The Size-Change Termination Principle for Constructor Based Languages

Pierre Hyvernat

To cite this version:

Pierre Hyvernat. The Size-Change Termination Principle for Constructor Based Languages. 2010. hal-00547440v1

HAL Id: hal-00547440
https://hal.science/hal-00547440v1

Preprint submitted on 16 Dec 2010 (v1), 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

Abstract. By keeping track of constructors and destructors inside the arguments of recursive calls, we are able to improve the *size-change termination principle* of Lee, Jones and Ben-Amram. The points of interest are that the new principle is able to ignore impossible compositions and detect which specific sub-arguments decrease even when the whole argument doesn't.

Introduction

Lee, Jones and Ben-Amram's size-change termination principle ([1]) is a simple, yet surprisingly strong termination checker for generic programming languages. The ingredients are a very general notion of size on values, a static analysis of the program to get a "call-graph" of the recursive definitions and a conceptually simple "transitive closure" computation on this graph. The static analysis is independent of the test, as long as it yields a safe description of the possible calls among the recursive functions.

The implementation described below was done in Caml for the PML programming language ([3]). The termination checker is crucial in this context because PML has notions of specification and proof. While non terminating programs are useful in practice, or at least programs whose termination proof is beyond any automated tool, proofs on the other hand must terminate.

The kind of programs validated by the improved principle is described via examples and "counter-examples" in sections 5.6 and 5.7. Readers not interested in the theoretical justification can skip directly to these sections and proceed to testing the termination checker with the PML language ([4]). The code is available in the PML distribution[1] or as a single file on the author's webpage for easy perusal.

The main difference with the original principle is that we do not reduce a recursive call to a single size information. Instead, we keep a (bounded) term representing the call. This makes it possible to ignore some compositions which do not occur in practice and allows to detect precisely which part of an argument decreases.

The context: an ML-style language. We are interested in automatically checking that (some) programs always terminate. The ambient programming language is a very generic language in ML-style with:

- variant constructors and pattern-matching,
- tuples and projections,
- (mutual) recursive definitions.

Since we do only inspect first-order argument of the definitions, λ-abstraction doesn't play any role in the sequel.

The syntax should be obvious to anyone familiar with an ML-style language with one proviso: all variant constructors are unary and written as `C[u]`. The argument `u` can be an n-ary tuple, where n can be 0. Other features like `let`

Date: Winter 2010.

[1] `http://lama.univ-savoie.fr/~pml/`

expressions, exceptions, subtyping, specification, proofs, etc. are of no concern here. Some of them are described in more details in [2]. As an example, here is the function computing the length of arbitrary lists:

```
val rec length l =
  match l with
      Nil[]  ->  Zero[]
    | Cons[_,l]  ->  Succ[length l]
```

The PML language has a rather advanced *constraint checking* algorithm which only accepts "safe" programs, *i.e.* programs which do not provoke an error of the evaluation machine. As far as this work is concerned, this could be replaced by any sound typing algorithm.

The other important property of the language should be that non-termination may only come from an infinite sequences of calls to recursive functions, which is indeed the case for the PML language ([2]). Because of that it seems mostly useful for typed programming languages.

Vocabulary and notation. We will deal quite a lot with function definitions. In the rest of this paper, the following convention will be used: for a recursive definition of the form:

```
val rec f x1 x2 x3  =
  ... g u1 u2 ...
and g x1 x2 = ...
```

where `x1`, `x2` and `x3` are variables and `u1` and `u2` are terms,

- `g u1 u2` is a *call from* `f` *to* `g`,
- `x1`, `x2` and `x3` are the *parameters*,
- `u1` and `u2` are the *arguments* of the call.

For simplicity, we assume that each function has an arity and is always fully applied.

For $b \geq 1$, $\mathbf{Z}_b$ will denote the set $\{-b, -b+1, ..., 0, ..., b-1, \infty\}$; and $\mathbf{Z}_\infty$ will denote the set $\mathbf{Z} \cup \{\infty\}$. Addition on $\mathbf{Z}_\infty$ is defined as expected and $w - m$ is a synonym for $w + (-m)$. There is a natural projection $\lfloor _ \rfloor_b : \mathbf{Z}_\infty \to \mathbf{Z}_b$:

$$\lfloor n \rfloor_b \quad = \quad \begin{cases} n & \text{if } -b \leq n < b \\ -b & \text{if } n < -b \\ \infty & \text{if } n \geq b\ . \end{cases}$$

Some of its obvious properties are:

- $\lfloor _ \rfloor_b$ is monotonic,
- $n \leq \lfloor n \rfloor_b$.

One last word about proofs: the lemmas of sections 3 and 4 have rather "pedestrian" induction proofs. Care has been taken to give the definitions and lemmas in the appropriate order to make everything as straightforward as possible. To keep the size of the paper reasonable, only the most interesting cases are spelled out.

1. The Combinatorial Part

The heart of the size-change termination principle is the following:

Proposition 1.1. *If G is a finite category,*[2] *for every infinite chain of arrows*

$$(*) \qquad A_0 \xrightarrow{c_0} A_1 \xrightarrow{c_1} \dots \xrightarrow{c_n} A_{n+1} \xrightarrow{c_{n+1}} \dots$$

[2]since we don't need identities, this amounts to a labeled graph where consecutive arcs can be composed in an associative manner.

there is an A such that we can decompose the chain as

$$A_0 \quad \underbrace{\xrightarrow{c_0} \dots \xrightarrow{c_{n_0-1}}}_{\textit{initial prefix}} \quad A \quad \underbrace{\xrightarrow{c_{n_0}} \dots \xrightarrow{c_{n_1-1}}}_{c} \quad A \quad \underbrace{\xrightarrow{c_{n_1}} \dots \xrightarrow{c_{n_2-1}}}_{c} \quad A \quad \dots$$

where:

- *all the compositions $c_{n_{k+1}-1} \circ \dots \circ c_{n_k}$ are equal to the same $c : A \to A$,*
- *c is idempotent: $c = c \circ c$.*

Proof. this is a consequence of the infinite Ramsey theorem. The argument from [1] is repeated for completeness: let $(c_n)_{n \geq 0}$ be an infinite chain in C. Define an equivalence relation on pairs (m, n) of natural numbers where $m < n$:

$$(m_1, n_1) \approx (m_2, n_2) \qquad \text{iff} \qquad \begin{cases} & c_{n_1-1} \circ \dots \circ c_{m_1} = c_{n_2-1} \circ \dots \circ c_{m_2} \\ \text{and} & A_{m_1} = A_{m_2} \\ \text{and} & A_{n_1} = A_{n_2} \ . \end{cases}$$

Because C is finite, the corresponding equivalence classes form a partition of the set $\{(m, n) \mid m < n\}$ into a finite collection of "colors". By the infinite Ramsey theorem, there is an infinite set $I \subseteq \mathbf{N}$ such all the (i, j) for $i < j \in I$ are in the same class. Write $I = \{n_0 < n_1 < \dots < n_k < \dots\}$. Those satisfy the conditions of the lemma.

Note that for this to work for a specific chain $(*)$, we only need that

- only finitely many different A_i's appear,
- there are finitely many different compositions $c_{m+n} \circ \dots \circ c_m$.

Then we don't actually need the initial category (labeled graph) to be finite. □

A consequence is that it is sometimes possible to deduce properties of infinite chains only by looking at the finite idempotent cycles. A very crude explanation of the size-change termination principle is then:

- compute the "call-graph" of the recursive definitions: a graph with function names as vertices and "size information" as label for each call between functions,
- compute its "transitive closure",
- check that for all idempotent loops (in the transitive closure), "something" decreases.

We refer to [1] (or the rest of this paper) for the details.

2. A Language for Arguments

For the "obvious" notion of size, the original principle doesn't keep much information. For example, the Ackermann function contains three recursive calls:

```
val rec ack x1 x2 = ... ack 1 (x2-1)
                    ... ack (x1-1) (ack x1 (x2-1))
```

The information kept in [1] is

$$\begin{pmatrix} ? & ? \\ ? & < \end{pmatrix}, \begin{pmatrix} < & ? \\ ? & ? \end{pmatrix}, \begin{pmatrix} = & ? \\ ? & < \end{pmatrix} \quad : \quad \texttt{ack} \to \texttt{ack} \ .$$

Each matrix represents a recursive call. For example, the third matrix corresponds to the last recursive call "`ack x1 (x2-1)`" and contains the following size information:

- the entry $(1, 1)$ is "`=`" because argument 1 ("`x1`") of the recursive call is at most of the same size as parameter 1 ("`x1`") of the definition,

- the entry $(2,2)$ is "<" because argument 2 ("x2-1") of the recursive call is strictly smaller than parameter 2 ("x2") of the definition,
- the other entries are "?" as we don't know anything about the size relationship between arguments and parameters.

In general, for a recursive call

```
val rec f x1 x2 x3 =
  ...  g u1 u2 ...
```

the entry $(2,3)$ represents the relation between the size of parameter "x3" and the size of argument "u2".

Even with this restricted knowledge, the size-change termination principle is able to infer termination of many functions (including the Ackermann function). In a language with constructors and pattern-matching, it is relatively easy to obtain more information, even with a naive and straightforward static analysis: the arguments u1 and u2 are built from pieces of the parameters x1, x2 and x3, and such pieces are obtained using projections and pattern-matching. The following term language describes the possible ways of constructing such arguments from parameters:

Definition 2.1. The language $\widehat{\mathcal{L}}(\mathtt{x}_1,\dots,\mathtt{x}_m)$ is defined as:

$$u, u_i \quad ::= \quad \bot \;\mid\; \underbrace{\mathtt{x}_i}_{i=1,\dots,m} \;\mid\; \underbrace{\mathtt{C}u \;\mid\; (u_1,\dots,u_n)}_{\text{constructors}} \;\mid\; \underbrace{\pi_i u \;\mid\; \mathtt{C}^- u}_{\text{destructors}}$$

where the C's are taken from a finite set of *variant constructors* and tuples may be empty.

The intended semantics of those terms is obvious, except perhaps for:

- "$\bot$" represents impossible cases. It would be the least element of the appropriate domain of values.[3]
- "$\mathtt{C}^- u$" represents a branch of a case analysis. Its semantics would be:

$$[\![\mathtt{C}^- u]\!] \quad = \quad \begin{cases} e & \text{if } [\![u]\!] = \mathtt{C}\, e \\ \bot & \text{otherwise .} \end{cases}$$

Let's look at the Ackermann function again, written this time in full with unary numbers:

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

The second argument to the last call: "n" is represented by $\mathtt{S}^-\mathtt{x}_2$ while the first argument: "S[m]" is represented by $\mathtt{S}\mathtt{S}^-\mathtt{x}_1$. For reasons that will become clear later, arguments which contain an unknown number of variant, the second argument to the second call will be represented by "()".[4]

There is an obvious notion of reduction on $\widehat{\mathcal{L}}$:

Definition 2.2. $\triangleright$ is the contextual closure of:

- $\pi_i(u_1,\dots,u_n) \triangleright u_i$,
- $\mathtt{C}^-\mathtt{C}u \triangleright u$.

The other reduction rules deal with "impossible cases", either by introducing them:

[3]Values would be interpreted in a domain $\mathbb{D} \simeq \mathrm{List}(\mathbb{D}) + \sum_{\mathtt{C}} \mathbb{D}$.

[4]More precisely, by a variant of "()" to be introduced in the next section.

- $\pi_i(u_1,\dots,u_n) \rhd \bot$ if $i > n$,
- $\mathtt{C}^-\mathtt{D}u \rhd \bot$ if $\mathtt{C} \neq \mathtt{D}$,
- $\pi_i\mathtt{C}u \rhd \bot$,
- $\mathtt{C}^-(u_1,\dots,u_n) \rhd \bot$,

or by propagating them:

- $\bot u \rhd \bot$, $u\bot \rhd \bot$ and $(\dots,\bot,\dots) \rhd \bot$.

We write $\mathcal{L}(\mathtt{x}_1,\dots,\mathtt{x}_n)$ for the normal forms of $\widehat{\mathcal{L}}(\mathtt{x}_1,\dots,\mathtt{x}_n)$ different from $\bot$.

Lemma 2.1. *This reduction is strongly normalizing. When restricted to terms which do not reduce to $\bot$, it is also confluent.*

Proof. Strong normalization is trivial: the size of terms decreases along reduction. For confluence, the only problematic critical pair is: $\pi_i(\dots,u_i,\dots,\bot,\dots)$ which reduces either to u_i or to $\bot$.

□

A note on "$\bot$". Arguments which reduce to $\bot$ can be ignored because they do not correspond to actual computations in the ambient language. More precisely, the two rule $\mathtt{C}^-\mathtt{D}u \rhd \bot$ and $\pi_i\mathtt{C}u \rhd \bot$ never occur because they correspond to ill-typed terms. Similarly, the rule $\pi_i(u_1,\dots,u_n) \rhd \bot$ if $i > n$ would have been detected by the typing algorithm.

The last rule $\mathtt{C}^-(u_1,\dots,u_n) \rhd \bot$ is more interesting: it is "well-typed" and does occur in practice. It is the evaluation mechanism that prevents it from ever being used during computation: for a "`match u with ...`", the evaluation mechanism uses the first matching branch (a kind of $\mathtt{C}^-\mathtt{C}\dots$) and raises an error if no such branch is found. This explains why we can safely ignore all terms reducing to $\bot$. In the actual implementation, the first three rules provoke an error while the last one raises an exception `Impossible_composition`.

3. A Bounded Language for Arguments

Finiteness is crucial for proposition 1.1. We thus need to devise a finite version of $\mathcal{L}$. In order to do that, we introduce "weighted approximations" of terms.

3.1. Weighted terms.

Definition 3.1. The terms of $\widehat{\mathcal{F}}(\mathtt{x}_1,\dots,\mathtt{x}_m)$ are generated by:

$$u, u_i \quad ::= \quad \bot \;\mid\; \underbrace{\mathtt{x}_i}_{i=1,\dots,m} \;\mid\; \underbrace{\delta_w u}_{w\in\mathbf{Z}_\infty} \;\mid\; \mathtt{C}u \;\mid\; (u_1,\dots,u_n) \;\mid\; \pi_i u \;\mid\; \mathtt{C}^- u \;.$$

We usually leave the variables implicit and write $\widehat{\mathcal{F}}$ whenever possible.

The intuition is that δ_w is a prefix for a term of weight bounded by w: it contains at most w more variant constructors than destructors. We can use those terms to approximate elements of $\mathcal{L}$: for example, $\delta_1 u$ is an approximation of $\mathtt{C}u$, $\mathtt{D}u$, or even $\mathtt{C}\,\mathtt{D}^-u$.

We can identify several "canonical" subsets of $\widehat{\mathcal{F}}$:

- The set $\mathcal{B}$ of *terms "without constructors":*

  $$b \quad ::= \quad \vec{d}\,\mathtt{x}_i \;\mid\; () \;\mid\; \delta_w\,\vec{d}\,\mathtt{x}_i \;\mid\; \delta_w()$$

  where $w \in \mathbf{Z}_\infty$ and $\vec{d}$ is a sequence of destructors ($\mathtt{C}^-$ and π_i). In this case, $||\vec{d}||$ is the number of variant destructors in the sequence.

- The set $\mathcal{F}$ of *normal forms:*[5]

$$u, u_i \quad ::= \quad b \in \mathcal{B} \quad | \quad \mathtt{C}u \quad | \quad (u_1, \dots, u_n) \ .$$

where the tuples are not empty.

The presence of the δ's makes the definition of reduction more subtle than before. In particular, we will need the following definition:

Definition 3.2. "$\sup_0(b_1, b_2)$" is defined for any $b_1, b_2 \in \mathcal{B}$ as:[6]

- $\sup_0(b, b) = b$ for any $b \in \mathcal{B}$,
- $\sup(b_1, b_2) = \delta_\infty()$ if b_1 and b_2 are "incompatible": either they end with $\mathtt{x}_i$ and $\mathtt{x}_j$ with $i \neq j$, or one ends with $\mathtt{x}_i$ and the other with $()$,
- $\sup_0(b_1, b_2) = \delta_k \, \vec{a} \, \mathtt{x}_i$ whenever
  - b_1 is of the form $\delta_w \vec{b} \cdot \vec{a} \, \mathtt{x}_i$,
  - b_2 is of the form $\delta_{w'} \vec{c} \cdot \vec{a} \, \mathtt{x}_i$,

  where
  - $\vec{a}$ is the longest common suffix,
  - $k = \max(w - ||\vec{b}||, w' - ||\vec{c}||)$.
- $\sup_0(\delta_w(), \delta_{w'}()) = \delta_{\max(w,w')}()$,
- $\sup_0(\delta_w \vec{a} y, b) = \sup_0(b, \delta_w \vec{a} y) = \sup_0(\delta_w \vec{a} y, \delta_0 b)$ if b doesn't start with a δ_w,
- $\sup_0(b_1, b_2) = \sup_0(\delta_0 b_1, \delta_0 b_2)$ if neither b_1 nor b_2 starts with a δ_w.

This operation is associative and the n-ary version is written $\sup_{0,1 \leq i \leq n}\{b_i\}$.

As a first approximation, it is safe to replace any $\vec{d}\mathtt{x}_i \in \mathcal{B}$ (resp. $()$) by $\delta_0 \vec{d} \mathtt{x}_i$ (resp. $\delta_0()$). Doing so simplifies the definition of $\sup_0$ and reduction and yields a sound but slightly weaker termination principle.

Definition 3.3. $\triangleright$ is the contextual closure of:

- $\pi_i(u_1, \dots, u_n) \triangleright u_i$,
- $\mathtt{C}^- \mathtt{C}u \triangleright u$,
- propagation of impossible cases (see definition 2.2),
- introduction of impossible cases (see definition 2.2),

with the added clauses:

- $\delta_w \delta_{w'} u \triangleright \delta_{w+w'} u$,
- $\pi_i \delta_w u \triangleright \delta_w u$,
- $\mathtt{C}^- \delta_w u \triangleright \delta_{w-1} u$,
- $\delta_w \mathtt{C} u \triangleright \delta_{w+1} u$,
- if $n \geq 1$, $\delta_w(u_1, \dots, u_n) \triangleright \sup_{0,1 \leq i \leq n} \big\{ \mathrm{nf}(\delta_w u_i) \big\}$,

where in the last clause, "$\mathrm{nf}(u)$" is the normal form of u with respect to $\triangleright$.

It is interesting to note that the definition of reduction assumes it is normalizing and that $\mathrm{nf}(\delta_w u) \in \mathcal{B}$. Both this properties are trivially true, making this definition well-founded. Since the reduction is not strictly confluent, the last clause is non-deterministic in general. In practice however, we will be in the confluent fragment.

It will be useful in the sequel to define the following notation:

$$\delta_w \, \vec{d} \cdot u \quad := \quad \mathrm{nf}(\delta_w \, \vec{d} \, u)$$

whenever u is in normal form.

Variant constructors have a "weight" of 1 as can be seen in $\mathtt{C}^- \delta_w u \triangleright \delta_{w-1} u$ and $\delta_w \mathtt{C} u \triangleright \delta_{w+1} u$. Tuple constructors on the other hand have a weight of 0. This could be changed but isn't interesting in practice: programming languages usually require the use of variants in order to define inductive types.

---

[5] reduction and normalization will be defined shortly

[6] This definition is rather tedious. The Caml code might be even easier to read, see on page 26.

Just like before, we have:

Lemma 3.1. *This reduction is strongly normalizing. Moreover, this reduction is confluent on terms which do not reduce to* $\bot$*.*

Proof. Strong normalization is trivial as the size of terms decreases during reduction. For the other part, note that the critical pairs which do not involve $\bot$ are:

$$\begin{array}{ccc} \delta_w \delta_{w'} \delta_{w''} u & \delta_w \delta_{w'} \mathtt{C} u & \delta_w \delta_{w'} (u_1, \dots, u_n) \\ \pi_i \delta_w \delta_{w'} u & \pi_i \delta_{w'} \mathtt{C} u & \pi_i \delta_{w'} (u_1, \dots, u_n) \\ \mathtt{C}^- \delta_w \delta_{w'} u & \mathtt{C}^- \delta_{w'} \mathtt{C} u & \mathtt{C}^- \delta_{w'} (u_1, \dots, u_n) \end{array} .$$

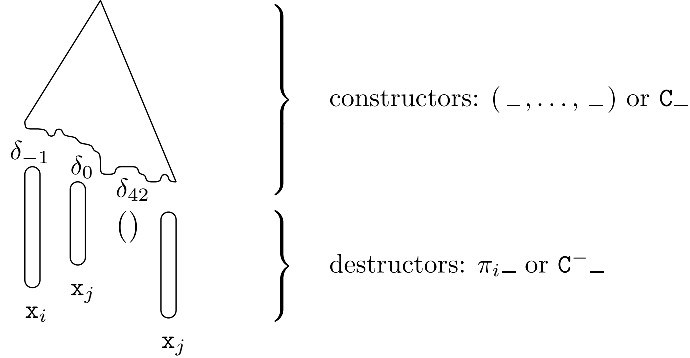
For the first two columns, the diagrams can be closed directly. For the last column, since $\delta_w \cdot u$ doesn't contain constructors, we have:

- $\pi_i \sup_{0,1 \le i \le n} \{\delta_{w'} \cdot u_i\} \rhd \sup_{0,1 \le i \le n} \{\delta_{w'} \cdot u_i\}$,
- $\mathtt{C}^- \sup_{0,1 \le i \le n} \{\delta_{w'} \cdot u_i\} \rhd \sup_{0,1 \le i \le n} \{\delta_{w'-1} \cdot u_i\}$,
- $\delta_w \sup_{0,1 \le i \le n} \{\delta_{w'} \cdot u_i\} \rhd \sup_{0,1 \le i \le n} \{\delta_{w+w'} \cdot u_i\}$

which closes the remaining critical pairs.

□

The δ's absorb constructors on the right (except for the empty tuple) and destructors on the left. The normal forms for this reduction have a simple shape: it may help to picture them as:



Definition 3.4. $\mathcal{F}(\mathtt{x}_1, \dots, \mathtt{x}_n)$ consists of the normal forms of $\widehat{\mathcal{F}}(\mathtt{x}_1, \dots, \mathtt{x}_n)$ different from $\bot$.

Lemma 3.2. $\mathcal{F}(\mathtt{x}_1, \dots, \mathtt{x}_m)$ *is exactly the set defined on page 6.*

Since the main object really is $\mathcal{F}$, the implementation defines a type for the normal forms and a kind of "restricted" reduction: we need only normalize terms of the form $u[\mathtt{x} := v]$ where u and v are already in $\mathcal{F}$. (See appendix C.)

We can now extend the definition of $\sup_0$ to a total (binary) function on normal forms:

Definition 3.5. "$\sup(u, v)$" is defined for all $u, v \in \mathcal{F}$:

- $\sup(u, v) = \sup_0(u, v)$ if $u, v \in \mathcal{B}$,
- $\sup(\mathtt{C}u, \mathtt{C}v) = \mathtt{C}\big(\sup(u, v)\big)$,
- $\sup\big((u_1, \dots, u_n), (v_1, \dots, v_n)\big) = \big(\sup(u_1, v_1), \dots, \sup(u_n, v_n)\big)$,
- $\sup(u, v) = \sup_0(\delta_0 \cdot u, \delta_0 \cdot v)$ for all other cases.[7]

This function satisfies the requirements to be a suppremum operation:

[7]That is, when u and v start with incompatible constructors, or when one of u and v is in $\mathcal{B}$ and the other is not.

Lemma 3.3.

- sup *is commutative:* $\sup(u, v) = \sup(v, u)$*;*
- sup *is idempotent:* $\sup(u, u) = u$*;*
- sup *is associative:* $\sup\big(u, \sup(v, t)\big) = \sup\big(\sup(u, v), t\big)$.

Proof. The first two points are direct induction proofs. For associativity, the complete proof rather tedious and delayed to the appendix on page 24.

□

We can now define the corresponding notion of order:

Definition 3.6. We write $u \sqsubseteq v$ for $\sup(u, v) = v$. This is read "v approximates u".

Lemma 3.4. *The relation* $\sqsubseteq$ *is a partial order on* $\mathcal{F}$*, with* sup *the suppremum operation. Moreover, the order has a greatest element:* $\delta_\infty()$.

Proof. That $\sqsubseteq$ is a partial order and sup is the suppremum is a direct consequence of lemma 3.3. To check that $\delta_\infty()$ is a greatest element, we only need to check that $\sup\big(u, \delta_\infty()\big) = \delta_\infty()$. This is obvious for any $u \in \mathcal{B}$, and all other cases reduce to this one...

□

It is now straightforward to expand the definition of $\sqsubseteq$ to find the following characterization:

Lemma 3.5. *For all* u, v, u_1, $\dots$, *we have:*

- $\mathtt{C}u \sqsubseteq \mathtt{D}v$ *iff* $\mathtt{C} = \mathtt{D}$ *and* $u \sqsubseteq v$,
- $(u_1, \dots, u_m) \sqsubseteq (v_1, \dots, v_n)$ *iff* $m = n$ *and* $u_i \sqsubseteq v_i$ *for all* $i = 1, \dots, n$,
- *if* $b \sqsubseteq v$ *with* $b \in \mathcal{B}$*, then* $v \in \mathcal{B}$,
- *if* $u \sqsubseteq b$ *with* $u \neq b$ *and* $b \in \mathcal{B}$*, then* b *starts with a* δ_w,
- *if* $u \sqsubseteq b$ *with* $b \in \mathcal{B}$*, then* $\delta_0 \cdot u \sqsubseteq b$,
- $\mathtt{C}u \sqsubseteq b$ *with* $b \in \mathcal{B}$ *iff* $\delta_1 \cdot u \sqsubseteq b$ *iff* $u \sqsubseteq \delta_{-1} \cdot b$,
- $(u_1, \dots, u_n) \sqsubseteq b$ *with* $b \in \mathcal{B}$ *iff* $\delta_0 \cdot u_i \sqsubseteq b$ *for all* $i = 1, \dots, n$ *iff* $u_i \sqsubseteq b$ *for all* $i = 1, \dots, n$,
- $\mathtt{C}u \not\sqsubseteq (v_1, \dots, v_n)$ *and* $(u_1, \dots, u_m) \not\sqsubseteq \mathtt{C}v$.

We close this section by a technical lemma:

Lemma 3.6. *For all* $u, v \in \mathcal{F}$ *and* $w \in \mathbf{Z}_\infty$*:*

$$u \quad \sqsubseteq \quad \delta_0 \cdot u$$

and

$$u \sqsubseteq v \qquad \Longrightarrow \qquad \delta_w \cdot u \sqsubseteq \delta_w \cdot v \ .$$

Proof. For the first point, we need to show that $\sup(u, \delta_0 \cdot u) = \delta_0 u$. This holds trivially by idempotence of $\sup_0$ and by definition of sup.

For the second point, we need the following fact:

Fact.

$$\begin{aligned} \delta_{w+w'} \cdot u \quad &= \quad \delta_w \cdot (\delta_{w'} \cdot u) \\ \sup\big(\delta_{w+w'} \cdot u \,,\, \delta_{w+w''} \cdot v\big) \quad &= \quad \delta_w \cdot \sup\big(\delta_{w'} \cdot u \,,\, \delta_{w''} \cdot v\big) \ . \end{aligned}$$

We can then proceed by induction on v:

- if $v = (v_1, \dots, v_n)$, then u is of the form $(u_1, \dots, u_n)$ with $u_i \sqsubseteq v_i$ for each $i = 1, \dots, n$ (by lemma 3.5); we know by induction that

$$\sup\big(\delta_w \cdot u_i, \delta_w \cdot v_i\big) \quad = \quad \delta_w \cdot v_i$$

for all $i = 1, \dots, n$. By associativity and commutativity of sup, we get

$$\begin{array}{rl} & \sup\Big(\delta_w\cdot(u_1,\dots,u_n)\ ,\ \delta_w\cdot(v_1,\dots,v_n)\Big) \\ = & \sup\Big(\sup_{1\le i\le n}\{\delta_w\cdot u_i\}\ ,\ \sup_{1\le i\le n}\{\delta_w\cdot v_i\}\Big) \\ = & \sup\Big(\sup_{1\le i\le n}\big\{\sup(\delta_w\cdot u_i,\delta_w\cdot v_i)\big\}\Big) \\ = & \sup\big(\sup_{1\le i\le n}\{\delta_w\cdot v_i\}\big) \\ = & \delta_w\cdot\big(\sup_{1\le i\le n}\{\delta_0\cdot v_i\}\big) \\ = & \delta_w\cdot(v_1,\dots,v_n)\ . \end{array}$$

- if $v = \mathtt{C}v'$, then u is necessarily of the form $\mathtt{C}u'$, with $u' \sqsubseteq v'$ (by lemma 3.5); we thus need to show that $\sup(\delta_w\cdot u, \delta_w\cdot v) = \delta_w\cdot v$. By induction hypothesis, we know that $\delta_w\cdot u' \sqsubseteq \delta_w\cdot v'$. By the previous fact, we have

$$\begin{array}{rcl} \sup\big(\delta_w\cdot u, \delta_w\cdot v\big) & = & \sup\big(\delta_{w+1}\cdot u', \delta_{w+1}\cdot v'\big) \\ & = & \delta_1\cdot \sup\big(\delta_w\cdot u', \delta_w\cdot v'\big) \\ & = & \delta_1\cdot(\delta_w\cdot v') \\ & = & \delta_{w+1}\cdot v' \\ & = & \delta_w\cdot v \end{array}$$

- if $v \in \mathcal{B}$ and $u \in \mathcal{B}$, the result is trivially true because $\delta_w\cdot_$ doesn't change the shape of its argument.
- the last case is when $v \in \mathcal{B}$ and $u \notin \mathcal{B}$. Since v necessarily starts with a δ, we have $v = \delta_0\cdot v$. By lemma 3.5, we have

$$u \sqsubseteq v \qquad \text{iff} \qquad \delta_0\cdot u \sqsubseteq \delta_0\cdot v$$

which implies that $\delta_w\cdot u \sqsubseteq \delta_w\cdot v$.

□

3.2. **Bounding the weight and depth.** The set $\mathcal{F}$ is still infinite. We will now ensure finiteness by approximating each element of $\mathcal{F}$ by an element of bounded depth and by restricting the possible weights of the δ's.

Definition 3.7. The set $\mathcal{F}_{d,b}(\mathtt{x}_1,\dots,\mathtt{x}_n)$ is the finite subset of $\mathcal{F}(\mathtt{x}_1,\dots,\mathtt{x}_n)$ of elements s.t.

1. all the δ's have weights in $\mathbf{Z}_b$,
2. the destructor depth is less than d,
3. the constructor depth is less than d.

The figure on page 7 should make it clear what is meant by "constructor depth" and "destructor depth", but a formal definition is of course possible. Note that both tuple and variant constructors are taken into account for this notion of depth.

Collapsing a term in $\mathcal{F}$ so that it satisfies the first condition is easy: replace each δ_w by $\delta_{\lfloor w\rfloor_b}$. We write $\mathrm{p}_{1,b}(u)$ for the term resulting from u.

Collapsing to ensure the second condition is also easy: we replace each $\delta_w\vec{a}\cdot\vec{b}\,\mathtt{x}_i$ by $\delta_{w-||\vec{a}||}\vec{b}\,\mathtt{x}_i$ where $\vec{b}$ is chosen of length d. We write $\mathrm{p}_{2,d}(u)$ for the term obtained from u.

The third function to collapse constructors is called $\mathrm{p}_{3,d}$. It "merges" just enough constructors into each δ_w to reduce the depth. It is defined by induction on d:

- $\mathrm{p}_{3,d+1}(\mathtt{C}u) = \mathtt{C}\,\mathrm{p}_{3,d}(u)$,
- $\mathrm{p}_{3,d+1}\big((u_1,\dots,u_n)\big) = \big(\mathrm{p}_{3,d}(u_1),\dots,\mathrm{p}_{3,d}(u_n)\big)$,
- $\mathrm{p}_{3,d}(b) = b$ whenever $b \in \mathcal{B}$,
- $\mathrm{p}_{3,0}(u) = \delta_0\cdot u$.

Definition 3.8. The collapsing function $\lfloor _ \rfloor_{d,b} : \mathcal{F} \to \mathcal{F}_{d,b}$ is defined by

$$\begin{array}{rcl} \mathcal{F}(\mathtt{x}_1,\dots,\mathtt{x}_m) & \to & \mathcal{F}_{d,b}(\mathtt{x}_1,\dots,\mathtt{x}_m) \\ u & \mapsto & \lfloor u \rfloor_{d,b} = \mathrm{p}_{1,b}\,\mathrm{p}_{2,d}\,\mathrm{p}_{3,d}(u) \ . \end{array}$$

Because p_2 deals with destructors and p_3 deals with constructors, they commute, so that the collapsing function could equivalently be defined as $u \mapsto \mathrm{p}_{1,b}\,\mathrm{p}_{3,d}\,\mathrm{p}_{2,d}(u)$. Note that since $\mathcal{L}(\mathtt{x}_1,\dots,\mathtt{x}_m)$ is a subset of $\mathcal{F}(\mathtt{x}_1,\dots,\mathtt{x}_m)$, this collapsing function can be restricted to $\mathcal{L}(\mathtt{x}_1,\dots,\mathtt{x}_m) \to \mathcal{F}_{d,b}(\mathtt{x}_1,\dots,\mathtt{x}_m)$.

The important result is that this is compatible with the approximation order:

Lemma 3.7. *For any $u \in \mathcal{F}$ we have*

- $u \sqsubseteq \lfloor u \rfloor_{d,b}$,
- $\lfloor _ \rfloor_{d,b}$ *is monotonic.*

Proof. It is enough to show that the same holds for each of $\mathrm{p}_{1,b}$, $\mathrm{p}_{2,d}$ and $\mathrm{p}_{3,d}$. It is fairly easy for both $\mathrm{p}_{1,b}$ and $\mathrm{p}_{2,d}$.

We prove that $u \sqsubseteq \mathrm{p}_{3,d}(u)$ by induction on d: when $d = 0$, this is just point 1 of lemma 3.6 and when $d > 0$, this is a direct application of the induction hypothesis.

We prove the monotonicity of $\mathrm{p}_{3,d}$ by induction on d, v and u. If $d = 0$, this is just point 2 of lemma 3.6, with $w = 0$. If $d > 0$, we do a case analysis on v:

- if $v = (v_1,\dots,v_n)$, then u is of the form $(u_1,\dots,u_n)$ with $u_i \sqsubseteq v_i$ for each $i = 1,\dots,n$. By induction, we know that each $\mathrm{p}_{3,d-1}(u_i) \sqsubseteq \mathrm{p}_{3,d-1}(v_i)$ and since
$$\begin{array}{rcl} \mathrm{p}_{3,d}\big((v_1,\dots,v_n)\big) & = & \big(\mathrm{p}_{3,d-1}(v_1),\dots,\mathrm{p}_{3,d-1}(v_n)\big) \\ \mathrm{p}_{3,d}\big((u_1,\dots,u_n)\big) & = & \big(\mathrm{p}_{3,d-1}(u_1),\dots,\mathrm{p}_{3,d-1}(u_n)\big) \end{array} ,$$
we can conclude by lemma 3.5.
- if $v = \mathtt{C}v'$, then u is necessarily of the form $\mathtt{C}u'$, with $u' \sqsubseteq v'$. By induction, we know that $\mathrm{p}_{3,d-1}(u') \sqsubseteq \mathrm{p}_{3,d-1}(v')$ and since $\mathrm{p}_{3,d}(u) = \mathtt{C}\,\mathrm{p}_{3,d-1}(u')$ and $\mathtt{C}\,\mathrm{p}_{3,d}(v) = \mathrm{p}_{3,d-1}(v')$, we can conclude by lemma 3.5.
- If $u, v \in \mathcal{B}$, we can conclude as $\mathrm{p}_{3,d}(u) = u$ and $\mathrm{p}_{3,d}(v) = v$.
- If $v \in \mathcal{B}$ and $u = (u_1,\dots,u_n)$, we know by lemma 3.5 that each $u_i \sqsubseteq v$. We can use the induction hypotheses for $d-1$, u_i and v to get
$$\mathrm{p}_{3,d-1}(u_i) \quad \sqsubseteq \quad \mathrm{p}_{3,d-1}(v) \quad = \quad v$$
for all $i = 1,\dots,n$. This implies that
$$\begin{array}{rcl} \mathrm{p}_{3,d}(u) & \sqsubseteq & \delta_0 \cdot \mathrm{p}_{3,d}(u) \\ & = & \sup_{1 \le i \le n} \mathrm{p}_{3,d-1}(u_i) \\ & \sqsubseteq & v \\ & = & \mathrm{p}_{3,d}(v) \ . \end{array}$$
- If $v \in \mathcal{B}$ and $u = \mathtt{C}u'$, we use the induction hypothesis for $d-1$, u' and $\delta_{-1} \cdot v$ (which has the same shape as v and satisfies $u' \sqsubseteq \delta_{-1} \cdot v$ by lemma 3.5). We get
$$\mathrm{p}_{3,d-1}(u') \quad \sqsubseteq \quad \delta_{-1} \cdot v$$
which implies that
$$\mathrm{p}_{3,d}(u) \quad = \quad \mathtt{C}\,\mathrm{p}_{3,d-1}(u') \quad \sqsubseteq \quad \delta_1 \cdot \delta_{-1} \cdot v \quad = \quad v \quad = \quad \mathrm{p}_{3,d}(v) \ .$$

□

Even if the next lemma isn't necessary for the correctness of the termination checker, it is nevertheless interesting to know this collapsing gives the best approximation possible:

Lemma 3.8. *For any* $u \in \mathcal{F}$,

$$\lfloor u \rfloor_{d,b} \quad = \quad \inf \left\{ v \in \mathcal{F}_{d,b} \ \mid \ u \sqsubseteq v \right\} .$$

More precisely, each projection $\mathrm{p}_{1,b}$, $\mathrm{p}_{2,d}$ *and* $\mathrm{p}_{3,d}$ *is optimal for the corresponding criterion.*

The proof is omitted...

4. Call-Graphs

From now on, we suppose a fixed bound b for the weights of the δ's and a fixed bound d for the depth has been chosen.

4.1. **Graphs and composition.**

Definition 4.1. A call-graph for a set of mutually recursive definitions is a labeled graph where:

- vertices are function names,
- each call from `f` to `g` corresponds exactly to one arc from "`f`" to "`g`",
- if the arity of `f` is m and the arity of `g` is n, the label of such an arc is a substitution $(\mathtt{x}_1 := u_1, \dots, \mathtt{x}_n := u_n)$ of terms of $\mathcal{F}(\mathtt{x}_1, \dots, \mathtt{x}_m)$.

Here is for example the call-graph corresponding to the Ackermann function as defined on page 4:

- there is a singe node: `ack`,
- there are three loops from `ack` to itself:
  - $\big(\mathtt{x}_1 := \mathtt{S}^-\mathtt{x}_1 \ , \ \mathtt{x}_2 := \mathtt{Z}()\big)$,
  - $\big(\mathtt{x}_1 := \mathtt{S}^-\mathtt{x}_1 \ , \ \mathtt{x}_2 := \delta_\infty()\big)$,
  - $\big(\mathtt{x}_1 := \mathtt{S}\mathtt{S}^-\mathtt{x}_1 \ , \ \mathtt{x}_2 := \mathtt{S}^-\mathtt{x}_2\big)$.

An example of call-graph for two mutually recursive definition is the following:

```
val rec f x y = match x with Z[] -> y  | S[x] -> g S[y] x
    and g x y = match y with Z[] -> x  | S[y] -> f S[y] S[x]
```

- the call-graph contains two nodes: `f` and `g`,
- there is one arc from `f` to `g`: $\big(\mathtt{x}_1 := \mathtt{S}\mathtt{x}_2 \ , \ \mathtt{x}_2 := \mathtt{S}^-\mathtt{x}_1\big)$,
- there is one arc from `g` to `f`: $(\mathtt{x}_1 := \mathtt{S}\mathtt{S}^-\mathtt{x}_2 \ , \ \mathtt{x}_2 := \mathtt{S}\mathtt{x}_1)$.

The static analysis of the code will probably produce, as in the above examples, a call-graph with labels in $\mathcal{L}$ (see for example appendix B). To be able to apply proposition 1.1, we need to work in $\mathcal{F}_{d,b}$. If G is a call-graph, we write $G_{d,b}$ for the graph where all labels have been collapsed by pointwise application of the collapsing function:

$$(\mathtt{x}_1 := u_1, \dots, \mathtt{x} := u_n) \quad \mapsto \quad \Big(\mathtt{x}_1 := \lfloor u_1 \rfloor_{d,b}, \dots, \mathtt{x}_n := \lfloor u_n \rfloor_{d,b}\Big) .$$

Definition 4.2. The *composition* of two consecutive substitutions

$$\sigma = \big(\mathtt{x}_1 := u_1, \dots, \mathtt{x}_n := u_n\big) \ : \ \mathtt{f} \to \mathtt{g}$$

and

$$\tau = \big(\mathtt{x}_1 := v_1, \dots, \mathtt{x}_m := v_m\big) \ : \ \mathtt{g} \to \mathtt{h}$$

is

$$\tau \circ \sigma \quad = \quad \big(\mathtt{x}_1 := \mathrm{nf}(v_1[\sigma]), \dots, \mathtt{x}_m := \mathrm{nf}(v_m[\sigma])\big) \quad : \quad \mathtt{f} \to \mathtt{h}$$

where $v[\sigma]$ is the syntactical substitution of all the $\mathtt{x}_i$'s by u_i. *If any of the* $v_i[\sigma]$ *reduces to* $\bot$*, the composition isn't defined;* otherwise, the normal form is unique by lemma 3.1.

The *collapsed substitution* of two consecutive substitutions is

$$\tau \diamond \sigma \quad = \quad \lfloor \tau \circ \sigma \rfloor_{d,b} \quad : \quad \mathtt{f} \to \mathtt{h}$$

where the collapsing function is applied pointwise.

Because of the impossible cases ($\perp$), composition of substitutions isn't total. Slightly more bothering for the rest is that the collapsed composition "$\diamond$" isn't associative because the operation $\oplus$ on $\mathbf{Z}_b$ defined as

$$w \oplus w' \quad = \quad \lfloor w + w' \rfloor_b$$

isn't associative, *except when* $b = 1$, as in the original principle (see section 5.4). Consider for example

$$\infty = \big((b-1) \oplus 1\big) \oplus -1 \quad \neq \quad (b-1) \oplus (1 \oplus -1) = b-1 \ .$$

This translates in $\mathcal{F}$ into such compositions as

$$(\mathtt{x} := \delta_{b-1}\mathtt{x}) \diamond (\mathtt{x} := \delta_1 \mathtt{x}) \diamond (\mathtt{x} := \delta_{-1}\mathtt{x}) \ .$$

Collapsed composition will however be associative up-to the weights inside δ's. If we define

$$\sigma \simeq \tau \quad \text{iff} \quad \begin{cases} \sigma = \tau & \text{when } b = 1 \\ \sigma \text{ and } \tau \text{ only differ on weights of } \delta\text{'s} & \text{otherwise;} \end{cases}$$

we have:

Lemma 4.1. *Composition $\circ$ is associative. Collapsed composition $\diamond$ is associative up-to $\simeq$. In particular, it is strictly associative when $b = 1$.*

Proof. That $\circ$ is associative follows from confluence (lemma 3.1). Since collapsing respects $\simeq$, $\diamond$ is associative up-to $\simeq$.

□

The order $\sqsubseteq$ is extended pointwise to substitutions, with the implicit assumption that $\rho \sqsubseteq \sigma$ means that if the expression ρ is defined, so is the expression σ.

Proposition 4.1. *The order on substitutions is compatible with composition:*

- *if $\rho \sqsubseteq \sigma$ then $\tau \circ \rho \sqsubseteq \tau \circ \sigma$,*
- *if $\rho \sqsubseteq \sigma$ then $\rho \circ \tau \sqsubseteq \sigma \circ \tau$.*

Because collapsing is monotonic (lemma 3.7), the same holds for collapsed composition $\diamond$.

Proof. This is once more a proof by induction. We decompose it and show

$$\begin{array}{lcl} u_1 \sqsubseteq u_2 & \Longrightarrow & \mathrm{nf}(v[\mathtt{x} := u_1]) \sqsubseteq \mathrm{nf}(v[\mathtt{x} := u_2]) \\ u_1 \sqsubseteq u_2 & \Longrightarrow & \mathrm{nf}(u_1[\mathtt{x} := v]) \sqsubseteq \mathrm{nf}(u_2[\mathtt{x} := v]) \end{array}$$

For the first point, the only interesting case is when v is $\delta_w \vec{a}\mathtt{x}$. Since $\delta_w \cdot _$ is monotonic (lemma 3.7), we only need to show that $\mathrm{nf}(\mathtt{C}^- _)$ and $\mathrm{nf}(\pi_i _)$ are monotonic. Those are straightforward inductive proofs.

The second point is proved by induction:

- direct application of the induction hypothesis when $u_2 \notin \mathcal{B}$,
- When $u_2 \in \mathcal{B}$ and $u_1 \notin \mathcal{B}$, u_2 necessarily starts with a δ (lemma 3.5). We thus have $u_2 = \delta_0 \cdot u_2$. By confluence of reduction, we also have that

$$\mathrm{nf}\left((\delta_0 \cdot u_i)[\mathtt{x} := v]\right) \quad = \quad \delta_0 \cdot \left(\mathrm{nf}(u_i[\mathtt{x} := v])\right) \ .$$

By definition, we have

$$\begin{array}{rl} & \sup\Big(\mathrm{nf}(u_1[\mathtt{x} := v]) \ , \ \mathrm{nf}(u_2[\mathtt{x} := v])\Big) \\ = & \sup_0\Big(\delta_0 \cdot \mathrm{nf}(u_1[\mathtt{x} := v]) \ , \ \delta_0 \cdot \mathrm{nf}(u_2[\mathtt{x} := v])\Big) \\ = & \sup\Big(\mathrm{nf}\left((\delta_0 \cdot u_1)[\mathtt{x} := v]\right) \ , \ \mathrm{nf}\left((\delta_0 \cdot u_2)[\mathtt{x} := v]\right)\Big) \\ = & \mathrm{nf}\Big(\sup\left(\delta_0 \cdot u_1 \ , \ \delta_0 \cdot u_2\right)[\mathtt{x} := v]\Big) \ . \end{array}$$

- The last case is when $u_1, u_2 \in \mathcal{B}$. We need yet another tedious induction proof, namely that

$$\mathrm{nf}(\delta_w \vec{d} v) \sqsubseteq \mathrm{nf}\left(\delta_{w-||\vec{d}||} v\right) .$$

  This implies that

$$\delta_w \vec{a}\mathtt{x} \sqsubseteq \delta_{w'} \vec{b}\mathtt{x} \qquad \Longrightarrow \qquad \mathrm{nf}(\delta_w \vec{a} v) \sqsubseteq \mathrm{nf}(\delta_{w'} \vec{b} v)$$

  which is the missing part for the case $u_1, u_2 \in \mathcal{B}$.

□

4.2. **Transitive closure.** The transitive closure of a graph G is the graph G^+ with the same vertices as G and arcs between a and b in G^+ correspond exactly to path between a and b in G. In our case, the graph is labeled, and the label of a path is the composition of the labels of its arcs.

Definition 4.3. If G is a call-graph with labels in $\mathcal{F}_{d,b}$, the graph G^+, the *transitive closure of* G, is the call-graph defined as follows:

- $G^0 = G$,
- in G^{n+1}, the arcs between $\mathtt{f}$ and $\mathtt{g}$ are

$$G^{n+1}[\mathtt{f},\mathtt{g}] \quad = \quad G^n[\mathtt{f},\mathtt{g}] \cup \left\{\sigma \diamond \tau \;\mid\; \tau \in G^n[\mathtt{f},\mathtt{h}], \sigma \in G^n[\mathtt{h},\mathtt{g}]\right\}$$

  where $\mathtt{g}$ ranges over all vertices of G^n,
- $G^+ = \bigcup_{n \geq 0} G^n$.

Because all the $\mathcal{F}_{d,b}(\mathtt{x}_1, \ldots, \mathtt{x}_m)$ are finite, we have

Lemma 4.2. *G^+ is finite and can be computed in finite time. More precisely, there is an n such that $G^n = G^{n+1}$ and G^+ is equal to this G^n.*

This is indeed the graph of "path" of $G_{d,b}$, and for any path $\sigma_1, \ldots, \sigma_n$ in G, it contains all the possible ways of computing $\sigma_n \diamond \ldots \diamond \sigma_1$. (Recall that "$\diamond$" isn't associative, except when $b = 1$.)

5. The Size-Change Principle

5.1. **Values, safety and diagonal nodes.** Given a sequence of substitutions, their composition models the actual reduction of the corresponding calls. Proposition 4.1 ensures that by collapsing the initial terms and using the "compose then collapse" reduction, we end up with an approximation of the sequence of calls. To link that with the actual computations performed by the language, we need a notion of value:

Definition 5.1. A *value* is an element of $\mathcal{L}(\emptyset)$.

Those values correspond exactly to first-order values of the programming language. They are built from tuples, variant constructors and constants (nullary variant constructors) but cannot use destructors. Approximations of values live in $\mathcal{F}(\emptyset)$ and do not contains destructors. It is relatively easy to characterize all the possible approximations of a value:

Lemma 5.1. *We have:*

1. *If v is a value, $v \sqsubseteq \delta_w()$ if and only if $w \geq h$ where h is the "variant-constructor" height of u.*
2. *If v is a value, then $v \sqsubseteq u$ if and only if u is obtained from v by replacing some sub-values v' by $\delta_{w'}()$ with $w' \geq h'$ where h' is the "variant-constructor" height of v'.*

Safety of the representation of a set of inductive definition by a call-graph simply means that we approximate the actual reduction:

Definition 5.2.

(1) suppose we have a call from f to g:

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

An arc $\mathtt{f} \xrightarrow{\sigma} \mathtt{g}$ in a call-graph is *safe* with respect to this particular call if for all substitutions of *values* $\rho = (\mathtt{x}_1 := v_1, \dots, \mathtt{x}_n := v_n)$, we have

$$\big(\mathtt{x}_1 := [\![u_1]\!]_\rho, \dots, \mathtt{x}_m := [\![u_m]\!]_\rho\big) \quad \sqsubseteq \quad \sigma \circ \rho$$

where $[\![u_1]\!]_\rho$ is the appropriate first-order semantical value. The semantics of higher-order values is simply $\delta_\infty()$.

(2) A set of mutually inductive definitions is *safely represented* by a call-graph if all calls have a corresponding safe arc in the graph.

Whenever the static analysis cannot infer an approximation of the reduction, it will use the top element $\delta_\infty()$ to ensure safety. This is the case for example when it encounters a call to an external function in one of the arguments. The most naive and straightforward static analysis is described in appendix B.

For example, the two call-graphs given on page 11 safely represent their corresponding definitions, precisely because they exactly describe the evolution of (first-order) arguments of the functions.

Lemma 3.7 and proposition 4.1 imply directly that

Lemma 5.2. *If G safely represents a set of inductive definitions, then $G_{d,b}$ safely represents the same set of inductive definitions.*

The last concept we need before stating the new size-change termination principle is that of "diagonal nodes". Those will play the same role as the entries on the diagonal of square matrices as in the original principle.

Definition 5.3. If $\sigma = (\mathtt{x}_1 := u_1, \dots, \mathtt{x}_n := u_n)$ is a substitution in $\mathcal{F}(\mathtt{x}_1, \dots, \mathtt{x}_n)$, a *diagonal node of* σ is given by a sequence of destructors $\vec{d}$ and an index i which satisfies:

$$\delta_0\, \vec{d} \cdot u_i \quad = \quad \delta_w\, \vec{d}\, \mathtt{x}_i\ .$$

In this case, we say that the *weight* of the node is w.

Looking at a couple of examples should make the notion easier to grasp:

- for the substitution $(\mathtt{x}_1 := \mathtt{AB}\delta_w\mathtt{B}^-\mathtt{A}^-\mathtt{x}_1\ ,\ \mathtt{x}_2 := \mathtt{C}\mathtt{x}_1)$, the only diagonal node is $(\mathtt{B}^-\mathtt{A}^-, 1)$, of weight w,
- In general, for a substitution $(\mathtt{x}_1 := u_1, \dots, \mathtt{x}_n := u_n)$, if u_i is of the form $\mathtt{A}_1 \dots \mathtt{A}_m \delta_w \mathtt{B}_n^- \dots \mathtt{B}_1^- \mathtt{x}_i$, $(\mathtt{B}_n^-, \dots, \mathtt{B}_1^-, i)$ is a diagonal node iff $\mathtt{A}_m^- \dots \mathtt{A}_1^-$ is a suffix of $\mathtt{B}_n^- \dots \mathtt{B}_1^-$. Its weight is then $w - n + m$.
- With tuple constructors, the situation isn't as simple. As an example, consider

$$\big(\mathtt{x}_1 := (\delta_3\pi_1\mathtt{x}_1, \delta_2\pi_2\mathtt{x}_2)\ ,\ \mathtt{x}_2 := (\delta_1\pi_2\mathtt{x}_1, \delta_\infty\pi_1\mathtt{x}_2)\big)$$

the only diagonal node is $(\pi_1, 1)$ of weight 3.

Here is the first use of the combinatorial principle of section 1:

Lemma 5.3. *Suppose we are given $\tau \simeq \tau \diamond \tau$ and an infinite sequence $(\tau_n)_{n\geq 0}$ of substitutions in $\mathcal{F}_{d,b}$ with $\tau_i \simeq \tau$.*

- *All the partial compositions $\tau_{m+n} \circ \ldots \circ \tau_m$ share the same diagonal nodes (possibly with different weights): they are exactly the diagonal nodes of τ.*
- *Moreover, we have:*
  (1) *either there are n and m s.t. all diagonal nodes have non-negative weight in $\tau_{m+n} \circ \ldots \circ \tau_m$,*
  (2) *or τ contains a diagonal node whose weight in $\tau_n \circ \ldots \circ \tau_1$ diverges to $-\infty$ as n tends to ∞.*[8]

Proof. For the first point, notice that being a diagonal node doesn't depend on the weights of the δ, only the weight of the diagonal node does. Thus, the sequence $(\tau_n)_n$ of the lemma induces a sequence of tuples of weights for all the diagonal nodes.

For the second point, the first thing to show is that diagonal nodes compose during substitution. More generally, confluence implies

Fact (1). If $\delta_0 \vec{a} \cdot u_i = \delta_w \vec{b} \, \mathtt{x}_j$ and $\delta_0 \vec{b} \cdot v_i = \delta_{w'} \vec{c} \, \mathtt{x}_k$ then

$$\delta_0 \vec{a} \cdot \big(u_i[\mathtt{x}_j := v_j]\big) \quad = \quad \delta_{w+w'} \vec{c} \, \mathtt{x}_k \ .$$

This shows that if $\tau_1 \simeq \tau_2$ and $(\vec{d}, i)$ is a diagonal node in τ_1 (and thus in τ_2), the weight of $(\vec{d}, i)$ in $\tau_2 \circ \tau_1$ is the sum of its weights in τ_1 and τ_2.

Fact (2). If $(\boldsymbol{w}_n)_n$ is a sequence of vectors in ${\mathbf{Z}_b}^k$ s.t. all pointwise initial partial sums are bounded from below:

$$\forall n \qquad (-B, \ldots, -B) \quad \leq \quad \sum_{i=1}^{n} \boldsymbol{w}_i$$

then there is a partial sum in which all components are positive:

$$\exists m, n \quad \sum_{i=m}^{m+n} \boldsymbol{w}_i \quad \geq \quad (0, \ldots, 0)$$

Proof of fact. Given such a sequence $(\boldsymbol{w}_n)_n$, we add some $(-1, 0, \ldots, 0)$ in enough places to make sure the initial sums of the first components is always in $\mathbf{Z}_{B'}$ where B' is $\max(B, b)$. This is always possible. We do the same for all the other components: we obtain a new sequence $(\boldsymbol{w}'_n)_n$ which satisfies

$$\forall n \quad \sum_{i=1}^{n} \boldsymbol{w'}_i \quad \in \quad {\mathbf{Z}_{B'}}^k \ .$$

This implies that we also have

$$\forall m, n \quad \sum_{i=m}^{m+n} \boldsymbol{w}_i \quad \in \quad {\mathbf{Z}_{2B'}}^k \ .$$

By (a slight generalization of) proposition 1.1,[9] we can decompose this sequence as:

$$\text{initial prefix}, \qquad \underbrace{\boldsymbol{w}'_{n_0}, \ldots \boldsymbol{w}'_{n_1 - 1}}_{\boldsymbol{w}'}, \quad \ldots, \quad \underbrace{\boldsymbol{w}'_{n_k}, \ldots \boldsymbol{w}'_{n_{k+1} - 1}}_{\boldsymbol{w}'}, \quad \ldots$$

where:

- all the $\boldsymbol{w}'_{n_k} + \cdots + \boldsymbol{w}'_{n_{k+1}-1}$ are equal to the same $\boldsymbol{w}'$,
- $\boldsymbol{w}' = \boldsymbol{w}' + \boldsymbol{w}'$.

---

[8] *i.e.* there is a diagonal node $(\vec{d}, i)$ of τ with $\forall B \geq 0 \; \exists n$ s.t. the weight of $(\vec{d}, i)$ in $\tau_n \circ \ldots \circ \tau_1$ is less than $-B$

[9] see the note at the end of the proof of proposition 1.1.

That $\boldsymbol{w}' = \boldsymbol{w}' + \boldsymbol{w}'$ implies trivially that all components of $\boldsymbol{w}'$ are either 0 or ∞.

Take the subsequence $\boldsymbol{w}'_{n_0}, \dots \boldsymbol{w}'_{n_1-1}$ and remove the additional $(-1, 0, \dots, 0)$ to obtain a subsequence of the original $(\boldsymbol{w}_n)_n$. The sum of this subsequence contains only non-negative components. □

Applying this fact to the sequence of the tuple of weights of diagonal nodes of $(\tau_n)_n$, we get the lemma: suppose the compositions $\tau_n \circ \dots \circ \tau_1$ do not contain a diagonal node whose weight diverges to $-\infty$. This translates into "the initial sequences of weights are bounded from above", and thus by the fact (2), there is a subsequence of the weights for which all sums are non-negative. This translates back to a subsequence $\tau_{m+n} \circ \dots \circ \tau_m$ in which all diagonal nodes have non-negative weights.

□

5.2. **Main result.**

Proposition 5.1 (Improved Size Change Termination Principle)**.** *If G safely represents some inductive definitions and all "weakly" idempotent loops $\tau \simeq \tau \diamond \tau$ in ${G_{d,b}}^+$ contain a diagonal node with strictly negative weight then the evaluation of the functions on values cannot produce an infinite sequence of calls to other functions.*[10]

Proof of proposition 5.1. Suppose the conditions of the proposition are satisfied and suppose that function `h` on values $v_1, \dots, v_n$ provokes an infinite sequence of calls $c_1, \dots, c_n, \dots$. Write ρ_n for the arguments of call c_n, in particular, ρ_0 corresponds to the initial arguments of `h`: $(\mathtt{x}_1 := v_1, \dots, \mathtt{x}_n := v_n)$. All those ρ_n contain (semantical) values.

Let $\sigma_1, \dots, \sigma_n, \dots$ be the substitutions in $G_{d,b}$ corresponding to the calls $c_1, \dots$. By considering substitutions up-to $\simeq$, we can use proposition 1.1 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 "$\simeq$" to the same substitution $\tau : \mathtt{f} \to \mathtt{f}$,
- τ is idempotent up-to $\simeq$: $\tau \simeq \tau \diamond \tau$.

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

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

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

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

Because $\circ$ is associative, and because collapsing and composition are monotonic, we obtain:

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

The composition $\sigma_{n_1-1} \diamond \dots \diamond \sigma_{n_0}$ is equal to some $\tau_1 \simeq \tau$. Doing the same thing several times, we get:

$$\rho_{n_k} \quad \sqsubseteq \quad \tau_k \circ \dots \circ \tau_1 \ \circ \ \rho_{n_0} \ .$$

Since the graph satisfies the size-change termination principle, no $\tau_{k+l} \circ \dots \circ \tau_k$ can have all diagonal weights positive. By lemma 5.3, one of the diagonal weights must diverge to $-\infty$. By taking a k big enough, this diagonal weight will be small

[10]Recall that $\sigma \simeq \tau$ means that σ and τ only differ on some weights of δ's.

enough to ensure that $\tau_k \circ \ldots \circ \tau_1 \circ \rho_{n_0}$ contains a negative δ. This contradicts the fact that $\rho_{n_k} \sqsubseteq \tau_k \circ \ldots \circ \tau_1 \circ \rho_{n_0}$ (by lemma 5.1).

Thus, we cannot get an infinite sequence of calls.

□

Lemma 5.4. *The size-change principle in monotonic with respect to d and b.*

5.3. **A stronger test?** It is tempting to replace idempotent loops up-to $\simeq$ by real idempotent loops in proposition 5.1 The resulting principle is still sound but isn't any stronger. More precisely:

Lemma 5.5. *The following are equivalent: in ${G_{d,b}}^+$,*

- *all* weakly *idempotents have a diagonal node with strictly negative weight,*
- *all* strictly *idempotents have a diagonal node with strictly negative weight.*

Proof. This follows from the fact that the weights of diagonal nodes are summed along a composition of weakly-idempotent substitutions (fact (1) in the proof of lemma 5.3). Thus, given a weakly-idempotent substitution, we can compose (with $\diamond$) it with itself enough time so that the weights of each diagonal node converges to a value.

It is not terribly difficult to show that the weights of *each* δ also converges to a value and that we obtain a truly idempotent loop. If the initial substitution contained a diagonal node of positive weight, the final one, contains a diagonal node of weight ∞.

□

Corollary. *If G safely represents some inductive definitions and all strictly idempotent loops $\sigma = \sigma \diamond \sigma$ in ${G_{d,b}}^+$ contain a diagonal branch whose weight is strictly negative, then the evaluation of the functions on values cannot produce an infinite sequence of calls to other functions.*

In practice there is no difference as checking equality and checking $\simeq$ have the same complexity.

5.4. **Comparison with original principle.** When using the original principle described in [1] for an ML-like language, it is quite natural to count variant constructors as the notion of size. This will ensure that we at least detect structural induction. Then, the original principle is a special case of the above test: choose the bounds $b = 1$ and $d = 0$. The substitution $(\mathtt{x}_1 := \delta_0 \mathtt{x}_1, \mathtt{x}_2 := \delta_{-1} \mathtt{x}_0, \mathtt{x}_3 := \delta_\infty \mathtt{x}_2)$ can be pictured as

$$\begin{pmatrix} = & < & ? \\ ? & ? & \infty \end{pmatrix} \quad \text{or} \quad \begin{array}{ccc} \mathtt{x}_1 & \xrightarrow{\delta_0} & \mathtt{x}_1 \\ \mathtt{x}_2 & \nearrow_{\delta_{-1}} & \mathtt{x}_2 \\ \mathtt{x}_3 & \nearrow_{\delta_\infty} & \end{array} \; .$$

The original paper used the graph representation with the notation "⇟" for δ_0, "↓" for δ_{-1} and did not draw δ_∞. What should be noted is that in this context (when $d = 0$), there is no difference between u and $\delta_0 u$ and that as far as termination is concerned, "?" (*i.e.* "δ_∞") carries the same information as $\delta_\infty \mathtt{x}_i$

5.5. **Using the order.** The following is rather obvious once the notion of diagonal node is understood:

Fact. If $\sigma \sqsubseteq \tau$ and $(\vec{d}, i)$ is a diagonal node of σ, then there is a diagonal node $(\vec{e}, i)$ of τ where $\vec{e}$ is a suffix of $\vec{d}$. The weight of $(\vec{d}, i)$ in σ is less than the weight of $(\vec{c}, i)$ in τ.

This makes it possible to improve the practical complexity of the test, in particular when the bounds d and b are "big".[11] By monotonicity of composition (propositions 4.1) and the previous fact, it is clear that we only need to keep the maximal substitutions between vertices $\mathtt{f}$ and $\mathtt{g}$. We compute the transitive closure with

$$G^{n+1}[\mathtt{f},\mathtt{g}] \quad = \quad \max\Big(G^n[\mathtt{f},\mathtt{g}] \cup \big\{\sigma \diamond \tau \ \mid\ \tau \in G^n[\mathtt{f},\mathtt{h}], \sigma \in G^n[\mathtt{h},\mathtt{g}]\big\}\Big)$$

where $\mathtt{g}$ ranges over all vertices of G^n and the max function takes *all* maximal elements of this set. Proving that the resulting criterion is equivalent is straightforward.

In the implementation, it amounts to replacing the function inserting a new element τ in the set of arcs by a function which removes any element smaller than τ and does nothing if it finds an element bigger than τ. Since sets in Caml are implemented by balanced binary trees, plain insertion in a set of of size n has complexity $O(\log(n))$ whereas the "improved" insertion has complexity $O(n\log(n))$.

In practice, this is more than compensated by the fact that sets of calls are smaller, making the computation of the transitive closure faster. It is possible to create examples where the sets get exponentially smaller when using the approximation order. A paradigmatic (if circumvoluted) example of this phenomenon is given by:

```
val rec f x1 x2 x3 x4 =
  f (f x1 x2 x3 x4) (f x2 x1 x3 x4) (f x1 x3 x2 x4) (f x1 x2 x4 x3)
```

which generates the following set of calls:

$$\begin{array}{ccc}
\{ & (\mathtt{x}_1 := \mathtt{x}_1, \mathtt{x}_2 := \mathtt{x}_2, \mathtt{x}_3 := \mathtt{x}_3, \mathtt{x}_4 := \mathtt{x}_4) & , \\
 & (\mathtt{x}_1 := \mathtt{x}_2, \mathtt{x}_2 := \mathtt{x}_1, \mathtt{x}_3 := \mathtt{x}_3, \mathtt{x}_4 := \mathtt{x}_4) & , \\
 & (\mathtt{x}_1 := \mathtt{x}_1, \mathtt{x}_2 := \mathtt{x}_3, \mathtt{x}_3 := \mathtt{x}_2, \mathtt{x}_4 := \mathtt{x}_4) & , \\
 & (\mathtt{x}_1 := \mathtt{x}_1, \mathtt{x}_2 := \mathtt{x}_2, \mathtt{x}_3 := \mathtt{x}_4, \mathtt{x}_4 := \mathtt{x}_3) & , \\
 & \big(\mathtt{x}_1 := \delta_\infty(), \mathtt{x}_2 := \delta_\infty(), \mathtt{x}_3 := \delta_\infty(), \mathtt{x}_4 := \delta_\infty()\big) & \}
\end{array}$$

The transitive closure without using approximation will generate all the

$$\big(\mathtt{x}_1 := \mathtt{x}_{p(1)}, \mathtt{x}_2 := \mathtt{x}_{p(2)}, \mathtt{x}_3 := \mathtt{x}_{p(3)}, \mathtt{x}_4 := \mathtt{x}_{p(4)}\big)$$

for all permutations p. If using the order, all those calls are approximated by the initial $\big(\mathtt{x}_1 := \delta_\infty(), \mathtt{x}_2 := \delta_\infty(), \mathtt{x}_3 := \delta_\infty(), \mathtt{x}_4 := \delta_\infty()\big)$ and the computation of the transitive closure takes only one step.

Similar examples do occur in practice, whether terminating or not.

5.6. **Examples.** All the examples in this section validate the size-change termination principle. They are chosen to illustrate specific properties and are not necessarily meant to compute anything useful.

This test validates any function which is validated by the original test when using "variant constructor height" as the size function. This includes in particular any function defined by structural induction but also functions which swap arguments around like:

```
val rec f x y = match x,y with
    S[x],S[y] -> if (...) then f x S[y] else f y S[x]
  | _,_ -> A[]
```

or similar behavior as described in the original paper.

[11]In practical examples, b rarely needs to be bigger than 2 or 3, and d rarely needs to be bigger than 4 or 5. The default values in the implementation are $b = 1$ and $d = 2$, but the user can dynamically change them.

The first novelty is that allowing restricted positive δ's, we can have a locally increasing size without loosing termination:

```
val rec f x = g S[x]
    and g x = match x with
                  S[S[x]] -> f x
                | _ -> Z[]
```

Even with the bound $d = 0$, since the increasing call ($\mathtt{x}_1 := \delta_1 \mathtt{x}_1$) from `f` to `g` is followed by a call ($\mathtt{x}_1 := \delta_{-2} \mathtt{x}_1$) from `g` to `f`, the transitive closure will keep enough information to see that these functions terminate. (This of course requires that $b > 1$.)

More importantly, the propagation of impossible cases can remove unwanted arcs in the call-graph:

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

If we were to keep only size information (*i.e.* use a bound $d = 0$), this wouldn't pass the test as the first call is represented by ($\mathtt{x}_1 := \delta_0 \mathtt{x}_1$), which is idempotent with only one diagonal node of weight 0. With $d > 0$ however, we get two recursive calls which are represented by the substitutions $\sigma_1 = (\mathtt{x}_1 := \mathtt{B}\mathtt{A}^- \mathtt{x}_1)$ and $\sigma_2 = (\mathtt{x}_1 := \mathtt{B}^- \mathtt{x}_1)$. Since σ_1 does not compose with itself, the transitive closure will only follow path of the form $\sigma_2^n \circ \sigma_1$ or σ_2^n.

One nice but subtle point is that while the original principle uses a global size notion, the present principle uses a "local" notion of size given by weight of diagonal nodes. In other words, it detects size decreasing in specific sub-parts of the arguments. Here are two functions to illustrate this:

```
val rec f = fun  T[S[x1],x2] -> f T[x1,S[x2]]  | _ -> A[]
val rec g = fun  T[x1,_[x2]] -> g T[S[x1],x2]  | _ -> A[]
```

For $d = 2$ and $b = 1$, the call initial call-graphs are:

- $\big(\mathtt{x}_1 := \mathtt{T}(\delta_{-1}\pi_2\mathtt{T}^-\mathtt{x}_1\ , \delta_\infty\pi_1\mathtt{T}^-\mathtt{x}_1)\big)$ for `f`,
- $\big(\mathtt{x}_1 := \mathtt{T}(\delta_\infty\pi_1\mathtt{T}^-\mathtt{x}_1\ ,\ \delta_{-1}\pi_2\mathtt{T}^-\mathtt{x}_1)\big)$ for `g`.

Those call-graphs are equal to their transitive closure. In both cases, the test is able to see that one branch is actually decreasing: there is only one diagonal node in each graph:

- $(\pi_1\mathtt{T}^-, 1)$ of weight -1 for `f`,
- $(\pi_2\mathtt{T}^-, 1)$ of weight -1 for `g`.

Each of those functions could pass the original test, provided the user chose the appropriate notion of size: look on the left-branch (for `f`) or look on the right-branch (for `g`). What seems to be new is that the test somehow includes the search of a "good" notion of size. This is what allows the next function to pass the termination test:

```
val rec sum : (list nat => nat) = fun
      Nil[] -> Z[]
    | Cons[acc,Nil[]] -> acc
    | Cons[acc,Cons[Z[],l]] -> sum Cons[acc,l]
    | Cons[acc,Cons[S[n],l]] -> sum Cons[S[acc],Cons[n,l]]
```

which uses the first element of a list to accumulate as many `S` variant constructors as there are in the original list. Since in the second call, we need to look at "$\pi_1\mathtt{Cons}^-\pi_2\mathtt{Cons}^-\mathtt{x}_1$" to reach the decreasing part, the bound d needs to be at

least 4. Because of this bound, the transitive closure computes a little more arcs than we'd like and we end up with a call-graph with 20 arcs.

In functional programming, it is customary to use functions with several arguments rather than functions with one tuple of arguments. As a final remark, we note that this doesn't change the result of the test, save that we may need to increase the bound d by one to account for the additional tuple constructor in the curried version of a function.

5.7. **Non-examples, limitations.** To see the limit of this test, it is helpful to look at some kinds of behavior that prevent it from detecting termination.

The first limitation is that since this is mostly based on terms (and not types), any call to a function is replaced by $\delta_\infty()$. For example:

```
val rec f = fun Nil[] -> Nil[]
              | Cons[n,l] -> Cons[twice a , list.map twice l]
```

is bound to fail because we don't know that map preserves the length of its second argument.

Future work include ideas to use "sized-types" in conjunction with size-change termination to deal with this kind of examples.

Another limitation comes from recursive calls on constant values. Consider

```
val rec f = fun
    Z[] -> Z[]
  | S[Z[]] -> f Z[]
  | S[n] -> f n
```

The first recursive called is represented by the substitution $(\mathtt{x}_1 := \mathtt{Z}())$ which is idempotent but without diagonal node. This function will be tagged as non-terminating. A hack to make this function terminating is to use "aliases" and replace "`| S[Z[]] -> f Z[]`" by

```
  | S[Z[] as x] -> f x
```

A more permanent fix for this is to add information to the substitution concerning the sequence of pattern-matching leading to the call. Here, the call would be represented by

$$(\mathtt{x}_1 := \mathtt{Z}()) \quad | \quad \underbrace{\mathtt{x}_1 \equiv \mathtt{SZ}()}_{\text{shape of parameters for the call}}$$

while the other call would be represented by

$$(\mathtt{x}_1 := \mathtt{Sx}_1) \quad | \quad \mathtt{x}_1 \equiv \mathtt{S}(_) \ .$$

The reason this hasn't been implemented is because PML's code analysis doesn't, at the moment, output this information. This is however easily extracted from the naive analysis presented in appendix B.

Adding the shape of arguments would also allow the test to detect more impossible cases, like in:

```
val rec f x = match x with
    B[y] -> f C[x]
  | C[y] -> f y
  | _ -> D[]
```

This is not found to be terminating because the recursive call is represented by the substitution $(\mathtt{x}_1 := \mathtt{Cx}_1)$. If the function were written as

```
val rec f x = match x with
    B[y] -> f C[B[y]]
  | C[y] -> D[]
```

```
  | _ -> D[]
```

the call would be represented by ($\mathtt{x}_1$:= $\mathtt{CBB}^-\mathtt{x}_1$) and the function would pass the termination test because this call cannot be composed with itself. Like above, adding the shape of the parameters would be enough to detect such impossible compositions:

$$(\mathtt{x}_1 := \mathtt{C}\mathtt{x}_1) \quad | \quad \mathtt{x}_1 \equiv \mathtt{B_}\ .$$

It might sometimes be necessary to give more weight to specific variant constructors. At the moment, each variant constructor has weight 1, as can be seen in the reduction rule $\delta_w \mathtt{C} u \triangleright \delta_{w+1} u$. This 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[]
```

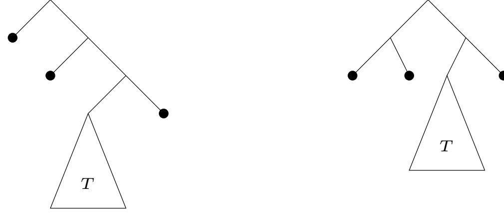
This function does pass the termination test if we choose a depth $d \geq 7$. If the function contained other recursive calls, it can make the test use more resources than reasonable.

Giving a weight of 3 to B and 1 to C would be enough to see this is terminating, even when $d = 0$.[12]

The last example seems to be outside the range of size-change termination. The combing function:

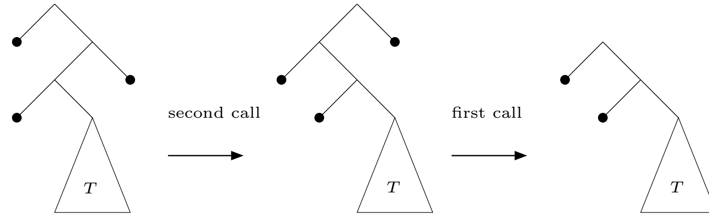
```
val rec comb t = match t with
     Leaf[] -> Leaf[]
   | Node[t,Leaf[]] -> Node[comb t,Leaf[]]
   | Node[t1,Node[t2,t3]] -> comb Node[Node[t1,t2],t3]
```

terminates for a very subtle reason. It seems impossible to capture this function for the following reason: for any bound d, for any branch $\vec{a}$ of lenght d, it is possible to find a tree t for which the subtree $\mathrm{nf}(\vec{d}t)$ increases arbitrarily during a sequence of recursive calls. For example, at $d = 4$ for $\vec{a} = \pi_1\mathtt{Node}^-\pi_2\mathtt{Node}^-$, 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 it possible: for $\pi_2\mathtt{Node}^-\pi_2\mathtt{Node}^-$, we need to use the second call and then the first call, for example on



[12]Believe it or not, similar situation did arise in practical examples.

This implies that during the transitive closure, all branches of lenght d will reach a δ_∞ because they can be composed with such a tree. Thus, all diagonal nodes have weight ∞.

Even if this is rather unfortunate, this example (and many similar ones) can be dealt with by adding additional arguments. The function `comb_size` with a second parameter for (a bound of) 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 outside the programming language) that the actual `comb` function simply is:[13]

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

Concluding Remarks

Complexity. Section 5.4 makes it clear that the complexity of the original principle hasn't improved. Computing the transitive closure of a call-graph is still P-space complete! It should be noted however that, like for the original principle, the complexity of the implemented algorithm is perfectly reasonable in practice. Testing termination for PML programs still takes a lot less resources than the other parts of the correctness checking (type checking, completeness of pattern matching etc.) except for some very specific examples.

Counting abstractions. Since the PML language computes only weak-head normal forms, it is possible to extend the principle to detect that functions such as

```
val rec glutton f = fun x -> (glutton f)
```

do indeed terminate: when applied to n arguments, it will "eat" through all of them and stop on the weak-head normal form `fun x -> (F f)`.

In order to do that, we add a virtual argument to all recursive functions: it simply counts the difference between the number of abstraction and the number of applications. Think of it as an additional "$\mathtt{x}_0 := \delta_w \mathtt{x}_0$" in all substitutions. Note that an abstraction counts positively and an application counts negatively so that in effect, it amounts to having a *constructor* Δ for abstraction and a *destructor* Δ^- for application.

This could prove useful in dealing with "frozen" variant constructors: variant constructors which block reduction until a pattern-matching. This might also be applied to the destructor based coinductive datatypes as advocated by Anton Setzer. The interplay between Δ, frozen / non-frozen variants and tuples isn't clear at the moment. This is left for future work.

Integrating Sized-Types. Besides a couple of optimizations, the next natural step seems to be able to integrate "size-types" in the principle. The idea would be to start dealing with applications in arguments. For example, if one were to know that `map` keeps the same number of list constructors, then a call to `map f l` could be replaced by $\delta_0 l$.

This is of course more difficult than that since while the number of list constructors doesn't change, the number of other constructors (in the elements of the list) could increase.

[13] One would of course like to tag the additional parameter `s` as "computationally irrelevant" so that it isn't used during real computation.

5.7.1. *Loose ends.* It was surprising that proposition 1.1 is used twice during the proof of correctness of the principle. It “feels” like both uses are on different levels, and factoring both uses in a single one doesn’t look easy: ones deals with terms up to $\simeq$, while the other deals with equality of weights.

References

[1] Chin Soon Lee, Neil D. Jones and Amir M. Ben-Amram, “The Size-Change Principle for Program Termination”. *ACM SIGPLAN Notices*, Volume 36 , Issue 3, 2001.

[2] Christophe Raffalli, “Realizability for programming languages”, course notes for the *École jeunes chercheurs du GDR IM*, 2010. Submitted for publication.

[3] Christophe Raffalli, “The PML programming language”, ...

[4] Christophe Raffalli, “PML online”, `http://lama.univ-savoie.fr/~pml/interactive.php`, an interactive online interpreter for PML.

Appendix A. Missing proof

Lemma A.1. sup *is associative:* $\sup\big(u, \sup(v,t)\big) = \sup\big(\sup(u,v), t\big)$.

Proof. The proof relies on the following fact:

Fact. For all u and v in normal form, we have

$$(*)\qquad \delta_w\cdot \sup(u,v) \quad=\quad \sup\big(\delta_w\cdot u\ ,\ \delta_w\cdot v\big) \quad=\quad \sup_0\big(\delta_w\cdot u\ ,\ \delta_w\cdot v\big)\ .$$

The proof of this is rather straightforward.

Let's look at the relevant cases for associativity:

- if all of u, v and t are in $\mathcal{B}$, this is just associativity of $\sup_0$.
- If all of u, v and t are compatible (they start with the same variant constructor, or they are tuples of equal length), the result holds directly by the induction hypothesis,
- If $u = b_1 \in \mathcal{B}$, $v = b_2 \in \mathcal{B}$ and $t = \mathtt{C}t'$, we can use the definition of sup and associativity of $\sup_0$:

$$\begin{aligned}
\sup\big(\sup(b_1\ ,\ b_2)\ ,\ \mathtt{C}t'\big) &= \sup\big(\sup(b_1\ ,\ b_2)\ ,\ \delta_0\cdot \mathtt{C}t'\big)\\
&= \sup_0\big(\sup_0(b_1\ ,\ b_2)\ ,\ \delta_1\cdot t'\big)\\
&= \sup_0\big(b_1\ ,\ \sup_0(b_2\ ,\ \delta_1\cdot t')\big)\\
&= \sup_0\big(b_1\ ,\ \sup_0(b_2\ ,\ \delta_0\cdot \mathtt{C}t')\big)\\
&= \sup\big(b_1\ ,\ \sup(b_2\ ,\ \mathtt{C}t')\big)
\end{aligned}$$

- If $u = b_1 \in \mathcal{B}$, $v = b_2 \in \mathcal{B}$ and $t = (t_1, \dots, t_n)$, this is very similar:

$$\begin{aligned}
&\sup\Big(\sup(b_1\ ,\ b_2)\ ,\ (t_1,\dots,t_n)\Big)\\
=\ &\sup_0\Big(\sup(b_1\ ,\ b_2)\ ,\ \delta_0\cdot(t_1,\dots,t_n)\Big)\\
=\ &\sup_0\Big(\sup_0(b_1\ ,\ b_2)\ ,\ \sup_{0,1\le i\le n}\{\delta_0\cdot t_i\}\Big)\\
=\ &\sup_0\Big(b_1\ ,\ \sup_0\big(b_2\ ,\ \sup_{0,1\le i\le n}\{\delta_0\cdot t_i\}\big)\Big)\\
=\ &\sup\Big(b_1\ ,\ \sup\big(b_2\ ,\ \delta_0\cdot(t_1,\dots,t_n)\big)\Big)\\
=\ &\sup\Big(b_1\ ,\ \sup\big(b_2\ ,\ (t_1,\dots,t_n)\big)\Big)
\end{aligned}$$

- If $u = b \in \mathcal{B}$ and $v = \mathtt{C}v'$ and $t = \mathtt{C}t'$, we need to use the above fact

$$\begin{aligned}
\sup\big(b\ ,\ \sup(\mathtt{C}v'\ ,\ \mathtt{C}t')\big) &= \sup\big(b\ ,\ \mathtt{C}\sup(v'\ ,\ t')\big)\\
&= \sup_0\big(b\ ,\ \delta_1\cdot \sup(v'\ ,\ t')\big)\\
&= \sup_0\big(b\ ,\ \sup_0(\delta_1\cdot v'\ ,\ \delta_1\cdot t')\big) \quad \text{by } (*)\\
&= \sup_0\big(\sup_0(b\ ,\ \delta_1\cdot v')\ ,\ \delta_1\cdot t'\big)\\
&= \sup_0\big(\sup(b\ ,\ \mathtt{C}v')\ ,\ \delta_1\cdot t'\big)\\
&= \sup\big(\sup(b\ ,\ \mathtt{C}v')\ ,\ \mathtt{C}t'\big)
\end{aligned}$$

- If $u = b \in \mathcal{B}$ and $v = \mathtt{C}v'$ and $t = \mathtt{D}t'$, with $\mathtt{C} \neq \mathtt{D}$:

$$\begin{aligned}
\sup\big(b\ ,\ \sup(\mathtt{C}v'\ ,\ \mathtt{D}t')\big) &= \sup_0\big(b\ ,\ \sup_0(\delta_1\cdot v'\ ,\ \delta_1\cdot t')\big)\\
&= \sup_0\big(\sup_0(b\ ,\ \delta_1\cdot v')\ ,\ \delta_1\cdot t'\big)\\
&= \sup_0\big(\sup(b\ ,\ \mathtt{C}\cdot v')\ ,\ \delta_1\cdot t'\big)\\
&= \sup\big(\sup(b\ ,\ \mathtt{C}\cdot v')\ ,\ \mathtt{D}\cdot t'\big)
\end{aligned}$$

- If $u = b \in \mathcal{B}$ and $v = (v_1, \ldots, v_n)$ and $t = (t_1, \ldots, t_n)$:

$$\begin{array}{rl} & \sup\Big(b\ ,\ \sup\big((v_1,\ldots,v_n)\ ,\ (t_1,\ldots,t_n)\big)\Big) \\ = & \sup\Big(b\ ,\ \big(\sup(v_1,t_1),\ldots,\sup(v_n,t_n)\big)\Big) \\ = & \sup_0\Big(b\ ,\ \delta_0\cdot\big(\sup(v_1,t_1),\ldots,\sup(v_n,t_n)\big)\Big) \\ = & \sup_0\Big(b\ ,\ \sup_{0,1\le i\le n}\{\delta_0\cdot\sup(v_i,t_i)\}\Big) \\ = & \sup_0\Big(b\ ,\ \sup_{0,1\le i\le n}\{\sup_0(\delta_0\cdot v_i\ ,\ \delta_0\cdot t_i)\}\Big) \quad \text{by } (*) \\ = & \sup_0\Big(\sup_0\big(b\ ,\ \sup_{0,1\le i\le n}\{\delta_0\cdot v_i\}\big)\ ,\ \sup_{0,1\le i\le n}\{\delta_0\cdot t_i\}\Big) \\ = & \sup_0\Big(\sup\big(b\ ,\ (v_1,\ldots,n_n)\big)\ ,\ \sup_{0,1\le i\le n}\{\delta_0\cdot t_i\}\Big) \\ = & \sup\Big(\sup\big(b\ ,\ (v_1,\ldots,n_n)\big)\ ,\ (t_1,\ldots,t_n)\Big) \end{array}$$

- If $u = b \in \mathcal{B}$ and $v = (v_1, \ldots, v_n)$ and $t = (t_1, \ldots, t_m)$ with $n \neq m$:

$$\begin{array}{rl} & \sup\Big(b\ ,\ \sup\big((v_1,\ldots,v_n)\ ,\ (t_1,\ldots,t_m)\big)\Big) \\ = & \sup\Big(b\ ,\ \sup_0\big(\delta_0\cdot(v_1,\ldots,v_n)\ ,\ \delta_0\cdot(t_1,\ldots,t_m)\big)\Big) \\ = & \sup_0\Big(b\ ,\ \sup_0\big(\sup_{0,1\le i\le n}\{\delta_0\cdot v_i\}\ ,\ \sup_{0,1\le i\le m}\{\delta_0\cdot t_i\}\big)\Big) \\ = & \sup_0\Big(\sup_0\big(b\ ,\ \sup_{0,1\le i\le n}\{\delta_0\cdot v_i\}\big)\ ,\ \sup_{0,1\le i\le m}\{\delta_0\cdot t_i\}\Big) \\ = & \sup_0\Big(\sup\big(b\ ,\ (v_1,\ldots,v_n)\big)\ ,\ \delta_0\cdot(t_1,\ldots,t_m)\Big) \\ = & \sup\Big(\sup\big(b\ ,\ (v_1,\ldots,v_n)\big)\ ,\ (t_1,\ldots,t_m)\Big) \end{array}$$

- If $u = b \in \mathcal{B}$ and $v = (v_1, \ldots, v_n)$ and $t = \mathtt{C}t'$:

$$\begin{array}{rl} & \sup\Big(b\ ,\ \sup\big((v_1,\ldots,v_n)\ ,\ \mathtt{C}t'\big)\Big) \\ = & \sup_0\Big(b\ ,\ \sup_0\big(\sup_{0,1\le i\le n}\{\delta_0\cdot v_i\}\ ,\ \delta_1\cdot t'\big)\Big) \\ = & \sup_0\Big(\sup_0\big(b\ ,\ \sup_{0,1\le i\le n}\{\delta_0\cdot v_i\}\big)\ ,\ \delta_1\cdot t'\Big) \\ = & \sup_0\Big(\sup\big(b\ ,\ (v_1,\ldots,v_n)\big)\ ,\ \delta_1\cdot t'\Big) \\ = & \sup\Big(\sup\big(b\ ,\ (v_1,\ldots,v_n)\big)\ ,\ \mathtt{C}t'\Big) \end{array}$$

- All the remaining cases are variants of those and can be handled using commutativity.

□

Appendix B. Naive Static Analysis

The simplest static analysis is just a syntactical analysis of the code. Each recursive call

```
val rec f x1 ... xn =
  ... (g u1 ... um)
    and g x1 ... xm = ...
```

is represented by the substitution

$$(\mathtt{x}_1 := \mathtt{u1}[\rho]\ ,\ \ldots\ ,\ \mathtt{x}_m := \mathtt{um}[\rho]) \qquad [*]$$

where ρ is the substitution that keeps track of the current pattern-matching branch. It initialized to $(\mathtt{x}_1 := \mathtt{x}_1, \ldots, \mathtt{x}_n := \mathtt{x}_n)$ and updated as follows:

$$\underbrace{\ldots}_{\rho} \quad \texttt{match u with ...}\quad \texttt{A[y1,...,yk] ->} \quad \underbrace{\ldots}_{\rho\cdot(\mathtt{y1}:=\pi_1\mathtt{A}^-\mathtt{u},\ldots,\mathtt{yk}:=\pi_k\mathtt{A}^-\mathtt{u})}$$

In $[*]$ above, every $\mathtt{ui}[\rho]$ which isn't an element of $\mathcal{F}$ is replaced by $\delta_\infty()$.

This analysis yield a safe representation, because each component in the substitution exactly represents the corresponding argument (or is $\delta_\infty()$ which is bigger than any value).

As an example, consider

```
val rec f x =
  match x with A[B [y1,y2,y3]] -> f C[y3,g y2,A[y2]]
               C[y1,y2,y3] -> ...
               ...
```

where g is some function in the environment. This call will be represented by

$$\sigma = \Big(\mathtt{x}_1 := \mathtt{C}\Big(\ \pi_3\mathtt{B}^-\mathtt{A}^-\mathtt{x}_1\ ,\ \delta_\infty(),\ \mathtt{A}\,\pi_2\mathtt{B}^-\mathtt{A}^-\mathtt{x}_1\ \Big)\Big)\ .$$

For any argument v to f that is fed to this recursive call, this argument is necessarily of the form $v = \mathtt{AB}(v_1, v_2, v_3)$. In that case, (the semantics of) the argument of the function g is $\mathtt{C}(v_3, ?, \mathtt{A}v_2)$. We have

$$\big(\mathtt{x}_1 := \mathtt{C}(v_3, [\![\mathtt{g}\ v_2]\!], \mathtt{A}v_2)\big) \quad \sqsubseteq \quad \big(\mathtt{x}_1 := \mathtt{C}(v_3, \delta_\infty(), \mathtt{A}v_2)\big) = \sigma \circ (\mathtt{x}_1 := v)$$

so that this is indeed a safe representation of the call.

Appendix C. Implementation bits and pieces

Here are some Caml pieces from the actual implementation. The whole file is available in the source of PML (http://www.lama.univ-savoie.fr/~pml) or directly at http://www.lama.univ-savoie.fr/~hyvernat/Files/sct.ml.

Without the various bureaucratic parts (printing and debug code) the implementation is about 600 lines of Caml code.

Types for $\mathcal{F}$. The type for $\mathbf{Z}_\infty$ and $\mathcal{F}$ are defined as:

```
type z_infty = Number of int | Infty
type destructor = Project of string | RemoveVariant of string
type argument =
    Variant of string*argument
  | Record of (string*argument) list
  | Epsilon of (destructor list)*int
  | Delta of z_infty*(((destructor list)*int) option)
```

- "Variant" takes two arguments: the name of the variant, and the recursive argument.
- "Record" is for tuples, but since PML has a notion of records with labels, we use association lists rather than simple lists.
- "Epsilon" is used for "exact" sequences of destructors. They are followed by a list of destructors, and a parameter number. Note that the exact term $() \in \mathcal{B}$ is already represented by "Record []".
- The last constructor "Delta" takes a weight, and either a list of destructors and an argument number, or nothing when it represents a $\delta_w()$.

Suppréma. The code for the suppremum, as needed in the implementation, is:

```
let rec sup u1 u2 = match u1,u2 with
    Epsilon(ds1,i1), Epsilon(ds2,i2) when ds1=ds2 && i1=i2 ->
      Epsilon(ds1,i1)
  | Epsilon(ds1,i1), Epsilon(ds2,i2) ->
      sup (Delta(Number 0, Some(ds1,i1)))
          (Delta(Number 0, Some(ds2,i2)))
  | Epsilon(ds,i), Delta(w,x)
  | Delta(w,x), Epsilon(ds,i) ->
      sup (Delta(Number 0,Some(ds,i))) (Delta(w,x))
```

```
  | Delta(w1,None), Delta(w2,None) -> Delta(max_infty w1 w2,None)
  | Delta(w1,Some(ds1,i)), Delta(w2,Some(ds2,i2)) when i=i2 ->
      let s,l1,l2 = suffix ds1 ds2 in
      let w1' = add_int w1 (-(weight l1)) in
      let w2' = add_int w2 (-(weight l2)) in
      Delta(max_infty w1' w2', Some(s,i))
  | Delta _, Delta _ -> Delta(Infty,None)
  | Delta(w,x), u | u, Delta(w,x) ->
      sup (Delta(w,x)) (reduce_Delta (Number 0) u)
  | Variant(c1,u1), Variant(c2,u2) when c1=c2 -> Variant(c1,sup u1 u2)
  | Variant _, Variant _ -> Delta(Infty,None)
  | Record l1, Record l2 ->
      let lab1,uu1 = List.split l1 in
      let lab2,uu2 = List.split l2 in
      sorted l1; sorted l2;
      if (lab1=lab2)
      then
        Record (List.combine lab1 (List.map2 sup uu1 uu2))
      else sup (reduce_Delta (Number 0) u1) (reduce_Delta (Number 0) u2)
  | _,_ -> sup (reduce_Delta (Number 0) u1) (reduce_Delta (Number 0) u2)
```

This function is mutually recursive with the function `reduce_Delta` computing $\delta_w \cdot u$. We thus avoid defining two functions for $\sup_0$ and sup.

Reduction, composition of substitutions. We only need a restricted notion of composition. The heart of it is the function `reduce_destructors` which computes the normal form of $\vec{a}\, u$ when u is in normal form:

```
(* reduce a branch of destructors (ds, which has already been
 * reversed) against an argument term.  *)
let rec reduce_destructors ds v = match ds,v with
    [],v -> v
  | ds, Delta(w,r) -> Delta(add_int w (-(weight ds)),r)
  | ds, Epsilon(ds',i) -> Epsilon(List.rev_append ds ds', i)
  | RemoveVariant c::ds, Variant (c',v) when c=c' ->
      reduce_destructors ds v
  | RemoveVariant _::_, Variant _ -> raise Impossible_case
  | Project c::ds, Record l ->
    begin
    try
      let v = List.assoc c l in
      reduce_destructors ds v
    with
      Not_found -> assert false (* typing problem *)
    end
  | _,_ -> assert false (* typing problem *)
```

The actual implementation has one more case, specific to the way PML deals with records.

Substituting and putting in normal form is rather direct from there.

Collapsing. The three collapsing functions $\mathrm{p}_{1,b}$, $\mathrm{p}_{2,d}$ and $\mathrm{p}_{3,d}$ are defined as:

```
let rec collapse1 b u = match u with
    Variant(c,u) -> Variant(c,collapse1 b u)
  | Record l ->
      let labels, args = List.split l in
      Record(List.combine labels (List.map (collapse1 b) args))
  | Epsilon(ds,i) -> Epsilon(ds,i)
  | Delta(w,x) -> Delta(collapse_z_infty b w, x)
```

```
let rec collapse2 d u = match u with
    Variant(c,u) -> Variant(c,collapse2 d u)
  | Record l ->
      let labels, args = List.split l in
      Record(List.combine labels (List.map (collapse2 d) args))
  | Epsilon(ds,i) ->
    begin
      let ds',r = get_suffix ds d in
      (* ds' is the suffix of "ds" of lenght d, and
         r is the rest of "ds" *)
      match r with
          [] -> Epsilon(ds,i)
        | _ -> Delta(Number(-(weight r)),Some(ds',i))
    end
  | Delta(w,None) -> Delta(w,None)
  | Delta(w,Some(ds,i)) ->
      let ds',r = get_suffix ds d in
      Delta(add_infty w (Number(-(weight r))),Some(ds',i))

let rec collapse3 d u =
  if d=0
  then
    match u with
        Epsilon _ -> u
      | _ -> reduce_Delta (Number 0) u
  else
    match u with
        Record l ->
          let labels,args = List.split l in
            Record (List.combine labels
                                 (List.map (collapse3 (d-1)) args))
      | Variant(c,u) -> Variant(c,collapse3 (d-1) u)
```

Weights of diagonal nodes. The diagonal nodes are computed with the following function. This function relies on several properties of diagonal nodes which aren't stated in the paper but should become clear when looking at the code.

```
let diagonal_nodes tau =
  let rec nodes_aux i u ds = match (reduce_Delta (Number 0) u) with
    | Delta(_,Some(ds',j)) when i<>j or incompatible ds ds' -> []
        (* the above case is a minor optimization *)
    | Delta(w,r) ->
      begin
        match u with
            Variant(c,u) -> nodes_aux i u ((RemoveVariant c)::ds)
          | Record(l) ->
            let n = match r with
                Some(ds',j) when i=j && ds=ds' -> [(i,ds,w)]
              | _ -> []
            in
            n @ (List.concat
                   (List.rev_map
                      (function label,arg ->
                          nodes_aux i arg ((Project label)::ds))
                      l))
          | Epsilon(ds',j) when ds=ds' && i=j-> [(i, ds', Number(0))]
            (* the above case could be ignored: they have positive weight *)
          | Delta(w',Some(ds',j)) ->
```

```
          begin
            let s,r,r' = suffix ds ds' in
            assert (i=j);
            match r with
                [] -> [ (i,
                         List.rev_append r' ds,
                         add_int w' (-(weight r'))) ]
              | _ -> assert (r' = []); []
          end
        | _ -> []
    end
  | _ -> assert false
in
List.concat (List.rev_map (function i,u -> nodes_aux i u []) tau)
```

Laboratoire de Mathématiques, CNRS UMR 5127 – 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/