *destructor depth* of a term with

$$\begin{aligned}
\mathrm{depth}_C\big(\mathtt{C}t\big) &\overset{\mathsf{def}}{=} 1 + \mathrm{depth}_C(t)\\
\mathrm{depth}_C\big((t_1,\dots,t_n)\big) &\overset{\mathsf{def}}{=} \max_{1\le i\le n}\big(1+\mathrm{depth}_C(t_i)\big)\\
\mathrm{depth}_C\big(\overline{d}\big) &\overset{\mathsf{def}}{=} 0\\
\mathrm{depth}_C\big(\langle w\rangle\overline{d}\big) &\overset{\mathsf{def}}{=} 0
\end{aligned}$$

and the *destructor depth* of simple terms as:

$$\begin{aligned}
\mathrm{depth}_D\big(\mathtt{C}t\big) &\overset{\mathsf{def}}{=} \mathrm{depth}_D(t)\\
\mathrm{depth}_D\big((t_1,\dots,t_n)\big) &\overset{\mathsf{def}}{=} \max_{1\le i\le n}\big(\mathrm{depth}_D(t_i)\big)\\
\mathrm{depth}_D\big(\overline{d}\big) &\overset{\mathsf{def}}{=} |\overline{d}\,|\\
\mathrm{depth}_D\big(\langle w\rangle\overline{d}\big) &\overset{\mathsf{def}}{=} |\overline{d}\,| \ .
\end{aligned}$$

The depth of a sum of simple terms is the maximum of the depth of its summands. This allows to define the following restriction for terms:

Definition 1.14. We write $\mathcal{T}_{D,B}$ for the subset of all $t \in \mathcal{T}$ s.t.

- t is in normal form
- for each $\langle w\rangle$ appearing in t, we have $w \in \mathbf{Z}_B = \{-B,\dots,0,1,\dots,B-1,\infty\}$,
- the constructor depth and the destructor depth of t are less or equal than D.

The aim is to send each element of $\mathcal{T}$ to an approximation that belongs to $\mathcal{T}_{D,B}$. Given $B > 0$ (fixed once and for all), it is easy to collapse all the weights in $\mathbf{Z}_\infty$ into the finite set $\mathbf{Z}_B$: send each w to $\lceil w\rceil_B$, with

$$\lceil w\rceil_B \quad \overset{\mathsf{def}}{=} \quad \begin{cases} -B & \text{if } w < -B\\ w & \text{if } -B \leqslant w < B\\ \infty & \text{if } w \geqslant B \ . \end{cases}$$

This gives rise to a function from simple terms to simple terms with bounded weights: using the grammar of simple terms from Lemma 1.5, we define

$$\begin{array}{rcl} \lceil \mathtt{C}t \rceil_B & \overset{\mathsf{def}}{=} & \mathtt{C}\lceil t \rceil_B \\ \lceil (t_1, ..., t_n) \rceil_B & \overset{\mathsf{def}}{=} & \big(\lceil t_1 \rceil_B, ..., \lceil t_n \rceil_B\big) \\ \lceil \langle w \rangle \overline{d}\, \rceil_B & \overset{\mathsf{def}}{=} & \langle \lceil w \rceil_B \rangle\, \overline{d} \\ \lceil \overline{d}\, \rceil_B & \overset{\mathsf{def}}{=} & \overline{d}\ . \end{array}$$

Ensuring that the depth is bounded is more subtle. Given $D \geqslant 0$ (fixed once and for all) and $t \in \mathcal{T}$ in normal form, we want to bound the constructor depth and the destructor depth by D. This is achieved with the following definition acting on simple terms and extended by linearity. Because of $(*)$, the clauses are not disjoint and only the first appropriate one is used:

$$\begin{array}{rcll} (\mathtt{C}t)_{\restriction_i} & \overset{\mathsf{def}}{=} & \mathtt{C}(t_{\restriction_{i-1}}) & \text{if } i > 0 \\ (t_1, \dots, t_n)_{\restriction_i} & \overset{\mathsf{def}}{=} & \big(t_1{}_{\restriction_{i-1}}, \dots, t_n{}_{\restriction_{i-1}}\big) & \text{if } i > 0 \\ \big(\langle w \rangle \overline{d}\big)_{\restriction_i} & \overset{\mathsf{def}}{=} & \langle w \rangle \big(\overline{d}_{\downharpoonright_D}\big) & \text{if } i > 0 \\ \overline{d}_{\restriction_i} & \overset{\mathsf{def}}{=} & \overline{d}_{\downharpoonright_D} & \text{if } i > 0 \\ t_{\restriction_0} & \overset{\mathsf{def}}{=} & \mathrm{nf}(\langle 0 \rangle t)_{\downharpoonright_D} & (*) \\ \big(\langle w \rangle \overline{d}\big)_{\downharpoonright_D} & \overset{\mathsf{def}}{=} & \langle w \rangle \big(\overline{d}_{\downharpoonright_D}\big) & (**) \\ \overline{d}_{\downharpoonright_D} & \overset{\mathsf{def}}{=} & \overline{d} & \text{if } |\overline{d}| \leqslant D \\ \big(\mathtt{C}^{\text{-}}\overline{d}\big)_{\downharpoonright_D} & \overset{\mathsf{def}}{=} & \langle -1 \rangle \big(\overline{d}_{\downharpoonright_D}\big) & \text{if } |\mathtt{C}^{\text{-}}\overline{d}| > D \\ \big(\pi_i \overline{d}\big)_{\downharpoonright_D} & \overset{\mathsf{def}}{=} & \langle -1 \rangle \big(\overline{d}_{\downharpoonright_D}\big) & \text{if } |\pi_i \overline{d}| > D\ . \end{array}$$

Note that we need to compute a normal form for clause $(*)$, and that since the normal form of $\langle 0 \rangle t$ doesn't contain any constructor (recall that approximations absorb constructors on their right), each summand of the result will match the left side of clause $(**)$.

The function $t \mapsto t_{\restriction_D}$ does several things:

- it keeps the constructors up to depth D (the first four clauses),
- it removes the remaining constructors with $t \mapsto \langle 0 \rangle t$ (clause $(*)$),
- it keeps a suffix of at most D destructors in front of each variable and incorporates the additional destructors into the preceding $\langle w \rangle$ (the last three clauses).

For example, we have

$$\big(\mathtt{ABCD}\langle w \rangle \mathtt{X}^{\text{-}}\mathtt{Y}^{\text{-}}\mathtt{Z}^{\text{-}}\mathtt{x}\big)_{\restriction_2} \quad = \quad \mathtt{AB}\langle w+1 \rangle \mathtt{Y}^{\text{-}}\mathtt{Z}^{\text{-}}\mathtt{x}$$

and

$$\big(\mathtt{A}(\mathtt{x}, \mathtt{B}\langle w \rangle \mathtt{X}^{\text{-}}\mathtt{Y}^{\text{-}}\mathtt{y})\big)_{\restriction_1} \quad = \quad \mathtt{A}\langle 1 \rangle \mathtt{x} + \mathtt{A}\langle w+1 \rangle \mathtt{Y}^{\text{-}}\mathtt{y}\ .$$

Lemma 1.15. *The collapsing function $t \mapsto \lceil t_{\restriction_D} \rceil_B$ is inflationary and monotonic:*

- $t \preccurlyeq \lceil t_{\restriction_D} \rceil_B$,
- *if $t \preccurlyeq u$ then $\lceil t_{\restriction_D} \rceil_B \preccurlyeq \lceil u_{\restriction_D} \rceil_B$,*

More precisely, both functions $t \mapsto \lceil t \rceil_B$ and $t \mapsto t_{\restriction_D}$ are inflationary and monotonic.

Proof. By definition of $\preccurlyeq$, $\lceil \_ \rceil_B$ is inflationary and monotonic. It follows from the fact that $\lceil \_ \rceil : \mathbf{Z}_\infty \to \mathbf{Z}_B$ is itself inflationary and monotonic.

That $\_{\restriction_D}$ is inflationary relies on the fact that $t \preccurlyeq \langle 0\rangle t$, $\mathtt{C}^{\text{-}} t \preccurlyeq \langle -1\rangle t$ and $\pi_i t \preccurlyeq \langle -1\rangle t$; it is a direct inductive proof. The proof that $\_{\restriction_D}$ is monotonic is a tedious inductive proof. It is omitted for sake of brevity.

Together, these facts imply that $\lceil \_{\restriction_D} \rceil_B$ is both inflationary and monotonic. $\square$

The next lemma justifies the use of this collapsing function.

Lemma 1.16. *For each $t \in \mathcal{T}$, we have $\lceil t_{\restriction_D} \rceil_B \in \mathcal{T}_{D,B}$. Moreover, $\lceil t_{\restriction_D} \rceil_B$ is the least term in $\mathcal{T}_{D,B}$ that approximates t. In particular, the function $t \mapsto \lceil t_{\restriction_D} \rceil_B$ is idempotent*

$$\left\lceil \lceil t_{\restriction_D} \rceil_{B\restriction_D} \right\rceil_B = \lceil t_{\restriction_D} \rceil_B \ .$$

Proof. It is easy to show that both $\lceil \_ \rceil_B$ and $\_{\restriction_D}$ are idempotent. Idempotence of $\lceil \_{\restriction_D} \rceil_B$ follows from the fact that $\lceil t_{\restriction_D} \rceil_{B\restriction_D} = \lceil t_{\restriction_D} \rceil_B$. That $\lceil t_{\restriction_D} \rceil_B \in \mathcal{T}_{D,B}$ follows directly from the definitions. Since it is not needed in this paper, the proof that $\lceil t_{\restriction_D} \rceil_B$ is the least term in $\mathcal{T}_{D,B}$ that approximates t is omitted. $\square$

An interesting corollary of Lemma 1.16 is that collapsing is monotonic with respect to the bound D and B:

Corollary 1.17. *If $0 \leqslant D \leqslant D'$ and $0 < B \leqslant B'$, then $\lceil t_{\restriction_{D'}} \rceil_{B'} \preccurlyeq \lceil t_{\restriction_D} \rceil_B$.*

Definition 1.18. If $\sigma = [\, \mathtt{x}_1 := t_1 \,;\, \dots \,;\, \mathtt{x}_n := t_n \,]$ and $\tau = [\, \mathtt{y}_1 := u_1 \,;\, \dots \,;\, \mathtt{y}_m := u_m \,]$ are substitutions, then $\tau \diamond \sigma$ is defined as the pointwise collapsing $\left\lceil (\tau \circ \sigma)_{\restriction_D} \right\rceil_B$.

This collapsed composition "$\diamond$" is a binary operation on $\mathcal{T}_{D,B}$. Unfortunately, it is not associative! For example, when $B = 2$, the composition

$$[\, \mathtt{r} := \langle -1\rangle \mathtt{x} \,] \diamond [\, \mathtt{x} := \langle 1\rangle \mathtt{y} \,] \diamond [\, \mathtt{y} := \langle 1\rangle \mathtt{z} \,]$$

can give $[\, \mathtt{r} := \langle 1\rangle \mathtt{z} \,]$ or $[\, \mathtt{r} := \langle \infty\rangle \mathtt{z} \,]$ depending on which composition we start with. Similarly, when $D = 1$, the composition

$$[\, \mathtt{r} := \mathtt{C}^{\text{-}} \mathtt{x} \,] \diamond [\, \mathtt{x} := \mathtt{C}\mathtt{y} \,] \diamond [\, \mathtt{y} := \mathtt{D}\mathtt{z} \,]$$

can give $[\, \mathtt{r} := \mathtt{D}\mathtt{z} \,]$ or $[\, \mathtt{r} := \langle 1\rangle \mathtt{z} \,]$. There is a special case: when $D = 0$ and $B = 1$, the operation $\diamond$ is becomes associative! This was the original case of SCT [6]. In general, we have:

Definition 1.19. Two terms u and v are called *compatible*, written $u \supset\!\subset v$, if there is some $t \not\approx \mathbf{0}$ that is finer than both, i.e., such that $t \preccurlyeq u$ and $t \preccurlyeq v$. Two substitutions are compatible if they are pointwise compatible.

Lemma 1.20. *If $\sigma_1, \dots, \sigma_n$ is a sequence of composable substitutions, and if τ_1 and τ_2 are the results of computing $\sigma_n \diamond \dots \diamond \sigma_1$ in different ways, then $\tau_1 \supset\!\subset \tau_2$.*

Proof. We have $\sigma_n \circ \dots \circ \sigma_1 \preccurlyeq \tau_1$ and $\sigma_n \circ \dots \circ \sigma_1 \preccurlyeq \tau_2$. $\square$

In order to simplify notations, we omit parenthesis and make this operation associate on the right: $\sigma_1 \diamond \sigma_2 \diamond \sigma_3 \overset{\text{def}}{=} \sigma_1 \diamond (\sigma_2 \diamond \sigma_3)$.

2. Size-Change Combinatorial Principle

2.1. **Combinatorial Lemma.** The heart of the criterion is the following combinatorial lemma

Lemma 2.1. *Let G be a control-flow graph; then, for every infinite path of composable substitutions*

$$\mathtt{f}_0 \xrightarrow{\sigma_0} \mathtt{f}_1 \xrightarrow{\sigma_1} \dots \xrightarrow{\sigma_n} \mathtt{f}_{n+1} \xrightarrow{\sigma_{n+1}} \dots$$

in the control-flow graph G, there is a node $\mathtt{f}$ such that the path can be decomposed as

$$\mathtt{f}_0 \quad \underbrace{\xrightarrow{\sigma_0} \dots \xrightarrow{\sigma_{n_0-1}}}_{\textit{initial prefix}} \quad \mathtt{f} \quad \underbrace{\xrightarrow{\sigma_{n_0}} \dots \xrightarrow{\sigma_{n_1-1}}}_{\tau} \quad \mathtt{f} \quad \underbrace{\xrightarrow{\sigma_{n_1}} \dots \xrightarrow{\sigma_{n_2-1}}}_{\tau} \quad \mathtt{f} \quad \dots$$

where:

- *all the $\sigma_{n_{k+1}-1} \diamond \dots \diamond \sigma_{n_k}$ are equal to the same $\tau : \mathtt{f} \to \mathtt{f}$,*
- *τ is* coherent*: $\tau \diamond \tau \supset \tau$.*

The proof is the same as the original SCT [6], with only a slight modification to deal with the fact that $\diamond$ isn't associative.

Proof. This is a consequence of the infinite Ramsey theorem. Let $(\sigma_n)_{n\geqslant 0}$ be an infinite path as in the lemma. We associate a "color" $c(m,n)$ to each pair (m,n) of natural numbers where $m < n$:

$$c(m,n) \quad \overset{\mathsf{def}}{=} \quad \Big(\mathtt{f}_m \ , \ \mathtt{f}_n \ , \ \sigma_{n-1} \diamond \dots \diamond \sigma_m\Big) \ .$$

Since the number of constructors and the arity of tuples that can arise from compositions in a control flow graph is finite, the number of possible colors is finite. By the infinite Ramsey theorem, there is an infinite set $I \subseteq \mathbf{N}$ such all the (i,j) for $i < j \in I$ have the same color $(\mathtt{f}, \mathtt{f'}, \tau)$. Write $I = \{n_0 < n_1 < \dots < n_k < \dots\}$. If $i < j < k \in I$, we have:

$$\begin{aligned}
\Big(\mathtt{f} \ , \ \mathtt{f}' \ , \ \tau\Big) &= \Big(\mathtt{f}_i \ , \ \mathtt{f}_j \ , \ \sigma_{j-1} \diamond \dots \diamond \sigma_i\Big) \\
&= \Big(\mathtt{f}_j \ , \ \mathtt{f}_k \ , \ \sigma_{k-1} \diamond \dots \diamond \sigma_j\Big) \\
&= \Big(\mathtt{f}_i \ , \ \mathtt{f}_k \ , \ \sigma_{k-1} \diamond \dots \diamond \sigma_i\Big)
\end{aligned}$$

which implies that $\mathtt{f} = \mathtt{f}' = \mathtt{f}_i = \mathtt{f}_j = \mathtt{f}_k$ and

$$\begin{aligned}
\tau &= \sigma_{j-1} \diamond \dots \diamond \sigma_i \\
&= \sigma_{k-1} \diamond \dots \diamond \sigma_j \\
&= \sigma_{k-1} \diamond \dots \diamond \sigma_j \diamond \sigma_{j-1} \diamond \dots \diamond \sigma_i \\
\tau \diamond \tau &= \big(\sigma_{k-1} \diamond \dots \diamond \sigma_j\big) \diamond \big(\sigma_{j-1} \diamond \dots \diamond \sigma_i\big) \ .
\end{aligned}$$

In the original SCT principle, composition was associative and we had $\tau \diamond \tau = \tau$. Here however, τ and $\tau \diamond \tau$ differ only in the order of compositions, and we only get that $\tau \supset \tau \diamond \tau$ (Lemma 1.20). □

2.2. **Graph of paths.** The graph of paths of a control-flow graph G is the graph G^+ with the same vertices as G and where arcs between a and b in G^+ correspond exactly to paths between a and b in G. In our case, the graph is labeled with substitutions and the label of a path is the composition of the labels of its arcs.

Definition 2.2. If G is a control-flow graph, the graph G^+, the *graph of paths of* G, is the control-flow graph defined as follows:

- $G^0 = G$,
- in G^{n+1}, the arcs from $\mathtt{f}$ to $\mathtt{g}$ are
$$G^{n+1}(\mathtt{f},\mathtt{g}) \quad = \quad G^n(\mathtt{f},\mathtt{g}) \cup \big\{\sigma \diamond \tau \;\mid\; \tau \in G^n(\mathtt{f},\mathtt{h}), \sigma \in G(\mathtt{h},\mathtt{g})\big\}$$
where $\mathtt{h}$ ranges over all vertices of G,
- $G^+ = \bigcup_{n \geqslant 0} G^n$.

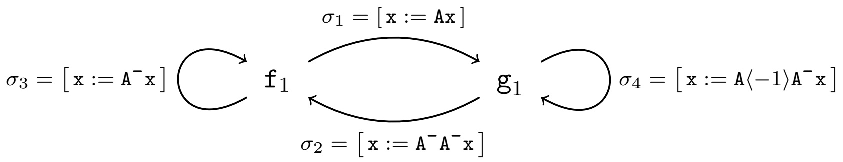
By definition, each path $\sigma_1 \cdots \sigma_n$ in G corresponds to an arc in $G^+_{D,B}$ that is labelled with $\sigma_n \diamond \ldots \diamond \sigma_1$. (Recall that "$\diamond$" associates on the right.) The restrictions of the sets $\mathcal{T}_{D,B}(\mathtt{x}_1, \ldots, \mathtt{x}_m)$ to terms that can appear in compositions of arcs in a control-flow graph G are finite because the number of variables and constructors in G is finite and the arity of tuples is bounded. We thus have:

Lemma 2.3. *G^+ is finite and can be computed in finite time. More precisely, $G^n = G^{n+1}$ for some n, and G^+ is equal to this G^n.*

As an example, here are the first steps of the computation of the graph of paths of the control-flow graph for the functions $\mathtt{f}_1$ and $\mathtt{g}_1$ (page 4) when $D = B = 1$. The initial control-flow graph G given by the static analysis is given on page 9. The graph $G^0 = G$ contains only two arcs:

- $\sigma_1 \overset{\text{def}}{=} [\,\mathtt{x} := \mathtt{Ax}\,]$ from $\mathtt{f}_1$ to $\mathtt{g}_1$;
- $\sigma_2 \overset{\text{def}}{=} [\,\mathtt{x} := \mathtt{A^-A^-x}\,]$ from $\mathtt{g}_1$ to $\mathtt{f}_1$.

The graph G^1 is then



$$\begin{tikzcd}
\mathtt{f}_1 \arrow[loop left, "{\sigma_3 = [\,\mathtt{x} := \mathtt{A^-x}\,]}"] \arrow[r, bend left, "{\sigma_1 = [\,\mathtt{x} := \mathtt{Ax}\,]}"] & \mathtt{g}_1 \arrow[l, bend left, "{\sigma_2 = [\,\mathtt{x} := \mathtt{A^-A^-x}\,]}"] \arrow[loop right, "{\sigma_4 = [\,\mathtt{x} := \mathtt{A}\langle -1\rangle\mathtt{A^-x}\,]}"]
\end{tikzcd}$$

where the loop σ_3 on the left is obtained as $\sigma_2 \diamond \sigma_1$ and the loop σ_4 on the right is obtained as $\sigma_1 \diamond \sigma_2$. The next iteration gives the following arcs for G^2:

- $\sigma_5 \overset{\text{def}}{=} \sigma_1 \diamond \sigma_3$ which gives $[\,\mathtt{x} := \mathtt{A}\langle -1\rangle\mathtt{x}\,]$ from $\mathtt{f}_1$ to $\mathtt{g}_1$,
- $\sigma_1 \diamond \sigma_2$ which gives σ_4 around $\mathtt{g}_1$,
- $\sigma_6 \overset{\text{def}}{=} \sigma_2 \diamond \sigma_4$ which gives $[\mathtt{x} := \langle -1\rangle\mathtt{A^-x}]$ from $\mathtt{g}_1$ to $\mathtt{f}_1$,
- $\sigma_2 \diamond \sigma_1$ which gives σ_3 around $\mathtt{f}_1$.

The next iteration G^3 yields a single new arc: $[\,\mathtt{x} := \langle -1\rangle\mathtt{x}\,]$ from $\mathtt{f}_1$ to $\mathtt{f}_1$. This graph G^3 with 7 arcs is the graph of paths of the starting control-flow graph.

2.3. Size-Change Termination Principle.

First, a small lemma:

Lemma 2.4. *If $v \in \mathcal{T}()$ is an exact value, then:*

- *the normal form of $\langle 0\rangle v$ is of the form $\sum_i \langle w_i\rangle()$ where $\max_i\big(w_i\big) = \mathrm{depth}(v)$,*
- *$\langle \mathrm{depth}(v)\rangle() \preccurlyeq \mathrm{nf}(\langle 0\rangle v) \preccurlyeq \langle \mathrm{depth}(v)\rangle()$,*
- *if $v \preccurlyeq \langle w\rangle()$ then $w \geqslant \mathrm{depth}(v)$.*

Proof. The first point is a simple inductive proof on v, and the second point follows directly.

For the third point, suppose that $v \preccurlyeq \langle w\rangle()$. By contextuality of $\preccurlyeq$ (refer to Definition 1.6), we get $\langle 0\rangle v \preccurlyeq \langle 0\rangle\langle w\rangle() \approx \langle w\rangle()$ and so, by the second point and transitivity, that $\langle \mathrm{depth}(v)\rangle() \preccurlyeq \langle w\rangle()$. We conclude by Lemma 1.9. □

A subvalue of a value can be accessed by a sequence of destructors. For example, the right subtree of a binary tree can be accessed with π_2`Node¯` in the sense that π_2`Node¯`v reduces exactly to the right subtree of v. By the previous lemma, we can get the depth of the right subtree by precomposing a value with $\langle 0\rangle\pi_2$`Node¯`. A *decreasing parameter* is a subvalue of a parameter whose depth decreases strictly over a given recursive call. For example, in the call

```
val rec push_left x =
  match x with Node[t₁, Node[t₂,t₃]] -> push_left Node[Node[t₁,t₂],t₃]
             | ...
```

the right subtree of `x` is decreasing, while neither its left subtree nor t itself are decreasing. In the control-flow graph, this call site becomes a loop labeled with

$$\tau \quad = \quad \left[\; \mathtt{x} := \mathtt{Node}\Big(\mathtt{Node}\big(\underbrace{\pi_1\mathtt{Node}^{\text{-}}\mathtt{x}}_{\mathtt{t}_1}, \underbrace{\pi_1\mathtt{Node}^{\text{-}}\pi_2\mathtt{Node}^{\text{-}}\mathtt{x}}_{\mathtt{t}_2}\big), \underbrace{\pi_2\mathtt{Node}^{\text{-}}\pi_2\mathtt{Node}^{\text{-}}\mathtt{x}}_{\mathtt{t}_3}\Big) \;\right] .$$

Looking at $\langle 0\rangle\pi_2\mathtt{Node}^{\text{-}}\mathtt{x}[\tau]$ to get the depth of the right subtree of the argument after the recursive call, we obtain

$$\langle 0\rangle\pi_2\mathtt{Node}^{\text{-}}\mathtt{x}[\tau] \approx \langle 0\rangle\pi_2\mathtt{Node}^{\text{-}}\pi_2\mathtt{Node}^{\text{-}}\mathtt{x} \quad \preccurlyeq \quad \langle -2\rangle\pi_2\mathtt{Node}^{\text{-}}\mathtt{x} \; .$$

This means that the depth of the right subtree of the argument `x` has decreased by (at least) 2 after the recursive call.

Definition 2.5. Let $\tau = [\,\mathtt{x}_1 := t_1\,;\,\dots\,;\,\mathtt{x}_n := t_n\,]$ be a loop in a control-flow graph. A *decreasing parameter* for τ is a branch of destructors: $\xi = \langle 0\rangle\overline{d}\,\mathtt{x}_i$ such that $\mathbf{0} \not\approx \xi[\tau] \preccurlyeq \langle w\rangle\xi$ with $w < 0$ and $\overline{d}$ minimal, i.e., no strict suffix of $\overline{d}$ satisfies the same condition. A loop is called *decreasing* when it has a decreasing parameter.

The minimality condition is purely technical: without it, the loop $\tau = [\,\mathtt{x} := \mathtt{A}\langle -1\rangle\mathtt{A}^{\text{-}}\mathtt{x}\,]$ would have $\xi = \langle 0\rangle\mathtt{X}^{\text{-}}\mathtt{A}^{\text{-}}\mathtt{x}$ as a decreasing argument because $\xi[\tau] \approx \langle -2\rangle\mathtt{A}^{\text{-}}\mathtt{x}$. The problem is that `X` has nothing to do with the definition and `X¯A¯` might not even represent a subvalue of the parameter! A good decreasing parameter would be $\langle 0\rangle\mathtt{A}^{\text{-}}\mathtt{x}$. The minimality condition is necessary to prove the following lemma:

Lemma 2.6. *If ξ is a decreasing parameter for τ and $\mathbf{0} \not\approx \sigma \preccurlyeq \tau \circ \rho$ then $\mathbf{0} \not\approx \xi[\sigma] \preccurlyeq \xi[\tau \circ \rho]$ and in particular, $\xi[\tau \circ \rho] \not\approx \mathbf{0}$.*

Proof. Suppose that $\xi = \langle 0\rangle d_1 \cdots d_n \mathtt{x}_i$. The "inequality" follows from monotonicity. The important point is that under the hypothesis, we have $\mathbf{0} \not\approx \xi \circ \sigma$. The term $\xi[\sigma]$ is equal to $\langle 0\rangle d_1 \cdots d_n \sigma(\mathtt{x}_i)$. Suppose by contradiction that this reduces to $\mathbf{0}$. Suppose also that σ is in normal form.

There is only one reduction sequence of $\langle 0\rangle d_1 \cdots d_n \sigma(\mathtt{x}_i)$ and for it to give $\mathbf{0}$, this reduction sequence needs to use a reduction step from group (3) of Definition 1.2. In other words, a destructor of $d_1 \cdots d_n$ has to reach an incompatible constructor in $\sigma(\mathtt{x}_i)$.

Since $\sigma \preccurlyeq \tau \circ \rho$ by hypothesis, we have that $\sigma(\mathtt{x}_i) \preccurlyeq \tau \circ \rho(\mathtt{x}_i) = \tau(\mathtt{x}_i)\rho$. By Lemma 1.8 we know that all the head constructors of $\tau(\mathtt{x}_i)\rho$ also appear in $\sigma(\mathtt{x}_i)$. It is not difficult to see that the head constructors appearing in the normal form of $\tau(\mathtt{x}_i)$ also appear in $\tau(\mathtt{x}_i)[\rho]$. This phenomenon is general and doesn't depend on τ or ρ. It comes from the fact that applying a substitution to a term in normal form doesn't interfere with its constructors...

All the constructors of $\tau(\mathtt{x}_i)$ thus appear in $\sigma(\mathtt{x}_i)$. Since $d_1 \cdots d_n$ reaches an incompatible constructor in $\sigma(\mathtt{x}_i)$, the only way for $d_1 \cdots d_n$ to not reach an incompatible constructor in $\tau(\mathtt{x}_i)$ is to reach the end of the constructors in $\tau(\mathtt{x}_i)$ before the end of $d_1 \cdots d_n$. There are two cases:

- either $d_1 \cdots d_n$ reaches an approximation:

$$\begin{aligned} \langle 0\rangle d_1 \cdots d_n \tau(\mathtt{x}_i) &\to \dots \\ &\vdots \quad n-k \text{ reductions} \\ &\to \langle 0\rangle d_1 \cdots d_k \langle w'\rangle \overline{b}\,\mathtt{x}_i \\ &\approx \langle w'-k\rangle \overline{b}\,\mathtt{x}_i \\ &\preccurlyeq \langle w\rangle \overline{d}\,\mathtt{x}_i \quad \text{with } w < 0 \ . \end{aligned}$$

  By Lemma 1.7 we get that $\overline{d}$ is a suffix of $\overline{b}$, and $w' - k + |\overline{d}\,| \leqslant w + |\overline{b}\,|$. But then, we have $\langle 0\rangle d_{k+1} \cdots d_n \tau(\mathtt{x}_i) \to^* \langle w'\rangle \overline{b}\,\mathtt{x}_i$, and we have that $d_{k+1} \cdots d_n$ is a suffix of $\overline{b}$, and $w' + |\overline{d}_{k+1} \dots d_n| \leqslant w + |b|$. This implies that the sequence $d_1 \cdots d_n$ wasn't minimal as we have $\langle 0\rangle d_{k+1} \cdots d_n \tau(\mathtt{x}_i) \preccurlyeq \langle w\rangle d_{k+1} \cdots d_n \mathtt{x}_i$.
- The other possibility is that $\overline{d}$ reaches directly a branch of destructors:

$$\begin{aligned} \langle 0\rangle d_1 \cdots d_n \tau(\mathtt{x}_i) &\to \dots \\ &\vdots \quad n-k \text{ reductions} \\ &\to \langle 0\rangle d_1 \cdots d_k \overline{b}\,\mathtt{x}_i \\ &\preccurlyeq \langle w\rangle \overline{d}\,\mathtt{x}_i \quad \text{with } w < 0 \ . \end{aligned}$$

  By Lemma 1.7, $\overline{d}$ is a suffix of $d_1 \cdots d_k \cdot \overline{b}$ and $|\overline{d}\,| \leqslant w + |\overline{b}\,| + k$. The sequence $d_1 \cdots d_n$ isn't minimal because we have $\langle 0\rangle d_{k+1} \cdots d_n \tau \mathtt{x}_i \approx \langle 0\rangle \overline{b}\,\mathtt{x}_i$ with $d_{k+1} \cdots d_n$ a suffix of $\overline{b}$ and $|d_{k+1} \cdots d_n| \leqslant w + |\overline{b}\,|$, i.e., $\langle 0\rangle d_{k+1} \cdots d_n \tau(\mathtt{x}_i) \preccurlyeq \langle w\rangle d_{k+1} \cdots d_n \mathtt{x}_i$.

□

We can now state, and prove, the size-change termination principle.

Proposition 2.7 (Size-Change Termination Principle with Constructors)**.** *If G safely represents some recursive definitions and all coherent loops $\tau \supset \tau \diamond \tau$ in G^+ are decreasing, then the evaluation of the functions on values cannot produce an infinite sequence of calls.*

Proof. Suppose the conditions of the proposition are satisfied and suppose that function $\mathtt{h}$ on values $v_1, \dots, v_m$ provokes an infinite sequence of calls $c_1 \cdots c_n \cdots$. Write ρ_n for the arguments of call c_n. The ρ_n's contain first-order values and in particular, ρ_0 corresponds to the initial arguments of $\mathtt{h}$: $\rho_0 = [\mathtt{x}_1 := v_1 ; \dots ; \mathtt{x}_m := v_m]$. Let $\sigma_1 \cdots \sigma_n \cdots$ be the substitutions that label the arcs of G corresponding to the calls $c_1 c_2 \cdots$. We can use

Lemma 2.1 to decompose this sequence as:

$$\mathtt{h} \quad \underbrace{\xrightarrow{\sigma_0} \dots \longrightarrow}_{\text{initial prefix}} \quad \mathtt{f} \quad \underbrace{\xrightarrow{\sigma_{n_0}} \dots \longrightarrow}_{\tau} \quad \mathtt{f} \quad \underbrace{\xrightarrow{\sigma_{n_1}} \dots \longrightarrow}_{\tau} \quad \mathtt{f} \quad \dots$$

where:

- all the $\sigma_{n_{k+1}-1} \diamond \dots \diamond \sigma_{n_k}$ are equal to the same $\tau : \mathtt{f} \to \mathtt{f}$,
- τ is coherent: $\tau \supset \tau \diamond \tau$.

The control-flow graph G is safe and we thus have

$$\rho_{n+1} \quad \preccurlyeq \quad \sigma_n \circ \rho_n$$

Since $\circ$ is monotonic, we also get

$$\rho_{n_1} \quad \preccurlyeq \quad \sigma_{n_1-1} \circ \dots \circ \sigma_{n_0} \circ \rho_{n_0} \ .$$

By associativity of $\circ$, and because collapsing and composition are monotonic, we get

$$\rho_{n_1} \quad \preccurlyeq \quad (\sigma_{n_1-1} \diamond \dots \diamond \sigma_{n_0}) \circ \rho_{n_0} \ = \ \tau \circ \rho_{n_0} \ .$$

Repeating this, we obtain:

$$\rho_{n_k} \quad \preccurlyeq \quad \underbrace{\tau \circ \dots \circ \tau}_{k} \circ \rho_{n_0} \ .$$

By hypothesis, τ has a decreasing parameter: some $\xi = \langle 0 \rangle \overline{d}\,\mathtt{x}$ s.t. $\xi[\tau] \preccurlyeq \langle w \rangle \xi$ with $w < 0$. We thus have

$$\xi[\rho_{n_k}] \quad \preccurlyeq \quad \xi[\tau \circ \dots \circ \tau \circ \rho_{n_0}] \quad \preccurlyeq \quad \cdots \quad \preccurlyeq \quad \langle w + \dots + w \rangle \xi[\rho_{n_0}] \ .$$

By Lemma 2.6, the right side cannot be $\mathbf{0}$. By Lemma 2.4, it is approximated by $\langle w' \rangle ()$, where w' is equal to the depth of the value $\overline{d}\,\rho_{n_0}(\mathtt{x})$. We can choose k large enough to ensure that $-kw$ is strictly more than w'. Lemma 2.4 also implies that $\xi[\rho_{n_k}]$ approximates $\langle w'' \rangle ()$, where w'' is equal to depth $\big(\overline{d}\,\rho_{n_k}(\mathtt{x})\big)$. But then, we have

$$\langle w'' \rangle () \quad \preccurlyeq \quad \xi[\rho_{n_k}] \quad \preccurlyeq \quad \langle kw \rangle \xi[\rho_{n_0}] \quad \preccurlyeq \quad \langle kw + w' \rangle ()$$

where $w'' = \text{depth}(\xi[\rho_{n_k}]) \geqslant 0$ and $kw + w' < 0$. This contradicts Lemma 2.4. □

Definition 2.8. A control-flow graph G that satisfies the condition of Proposition 2.7 is said to be *size-change terminating for D and B*.

We have:

Proposition 2.9. *If G is size-change terminating for some $D \geqslant 0$ and $B > 0$, then G is also size-change terminating for all $D' \geqslant D$ and $B' \geqslant B$.*

Proof. Let G be a control-flow graph, and let $B' \geqslant B$ and $D' \geqslant D$. Suppose that G is size-change terminating for D and B; we want to show that it is also size-change terminating for D' and B'.

Let τ' be a coherent loop in $G^+_{D',B'}$. By construction, $\tau' = \sigma_1 \diamond_{D',B'} \cdots \diamond_{D',B'} \sigma_n$ for a path $\sigma_1 \dots \sigma_n$ in G. We can define $\tau = \sigma_1 \diamond_{D,B} \cdots \diamond_{D,B} \sigma_n$, which is a loop in $G^+_{D,B}$.

Since collapsing is monotonic (Lemma 1.17), we have that $\tau' \preccurlyeq \tau$. We also have that $\tau' \diamond_{D',B'} \tau' \preccurlyeq \tau \diamond_{D,B} \tau$ and because τ' is coherent, τ is also coherent. By hypothesis, τ has a decreasing parameter ξ: we have $\xi[\tau] \preccurlyeq \langle w \rangle \xi$, with $w < 0$. As τ': $\xi[\tau'] \preccurlyeq \xi[\tau] \preccurlyeq \langle w \rangle \xi$, there is a minimal suffix of ξ that is a decreasing argument for τ'. □

2.3.1. *The Algorithm.* The procedure checking if a set of mutually recursive definitions is terminating is thus:

1- static analysis: compute a safe representation of the recursive definitions as a control-flow graph G. The simple static analysis described in Appendix C is enough for all the examples in the paper and can be done in linear time.

2- choose bounds B and D: in our implementation, the bounds do not depend on G and are $B = 1$, $D = 2$ by default. The user can also change them by inserting pragmas together with the code of the recursive definitions.

3- compute the graph of paths: compute the graph of paths G^+ of G incrementally, with the bounds B and D. This step can take an exponential amount of space, as the example of the function `perms` (page 20) demonstrates.

4- check coherent loops: check that all the coherent loops of the graph G^+ computed previously are decreasing. If so, the functions of the definitions terminate; otherwise, the procedure cannot answer.

For this, it must be possible:
- to check the coherence relation $\asymp$,
- to look for decreasing arguments of a loop.

Some implementation details are given in Appendix B

Failure of Completeness. The original SCT satisfied a notion of *completeness* stating roughly that "all infinite paths are infinitely decreasing *iff* all coherent loops have a decreasing parameter". We capture more programs (Section 2.5) than the original SCT, but completeness doesn't hold anymore. Here is a counter example for $D = 0$ and $B = 2$:

```
val rec h1 x = match x with A[A[A[x]]] -> h2 x
  and   h2 x = h3 A[X]
  and   h3 x = h1 A[X]
```

The corresponding control-flow graph is

$$\begin{tikzcd}
& \mathtt{h}_1 \arrow[dl, bend right, "{[\mathtt{x} := \mathtt{A}^{-}\mathtt{A}^{-}\mathtt{A}^{-}\mathtt{x}]}"'] & \\
\mathtt{h}_2 \arrow[rr, bend right, "{[\mathtt{x} := \mathtt{Ax}]}"'] & & \mathtt{h}_3 \arrow[ul, bend right, "{[\mathtt{x} := \mathtt{Ax}]}"']
\end{tikzcd}$$

For every conceivable definition of "decreasing path", all the infinite paths in this graph should decrease infinitely. However, because of the consecutive "$[\,\mathtt{x} := \mathtt{Ax}\,]$" arcs, we will get a "$[\,\mathtt{x} := \langle\infty\rangle\mathtt{x}\,]$" arc in the graph of paths, corresponding to their composition. This will propagate and give coherent loops $[\,\mathtt{x} := \langle\infty\rangle\mathtt{x}\,]$ around each node. This graph is not size-change terminating for $D = 0$ and $B = 2$.

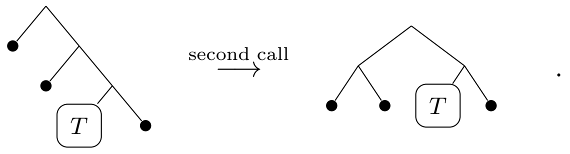
The previous example is size-change terminating whenever $B > 2$; but completeness doesn't even hold if we can choose the bounds B and D. Call a graph G *decreasing* if no infinite path comes from actual computation or, more precisely, if for every infinite path $(\sigma_k)_{k>0}$ and substitution ρ of values, there is a finite prefix $\sigma_1 \cdots \sigma_n$ s.t. $\rho \circ \sigma_1 \circ \cdots \sigma_n \approx \mathbf{0}$. The combing function transforming a binary tree into a right-leaning tree terminates for a subtle reason. Its definition is

```
val rec comb x = match x with
      Leaf[] -> Leaf[]
    | Node[t,Leaf[]] -> Node[comb t,Leaf[]]
    | Node[t1,Node[t2,t3]] -> comb Node[Node[t1,t2],t3]
```

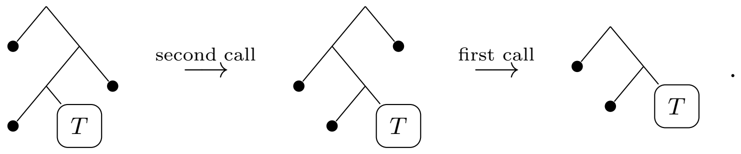
and it is safely represented by the graph with a single node `comb` and two loops:

- $\big[\,\mathtt{x} := \pi_1\mathtt{Node}^{-}\mathtt{x}\,\big]$
- $\big[\,\mathtt{x} := \mathtt{Node}\big(\mathtt{Node}(\pi_1\mathtt{Node}^{-}\mathtt{x}, \pi_1\mathtt{Node}^{-}\pi_2\mathtt{Node}^{-}\mathtt{x}), \pi_2\mathtt{Node}^{-}\pi_2\mathtt{Node}^{-}\mathtt{x}\big)\,\big]$.

This graph is terminating in the above sense precisely because `comb` terminates. It can however be shown that for every choice of D and B, this graph is *never* size-change terminating. The reason is that for any bound D and sequence $d_1 \cdots d_D$ of length D, there is a tree t for which the depth of the subtree $d_1 \cdots d_D t$ increases arbitrarily during a sequence of recursive calls. For example, at $D = 4$ for $\pi_1\mathtt{Node}^{-}\pi_2\mathtt{Node}^{-}$, consider the tree on the left:



By the second recursive call, the tree on the right will be used as the new argument. While $\pi_1\mathtt{Node}^{-}\pi_2\mathtt{Node}^{-}$ corresponds to the empty tree on the left, it corresponds to T on the right! Note that it is the conjunction of the two recursive calls that makes this possible: for $\pi_2\mathtt{Node}^{-}\pi_2\mathtt{Node}^{-}$, we need to use the second call and then the first call:



This implies that there can be no decreasing argument in the argument of `comb`!

Surprisingly, adding a second argument representing the size of the tree makes the function size-change terminating, i.e., the following definition *is* size-change terminating even though the second argument doesn't decrease at the second call site.

```
val rec comb_size t s = match t,s with
      Leaf[],_ -> Leaf[]
    | Node[t,Leaf[]],S[n] -> Node[comb_size t n,Leaf[]]
    | Node[t1,Node[t2,t3]],n -> comb_size Node[Node[t1,t2],t3],n
    | _,_ -> raise Error[]
```

In other words, we can define the combing function as

```
val comb t = comb_size t (size t)
```

and have the system automatically infer that it is terminating.

2.4. Complexity.

Lee, Jones and ben Amram showed that deciding whether a graph is size-change terminating in the original sense is P-space hard [6]. We can encode the same P-space complete problem as an instance of our version of size-change termination for $D = 0$ and $B = 1$. By monotonicity (Lemma 2.9), all other instances of size-change termination are P-space hard.

It is not difficult to construct ad-hoc small programs that require an exponential amount of space, even when $D = 0$ and $B = 1$. The simplest is probably the following:

```
val rec perms x₁ x₂ x₃ x₄ =
  g (perms x₂ x₁ x₃ x₄)
    (perms x₁ x₃ x₂ x₄)
    (perms x₁ x₂ x₄ x₃)
    (perms x₄ x₂ x₃ x₁)
```

where g is a previously defined function. The initial control-flow graph will contain a single node with 4 loops, and the graph of paths will contain 24 loops: one for each permutation of the parameters $\mathtt{x}_1$ through $\mathtt{x}_4$. More generally we can construct, for each n, a program of size n^2 for which the graph of paths will contain $n!$ loops. However, just like with the original SCT, checking termination of definitions written by hand with reasonable bounds B and D seems to remain practical.

2.5. Comparison with other SCT-Based Criterion.

In the original SCT, an arc in the control-flow graph was a bipartite graph with the parameters of the calling function on the left and the arguments of the called function on the right. A link from x to u can have label:

- $\downarrow$, meaning that the size of u is strictly smaller than the size of x,
- $\overline{\downarrow}$, meaning that the size of u is smaller or equal than the size of x.

Such a graph is said to be *fan-in free* if no u on the right is the target of more than one arc. We can encode such a bipartite graph as a substitution $\sigma = [\,\mathtt{y}_1 := t_1\,;\,\dots\,;\,\mathtt{y}_m := t_m\,]$ where:

- $t_k = \langle -1\rangle \mathtt{x}_i$ if there is an arc $\downarrow$ from $\mathtt{x}_i$ to u_i,
- $t_k = \langle 0\rangle \mathtt{x}_i$ if there is an arc $\overline{\downarrow}$ from $\mathtt{x}_i$ to u_i,
- $t_k = \langle \infty\rangle()$ otherwise.

It can be checked that when $D = 0$ and $B = 1$, composition and the size-change termination condition on G^+ correspond exactly to composition and the size-change termination condition from [6]. Note in particular that composition is associative in this context. Our criterion with $D = 0$ and $B = 1$ is roughly equivalent to the original SCT for fan-in free graph and with "depth" as the notion of size where all arcs have been initially collapsed. A small lemma stating that checking all coherent loops $\tau \supset \tau \diamond \tau$ is equivalent to checking only the idempotent loops $\tau = \tau \diamond \tau$ is necessary. (A more general conjecture which we have been unable to prove is that for checking size-change termination for arbitrary D and B, it is always sufficient to only check that idempotent loops have a decreasing argument.)

SCT with Difference Constraints. A. Ben-Amram considered a generalisation of the original SCT which, in our terminology, could be seen as choosing the bounds $D = 0$ and $B = \infty$ by allowing unbounded weights in the control-flow graphs [2]. The general problem is undecidable, but the restriction to fan-in free graph is decidable. The cost of this generality is the introduction of arithmetic in the decision procedure: deciding if a graph is size-change terminating involves integer linear programming. Our control-flow graphs are fan-in free and the criterion avoids arithmetics by putting a bound on the weights. We lose completeness as shown by the example on page 18, but this doesn't seem to be a problem in practice because the user may increase the bound B (at the cost of speed) and we've rarely found it necessary to go beyond 2 or 3. It would nevertheless be interesting to see if the approach of [2] can be combined with our approach to get a criterion for $B = \infty$ and arbitrary D.

Using "Calling Contexts". P. Manolios and D. Vroon generalized the SCT principle by adding "calling contexts" to the control-flow graph [7]. A calling context from `f` to `g` amounts to:

- a substitution describing the arguments of `g` as terms with free variables among the parameters of `f`,
- a set of expressions whose free variables are among the parameters of `f`.

The substitutions are built from the ambient language, as are the expressions in the set. The intuition of having such a calling context from `f` to `g` is that *if all the expressions of the set evaluate to* `True`*, then there can be a call to* `g` *from* `f`*, and the arguments of* `g` *are given by the substitution.*

This is much more expressive than our approach as the contexts may contain terms representing arbitrary conditions, like "`Prime(x)`" expressing that a parameter is a prime number. The drawback is that because the conditions contain free variables, an automatic theorem prover is necessary to decide when they evaluate to `True`. This version has been formalized and implemented [4, 5] in Isabelle [8], a proof assistant based on higher-order logic. The formalization relies Isabelle's "`auto`matic" tactic for checking those conditions.

Our approach uses a similar idea but restricts to the "constructors/destructors contexts" that were necessary to build the arguments of a call. This simplifies the problem so that everything can be handled combinatorially in a uniform way and makes it more appropriate for a proof assistant based on type theory like Coq, or the Agda programming language.

2.6. **Extensions.**

Weighted constructors. At the moment, each constructor has weight 1, as can be seen from the reduction $\langle w\rangle \mathtt{C}t \to \langle w+1\rangle t$. Choosing different weights for constructors could be useful in cases such as

```
val rec f = fun
    A[A[A[A[A[B[x]]]]]] ->  f A[A[A[A[A[C[C[x]]]]]]]
  | A[A[A[A[A[C[x]]]]]] ->  f A[A[A[A[A[x]]]]]
  | _ -> A[]
```

This function is size-change terminating if the bound D is greater than 7. If the definition contained other recursive calls, it can make the testing procedure use more resources than reasonable. Giving a weight of 3 to `B` and 1 to `C` would make this function size-change terminating, even when $D = 0$. Trying to choose the appropriate weights automatically might not be worth the trouble but this is still an interesting question.

Counting abstractions. The PML language for which this criterion was developed is more complete than the ambient language presented here. In particular, function abstractions and partially applied functions are allowed. Like OCaml, PML only computes weak-head normal forms and the function

```
val rec glutton x = glutton
```

terminates: when applied to n arguments, it discards all of them and stops on the weak-head normal form `fun x -> glutton`.

We can make such functions size-change terminating by adding a virtual parameter $\mathtt{x}_{ac}$ to all functions. This parameter counts the difference between the number of abstraction and the number of applications above the call-site: it gives the "applicative context" of the call. This parameter records an additional constructor "`App`" introduced by function application and removed ("$\texttt{App}^-$") by function abstraction. The previous function is

```
val rec glutton = fun x -> glutton
```

which contains an abstraction and no application. The corresponding arc in the control-flow graph will thus be $[\,\mathtt{x}_{ac} := \mathtt{App}^-\mathtt{x}_{ac}\,]$, where $\mathtt{x}_{ac}$ is the virtual parameter giving the applicative context of the call. This virtual parameter makes the definition size-change terminating (for any choice of bounds B and D).

This is interesting because dummy abstractions and applications is the usual way to freeze evaluation and define "infinite" data structures in OCaml.[2] In this context, the size-change termination principle can be used to detect some notion of *productivity*. For example, let the type of infinite streams of integers be the coinductive type `S` = `unit -> int*S` where `unit` is the type with a single constructor `U[]`. The stream of all even integers can be defined with

```
val rec arith n d = fun _ -> (n, arith (n+d) d)
val even = arith Z[] S[S[Z[]]]
```

The call "`arith n r`" constructs the stream of integers in arithmetic progression, starting from `n` with common difference `d`. ("`_`" stands for a dummy variable and "`+`" stands for the addition of unary natural numbers.) The following definition then corresponds to the `map` function on streams:

```
val rec map_stream f s = fun _ ->
  match s U[] with
    (n, s) -> (f n, map_stream f s)
```

Like `glutton`, the functions `arith` and `map_stream` have a deficit of applications: the call-sites are bellow 3 abstractions but only 2 applications. The parameter $\mathtt{x}_{ac}$ is thus represented by $\mathtt{App}^-\mathtt{App}^-\mathtt{App}^-\mathtt{App}\,\mathtt{App}\mathtt{x}_{ac} \approx \mathtt{App}^-\mathtt{x}_{ac}$. Those functions are size-change terminating with this extension: their control-flow graphs consist of a single node with label

- $[\,\mathtt{x}_{ac} := \mathtt{App}^-\mathtt{x}_{ac}\,;\, \mathtt{n} := \langle\infty\rangle()\,;\, \mathtt{d} := \mathtt{d}\,]$ for `arith`
- $[\,\mathtt{x}_{ac} := \mathtt{App}^-\mathtt{x}_{ac}\,;\, \mathtt{f} := \mathtt{f}\,;\, \mathtt{s} := \langle\infty\rangle()\,]$ for `map_stream`.

It is possible to mix finite (inductive) and infinite (coinductive) structures: here is the function that removes a given number of 0s in a stream of integers.

```
val rec remove_zeros n s =
  match n with
      Z[] -> s
    | S[m] -> (match s U[] with
                 (Z[], s) -> remove_zeros m s
               | (S[h], s) -> fun _ -> (S[h], remove_zeros n s))
```

The function `remove_zeros` is size-change terminating with this extension and works for arbitrary streams, i.e., even for those that do not contain any 0. Its control-flow graph contains two loops:

- $[\,\mathtt{x}_{ac} := \mathtt{x}_{ac}\,;\, \mathtt{n} := \mathtt{S}^-\mathtt{n}\,;\, \mathtt{s} := \langle\infty\rangle()\,]$
- $[\,\mathtt{x}_{ac} := \mathtt{App}^-\mathtt{x}_{ac}\,;\, \mathtt{n} := \mathtt{n}\,;\, \mathtt{s} := \langle\infty\rangle()\,]$.

A more complete investigation of this phenomenon is pending...

[2]Refer to the implementation of the "`Lazy`" module.

2.6.1. *Higher-Order Arguments.* The PML language for which the size-change termination principle was implemented allows higher-order arguments for functions. It is not possible to just ignore higher order arguments: the definition

```
val app_zero f = f Z[]
val rec f x = app_zero f
```

might be seen as terminating!

To deal with those, the simplest is to have the static analysis to tag each instance of a recursively defined function appearing as an *argument* of another function as non terminating. This makes it possible to define all the usual functions that have functions in their parameters, like the real `map` function:

```
val rec map f x = match x with Nil[]  ->  Nil[]
                             | Cons[a,y]  ->  Cons[f a, map f y]
```

whose control-flow graph consists of a single loop $[\,\mathtt{f} := \mathtt{f}\,;\,\mathtt{x} := \pi_2\mathtt{Cons}^{\text{-}}\mathtt{x}\,]$. It is also possible to think of smarter static analysis that would see that the definition

```
val phi f = fun n -> math n with Z[] -> Z[]
                               | S[m] -> n + f m
val rec f x = phi f x
```

is size-change terminating. The PML language uses a constraint checking algorithm to check that the definitions are well formed, i.e., that their semantics is well defined. This algorithm builds a kind of *data-flow graph* to compute an accessibility relation between different parts of the code and check, for example, that tuples never reach a "`match`" [12]. The static analysis is inferred from this data-flow graph, and it detects that the function `phi` defined previously acts in such a way that "`phi f` u" may only yield a call "$\mathtt{f}\mathtt{S}^{\text{-}}u$". Because of this the control-flow graph of the function `f` will contain a single loop $[\,\mathtt{x} := \mathtt{S}^{\text{-}}\mathtt{x}\,]$, and will thus pass the termination test.

Unfortunately, we currently don't have a proof that this static analysis is safe! We are currently working on this aspect and are trying to unify the "data-flow graph" used for checking that a definition is well-formed and the "control-flow graph" used for the SCT.

Finishing the proof that this analysis is safe is interesting because it allows for a very powerful static analysis. As an example, the following piece of code is accepted as terminating in the PML language:

```
val rec map f l = (* map on lists *)
  match l with
      Nil[] -> Nil[]
    | Cons[a,l] -> Cons[f a, map f l]
type rec rose_tree A = [  Node[A * list(rose_tree A)] ]
val rec rmap f t = (* map on rose trees *)
  match t with
      Node[a,l] -> Node[ f a , map (rmap f) l ]
```

The data-flow analysis detects that the list `l` contains trees that are smaller than `t` and that those elements are fed to the partially applied `rmap f`. The control flow-graph for `rmap` contains a single loop $[\mathtt{f} := \mathtt{f}; \mathtt{t} := \pi_1\mathtt{Cons}^{\text{-}}\pi_2\mathtt{Node}^{\text{-}}\mathtt{t}]$. This will be enough for the termination criterion to accept the function.

References

1. Andreas Abel and Thorsten Altenkirch, *A predicative analysis of structural recursion*, Journal of Functional Programming **12** (2002), 1–41.

2. Amir Ben-Amram, *Size-change termination with difference constraints*, ACM Transactions on Programming Languages and Systems **30** (2008), no. 3, 1–31.
3. Neil D. Jones and Nina Bohr, *Call-by-value termination in the untyped lambda-calculus*, Logical Methods in Computer Science **4** (2008), no. 1.
4. Alexander Krauss, *Certified size-change termination*, 11th International Conference on Automated Deduction, LNAI, Springer-Verlag, July 2007.
5. Alexander Krauss and Armin Heller, *A mechanized proof reconstruction for SCNP termination*, Presented in the Tenth International Workshop on Termination WST'09, Leipzig, 2009.
6. Chin Soon Lee, Neil D. Jones, and Amir Ben-Amram, *The size-change principle for program termination*, Symposium on Principles of Programming Languages, vol. 28, ACM press, january 2001, pp. 81–92.
7. Panagiotis Manolios and Daron Vroon, *Termination analysis with calling context graphs*, Computer Aided Verification (Thomas Ball and Robert Jones, eds.), Lecture Notes in Computer Science, vol. 4144, Springer Berlin / Heidelberg, 2006, pp. 401–414.
8. et all Markus Wenzel, *The isabelle/isar reference manual*, 2007.
9. The Coq development team, *The coq proof assistant reference manual*, LogiCal Project, 2004.
10. Robin Milner, *A theory of type polymorphism in programming*, Journal of Computer and System Sciences **17** (1978), 348–375.
11. Ulf Norell, *Dependently typed programming in agda*, In Lecture Notes from the Summer School in Advanced Functional Programming, 2008.
12. Christophe Raffalli, *Realizability for programming languages*, course notes for the *École jeunes chercheurs du GDR IM*, submitted, 2010.
13. Damien Sereni and Neil D. Jones, *Termination analysis of higher-order functional programs*, Proceedings of the Third Asian conference on Programming Languages and Systems (Berlin, Heidelberg), APLAS'05, Springer-Verlag, 2005, pp. 281–297.

Appendix A. Other Definition of the Approximation Preorder

We give a more concrete characterization of the approximation preorder. This is crucial in the implementation of the termination test but is also used in the proof of Lemma 1.7. The proofs are rather verbose and not very surprising.

Definition A.1. The relation $\sqsubseteq$ is the relation on terms in normal forms generated by

$$\frac{u \sqsubseteq v}{\mathtt{C}u \sqsubseteq \mathtt{C}v}(1) \qquad \frac{u_1 \sqsubseteq v_1 \quad \dots \quad u_n \sqsubseteq v_n}{(u_1, \dots, u_n) \sqsubseteq (v_1, \dots, v_n)}(2)$$

$$\frac{\mathrm{nf}\left(\langle 0\rangle u\right) \sqsubseteq \mathrm{nf}\left(\langle w\rangle v\right)}{u \sqsubseteq \mathrm{nf}\left(\langle w\rangle v\right)}(3)$$

$$\frac{\forall i = 1, \dots, n\ \exists j = 1, \dots, m \quad \langle w_i\rangle\overline{d_i} \sqsubseteq \langle w'_j\rangle\overline{b_j}}{\sum_{i=1}^n \langle w_i\rangle\overline{d_i} \sqsubseteq \sum_{j=1}^m \langle w'_j\rangle\overline{b_j}}(4)$$

$$\frac{\overline{d} \text{ is a suffix of } \overline{b} \text{ and } w' + |\overline{d}| \leqslant w + |\overline{b}|}{\langle w'\rangle\overline{b} \sqsubseteq \langle w\rangle\overline{d}}(5) \qquad \frac{}{\overline{d} \sqsubseteq \overline{d}}(6)$$

where we identify $\mathbf{0}$ *and the empty sum.*

We will usually drop the nf(_) and reason up-to $\approx$.

Lemma A.2. *For every* $u \sqsubseteq v$ *and* $w \in \mathbf{Z}_\infty$*, we have* $\langle w\rangle u \sqsubseteq \langle w\rangle v$.

Proof. By induction on the proof that $u \sqsubseteq v$:

- if the last rule used for $u \sqsubseteq v$ was (1), then $u = \mathtt{C}u$ and $v = \mathtt{C}v$ and we have $u \sqsubseteq v$. By induction, we have $\langle w\rangle u \sqsubseteq \langle w\rangle v$ for all $w \in \mathbf{Z}_\infty$. From this, we conclude that $\langle w\rangle \mathtt{C}u \approx \langle w+1\rangle u \sqsubseteq \langle w+1\rangle v \approx \langle w\rangle \mathtt{C}v$.
- If the last rule was (2), then $u = (u_1, \dots, u_n)$ and $v = (v_1, \dots, v_n)$ and $u_i \sqsubseteq v_i$ for all $i = 1, \dots, n$. By induction, we get $\langle w\rangle u_i \sqsubseteq \langle w\rangle v_i$ for all $i = 1, \dots, n$ and $w \in \mathbf{Z}_\infty$. We thus have $\langle w\rangle(u_1, \dots, u_n) \approx \sum_i \langle w+1\rangle u_i \sqsubseteq \sum_i \langle w+1\rangle v_i \approx \langle w\rangle(v_1, \dots, v_n)$.
- If the last rule was (3), then $v \approx \langle w'\rangle v$ and $\langle 0\rangle u \sqsubseteq \langle w'\rangle v$. By induction, we get $\langle w\rangle u \approx \langle w\rangle\langle 0\rangle u \sqsubseteq \langle w\rangle\langle w'\rangle v$ for all $w \in \mathbf{Z}_\infty$.
- If the last rule was (4), then $u = \sum_i \langle w_i\rangle\overline{d_i}$ and $v = \sum_j \langle w'_j\rangle\overline{b_j}$ and forall i, there is a j s.t. $\langle w_i\rangle\overline{d_i} \sqsubseteq \langle w'_j\rangle\overline{b_j}$. By induction, for all i, there is a j s.t. $\langle w\rangle\langle w_i\rangle\overline{d_i} \sqsubseteq \langle w\rangle\langle w'_j\rangle\overline{b_j}$. This implies that $\sum_i \langle w\rangle\langle w_i\rangle\overline{d_i} \sqsubseteq \sum_j \langle w\rangle\langle w'_j\rangle\overline{b_j}$ and because the left side is equal to $\langle w\rangle u$ and the right side is equal to $\langle w\rangle v$, we get $\langle w\rangle u \sqsubseteq \langle w\rangle v$.
- If the last rule was (5), then $u = \langle w''\rangle\overline{b}$ and $v = \langle w'\rangle\overline{d}$ with $\overline{d}$ a suffix of $\overline{b}$ and $w'' + |\overline{b}| \leqslant w' + |\overline{d}|$. We have $\langle w\rangle\langle w''\rangle\overline{b} \approx \langle w+w''\rangle\overline{b} \sqsubseteq \langle w+w'\rangle\overline{d} \approx \langle w\rangle\langle w'\rangle\overline{d}$ because $\overline{d}$ is a suffix of $\overline{b}$ and $w + w'' + |\overline{b}| \leqslant w + w' + |\overline{d}|$.
- If the last rule was (6), then $u = v = \overline{d}$. We have $< w > \overline{d} \sqsubseteq \langle w\rangle\overline{d}$ because $\overline{d}$ is a suffix of $\overline{d}$ and $w + |\overline{d}| \leqslant w + |\overline{d}|$.

□

Lemma A.3. *The relation $\sqsubseteq$ is transitive.*

Proof. We prove that $u_1 \sqsubseteq u_2$ and $u_2 \sqsubseteq u_3$ implies $u_1 \sqsubseteq u_3$ (where each u_i is in normal form) by induction on the proofs of $u_2 \sqsubseteq u_3$ and $u_1 \sqsubseteq u_2$. We look at the last rule of $u_2 \sqsubseteq u_3$:

- If the last rule was (1), then u_2 is of the form $\mathtt{C}v_2$, and the last rule of $u_1 \sqsubseteq u_2$ is necessarily (1). We thus have $v_1 \sqsubseteq v_2 \sqsubseteq v_3$, which implies by induction, that $v_1 \sqsubseteq v_3$. This implies that $\mathtt{C}v_1 \sqsubseteq \mathtt{C}v_3$.
- If the last rule of $u_2 \sqsubseteq u_3$ is (2), the proof is similar.
- If the last rule in $u_2 \sqsubseteq u_3$ was (3), we have $u_3 = \langle w\rangle v_3$ and $\langle 0\rangle u_2 \sqsubseteq \langle w\rangle v_3$. By Lemma A.2, we have $\langle 0\rangle u_1 \sqsubseteq \langle 0\rangle u_2$, and we know by induction that $\langle 0\rangle u_1 \sqsubseteq \langle w\rangle v_3$. We get $u_1 \sqsubseteq \langle w\rangle v_3$ by rule (3).
- If the last rule in $u_2 \sqsubseteq u_3$ was (4), then u_2 is of the form $\sum_j \langle w_{2,j}\rangle\overline{d_{2,j}}$ and u_3 is of the form $\sum_k \langle w_{3,k}\rangle\overline{d_{3,k}}$ and we have $\forall j, \exists k, \langle w_{2,j}\rangle\overline{d_{2,j}} \sqsubseteq \langle w_{3,k}\rangle\overline{d_{3,k}}$. We look at the last rule of the proof that $u_1 \sqsubseteq u_2$:
  - if $u_1 \sqsubseteq u_2$ ended with (3), we have $\langle 0\rangle u_1 \sqsubseteq \langle 0\rangle u_2 \approx u_2 \sqsubseteq u_3$. By induction hypothesis, $\langle 0\rangle \sqsubseteq u_3 \approx \langle 0\rangle u_3$. We conclude that $u_1 \sqsubseteq u_3$ by rule (3).
  - if $u_1 \sqsubseteq u_2$ ended with (4), then u_1 is of the form $\sum_i \langle w_{1,i}\rangle\overline{d_{1,i}}$ and for all i, there is a j s.t. $\langle w_{1,i}\rangle\overline{d_{1,i}} \sqsubseteq \langle w_{2,j}\rangle\overline{d_{2,j}}$. By the previous remark about $u_2 \sqsubseteq u_3$, we thus have $\forall i, \exists k \langle w_{1,i}\rangle\overline{d_{1,i}} \sqsubseteq \langle w_{3,k}\rangle\overline{d_{3,k}}$. We conclude that $u_1 \sqsubseteq u_3$ by rule (4).
  - if $u_1 \sqsubseteq u_2$ ended with (5), then u_2 is a sum with a single summand and we have $u_1 \sqsubseteq u_2 \sqsubseteq \langle w_{3,k}\rangle\overline{d_{3,k}}$ for some k. We have $u_1 \sqsubseteq \langle w_{3,k}\rangle\overline{d_{3,k}}$ by induction and we get $u_1 \sqsubseteq \sum_k \langle w_{3,k}\rangle\overline{d_{3,k}}$ by rule (4).
- If the last rule in $u_2 \sqsubseteq u_3$ was (5), then $u_3 = \langle w_3\rangle\overline{d_3}$ and $u_2 = \langle w_2\rangle\overline{d_2}$. We look at the last rule of the proof that $u_1 \sqsubseteq u_2$:
  - the proof that $u_1 \sqsubseteq u_2$ ended with (3). We have $\langle 0\rangle u_1 \sqsubseteq \langle w_2\rangle\overline{d_2} \sqsubseteq \langle w_3\rangle\overline{d_3}$, and thus, by induction, that $\langle 0\rangle u_1 \sqsubseteq \langle w_3\rangle\overline{d_3}$. We can use rule (3) to deduce that $u_1 \sqsubseteq \langle w_3\rangle\overline{d_3}$.
  - the proof that $u_1 \sqsubseteq u_2$ ended with rule (4), with $u_1 = \sum_i \langle w_{1,i}\rangle\overline{d_{1,i}}$. We thus have $\langle w_{1,i}\rangle\overline{d_{1,i}} \sqsubseteq \langle w_2\rangle\overline{d_2}$ for all is. By induction, we get $u_{1,i} \sqsubseteq \langle w_3\rangle\overline{d_3}$ for all is, and we can conclude that $\sum_i u_{1,i} \sqsubseteq \langle w_1\rangle\overline{d_3}$.
  - the proof that $u_1 \sqsubseteq u_2$ ended with rule (5). We have $u_1 = \langle w_1\rangle\overline{d_1}$ and we get $\langle w_1\rangle\overline{d_1} \sqsubseteq \langle w_3\rangle\overline{d_3}$ by rule (5).
- If the last rule of $u_2 \sqsubseteq u_3$ is (6), then the last rule of $u_1 \sqsubseteq u_2$ is necessarily (6). We have $u_1 = u_2 = u_3 = \overline{d}$. Transitivity holds by rule (6).

□

Lemma A.4. *For every u in normal form, we have:*

1. *for every sequence of destructors $d_1 \cdots d_k$, we have $d_1 \cdots d_k u \sqsubseteq \langle -n\rangle u$,*
2. *for every sequence of destructors $d_1 \cdots d_k$, if $u \sqsubseteq v$, then $d_1 \cdots d_k u \sqsubseteq d_1 \cdots d_k v$,*
3. *for every $t_1 \sqsubseteq t_2$, $u[\mathtt{x} := t_1] \sqsubseteq u[\mathtt{x} := t_2]$.*

Proof. The first point is a simple induction on k:

- if $k = 0]$, the result amounts to $u \sqsubseteq \langle 0\rangle u$. It follows from rule (3), (4) and (5).
- if $\overline{d} = d_1 \cdots d_k \cdot d_{k+1}$: suppose that $d_{k+1} = \mathtt{C}^-$. We look at u:
  - if $u = \mathtt{C}v$, we have $d_1 \cdots d_k \cdot \mathtt{C}^-\mathtt{C}v \approx d_1 \cdots d_k v \sqsubseteq \langle -k\rangle v \approx \langle -k-1\rangle \mathtt{C}v$, where the "inequality" comes from the induction hypothesis.

- if $u = \mathtt{D} \neq \mathtt{C}$ or $u = (v_1, \dots, v_n)$, we have $d_1 \cdots d_{k+1} u \approx \mathbf{0} \sqsubseteq \langle -k-1 \rangle u$ by rule (4) of the definition of $\sqsubseteq$.
- if $u = \overline{b}\,\mathtt{x}$ we need to show that $d_1 \cdots d_{k+1} \cdot \overline{b}\,\mathtt{x} \sqsubseteq \langle -k-1 \rangle \overline{b}\,\mathtt{x}$. By rule (3) of the definition of $\sqsubseteq$, it is enough to show that $\langle 0 \rangle d_1 \cdots d_{k+1} \cdot \overline{b}\,\mathtt{x} \sqsubseteq \langle -k-1 \rangle \overline{b}\,\mathtt{x}$. This holds by rule (5).
- if $u = \langle w \rangle \overline{b}\,\mathtt{x}$ we need to show $d_1 \cdots d_{k+1} \langle w \rangle \overline{b}\,\mathtt{x} \approx \langle w-k-1 \rangle \overline{b}\,\mathtt{x}$ is approximated by $\langle -k-1 \rangle \langle w \rangle \overline{b}\,\mathtt{x} \approx \langle w-k-1 \rangle \overline{b}\,\mathtt{x}$. This holds by rule (5).

The proof is similar when $d_{k+1} = \pi_i$.

The second point is an induction on the proof that $t_1 \sqsubseteq t_2$. If $k = 0$, the result holds trivially. Otherwise, let's assume that the sequence of destructors is of the form $d_1 \cdots d_k \mathtt{C}^{-}$.

- If the proof that $t_1 \sqsubseteq t_2$ ended with rule (1), with the same constructor $\mathtt{C}$, we have $u = \mathtt{C}u'$ and $v = \mathtt{C}v'$ with $u' \sqsubseteq v'$. By induction, we can thus conclude that $d_1 \cdots d_k \mathtt{C}^{-} u \approx d_1 \cdots d_k u' \sqsubseteq d_1 \cdots d_k v' \approx d_1 \cdots d_k \mathtt{C}^{-} v$.
- If the proof that $t_1 \sqsubseteq t_2$ ended with rule (1) but with a different constructor, or with rule (2), both $d_1 \cdots d_k \mathtt{C}^{-} u$ and $d_1 \cdots d_k \mathtt{C}^{-} v$ reduce to $\mathbf{0}$, and we conclude with rule (4).
- If the proof that $u \sqsubseteq v$ ended with (3), we know that $v \approx \langle w \rangle v'$ and $\langle 0 \rangle u \sqsubseteq \langle w \rangle v'$. We have $d_1 \cdots d_k \mathtt{C}^{-} \langle 0 \rangle u \approx \langle -k-1 \rangle u \sqsubseteq d_1 \cdots d_k \mathtt{C}^{-} \langle w \rangle v' \approx \langle w-k-1 \rangle v'$ by induction. By the previous point, we also have $d_1 \cdots d_k \mathtt{C}^{-} u \sqsubseteq \langle -k-1 \rangle u$, and by transitivity, we conclude that $d_1 \cdots d_k \mathtt{C}^{-} u \sqsubseteq d_1 \cdots d_k \mathtt{C}^{-} \langle w \rangle v'$.
- If the proof that $u \sqsubseteq v$ ended with (4), we just need to apply the induction hypothesis and rule (4).
- If the proof that $u \sqsubseteq v$ ended with (5) or (6), we can conclude directly.

The proof is similar when the sequence of destructors ends with π_i.

The third point is a simple inductive proof on u:

- if u is $\mathtt{C}v$ or $(v_1, \dots, v_n)$, we just need the induction hypothesis and rules (1) or (2).
- If u is $\langle w \rangle \overline{d}\,\mathtt{y}$ with $\mathtt{y} \neq \mathtt{x}$, we just need rule (5). If $\mathtt{y} = \mathtt{x}$, we need the induction hypothesis and Lemma A.2.
- If u is $\overline{d}\,\mathtt{y}$ with $\mathtt{y} \neq \mathtt{x}$, we just need rule (6). If $\mathtt{y} = \mathtt{x}$, we use the previous point.

□

The relation $\sqsubseteq$ isn't quite the same as $\preccurlyeq$ because it doesn't interact with $+$. For example, we don't have $\mathtt{x} \sqsubseteq \mathtt{x} + \mathtt{y}$ or $\mathtt{Cx} + \mathtt{Dx} \sqsubseteq \langle 1 \rangle \mathtt{x}$. However, we have:

Lemma A.5. *We have $u \preccurlyeq v$ if and only if u can be written as $\sum_i u_i$ and v can be written as $\sum_j v_j$ with $\forall i, \exists j, u_i \sqsubseteq v_j$.*

Proof. To show that if $u \preccurlyeq v$ then u and v can be written as sums as in the lemma, we define a new relation $u \sqsubseteq' v$ as "*u can be written as $\sum_i u_i$, v can be written as $\sum_j v_j$ and $\forall i, \exists j, u_i \sqsubseteq v_j$*". We need to prove that (refer to Definition 1.6 page 7):

- $\sqsubseteq'$ is a preorder,
- $\sqsubseteq'$ is contextual,
- $\sqsubseteq'$ is compatible with $\approx$,
- $\sqsubseteq'$ is compatible with $+$,
- if $w \leqslant w'$ in $\mathbf{Z}_\infty$, then $\langle w \rangle t \sqsubseteq' \langle w' \rangle t$,
- $t \sqsubseteq' \langle 0 \rangle t$.

Since $\preccurlyeq$ is the least such relation, we will get that $u \preccurlyeq v$ implies $u \sqsubseteq' v$. We only sketch the proofs:

- $\sqsubseteq'$ is transitive because $\sqsubseteq$ is transitive (Lemma A.3).
- $\sqsubseteq'$ is reflexive because $\sqsubseteq$ is reflexive on simple terms (easy inductive proof).
- $\sqsubseteq'$ is contextual because $\sqsubseteq$ is contextual (Lemma A.4).
- $\sqsubseteq'$ is compatible with $\approx$ because by definition, $\sqsubseteq$ is compatible with $\approx$.
- $\sqsubseteq'$ is compatible with $+$ by definition.
- That $\langle w\rangle t \sqsubseteq \langle w'\rangle t$ when $w \leqslant w'$ in $\mathbf{Z}_\infty$ is an easy inductive proof. It lifts to $\sqsubseteq'$.
- We have $t \sqsubseteq \langle 0\rangle t$ by Lemma A.2, and this property lifts to $\sqsubseteq'$.

The proof that $\forall i, \exists j, u_i \sqsubseteq v_j$ implies that $\sum_i u_i \preccurlyeq \sum_j v_j$ is left as an exercise. It amounts to showing that all the rule for $\sqsubseteq$ are valid for $\preccurlyeq$ and that $\forall i, \exists j, u_i \preccurlyeq v_j$ implies $\sum_i u_i \preccurlyeq \sum_j v_j$. □

Note that some care is needed to use this lemma to decide approximation on arbitrary terms. Since $+$ is associative, commutative and idempotent, there is a choice to make when writing v as a sum. For example, we have $\mathtt{A}(\mathtt{x},\mathtt{y}) + \mathtt{B}(\mathtt{x},\mathtt{z}) \preccurlyeq v = \langle 2\rangle\mathtt{x} + \langle 2\rangle\mathtt{y} + \langle 2\rangle\mathtt{z} + \langle 1\rangle()$ because we can write v as "$\big(\langle 2\rangle\mathtt{x} + \langle 2\rangle\mathtt{y}\big) + \big(\langle 2\rangle\mathtt{x} + \langle 2\rangle\mathtt{z}\big) + \dots$", and we have:

- $\mathtt{A}(\mathtt{x},\mathtt{y}) \sqsubseteq \langle 2\rangle\mathtt{x} + \langle 2\rangle\mathtt{y}$,
- $\mathtt{B}(\mathtt{x},\mathtt{y}) \sqsubseteq \langle 2\rangle\mathtt{x} + \langle 2\rangle\mathtt{z}$.

Appendix B. Implementation Issues

In order to make the presentation readable, the paper followed a rather abstract description of the criterion. The initial goal was to get a concrete termination checker for the PML language [12] and ease of implementation was very important. The code for the criterion can be found at `http://lama.univ-savoie.fr/~hyvernat/Files/basic-SCT.tar.gz`: it consists of the implementation done for the PML language with a very simple static analysis for a very simple language. (There are no dependencies for this.) The full code of PML can be found at `http://lama.univ-savoie.fr/~pml/`.

The main points that make the task relatively straightforward are the following:

(1) we only manipulate terms in normal forms and use a representation similar to the grammar given in Lemma 1.5,
(2) computing if $t \preccurlyeq u$ and if $t \circ u$ is easy for those terms,
(3) checking if a loop is decreasing (Definition 2.5) is easy.

Even for terms in normal forms, we need a uniform way to deal with sums. As n-tuples are n-linear, applying linearity to get sums of simple terms can lead to an exponential blow-up and was ruled out. We instead start by making sure the initial control-flow graph doesn't contain any sum. In order to do that, we replace each arc labeled by a sum with as many arcs as summands. No exponential blow-up occurs in practice because PML's static analysis doesn't introduce sums. Then, sums only appear through collapsing of compositions, i.e. from the reduction rule $\langle w\rangle(t_1,\dots,t_n) \to \sum_i \langle w+1\rangle t_i$. Those sums can always be pushed under all constructors and all summands start with a $\langle w\rangle$. We thus use the following grammar for terms:

$$\begin{array}{rcl} t & ::= & \mathtt{C}t \quad | \quad (t_1,\dots,t_n) \quad | \quad \overline{d} \quad | \quad \sum_i \langle w_i\rangle \overline{d_i} \\ \overline{d} & ::= & \mathtt{x} \quad | \quad \pi_i \overline{d} \quad | \quad \mathtt{C}^{-}\overline{d} \end{array}$$

where the sums are not empty. Note that $()$ isn't part of the grammar. It was only used as a presentational artifact and can be removed from the implementation. Its only concrete use was to represent an argument whose shape in unknown: $\langle\infty\rangle()$. In the implementation, we use $\sum_{1\leqslant i\leqslant a}\langle\infty\rangle\mathtt{x}_i$ instead, where a is the arity of the calling function.

All the substitutions are in normal form and composition needs to do some reduction. This is done using the rules from Definition 1.2, with a particular proviso for group (3):

- rules $\pi_i\mathtt{C}t \to \mathbf{0}$, $\mathtt{C}^{-}(t_1,\dots,t_n) \to \mathbf{0}$ and $\pi_i(t_1,\dots,t_n) \to \mathbf{0}$ (when $i > n$) all raise an *error* `TypingError`. Encountering such a reduction means that the definitions where not valid to begin with and that the initial type-checking / constraint solving of the definitions is broken.
- the rule $\mathtt{C}^{-}\mathtt{D}t \to \mathbf{0}$ raises an *exception* `ImpossibleCase`. Even safe definitions may introduce such reductions, but we know that evaluation will never go along such a path: evaluation of `match v with ...` may only enter a branch if the corresponding pattern matches `v`. Compositions raising this exception are simply ignored.

B.1. **Order, Compatibility and Decreasing Arguments.** When the terms are generated by the above grammar, we can give an inductive definition of both the order and the compatibility relation. The inductive definition of the order corresponds in fact to Definition A.1: $\sqsubseteq$ is exactly the restriction of $\preccurlyeq$ on the terms used in the implementation.

Checking compatibility for arbitrary terms can be subtle. For example, we have

$$\big(\langle 0\rangle s,(u,v)\big)+\big(\langle 0\rangle t,(u,v)\big) \quad \supset \quad \big((s,t),\langle 0\rangle u\big)+\big((s,t),\langle 0\rangle v\big)$$

even though no summand on the left is compatible with a summand on the right. However, for the restriction used in the implementation, we can give a purely inductive definition of compatibility:

Lemma B.1. *Compatibility on terms given by the grammar on page 28 is generated by the following rules:*

$$\frac{u \supset v}{\mathtt{C}u \supset \mathtt{C}v}(1) \qquad \frac{u_1 \supset v_1 \quad \dots \quad u_n \supset v_n}{(u_1,\dots,u_n) \supset (v_1,\dots,v_n)}(2)$$

$$\frac{u \supset \sum_{j=1}^{m}\langle w_j\rangle\overline{d_j}}{\mathtt{C}u \supset \sum_{j=1}^{m}\langle w_j\rangle\overline{d_j}}(3) \qquad \textit{and symmetric}$$

$$\frac{\forall i=1,\dots,n \quad u_i \supset \sum_{j=1}^{m}\langle w_j\rangle\overline{d_j}}{(u_1,\dots,u_n) \supset \sum_{j=1}^{m}\langle w_j\rangle\overline{d_j}}(4) \qquad \textit{and symmetric}$$

$$\frac{\exists i=1,\dots,n\ \exists j=1,\dots,m \quad \langle w_i\rangle\overline{d_i} \supset \langle w'_j\rangle\overline{b_j}}{\sum_{i=1}^{n}\langle w_i\rangle\overline{d_i} \supset \sum_{j=1}^{m}\langle w'_j\rangle\overline{b_j}}(4)$$

$$\frac{\overline{d}\ \textit{is a suffix of}\ \overline{b}\ \textit{or}\ \overline{b}\ \textit{is a suffix of}\ \overline{d}}{\langle w'\rangle\overline{b} \supset \langle w\rangle\overline{d}}(5) \qquad \frac{}{\overline{d} \supset \overline{d}}(6)$$

Both definitions can be implemented easily using ML pattern matching.

Looking for decreasing arguments in a substitution is simple: the minimality condition means that a decreasing argument is a subterm of one component of the substitution. It is thus enough to check all subterms!

B.2. **Complexity.** We saw in section 2.4 that the problem of deciding size-change termination is P-space hard. In practice, we have found the algorithm described on page 18 to perform quite well. In our experience, we have found that checking termination of functions written by hand in PML doesn't require too much resources. There are concrete examples where D needs to be more than 4, but choosing a bound B greater than 1 is very rarely necessary. The default is to have $D = 2$ and $B = 1$, and let the user change the bounds. With this default, termination checking is an order of magnitude faster than sanity checking of the definitions, except for those examples specifically designed to stress the system.

There are however two points that help make the criterion perform well, especially when the bounds B and D are greater than their default values:

- we make sure that sums are minimal by keeping only maximal summands: $\overline{d}$ is equivalent ("$\preccurlyeq$ and $\succcurlyeq$") to $\overline{d} + \langle 12\rangle\overline{d} + \langle -1\rangle \mathtt{C}^{\texttt{-}}\overline{d}$ and is a much better choice because it keeps the size of the graph smaller.
- since everything is monotonic with respect to $\preccurlyeq$, we don't need to keep arcs that are approximated by another arc ("subsumption").

These points are trivial to implement and lower the complexity of the algorithm in practice.

Appendix C. Static Analysis

The simplest interesting static analysis only records pattern matching and projection: for each call-site "$\mathtt{g}\ u_1\ \ldots\ u_m$" in the definition of "$\mathtt{f}\ \mathtt{x}_1\ \ldots\ \mathtt{x}_n$", we construct the substitution $[\mathtt{y}_1 := u_1; \ldots ; \mathtt{y}_m := u_m]$ where each $u_i \in \mathcal{T}(\mathtt{x}_1, \ldots, \mathtt{x}_n)$ is

- a simple term without $\langle w\rangle$s if u_i is syntactically built from projections and pattern-matching variable coming from $\mathtt{x}_1$, ..., $\mathtt{x}_n$;
- $\langle\infty\rangle()$ otherwise.

For example, all the examples `map`, $\mathtt{f}_1$, $\mathtt{g}_1$, $\mathtt{f}_2$ and `push_left` (page 3 and 4) yield substitutions without $\langle\infty\rangle()$. For the `ack` function however, the three recursive calls are represented by:

- $[\mathtt{x}_1 := \mathtt{S}^{\texttt{-}}\mathtt{x}_1; \mathtt{SZ}()]$,
- $[\mathtt{x}_1 := \mathtt{SS}^{\texttt{-}}\mathtt{x}_1; \mathtt{x}_2 := \mathtt{S}^{\texttt{-}}\mathtt{x}_2]$,
- $[\mathtt{x}_1 := \mathtt{S}^{\texttt{-}}\mathtt{x}_1; \mathtt{x}_2 := \langle\infty\rangle()]$.

The "$\langle\infty\rangle()$" comes from the call "`ack m (ack ...)`": because the second argument is an application, it isn't syntactically built from the parameters. Note that this doesn't prevent the criterion from tagging the `ack` function as terminating.

It should be noted that this static analysis is entirely syntactical and can be done in linear time in the size of the recursive definitions. This is similar to the static analysis done in http://lama.univ-savoie.fr/~hyvernat/Files/basic-SCT.tar.gz. The only differences are that:

- the syntax of the definitions is much simpler,
- we use $\langle\infty\rangle\mathtt{x}_1 + \cdots + \langle\infty\rangle\mathtt{x}_a$ instead of $\langle\infty\rangle()$.