



HAL
open science

The Size-Change Termination Principle for Constructor Based Languages

Pierre Hyvernât

► **To cite this version:**

Pierre Hyvernât. The Size-Change Termination Principle for Constructor Based Languages. 2013.
hal-00547440v3

HAL Id: hal-00547440

<https://hal.science/hal-00547440v3>

Preprint submitted on 17 Jun 2013 (v3), last revised 2 Jan 2014 (v5)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

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 value are built from variants and tuples to keep some information about how arguments of recursive call evolve during evaluation.

The result is a criterion for termination extending the *size-change termination principle* of Lee, Jones and Ben-Amram [4] 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 (some) mutually recursive definitions written in a first-order call-by-value language in ML style. The problem is of course undecidable in general but there is a constant struggle for finding stronger sufficient conditions that are decidable. Lee, Jones and Ben-Amram’s size-change termination principle (SCT) is a simple, yet surprisingly strong sufficient condition for termination of programs [4]. 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 algorithm checking that such a graph is “terminating” amounts to a (conceptually) simple transitive closure computation.

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 variant 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 points is that because we know some of

Received by the editors June 17, 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.

the constructors present in the arguments, it is possible to ignore some path in the control-flow graph because they can not correspond to real evaluation steps. Moreover, it makes it possible to inspect *subvalues* of the arguments and detect “local” size change. Another important point is that there is a corresponding powerful syntax directed static analysis that can be done in linear time.

The criterion has been implemented as part of the PML [7] language, where it plays a central role: PML has a notion of *proofs*, which need, obviously, to be well founded. As far as usability is concerned, this was a success: the criterion is strong enough, its output is usually easy to predict, and its implementation was rather simple.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 a *control-flow graph*. 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 look at the actual criterion. We finish with an appendix giving some details about the implementation and describing a simple static analysis.

Comparison with other works. Using the size-change termination principle for a language with tuples and variants was already done by A. Abel, who implemented a termination checker for the Agda programming language² based on the “foetus” termination checker [1]. This implementation incorporates a part of SCT which unfortunately isn’t described in the corresponding publication.

Similar ideas appeared in [2] and [6] (see Section 2.4). 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 a linear syntactical analysis of the definitions– one needs only compute a transitive closure of the control-flow graph and inspect some loops. This makes it particularly easy to implement from scratch as it needs not rely on external automatic proof-checker (as in [6]) or integer linear programming libraries (as in [2]). One advantage of this minimalistic approach is that a formal proof of the criterion (bootstrap of the proof assistant) is probably easier.

This criterion is particularly well-suited for proof-assistants like Coq³ or Agda. 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 assistant seldom rely on such native types. Complementing the present approach with such internal datatypes and analysis of higher-order programs as in [8] 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 variants, 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 only proviso is that all variants are unary and written as “C[u]”. Other features like `let` expressions, exceptions, (sub)typing etc. can easily be added as they don’t interfere with the criterion. (They might make the static analysis harder though.)

¹The core consists of about 600 lines of OCaml code and doesn’t use any external library.

²A language based on dependent type theory, see <http://wiki.portal.chalmers.se/agda/pmwiki.php>

³A type theory based proof assistant with inductive types, see <http://coq.inria.fr/>

$$\begin{aligned}
 \text{program} & ::= \text{val rec } \text{def} \ (\text{and } \text{def})^* \\
 \text{def} & ::= \text{f } \mathbf{x}_1 \ \mathbf{x}_2 \ \dots \ \mathbf{x}_n = \text{term} \\
 \text{term} & ::= \mathbf{x} \mid \text{cste} \mid \text{f} \mid \text{term term}^+ \mid \\
 & \quad \text{C}[\text{term}] \mid (\text{term}, \dots, \text{term}) \mid \\
 & \quad \text{term}.i \mid \text{match term with branch}^+ \\
 \text{branch} & ::= \mid \text{C}[\mathbf{x}] \rightarrow \text{term}
 \end{aligned}$$

Figure 1: syntax of the programming language

The operational semantics is the usual one and we only consider *safe* programs, i.e., those whose semantics is well defined. This can be achieved using traditional type checking or type inference or as in PML [7], a *constraint checking* algorithm ensuring that

- a variant 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 also assume that functions have an arity and are always fully applied and that the arguments of functions are all first-order values. These constraints need not be present in the actual implementation: we can η -expand the functions that are not fully applied and ignore higher-order arguments (including some of the newly η -expanded arguments).

An important property of this language is that non-termination can only be the result of evaluation going through an infinite sequences of calls to recursive functions, see [7]. Because of 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 along the lines of [3] might be possible, at the cost of a greatly increased complexity of implementation.

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

$$\begin{aligned}
 \text{depth}(\text{C}[u]) & \stackrel{\text{def}}{=} 1 + \text{depth}(u) \\
 \text{depth}((u_1, \dots, u_n)) & \stackrel{\text{def}}{=} \max_{1 \leq i \leq n} (1 + \text{depth}(u_i)) .
 \end{aligned}$$

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

```

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

Without this convention, the definition would be

```

val rec map l = match l with Nil[x] -> Nil[()]
                    | Cons[x] -> Cons[(f x.1, map x.2)]
    
```

Note that because we restrict to first-order arguments, we cannot formally make `f` an argument of `map`. We thus assume that `f` is a predefined function...

Vocabulary and notation. For a set of mutual recursive definitions

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

where x_1 , x_2 and x_3 are variables and u_1 and u_2 are terms,

- “ x_1 ”, “ x_2 ” and “ 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.
- Because our criterion generalizes the original SCT (where we use the depth of a value as its size), all the examples from [4] are accepted. For example, the Ackermann function is accepted as:

```
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, as soon as the size of a value increased, all information would be lost. We do support a local 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 `f1` to `g1` (that increases the depth by 1) is harmless because it is followed by a call from `g1` to `f1` (that decreases the depth by 2).

- In the definition

```
val rec f2 x = match x with
  A[x] -> f2 B[x]
  | B[x] -> f2 x
  | _ -> ()
```

the size of the argument is constant in the first recursive call. This alone would make it non size-change terminating for the original SCT. However, because of the constructors, the first recursive call is necessarily followed by the second one, where the size decreases. This function is accepted by the improved criterion.

- In the definition

```
val rec push_left t =
  match t 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 the right subtree does. In the original SCT, the user could choose the ad-hoc notion of size “depth of the right-subtree”, but in our case, the algorithm will discover it automatically.

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 part is done by a static analysis, independent of the actual algorithm described in this paper. A simple, syntactical static analysis that allows to deal with the examples of the paper is described in the appendix. This control-flow graph represents the evolution of arguments of recursive call, and nothing more. For example both the `map` function and the `last` function

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

have the same control-flow graph: when applied to a non-empty list, they make a recursive call to its tail. 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 “ $\pi_2 \text{Cons}^- 1$ ”: starting from parameter `l`, we remove a `Cons`⁻ variant and take the second component of the resulting tuple. For a call to a function with several argument, a substitution is used: each parameter of the called function is described by a term with free variables among the parameter of the calling function.

Checking terminations 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 real sequence of calls are:

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

In order to do that, we will identify loops that every infinite path must go through, and check that for all of these “coherent” loops, there is some part of an argument that decreases strictly. For example, in the definition of `push_left` (page 4), the right subtree of the argument is a decreasing argument, while neither the left subtree, nor the argument itself is decreasing.

1. INTERPRETING CALLS

1.1. Terms and Reduction. The first 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. The term language is described by the following grammar

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

where x can be any variable, $n \geq 0$, $i \geq 1$ and $w \in \mathbf{Z}_\infty = \mathbf{Z} \cup \{\infty\}$. We write $\mathcal{T}(x_1, \dots, x_n)$ for the set of terms whose variables are chosen among x_1, \dots, x_n .

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

$$\begin{array}{ll}
\mathbf{C}\mathbf{0} = \mathbf{0} & \mathbf{C}(t_1 + t_2) = \mathbf{C}t_1 + \mathbf{C}t_2 \\
(\dots, \mathbf{0}, \dots) = \mathbf{0} & (\dots, t_1 + t_2, \dots) = (\dots, t_1, \dots) + (\dots, t_2, \dots) \\
\mathbf{C}^-\mathbf{0} = \mathbf{0} & \mathbf{C}^-(t_1 + t_2) = \mathbf{C}^-t_1 + \mathbf{C}^-t_2 \\
\pi_i\mathbf{0} = \mathbf{0} & \pi_i(t_1 + t_2) = \pi_it_1 + \pi_it_2 \\
\langle w \rangle \mathbf{0} = \mathbf{0} & \langle w \rangle (t_1 + t_2) = \langle w \rangle t_1 + \langle w \rangle t_2 .
\end{array}$$

Moreover, we implicitly quotient \mathcal{T} by associativity, commutativity, neutrality of $\mathbf{0}$, and idempotence of $+$:

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

The intuition is that:

- x is a parameter of the calling function,
- \mathbf{C} is a variant constructor and $(_, \dots, _)$ is a tuple constructor,
- π_i is a projection, giving to access the i th component of a tuple,
- \mathbf{C}^- corresponds to a branch of pattern matching: it removes the \mathbf{C} from a value,
- $t_1 + t_2$ can be seen as as a non-deterministic choice: we don't know which of t_1 and t_2 is going to be chosen. Such sums will play a central role when analysing definitions even when the initial static analysis doesn't introduce them.
- $\mathbf{0}$ is an artifact used to represent an error during evaluation. Because we only look at safe definitions, any $\mathbf{0}$ that appears during analysis can be ignored as it cannot come from an actual computation.
- $\langle w \rangle$ stands for anything that acts on values in such a way that the depth is increased by at most w . In particular, if $w < 0$, then the depth must decrease. For example, both $\mathbf{C}t$ and $\mathbf{D}t$ can be approximated by $\langle 1 \rangle t$, even though we loose information when doing so.

There is a natural notion of reduction on terms:

Definition 1.2. We define a reduction relation between terms:

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

The symbol “+” (addition) used for elements of \mathbf{Z}_∞ has nothing to do with the symbol “+” (non-deterministic choice) used for terms.⁴

This reduction extends the evaluation 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:

⁴adding elements in \mathbf{Z}_∞ is defined in the obvious way

- a $\langle w \rangle$ absorbs constructors on its right and destructors on its left,
- because a $\langle w \rangle$ may approximate some projections, we don't know which of the components of a tuple it may access, which explains why non-deterministic sums appear.

Lemma 1.3. *The reduction \rightarrow is strongly normalizing and confluent.*

Proof. Strong normalization is easy as the depth of terms decreases strictly during reduction. Confluence thus follows from local confluence which follows from examination of the critical pairs:

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

□

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,*

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. □

Because of the way reduction is defined, 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*

$$\begin{array}{l} t ::= Ct \mid (t_1, \dots, t_n) \mid \bar{d} \mid \langle w \rangle \bar{d} \\ \bar{d} ::= x \mid \pi_i \bar{d} \mid C^- \bar{d} \end{array}$$

The length $|\bar{d}|$ of a destructor sequence is the number of destructors C^-/π_i it contains.

We now introduce a preorder describing approximations between terms.

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

- \preceq is compatible with reduction: if $t \rightarrow u$ then $u \preceq t$ and $t \preceq u$,
- \preceq is compatible with $+$ and $\mathbf{0}$: $\mathbf{0} \preceq t$ and $t \preceq t + u$,
- if $v \preceq w$ then $\langle v \rangle t \preceq \langle w \rangle t$,
- $t \preceq \langle 0 \rangle t$.

When $t \preceq u$, we say that “ t is finer than u ” or that “ u is an approximation of t ”. When both $t \preceq u$ and $u \preceq t$, we say that t and u are equivalent, and write $t \approx u$.

This definition implies for example that $Cx \preceq \langle 0 \rangle Cx \approx \langle 1 \rangle x$, and thus, by contextuality, that $Ct \preceq \langle 1 \rangle t$. The next lemma gives some facts about the preorder (\mathcal{T}, \preceq) . They may help get an intuition about the objects, but are not necessary for the rest of the paper. Their proof is omitted.

Lemma 1.7. *We have*

- $\mathbf{0}$ is the least element,
- $+$ is a least-upper bound, i.e., $t_1 + t_2 \preceq u$ iff $t_1 \preceq u$ and $t_2 \preceq u$,
- if t and u are simple, then $t \preceq u$ and $u \preceq t$ iff $t = u$,
- a term t is an atom iff it reduces to a simple terms without any $\langle w \rangle$.

1.2. Substitutions and Control-Flow Graphs. Just like a term is meant to represent a single argument of a recursive call, a substitution $[x_1 := u_1; \dots; x_n := u_n]$ 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, \dots, u_n in \mathcal{T} , we define $t[x_1 := u_1; \dots; x_n := u_n]$ as the parallel substitution of each x_i by u_i . The composition $\tau \circ \sigma$ of two substitutions $\tau = [x_1 := u_1; \dots; x_n := u_n]$ and σ is simply the substitution $\tau \circ \sigma = [x_1 := u_1\sigma; \dots; x_n := u_n\sigma]$.

Lemma 1.8. *Composition of substitutions is associative and monotonic (for the pointwise order) on the right and on the left.*

Proof. Associativity is obvious. Monotonicity follows from the fact that \preceq is contextual. \square

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

Definition 1.9. A *control-flow graph* for some mutually recursive definitions is a labeled graph where:

- vertices are function names,
- if the parameters of \mathbf{f} are $\mathbf{y}_1, \dots, \mathbf{y}_m$ and the parameters of \mathbf{g} are $\mathbf{x}_1, \dots, \mathbf{x}_n$, the labels of arcs from \mathbf{f} to \mathbf{g} are substitutions $[x_1 := u_1; \dots; x_n := u_n]$ of terms in $\mathcal{T}(\mathbf{y}_1, \dots, \mathbf{y}_m)$.

That a control-flow graph is *safe* (Definition 1.11) means that it gives approximations of the real evolution of arguments of the recursive calls during evaluation. Those arguments are first-order values of the ambient language (see page 2) and we can embed them in $\mathcal{T}()$.

Definition 1.10. A *first-order value* is a simple term of $\mathcal{T}()$. An *atomic value* is a value which doesn't contain any $\langle w \rangle$.

Note that first-order values of the ambient language correspond exactly to atomic values in \mathcal{T} . We can now define safety formally:

Definition 1.11. Let G be a control-flow graph for some recursive definitions,

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

$$\begin{array}{l} \text{val rec } \mathbf{f} \ \mathbf{x}_1 \ \mathbf{x}_2 \ \dots \ \mathbf{x}_n = \\ \quad \dots \ \underline{\mathbf{g} \ \mathbf{u}_1 \ \dots \ \mathbf{u}_m} \\ \quad \dots \end{array}$$

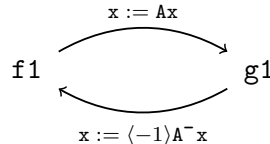
An arc $\mathbf{f} \xrightarrow{\sigma} \mathbf{g}$ in G is *safe* with respect to this particular call site if for all substitutions of *values* $\rho = [\mathbf{x}_1 := v_1; \dots; \mathbf{x}_n := v_n]$, we have

$$[\mathbf{x}_1 = \llbracket \mathbf{u}_1 \rrbracket_\rho; \dots; \mathbf{x}_m = \llbracket \mathbf{u}_m \rrbracket_\rho] \preceq \sigma \circ \rho$$

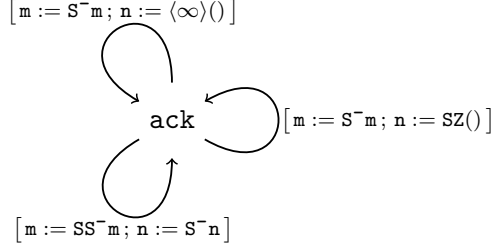
where each $\llbracket \mathbf{u}_i \rrbracket_\rho$ is the semantics of \mathbf{u}_i in context ρ .

- (2) A set of mutually inductive 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, an exact control-flow graph for \mathbf{f}_1 and \mathbf{g}_1 from page 4 could be:



while a safe control-flow graph for the Ackermann function could be:



1.3. Collapsing. For combinatorial reasons, we will need the labels of the control-flow graph (substitutions) to live in a finite set. Because there can only be finitely many variant constructors used in a control-flow graph, the two obstructions for finiteness of the term language are that the depth of terms is unbounded and that there are infinitely many possible weights for the $\langle w \rangle$ s.

Given $B > 0$ (fixed once and for all), it is easy to collapse all the weights appearing in a simple term into $\mathbf{Z}_B = \{-B, \dots, 0, 1, \dots, B-1, \infty\}$: for each $\langle w \rangle$ we replace w by:

$$[w]_B \stackrel{\text{def}}{=} \begin{cases} w & \text{if } -B \leq w < B \\ -B & \text{if } w < -B \\ \infty & \text{if } w \geq B. \end{cases}$$

Collapsing to ensure that the depth is bounded is more subtle. Given $D \geq 0$ (fixed once and for all) and $t \in \mathcal{T}$ in normal form, we want to make sure that the constructor part of t has depth less than D and that the destructor parts of t have length less than D . This is achieved with the following definition *acting on simple terms* (see Lemma 1.5) and extended by linearity. Note that the clauses are not disjoint and only the first appropriate one is used:

$$\begin{aligned} (\mathbf{C}t)_{\uparrow_i} &\stackrel{\text{def}}{=} \mathbf{C}(t_{\uparrow_{i-1}}) && \text{if } i > 0 \\ (t_1, \dots, t_n)_{\uparrow_i} &\stackrel{\text{def}}{=} (t_{1\uparrow_{i-1}}, \dots, t_{n\uparrow_{i-1}}) && \text{if } i > 0 \\ (\langle w \rangle \bar{d})_{\uparrow_i} &\stackrel{\text{def}}{=} \langle w \rangle (\bar{d}_{\downarrow_D}) && \text{if } i > 0 \\ \bar{d}_{\uparrow_i} &\stackrel{\text{def}}{=} \bar{d}_{\downarrow_D} && \text{if } i > 0 \\ t_{\uparrow_0} &\stackrel{\text{def}}{=} (\langle 0 \rangle t)_{\downarrow_D} && (*) \\ \bar{d}_{\downarrow_D} &\stackrel{\text{def}}{=} \bar{d} && \text{if } |\bar{d}| \leq D \\ (\mathbf{C}^- \bar{d})_{\downarrow_D} &\stackrel{\text{def}}{=} \langle -1 \rangle (\bar{d}_{\downarrow_D}) && \text{if } |\mathbf{C}^- \bar{d}| > D \\ (\pi_i \bar{d})_{\downarrow_D} &\stackrel{\text{def}}{=} \langle -1 \rangle (\bar{d}_{\downarrow_D}) && \text{if } |\pi_i \bar{d}| > D \end{aligned}$$

Note that the definition of t_{\uparrow_i} is done by double induction on i and t while the definition of \bar{d}_{\downarrow_D} is only by induction on \bar{d} , and that we need to compute a normal form for clause (*). The function $t \mapsto t_{\downarrow_D}$ 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).

We have:

Lemma 1.12. *The collapsing function $t \mapsto [t]_{\uparrow D}_B$ is inflationary and monotonic:*

- $t \preceq [t]_{\uparrow D}_B$,
- if $t \preceq u$ then $[t]_{\uparrow D}_B \preceq [u]_{\uparrow D}_B$,

Proof. We have

- by definition of \preceq , that $[-]_B$ is inflationary and monotonic follows from the fact that $[-] : \mathbf{Z}_\infty \rightarrow \mathbf{Z}_B$ is inflationary and monotonic.
- The proof that $[-]_{\uparrow D}$ is inflationary is a direct inductive proof. It relies on the fact that $t \preceq \langle 0 \rangle t$ and that $\mathbf{C}^- t \preceq \langle -1 \rangle t$ and $\pi_i t \preceq \langle -1 \rangle t$.
- The proof that $[-]_{\uparrow D}$ is monotonic is a tedious inductive proof. It is omitted for sake of brevity.

Together, these facts imply that $[-]_{\uparrow D}_B$ is both inflationary and monotonic. \square

Definition 1.13. Define the *constructor depth* of simple terms as:

$$\begin{aligned} \text{depth}_C(\mathbf{C}v) &\stackrel{\text{def}}{=} 1 + \text{depth}_C(v) \\ \text{depth}_C((u_1, \dots, v_n)) &\stackrel{\text{def}}{=} \max_{1 \leq i \leq n} (1 + \text{depth}_C(v_i)) \\ \text{depth}_C(\overline{d}) &\stackrel{\text{def}}{=} 0 \\ \text{depth}_C(\langle w \rangle \overline{d}) &\stackrel{\text{def}}{=} 0 \end{aligned}$$

and the *destructor depth* of simple terms as:

$$\begin{aligned} \text{depth}_D(\mathbf{C}v) &\stackrel{\text{def}}{=} \text{depth}_D(v) \\ \text{depth}_D((u_1, \dots, v_n)) &\stackrel{\text{def}}{=} \max_{1 \leq i \leq n} (\text{depth}_D(v_i)) \\ \text{depth}_D(\overline{d}) &\stackrel{\text{def}}{=} |\overline{d}| \\ \text{depth}_D(\langle w \rangle \overline{d}) &\stackrel{\text{def}}{=} |\overline{d}| \end{aligned}$$

The depth of a sum of simple terms is the maximum of the depth of the summands.

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

- t is in normal form
- each $\langle w \rangle$ appearing in t has $w \in \mathbf{Z}_B$,
- the constructor depth and the destructor depth of t are less or equal than D .

If $\sigma = [x_1 := t_1; \dots; x_n := t_n]$ and $\tau = [y_1 := u_1; \dots; y_m := u_m]$ are substitutions, then $\tau \diamond \sigma$ is defined as the pointwise collapsing $\left[(\tau \circ \sigma) \right]_{\uparrow D}_B$.

The next lemma justifies the use of this collapsing function. Because it is not necessary in this paper, we only sketch the proof.

Lemma 1.14. *For each $t \in \mathcal{T}$, we have $[t]_{\uparrow D}_B \in \mathcal{T}_{D,B}$. Moreover, $[t]_{\uparrow D}_B$ is the least term in $\mathcal{T}_{D,B}$ that approximates t . In particular, the function $t \mapsto [t]_{\uparrow D}_B$ is idempotent*

$$\left[[t]_{\uparrow D}_B \right]_{\uparrow D}_B = [t]_{\uparrow D}_B.$$

Because it is also monotonic and inflationary, $[-]_{\uparrow D}_B$ is thus a closure operator.

Proof. We already know that collapsing is inflationary and monotonic. It is easy to show that both $[-]_B$ and $[-]_D$ are idempotent. Idempotence of $[-]_B$ follows from the fact that $[t]_B]_B = [t]_B$. That $[t]_B \in \mathcal{T}_{D,B}$ follows directly from the definition. The proof that $[t]_B$ is the least term in $\mathcal{T}_{D,B}$ that approximates t is a tedious inductive proof. \square

This implies in particular that collapsing is monotonic with respect to the bound D and B :

Corollary 1.15. *If $0 \leq D' \leq D$ and $0 < B' \leq B$, then $[t]_B \preceq [t]_{B'}$.*

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

$$[y := \langle -1 \rangle z] \diamond [x := \langle 1 \rangle y] \diamond [r := \langle 1 \rangle x]$$

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

$$[y := C^- z] \diamond [x := Cy] \diamond [r := Dx]$$

can give $[r := Dz]$ or $[r := D\langle 0 \rangle z]$.⁵ We have however the following:

Definition 1.16. Two terms u and v are called *compatible*, written $u \circ v$, if there is some $t \not\approx \mathbf{0}$ such that $t \preceq u$ and $t \preceq v$. Two substitutions are compatible if they are pointwise compatible.

Lemma 1.17. *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 $\tau_1 \circ \tau_2$.*

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

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

2. SIZE-CHANGE COMBINATORIAL PRINCIPLE

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

Lemma 2.1. *Let G be a control-flow graph with substitution components in $\mathcal{T}_{D,B}$; then, for every infinite path of composable substitutions*

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

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

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

where:

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

⁵There is a special case: when $D = 0$ and $B = 1$, the operation \diamond is indeed associative! This was the original case of SCT [4].

The idea of this decomposition (together with its proof) seems to go back to [5]. The proof is very similar to the one appearing in [4], with only a slight modification to deal with the fact that \diamond isn't associative in general.

Proof. This is a consequence of the infinite Ramsey theorem. Let $(\sigma_n)_{n \geq 0}$ be an infinite path in G . We associate a “color” $c(m, n)$ to each pairs (m, n) of natural numbers where $m < n$:

$$c(m, n) \stackrel{\text{def}}{=} \left(\mathbf{f}_m, \mathbf{f}_n, \sigma_{n-1} \diamond \cdots \diamond \sigma_m \right).$$

Because the number of possible colors for a given control-flow graph is finite, the infinite Ramsey theorem asserts the existence of an infinite set $I \subseteq \mathbf{N}$ such all the (i, j) for $i < j \in I$ have the same color $(\mathbf{f}, \mathbf{f}', \tau)$. Write $I = \{n_0 < n_1 < \cdots < n_k < \cdots\}$. If $i < j < k \in I$, we have:

$$\begin{aligned} (\mathbf{f}, \mathbf{f}', \tau) &= \left(\mathbf{f}_i, \mathbf{f}_j, \sigma_{j-1} \diamond \cdots \diamond \sigma_i \right) \\ &= \left(\mathbf{f}_j, \mathbf{f}_k, \sigma_{k-1} \diamond \cdots \diamond \sigma_j \right) \\ &= \left(\mathbf{f}_i, \mathbf{f}_k, \sigma_{k-1} \diamond \cdots \diamond \sigma_i \right) \end{aligned}$$

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

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

In the original size-change termination principle, composition was associative and we could deduce that $\tau \diamond \tau = \tau$. 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 1.17). \square

2.2. Transitive Closure. The transitive closure of a control-flow graph G is the graph G^+ with the same vertices as G and edges between a and b in G^+ correspond exactly to path 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 edges.

Definition 2.2. If G is a control-flow graph with labels in \mathcal{T} , the graph G^+ , the *transitive closure* of G , is the control-flow graph defined as follows:

- $G^0 = G_{D,B}$ the collapsing of G into $\mathcal{T}_{D,B}$,
- in G^{n+1} , the edges between \mathbf{f} and \mathbf{g} are

$$G^{n+1}(\mathbf{f}, \mathbf{g}) = G^n(\mathbf{f}, \mathbf{g}) \cup \{ \sigma \diamond \tau \mid \tau \in G^n(\mathbf{f}, \mathbf{h}), \sigma \in G^0(\mathbf{h}, \mathbf{g}) \}$$

where \mathbf{h} ranges over all vertices of G ,

- $G^+ = \bigcup_{n \geq 0} G^n$.

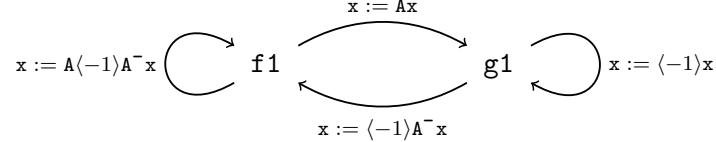
Because all the $\mathcal{T}_{D,B}(\mathbf{x}_1, \dots, \mathbf{x}_m)$ are finite,⁶ we have

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 .

⁶more precisely, because the number of variables and variants that can appear is finite

The transitive closure of G is the graph of path of $G_{D,B}$, and for each path $\sigma_1 \cdots \sigma_n$ in G , it contains an edge for $\sigma_n \diamond \dots \diamond \sigma_1$. (Recall that “ \diamond ” associates on the right.)

As an example, here is the first step of the computation of the transitive closure of the control-flow graph for the functions **f1** and **g1** (page 4) when $D = B = 1$:



For example, the substitution $x := \langle -1 \rangle x$ is obtained as the normal form of the composition $[x := \langle -1 \rangle A^{-}x] \circ [x := Ax]$. Note that no collapsing is necessary here because the result already lives in $\mathcal{T}_{D,B}$. The next iteration gives the following compositions:

- $[x := A\langle -1 \rangle A^{-}x]$ composed with itself, giving $[x := A\langle -1 \rangle A^{-}A\langle -1 \rangle A^{-}x]$ which reduces to $[x := A\langle -2 \rangle A^{-}x]$ and collapses to $[x := A\langle -1 \rangle A^{-}x]$,
- $[x := \langle -1 \rangle x]$ composed with itself, which collapses to $[x := \langle -1 \rangle x]$,
- $[x := \langle -1 \rangle x]$ composed with $[x := \langle -1 \rangle A^{-}x]$ which collapses to $[x := \langle -1 \rangle A^{-}x]$,
- $[x := \langle -1 \rangle A^{-}x]$ composed with $[x := A\langle -1 \rangle A^{-}x]$ which collapses to $[x := \langle -1 \rangle A^{-}x]$,
- $[x := Ax]$ composed with $[x := \langle -1 \rangle x]$ which collapses to $[x := A\langle -1 \rangle x]$,
- $[x := A\langle -1 \rangle A^{-}x]$ composed with $[x := Ax]$ which also collapses to $[x := A\langle -1 \rangle x]$.

The first four composition do not contribute to the transitive closure, as they already exist in the control-flow graph. The last two compositions are equal, and an arc $[x := A\langle -1 \rangle x]$ should be added from **f1** to **g1**. The resulting control-flow graph with 5 arcs is in fact the transitive closure of the initial graph (page 9).

2.3. Size-Change Termination Principle. In order to say that a part of an argument decreases during a call, we must be able to track it before and after the call. The next definition formalizes this: we use a branch of destructors to specify a subterm of an argument and require that the same subterm exists after composition with the substitution representing the call. Because this is enough for the rest, we only do that for loops, i.e., for calls to **f** inside **f**.

Definition 2.4. If $\tau = [x_1 := t_1; \dots; x_n := t_n]$ is a substitution from **f** to itself. A *decreasing parameter* for τ is given by a substitution $\xi = [\underline{p} := \bar{d}x_i]$ such that $\mathbf{0} \not\preceq \xi \circ \tau \preceq \langle w \rangle \xi$ with $w < 0$ and \bar{d} is minimal (i.e., no strict suffix of \bar{d} satisfies the same conditions). A substitution τ is called *decreasing* when it has a decreasing parameter.

For example, for the `push_left` function (page 4), $\xi = [t := \pi_2 \text{Node}^{-}t]$ is a decreasing argument: it accesses the right subtree of **t**. The only loop in the control flow-graph is approximated by $\tau = [t := \text{Node}(\langle \infty \rangle \text{Node}^{-}t, \langle -1 \rangle \pi_2 \text{Node}^{-}t)]$ and the composition $\xi \circ \tau$ collapses to $[t := \langle -1 \rangle \pi_2 \text{Node}^{-}t]$.

The minimality condition means that when computing the normal form of $\bar{d}\tau(x_i)$, we only use reduction rules from group (1) in Definition 1.2. If this were not the case we could cut \bar{d} to strict suffix. For example, for the loop $\tau = [x := A\langle 0 \rangle B^{-}A^{-}x]$, we don't want $\xi = [p := X^{-}A^{-}x]$ to be a decreasing argument, even though $\xi \circ \tau \approx [p := \langle -1 \rangle B^{-}A^{-}x]$. The reason is that the terms we will apply ξ may not have an **X** constructor. We should instead use $\xi = [x := A^{-}x]$. The minimality condition is important for the following lemma, which will be of crucial importance in the proof of soundness of the termination criterion.

Lemma 2.5. *If ξ is a decreasing parameter for τ , we have:*

$$\mathbf{0} \not\preceq \sigma \preceq \tau \circ \rho \quad \Rightarrow \quad \mathbf{0} \not\preceq \xi \circ \sigma \preceq \xi \circ \tau \circ \rho .$$

In particular, under the same hypothesis $\xi \circ \tau \circ \rho \not\preceq \mathbf{0}$.

Proof. The important part is that under the hypothesis of the lemma, we have $\mathbf{0} \not\preceq \xi \circ \sigma$. The substitution $\xi \circ \sigma$ is of the form $[\mathbf{p} := \bar{d}\sigma(\mathbf{x}_i)]$. Suppose by contradiction that this reduces to $\mathbf{0}$. Suppose also that all the substitutions are in normal forms.

If we look at the only reduction sequence of $\bar{d}\sigma(\mathbf{x}_i)$, we can find a reduction from group (3) of Definition 1.2: the sequence \bar{d} reaches an incompatible constructor in $\sigma(\mathbf{x}_i)$. Because $\sigma(\mathbf{x}_i) \preceq \tau \circ \rho(\mathbf{x}_i) = \tau(\mathbf{x}_i)\rho$, all the constructors reachable in $\tau(\mathbf{x}_i)\rho$ are also reachable in $\sigma(\mathbf{x}_i)$. Note that $\sigma(\mathbf{x}_i)\rho$ starts with the same constructors as $\sigma(\mathbf{x}_i)$ and so, the reduction of $\bar{d}\tau(\mathbf{x}_i)\rho$ must have reached the end of the constructors in $\sigma(\mathbf{x}_i)$ before the end of \bar{d} . (Otherwise, we would have $\mathbf{0} \approx \xi \circ \tau$.) This implies that \bar{d} is not minimal. \square

Proposition 2.6 (Size-Change Termination Principle with Constructors). *If G safely represents some recursive definitions and all coherent loops $\tau \circ \tau \diamond \tau$ in $G_{D,B}^+$ are decreasing, then the evaluation of the functions on values cannot produce an infinite sequence of calls to other functions.*

First, a small lemma:

Lemma 2.7. *If v is a value, $v \preceq \langle w \rangle()$ if and only if $w \geq \text{depth}(v)$.*

Proof. If we take the normal form of the value v , we can prove the result inductively using Lemma 1.5:

- if $v = ()$: we have $() \preceq \langle w \rangle() \Rightarrow \langle 0 \rangle() \preceq \langle w \rangle()$. This implies that $0 \leq w$.
- if $v = \mathbf{C}v'$: we have $\mathbf{C}v' \preceq \langle w \rangle() \Rightarrow v' \preceq \langle w - 1 \rangle()$. By induction hypothesis, we have $w - 1 \geq \text{depth}(v')$, which implies $w \geq \text{depth}(v)$.
- if $v = (v_1, \dots, v_n)$ with $n > 0$, we have $(v_1, \dots, v_n) \preceq \langle w \rangle() \Rightarrow v_i \preceq \langle w - 1 \rangle()$ for all $i = 1, \dots, n$. This implies that $w - 1 \geq \text{depth}(v_i)$ for all $i = 1, \dots, n$, which implies that $w \geq \text{depth}(v)$. \square

Proof of Proposition 2.6. Suppose the conditions of the proposition are satisfied and suppose that function \mathbf{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 \mathbf{h} : $\rho_0 = [\mathbf{x}_1 := v_1; \dots; \mathbf{x}_m := v_m]$. Let $\sigma_1 \cdots \sigma_n \cdots$ be the substitutions that label the edges of $G_{D,B}$ corresponding to the calls $c_1 c_2 \cdots$. We can use Lemma 2.1 to decompose this sequence as:

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

where:

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

Because G , and thus $G_{d,b}$ is safe, we have

$$\rho_{n+1} \preceq \sigma_n \circ \rho_n$$

and so, because \circ is monotonic, starting from n_0 until n_1 :

$$\rho_{n_1} \preceq \sigma_{n_1-1} \circ \cdots \circ \sigma_{n_0} \circ \rho_{n_0} .$$

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

$$\rho_{n_1} \preceq (\sigma_{n_1-1} \diamond \cdots \diamond \sigma_{n_0}) \circ \rho_{n_0} = \tau \circ \rho_{n_0} .$$

Repeating this, we obtain:

$$\rho_{n_k} \preceq \underbrace{\tau \circ \cdots \circ \tau}_k \circ \rho_{n_0}$$

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

$$\xi \circ \rho_{n_k} \preceq \xi \circ \tau \circ \cdots \circ \tau \circ \rho_{n_0} \preceq \cdots \preceq \langle w + \cdots + w \rangle \xi \circ \rho_{n_0}$$

By Lemma 2.5, the right side of the inequality cannot be $\mathbf{0}$ and the substitution $\xi \circ \rho_{n_0}$ thus consists of a single value $[\mathbf{p} := v]$. We can now choose k large enough to ensure that $\text{depth}(v) + kw < 0$. Because $\xi \circ \rho_{n_k}$ is also a first-order value, this contradicts Lemma 2.7. \square

Definition 2.8. A control-flow graph G that satisfies the condition of Proposition 2.6 is said to be *size-change terminating for D and B* .

This gives a family of criteria indexed by $D \geq 0$ and $B > 0$ of criteria. We have:

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

Proof. Let G be an uncollapsed control-flow graph, and let $B' \geq B$ and $D' \geq D$. Suppose $G_{D,B}$ is size-change terminating; we want to show that $G_{D',B'}$ is also size-change terminating.

Let τ' be a coherent loop in $G_{D',B'}^+$. By construction, τ' is equal to $\sigma'_1 \diamond_{D',B'} \cdots \diamond_{D',B'} \sigma'_n$ for a path $\sigma'_1 \dots \sigma'_n$ in $G_{D',B'}$. Because each σ'_k is a collapsing of some arc in G , there is a corresponding arc σ_k in $G_{D,B}$. Thus, we can find a loop $\tau = \sigma_1 \diamond_{D,B} \cdots \diamond_{D,B} \sigma_n$ in $G_{D,B}$.

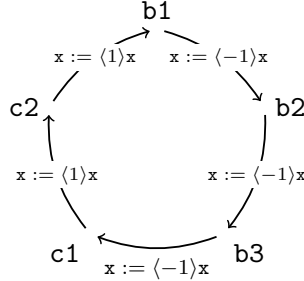
Because collapsing is monotonic in D and B (Lemma 1.15), we have that each $\sigma'_k \preceq \sigma_k$ and so $\tau' \preceq \tau$. Moreover, $\tau' \diamond_{D',B'} \tau' \preceq \tau \diamond_{D,B} \tau$. Because τ' is coherent, τ is also coherent. By hypothesis, τ has a decreasing parameter ξ : we have $\xi \circ \tau \preceq \langle w \rangle \xi$, with $w < 0$. This ξ is also a decreasing parameter for τ' : $\xi \circ \tau' \preceq \xi \circ \tau \preceq \langle w \rangle \xi$. \square

Failure of Completeness. The original SCT satisfied a notion of *completeness* stating roughly that “all infinite path are infinitely decreasing *iff* all (idempotent) loops have a decreasing parameter”. We capture more programs (Section 2.4) than the original SCT, but completeness doesn't hold anymore. Here is a counter example for $D = 0$ and $B = 2$:

```

val rec b1 x = match x with A[x] -> b2 x
    and b2 x = match x with A[x] -> b3 x
    and b3 x = match x with A[x] -> c1 x
    and c1 x = c2 A[X]
    and c2 x = b1 A[X]
    
```


The corresponding control-flow graph is



Whatever our definition of “decreasing path”, all the infinite path in $G_{0,2}$ ought to decrease infinitely. However, because there are two “ $\langle 1 \rangle x$ ” arcs in sequence, we will get some $\langle \infty \rangle x$ in the transitive closure, and thus some coherent loops $\langle \infty \rangle x$ around each node. Those are not decreasing.

The previous example is however size-change terminating whenever $B > 2$. However, completeness doesn’t even hold in this weaker sense. Call a graph G *decreasing* if no infinite path comes from actual computation. More precisely, a graph is decreasing 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. $\rho \circ \sigma_1 \circ \cdots \circ \sigma_n \approx \mathbf{0}$.

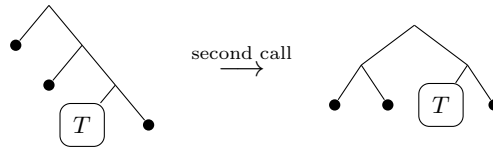
The combing function transforming a binary tree into a right-leaning tree terminates for a subtle reason. Its definition is

```
val rec comb t = match t 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:

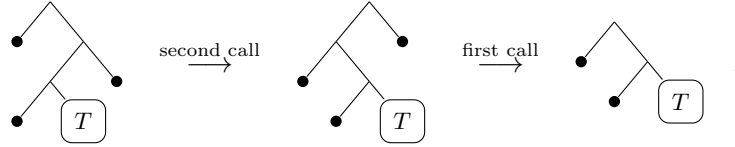
- $[t := \pi_1 \text{Node}^- t]$
- $[t := \text{Node}(\text{Node}(\pi_1 \text{Node}^- t, \pi_1 \text{Node}^- \pi_2 \text{Node}^- t), \pi_2 \text{Node}^- \pi_2 \text{Node}^- t)]$.

This graph is terminating in the above sense, exactly for the same reason that `comb` terminates, but it can be shown that whatever the values of D and B , this graph is *never* size-change terminating. Here is a sketch of why this is the case: for each bound D and sequence \bar{d} of length D , it is possible to find a tree for which the depth of the subtree $\bar{d}t$ increases arbitrarily during a sequence of recursive calls. For example, at $D = 4$ for $\bar{d} = \pi_1 \text{Node}^- \pi_2 \text{Node}^-$, consider the tree on the left:



By the second recursive call, the tree on the right will be used as the new argument. While $\pi_1 \text{Node}^- \pi_2 \text{Node}^-$ 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 $\pi_2\text{Node}^- \pi_2\text{Node}^-$, we need to use the second call and then the first call:



This implies that during the transitive closure, all sequences of destructors of length D will reach a weight of ∞ because they can be composed with such a tree.

Even if this is rather unfortunate, this example can be dealt with by adding additional arguments. The function `comb_size` with a second parameter for the size of the tree passes the termination test:

```
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 []
```

It is then possible to prove (in the system or in the meta language) that the `comb` function is equal to:

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

One would of course like to tag the additional parameter `s` as “computationally irrelevant” so that it isn’t used during real computation.

2.4. Comparison with other SCT-Based Criterion. In the original SCT, an arc in the control-flow graph is 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:

- $\bar{\downarrow}$, meaning that the size of u is strictly smaller than the size of x ,
- \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 is the target of more than one arc. We can encode such a bipartite graph as a substitution $\sigma = [y_1 := t_1; \dots; y_m := t_m]$ where:

- $t_k = \langle -1 \rangle x_i$ if there is an arc \downarrow from x_i to u_i ,
- $t_k = \langle 0 \rangle x_i$ if there is an arc $\bar{\downarrow}$ from 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 [4]. Note in particular that composition is associative in this context. Thus, 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.

SCT with Difference Constraints. Amir Ben-Amram considers in [2] a generalisation of the original SCT which, in our terminology, could be seen as choosing the bounds $D = 0$ but allowing unbounded weights in the control-flow graphs, i.e. choosing “ $B = \infty$ ”. 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. In particular, 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 loose completeness as shown by the example on page 16, but this is seldom a problem in practice because the user may increase the bound B (at the cost of speed). 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”. The authors of [6] use control-flow graphs where arcs are labeled with “calling contexts”. A calling context from \mathbf{f} to \mathbf{g} amounts to:

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

Note that the permutations are build from the ambient language, as are the expressions in the set. The intuition of having such a calling context from \mathbf{f} to \mathbf{g} is that *if all the expressions of the set evaluate to True, then there can be a call to \mathbf{g} from \mathbf{f} , and the arguments of \mathbf{g} are given by the permutation.*

The drawback of this approach is that an automatic theorem prover is necessary to deal with the conditions appearing in the contexts. 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 internally in a uniform way.

2.5. Extensions.

Weighted constructors. At the moment, each variant has weight 1, as can be seen from the reduction $\langle w \rangle Ct \rightarrow \langle w + 1 \rangle t$. 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[x]]]]]]
  | A[A[A[A[A[C[x]]]]]] -> f A[A[A[A[A[x]]]]]
  | _ -> A[]
```

This function does pass the termination test if we choose $D \geq 7$. If the definition contained other recursive calls, it can make the test use more resources than reasonable. Giving a weight of 3 to B and 1 to C would be enough to see this is 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 is much more complete than the ambient language we have been using and in particular, function abstraction is possible. Like OCaml, the PML language computes weak-head normal forms. Because partially applied functions are allowed, it is possible to extend the principle to detect that functions such as

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

terminate: when applied to n arguments, it “eats” through all of them and stop on the weak-head normal form `fun x -> glutton`.

In order to do that, it is enough to add a virtual argument to all recursive functions: it counts the difference between the number of abstraction and the number of applications. Think of it as an additional “ $\mathbf{x}_0 := \langle w \rangle \mathbf{x}_0$ ” in all substitutions. Note that an abstraction counts positively and an application counts negatively so that in effect, it amounts to having

a constructor for abstraction and a destructor for application. This is interesting because adding dummy abstractions and applications is the usual way to freeze evaluation and define “infinite” data structures in OCaml.⁷ In this context, the size-change termination principle can be used to detect *productivity*.

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. 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.
5. Naomi Lindenstrauss and Yehoshua Sagiv, *Automatic termination analysis of logic programs*, Proceedings of the Fourteenth International Conference on Logic Programming, MIT Press, 1997, pp. 63–77.
6. 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.
7. Christophe Raffalli, *Realizability for programming languages*, course notes for the *École jeunes chercheurs du GDR IM*, submitted, 2010.
8. Damien Sereni and Damien Sereni, *Termination analysis of higher-order functional programs*, In APLAS 2005: The Third Asian Symposium on Programming Languages and Systems (Kwangkeun, Springer. November, 2005, pp. 281–297.

⁷refer to the implementation of the “Lazy” module

APPENDIX A. IMPLEMENTATION ISSUES

In order to make the presentation readable, the paper followed a rather “high-level” description of the criterion. Because the initial goal was to get a concrete termination checker for the PML language [7], ease of implementation was very important. 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 whether $t \preceq u$ and whether $t \supset u$ is easy for those normal forms,
- (3) checking if a loop is decreasing (Definition 2.4) is easy.

Even for terms in normal forms, we need a uniform way to deal with sums. Because 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. Because PML’s static analysis doesn’t introduce many sums, no exponential blow-up occurs in practice. Then, sums only appear through collapsing of substitutions, i.e. from the reduction rule $\langle w \rangle(t_1, \dots, t_n) \rightarrow \sum_i \langle w + 1 \rangle t_i$. In particular, sums can always be pushed under all constructors implying that all summands start with a $\langle w \rangle$. We thus use the following grammar for terms:

$$t ::= Ct \mid (t_1, \dots, t_n) \mid \bar{d} \mid \sum_i \langle w_i \rangle \bar{d}_i$$

$$\bar{d} ::= x \mid \pi_i \bar{d} \mid C^- \bar{d}$$

where the sums are not empty. This can be represented by the following inductive type (Caml syntax):

```

type z_infty = Number of int | Infty
type destructor = Project of string | RemoveVariant of string
type parameter = int
type term = Variant of string*argument
            | Tuple of argument list
            | Epsilon of (destructor list)*parameter
            | Sum of (z_infty*(destructor list)*parameter) list

```

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

- rules $\pi_i Ct \rightarrow \mathbf{0}$, $C^-(t_1, \dots, t_n) \rightarrow \mathbf{0}$ and $\pi_i(t_1, \dots, t_n) \rightarrow \mathbf{0}$ (when $i > n$) all raise an *error* `TypingError`. Encountering such a reduction means that the definitions were not *safe* to begin with and that the initial type-checking / constraint solving of the definitions is broken.
- the rule $C^- D t \rightarrow \mathbf{0}$ raises an *exception* `ImpossibleCase`. Even safe definitions may introduce such reductions, but we know 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.

A.1. Order and Compatibility. When the terms are in the form given by the above grammar, we can give an inductive definition of both the order and compatibility.

Lemma A.1. *The relation \preceq on terms as above is the least preorder satisfying:*

- $\bar{d} \preceq \bar{d}$,
- $t \preceq u \Rightarrow Ct \preceq Cu$,
- $\forall i, t_i \preceq u_i \Rightarrow (t_1, \dots, t_n) \preceq (u_1, \dots, u_n)$,
- $\forall j, \langle v_j \rangle \bar{e}_j \preceq \sum_i \langle w_i \rangle \bar{d}_i \Rightarrow \sum_j \langle v_j \rangle \bar{e}_j \preceq \sum_i \langle w_i \rangle \bar{d}_i$,
- $\exists i, \bar{d} \preceq \langle w_i \rangle \bar{d}_i \Rightarrow \bar{d} \preceq \sum_i \langle w_i \rangle \bar{d}_i$,
- $t \preceq \sum_i \langle w_i - 1 \rangle \bar{d}_i \Rightarrow Ct \preceq \sum_i \langle w_i \rangle \bar{d}_i$,
- $\forall j, t_j \preceq \sum_i \langle w_i - 1 \rangle \bar{d}_i \Rightarrow (t_1, \dots, t_n) \preceq \sum_i \langle w_i \rangle \bar{d}_i$,
- $\langle 0 \rangle \bar{e} \preceq \langle w \rangle \bar{d} \Rightarrow \bar{e} \preceq \langle w \rangle \bar{d}$,
- if \bar{e} is a suffix of \bar{d} and $v - |\bar{e}| \leq w - |\bar{d}|$ then $\langle v \rangle \bar{e} \preceq \langle w \rangle \bar{d}$.

This follows rather directly from the definition of \preceq , but the proof is a little tedious to write out in full detail. Checking compatibility is done in a similar way:

Lemma A.2. *Compatibility on terms as above is the least symmetrical relation satisfying:*

- $t \circ u \Rightarrow Ct \circ Cu$,
- $\forall i, t_i \circ u_i \Rightarrow (t_1, \dots, t_n) \circ (u_1, \dots, u_n)$,
- $\bar{d} \circ \bar{d}$,
- $t \circ \sum_i \langle w_i - 1 \rangle \bar{d}_i \Rightarrow Ct \circ \sum_i \langle w_i \rangle \bar{d}_i$,
- $\forall j, t_j \circ \sum_i \langle w_i - 1 \rangle \bar{d}_i \Rightarrow (t_1, \dots, t_n) \circ \sum_i \langle w_i \rangle \bar{d}_i$,
- $\exists i, \bar{d} \preceq \langle w_i \rangle \bar{e}_i \Rightarrow \bar{d} \circ \sum_i \langle w_i \rangle \bar{e}_i$,
- $\exists i, j, \langle w_i \rangle \bar{d}_i \circ \langle v_j \rangle \bar{e}_j \Rightarrow \sum_i \langle w_i \rangle \bar{d}_i \circ \sum_j \langle v_j \rangle \bar{e}_j$,
- if \bar{d} is a suffix of \bar{e} or \bar{e} is a suffix of \bar{d} , then $\langle w \rangle \bar{d} \circ \langle v \rangle \bar{e}$.

Both lemmas can be implemented easily using pattern matching. For example, here is the beginning of the Caml definition of compatibility:

```

let rec compatible t1 t2 = match t1,t2 with
  Variant(c1,t1), Variant(c2,t2) -> c1=c2 && compatible t1 t2
| Tuple(l1), Tuple(l2) ->
  List.length l1 = List.length l2 && List.for_all2 compatible l1 l2
| Epsilon(ds,x), Epsilon(es,y) -> ds=es && x=y
| Variant(_,t), Sum(s)
| Sum(s), Variant(_,t) ->
  let s_minus = List.map (fun (w, ds, x)->(decr w, ds, x)) in
  compatible t (Sum(s_minus))
...
...
| _,_ -> false

```

A.2. Complexity. The authors of [4] showed that deciding whether a graph is size-change terminating is P-space complete. The same is true here, and this can be shown by encoding the same problem. Because this encoding works for $D = 0$ and $B = 1$, all versions of SCT are P-space complete by Proposition 2.9.

In practice however, mutually recursive definitions written by hand do not contain enough functions and enough calls to make that a problem. Complexity comes when the bounds D and B get bigger. The following two points make the problem more tractable:

- make sure sums are minimal by keeping only maximal summands: \bar{d} is equivalent (“ \approx ”) to $\bar{d} + \langle 12 \rangle \bar{d} + \langle -1 \rangle C^- \bar{d}$ and is a much better choice;

- during the construction of G^+ , remove the arcs that are approximated by some other arc (“subsumption”).

In practice, choosing a bound $B > 1$ is rarely necessary; but there are example where D needs to be more than 4. The default is to have $D = 2$ and $B = 1$, and let the user change the bounds. This is strong enough to detect most function as terminating with almost no overhead: parsing and type checking take more than an order of magnitude more than termination analysis.

APPENDIX B. STATIC ANALYSIS

The simplest interesting static analysis simply records pattern matching and projection: for each call-site “ $g\ u_1 \dots u_m$ ” in the definition of “ $f\ x_1 \dots x_n$ ”, we use the substitution $[y_1 := u_1; \dots; y_m := u_m]$ where each $u_i \in \mathcal{T}(x_1, \dots, x_n)$ is

- a simple term without $\langle w \rangle$ s if u_i is syntactically built from projections and pattern-matching variable coming from x_1, \dots, x_n ;
- $\langle \infty \rangle()$ otherwise.

For example, all the examples `map`, `f1`, `g1`, `f2` 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:

- $[x_1 := S^-x_1; SZ()]$,
- $[x_1 := SS^-x_1; x_2 := S^-x_2]$,
- $[x_1 := S^-x_1; x_2 := \langle \infty \rangle()]$.

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 just a syntactical analysis of the code for the recursive definitions. In particular, it can be done in linear time.

It is also possible to think of smarter 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 ]
```

Note that this definition isn’t strictly speaking in our ambient language because it contains higher-order arguments (`f`) and the functions are not fully applied (`rmap`). Those constraints are not fundamental and were only added to simplify the presentation.

The static analysis from PML is rather involved and we refer to [7] for the details. What is important, is that it will detect that the list `l` contains smaller trees than `t`, giving us the following interpretation for the unique recursive call: $[f := f; t := \pi_1 \text{Cons}^- \pi_2 \text{Node}^- t]$. This will be enough for the termination criterion to accept the function.