



The Size-Change Principle for Mixed Inductive and Coinductive types

Pierre Hyvernât

► To cite this version:

Pierre Hyvernât. The Size-Change Principle for Mixed Inductive and Coinductive types. 2019. hal-01989688v1

HAL Id: hal-01989688

<https://hal.science/hal-01989688v1>

Preprint submitted on 22 Jan 2019 (v1), last revised 25 Jun 2024 (v4)

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 PRINCIPLE FOR MIXED INDUCTIVE AND COINDUCTIVE TYPES

PIERRE HYVERNAT

Université Grenoble Alpes, Université Savoie Mont Blanc, CNRS, LAMA, 73000 Chambéry, France.

e-mail address: pierre.hyvernats@univ-smb.fr

URL: <http://lama.univ-savoie.fr/~hyvernats/>

ABSTRACT. This paper describes how to use Lee, Jones and Ben Amram’s size-change principle to check correctness of arbitrary recursive definitions in an ML / Haskell like programming language. The main point is that the size-change principle isn’t only used to check termination, but also productivity for infinite objects. The main point is that the resulting principle is sound even in the presence of arbitrary nestings of inductive and coinductive types. A small prototype has been implemented and gives a practical argument in favor of this principle.

This work relies on a characterization of least and greatest fixed points as sets of winning strategies for parity games that was developed by L. Santocanale in his work on circular proofs.

Half of the paper is devoted to the proof of correctness of the criterion, which relies on an untyped extension of the language’s denotational semantics to a domain of values extended with non-deterministic sums. We can recast all the syntactical constructions in this domain and check they are semantically sound.

CONTENTS

Introduction	2
Related Works	3
Plan of the Paper	6
1. The Language and its Semantics	6
1.1. Values	6
1.2. Type Definitions	7
1.3. Semantics in Domains	8
1.4. Semantics in Domains with Totality	9
1.5. Recursive Definitions	10
2. Combinatorial Description of Totality	12
2.1. Parity Games	13
2.2. Parity Games from Types	14
2.3. Forgetting Types	17
3. Call-Graph and Totality	17
3.1. Introduction	17

Received by the editors January 22, 2019.

Key words and phrases: coinductive types; nested fixed points; size-change principle; functional programming; recursive definitions.

3.2. Interpreting Calls	18
3.3. Call-Graph and Composition	21
3.4. Collapsing	23
3.5. Size-Change Principle	25
3.6. The Algorithm	26
4. Semantics and Correctness of the Call-Graph	26
4.1. Smyth power domain	26
4.2. Preliminaries: Recursion and Fixed Points	28
4.3. A Language for Operators	29
4.4. Interpreting recursive definitions	34
4.5. Removing finite path	35
4.6. The Call-Graph	37
4.7. Weights	39
4.8. The Size-Change Principle	41
4.9. Back to the Previous Section	44
Concluding Remarks	44
References	47

INTRODUCTION

Inductive types (also called algebraic datatypes) have been a cornerstone for typed functional programming: Haskell and Caml both rely heavily on those. One mismatch between the two languages is that Haskell is *lazy* while Caml is *strict*. A definition¹ like

```
val nats : nat -> nat
| nats n = n::(nats (n+1))
```

is useless (but valid) in Caml because the evaluation mechanism will try to evaluate it completely (call-by-value evaluation). In Haskell, because evaluation is lazy (call-by-need), such a definition can be used productively. Naively, it seems that types in Caml correspond to “least fixed points” while they correspond to “greatest fixed points” in Haskell.

The aim of this paper is to introduce a language, called **chariot**,² where the distinction between least and greatest fixed points makes sense and where one can define datatypes with an arbitrary nesting of polarities. To allow a familiar programming experience, definitions are not restricted: any (well-typed) recursive definition is allowed. In particular, it is possible to write badly behaved definitions like

```
val f : nat -> nat
| f 0 = 1
| f (n+1) = f(f n)
```

To guarantee that a definition is correct, two-steps are necessary:

- (1) Hindley-Milner type-checking [Mil78] to guarantee that evaluation doesn’t provoke run-time errors,
- (2) a *totality test* to check that the defined function respects the fixed points polarities involved in its type.

¹The examples in the paper are all given using the syntax of **chariot** which is briefly described in sections 1.2 and 1.5.

²A prototype implementation in Caml is available from <https://github.com/phyver/chariot>.

The second point generalizes a previous termination checker [Hyv14]: when no coinductive type is involved, totality amount to termination. It is important to keep in mind that any definition that passes this test is guaranteed to be correct but there are correct definitions that are rejected.³ In a programming context, the result of this second step can be ignored when the programmer (thinks he) knows better. In a proof-assistant context however, it cannot be ignored: non total definitions can lead to inconsistencies. The most obvious example is the definition

```
val undefined = undefined
```

which is non-terminating but belongs to all types, even the empty one! There are subtler examples of definitions that normalize to values but still lead to inconsistencies [AD12] (c.f. example on page 12).

In Coq [The04], the productivity condition for coinductive definitions is ensured by a very strict syntactic condition (guardedness [Coq93]) similar to the condition that inductive definitions need to have one structurally decreasing argument. In Agda [Nor08], the user can write arbitrary recursive definitions and the productivity condition is ensured by the termination checker. The implemented checker extends a published version [AA02] to deal with coinductive types, but while this is sound for simple types like streams, it is known to be unsound for nested coinductive and inductive types [AD12]. This paper provides a first step toward a solution for this problem.

Related Works.

Circular proofs. The primary inspiration for this work comes from the ideas developed by L. Santocanale in his work on circular proofs [San02c, San02a, San02b]. Circular proofs are defined for a linear proof system and are interpreted in categories with products, coproducts and enough initial algebras / terminal coalgebras. In order to get a functional language, we need to add rules and interpret them in *cartesian closed categories* with coproducts and enough initial algebras / terminal coalgebras (like the category of sets and functions, or the category of domains).

What is described in this paper seems to amount to using a strong combinatorial principle (the size-change principle) to check a sanity condition on a circular “preproof”. This condition implies that the corresponding cut free preproof (an infinite object) can be interpreted in a sufficiently well behaved category. This condition is strictly stronger than the one L. Santocanale and G. Fortier used in their work, which corresponded to the syntactical structurally decreasing / guardedness condition on recursive definitions.

Note however that while circular proofs were a primary inspiration, this work cannot be reduced to a circular proof system. The main problem is that all such proof systems are linear and do not enjoy a simple cut-elimination procedure. Cuts and exponentials are needed to interpret the full **chariot** language and while cuts can added to the original system of circular proofs [FS14, For14], adding exponentials looks extremely difficult and hasn’t been done.

Note also that more recent works in circular proof theory replace L. Santocanale’s criterion by a much stronger combinatorial condition. Without going into the details, it is

³The halting problem is, after all, undecidable [Tur36] ☹.

equivalent to some infinite word being recognized by a parity automata (which is decidable) [Dou17b, Dou17a]. The presence of parity automata points to a relation between this work and the present paper, but the different contexts make it all but obvious.

Size-change principle. The main tool used for checking totality is the *size-change principle* (SCP) from C. S. Lee, N. D. Jones and A. M. Ben-Amram [LJBA01]. The problem of totality is however subtler than termination of programs. While the principle used to check termination of ML-like recursive definitions [Hyv14] was inherently untyped, totality checking needs to be somewhat type aware. For example, in **chariot**, records are lazy and are used to define coinductive types. The following definition

```
val inf = Node { Left = inf; Right = inf }
```

yields an infinite, lazy binary tree. Depending on the types of **Node**, **Fst** and **Snd**, the definition may be correct or incorrect (refer to page 12 for more details)!

Charity. The closest ancestor to **chariot** is the language **charity**⁴ [CF92, Coc96], developed by R. Cockett and T. Fukushima, allows the user to define types with arbitrary nesting of induction and coinduction. Values in these types are defined using categorical principles.

- Inductive types are *initial* algebras: defining a function *from* an inductive type amounts to defining an algebra for the corresponding operator.
- Coinductive types are *terminal* coalgebras: defining a function *to* an inductive type amount to defining a coalgebra for the corresponding operator.

Concretely, it means the user can only define recursive functions that are “trivially” structurally decreasing on one argument, or “trivially” guarded. In particular, *all* functions terminate and the language is not Turing complete.

This is very different from the way one can write, for example, the Ackermann function with pattern matching:

```
val ack 0 n = n+1
  | ack (m+1) 0 = ack m 1
  | ack (m+1) (n+1) = ack m (ack (m+1) n)
```

Guarded recursion. Another approach to checking correctness of recursive definitions is based on “guarded recursion”, initiated by H. Nakano [Nak00] and later extended in several directions [CBGB16, Gua18]. In this approach, a new modality “later” (usually written “▷”) is introduced in the type theory. The new type $\triangleright T$ gives a syntactical way to talk about terms that “will later (after some computation) have type T ”. This work is rather successful and has been extended to very expressive type systems. The drawbacks are that this requires a non-standard type theory with a not quite standard denotational semantics (topos of trees). Moreover, it makes programming more difficult as it introduces new constructs in types and terms. Finally, these works only consider greatest fixed points (as in Haskell) and are thus of limited interest for systems such as Agda or Coq.

⁴By the way, the name **chariot** was chosen as a reminder of this genealogy.

Sized-types. This approach also extends type theory with a notion of “size” for types. It has been successful and is implemented in Agda [Abe10, Abe12]. This makes it possible, for example, to specify that the `map` function on list has type $\forall n, \mathbf{list}^n(T) \rightarrow \mathbf{list}^n(T)$, where $\mathbf{list}^n(T)$ is the type of lists with n elements of type T . These extra parameters allow to gather information about recursive functions and make it easier to check termination. A drawback is that functions on sized-types must take extra size parameters. This complexity is balanced by the fact that most of them can be inferred automatically and are thus mostly invisible to the casual user.⁵ Note however that this approach still needs to have a way to check that definition respect the sizes.

Fixed points in game semantics. An important tool for checking totality of definitions in this paper is the notion of *parity game*. P. Clairambault [Cla13] explored a notion of game (from a categorical, games semantics point of view) enriched with winning conditions for infinite plays. The way the winning condition is defined for least and greatest fixed points is reminiscent of L. Santocanale’s work on circular proofs and the corresponding category is cartesian closed.

Because this work is done in a more complex setting (categories of games) and aims for generality, it seems difficult to extract a practical test for totality from it. The present paper aims for specificity and practicality by devising a totality test for the “intended” semantics (i.e. in plain sets and functions) of recursion.

SubML. C. Raffalli and R. Lepigre also used the size-change principle in order to check correctness of recursive definitions in the language SubML [LR18]. Their approach uses a powerful but non-standard type theory with many features: subtyping, polymorphism, sized-types, control operators, some kind of dependent types, etc. On the downside, it makes their type theory more difficult to compare with other approaches. Note that like in Agda or `chariot`, they do allow arbitrary definitions that are checked by an incomplete totality checker. The similarity of the approach isn’t surprising considering previous collaborations between the authors. One interesting point of their work is that the size-change termination is only used to check that some object (a proof tree) is well-founded: even coinductive types are justified with well-founded proofs.

Nax. Another programming language with nested inductive / coinductive types is the Nax language [Ahn14], based on so called “Mendler style recursion” [Men91]. One key difference is that the Nax language is very permissive on the definitions of types (it is for example possible to define fixed points for non positive type operators) and rather restrictive on the definition of values: they are defined using various combinators similar (but stronger than) to the way values are defined in `charity`. From the point of view of a Haskell / Caml programmer, the restriction on the way programs are written is difficult to accept.⁶

⁵The libraries’ implementors still needs to give the appropriate type to `map` though.

⁶No implementation of Nax language is available, it is thus difficult to experiment with it.

Plan of the Paper. We start by introducing the language **chariot** and its denotational semantics in section 1, together with the notion of totality for functions. Briefly, totality generalizes termination in a way that accounts for inductive and coinductive types. An interesting point is that this notion is semantical rather than syntactical. We then describe, in section 2, a combinatorial approach to totality that comes from L. Santocanale’s work on circular proofs. This reduces checking totality of a definition to checking that the definitions gives a winning strategy in a parity game associated to the type of the definition. Section 3 describes how the size-change principle can be applied to this problem: a recursive definition gives a call-graph, and the size-change principle can be used to check a totality condition on all infinite path in this call-graph. This section is written from the implementor’s point of view, and most proofs are omitted: they are given in the following section. The last section is the longest and gives the proof of correctness. This works by showing that the call-graph and the operations defined on it have a sound semantics in domains.

1. THE LANGUAGE AND ITS SEMANTICS

1.1. Values. We are interested in a condition on the semantics of recursive definitions. What is interesting is that this doesn’t mention the reduction strategy: everything takes place in the realm of values.

Definition 1.1. The set of *values with leaves in* $X_1, \dots, X_n$, written $\mathcal{V}(X_1, \dots, X_n)$ is defined coinductively⁷ by the grammar

$$v ::= \perp \mid x \mid \mathbf{C} v \mid \{\mathbf{D}_1 = v_1; \dots; \mathbf{D}_k = v_k\}$$

where

- each x is in one of the X_i ,
- each $\mathbf{C}$ belongs to a finite set of *constructors*,
- each $\mathbf{D}_i$ belongs to a finite set of *destructors*,
- the order of fields inside records is irrelevant,
- k can be 0.

Locally, only a finite number of constructors / destructors will be necessary: those appearing in the type definitions involved in the definitions we are checking. There is a natural ordering on finite values, which can be extended to infinite ones (c.f. remark about ideal completion on page 26).

Definition 1.2. If the X_i are ordered sets, the order $\leq$ on $\mathcal{V}(X_1, \dots, X_n)$ is generated by

- (1) $\perp \leq u$ for all values u ,
- (2) if $x \leq x'$ in X_i , then $x \leq x'$ in $\mathcal{V}(X_1, \dots, X_n)$,
- (3) if $u \leq v$ then $C[u] \leq C[v]$ for any context $C[\]$.

⁷It is natural to give an infinite semantics for coinductive types, and infinite values are thus allowed.

1.2. Type Definitions. The approach described in this paper is entirely first-order. We are only interested in the way values in datatypes are constructed and destructed. Higher order parameters are allowed in the implementation but they are ignored by the totality checker. The examples in the paper will use such higher order parameters but for simplicity's sake, they are not formalized. Note that it is not possible to just ignore higher order parameters as they can hide some recursive calls:

```
val app f x = f x      -- non recursive
val g x = app g x
```

In order to deal with that, the implementation first checks that all recursive functions are fully applied. If that is not the case, the checker aborts and gives a negative answer.

Just like in **charity**, types in **chariot** come in two flavors: those corresponding to sum types (i.e. colimits) and those corresponding to product types (i.e. limits). The syntax is itself similar to that of **charity**:

- a data comes with a list of *constructors* whose codomain is the type being defined,
- a codata comes with a list of *destructors* whose domain is the type being defined.

The syntax is

<pre>data new_type where C₁ : T₁ -> new_type ... C_k : T_k -> new_type</pre>	<pre>codata new_type where D₁ : new_type -> T₁ ... D_k : new_type -> T_k</pre>
--	--

Each T_i is built from earlier types, parameters and *new_type*. Types parameters are written with a quote as in Caml but the parameters of *new_type* cannot change in the definition. Mutually recursive types are possible, but they need to be of the same polarity (all **data** or all **codata**). We can always suppose that all the mutually defined types have the same parameters as otherwise, the definition could be split in several, non mutual definitions. Here are some examples:

```
codata unit where      -- no destructor

codata prod('x,'y) where  Fst : prod('x,'y) -> 'x
                        | Snd : prod('x,'y) -> 'y

data nat where  Zero : unit -> nat
               | Succ : nat -> nat

data list('x) where  Nil : unit -> list('x)
                   | Cons : prod('x, list('x)) -> list('x)

codata stream('x) where  Head : stream('x) -> 'x
                       | Tail : stream('x) -> stream('x)
```

The examples given in the paper (and the implementation) do not adhere strictly to this syntax: n -ary constructors are allowed, and **Zero** will have type **nat** (instead of **unit** -> **nat**) while **Cons** will be uncurried and have type **'x** -> **list('x)** -> **list('x)** (instead of **prod('x, list('x))** -> **list('x)**).

Because destructors act as projections, it is useful to think about elements of a co-datatype as records. This is reflected in the syntax of terms, and the following defines the stream with infinitely many 0s.

```

val zeros : stream(nat)
  | zeros = { Head = Zero ; Tail = zeros }

```

As the examples show, codata are going to be interpreted as *coinductive* types, while data are going to be *inductive*. The denotational semantics will reflect that, and in order to have an operational semantics that is sound, codata need to be *lazy*. The simplest is to stop evaluation on records: evaluating “**zeros**” will give “{Head = ???; Tail = ???}” where the “???” are not evaluated. Surprisingly, the details are irrelevant to rest of paper.

We will use the following conventions:

- outside of actual type definitions (given using **chariot**’s syntax), type parameters will be written without quote: $\mathbf{x}, \mathbf{x}_1, \dots$
- an unknown datatype will be called $\theta_\mu(\mathbf{x}_1, \dots, \mathbf{x}_n)$ and an unknown codatatype will be called $\theta_\nu(\mathbf{x}_1, \dots, \mathbf{x}_n)$,
- an unknown type of unspecified polarity will be called $\theta(\mathbf{x}_1, \dots, \mathbf{x}_n)$.

1.3. Semantics in Domains. Our notion of domain is the historical one: a domain is a

- consistently complete (finite bounded sets have a least upper bound)
- algebraic (with a basis of compact elements)
- directed-complete partial order (DCPO: every directed set has a least upper bound).

While helpful in section 4, intimate knowledge of domain theory is not necessary to follow the description of the totality checker.

There is a natural interpretation of types in the category **Dom** of domains, where morphisms are continuous functions. (Note that morphisms are *not* required to preserve the least element.) The category theory aspect is not important because all the types are in fact subdomains of $\mathcal{V}$. The following can be proved directly but is also a direct consequence of a general fact about orders and their “ideal completion”.

Lemma 1.3. *If the X_i s are domains, then $(\mathcal{V}(X_1, \dots, X_n), \leq)$ is a domain.*

Type expressions with parameters are generated by the grammar

$$T ::= X \mid \mathbf{x} \mid \theta_\mu(T_1, \dots, T_n) \mid \theta_\nu(T_1, \dots, T_n)$$

where X is any domain (or set, depending on the context) called a parameter, and θ_μ is the name of a datatype of arity n and θ_ν is the name of a codatatype of arity n . A type is *closed* if it doesn’t contain variables. (It may contains parameters, that is, subdomains of the domain of values.)

Definition 1.4. The interpretation of a closed type $T(\overline{X})$ with domain parameters is defined *coinductively* as the set of (possibly infinite) values well typed according to:

- (1) $\frac{}{\perp : T}$ for any type T ,
- (2) $\frac{u \in X}{u : X}$ for any parameter X ,
- (3) $\frac{u : T[\sigma]}{\mathbf{C} u : \theta_\mu(\sigma)}$ where $\mathbf{C} : T \rightarrow \theta_\mu(\sigma)$ is a constructor of θ_μ ,

destructors for type θ_ν .

If T is a type with free variables $\mathbf{x}_1, \dots, \mathbf{x}_n$, we write $\llbracket T \rrbracket (\bar{X})$ for the interpretation of $T[\sigma]$ where σ is the substitution $[\mathbf{x}_1 := X_1, \dots, \mathbf{x}_n := X_n]$.

Proposition 1.5. *Let $X_1, \dots, X_n$ be domains, if T is a type then*

- $$\begin{aligned} \llbracket T \rrbracket (\overline{X}) &= \left\{ c_i u_i \mid i = 1, \dots, n \text{ and } u_i \in \llbracket T_i \rrbracket \right\} \cup \{\perp\} \\ &\cong \left(\llbracket T_1 \rrbracket (\overline{X}) + \dots + \llbracket T_k \rrbracket (\overline{X}) \right) \end{aligned}$$

- $$\begin{aligned} \llbracket T \rrbracket(\overline{X}) &= \left\{ \{ \dots; \mathbf{D}_i = u_i; \dots \} \mid i = 1, \dots, n \text{ and } u_i \in \llbracket T_i \rrbracket \right\} \cup \{\perp\} \\ &\cong \left(\llbracket T_1 \rrbracket(\overline{X}) \times \dots \times \llbracket T_k \rrbracket(\overline{X}) \right)_\perp \end{aligned}$$

```

graph BT
    bot["⊥"] --> Zero
    bot --> SuccBot["Succ ⊥"]
    SuccBot --> SuccZero["Succ Zero"]
    SuccBot --> SuccSuccBot["Succ(Succ ⊥)"]
    SuccZero --> SuccSuccZero["Succ(Succ Zero)"]

```

- $\perp$,
- $\{\text{Head} = \text{Succ } \perp; \text{Tail} = \perp\}$
- $\{\text{Head} = \text{Zero}; \text{Tail} = \{\text{Head} = \text{Zero}; \text{Tail} = \{\text{Head} = \text{Zero}; \dots\}\}\}$

1.4. Semantics in Domains with Totality. At this stage, there is no distinction between greatest and least fixed point: the functors defined by types are *algebraically compact* [Bar92], i.e. their initial algebras and terminal coalgebras are isomorphic. For example, $\text{Succ}(\text{Succ}(\text{Succ}(\dots)))$ is an element of $\llbracket \text{nat} \rrbracket$ as the limit of the chain $\perp \leq \text{Succ } \perp \leq$

$\text{Succ}(\text{Succ } \perp) \leq \dots$. In order to distinguish between inductive and coinductive types, we add a notion of *totality*⁸ to the domains.

Definition 1.6.

- (1) A *domain with totality* $(D, |D|)$ is a domain D together with a subset $|D| \subseteq D$.
- (2) An element of D is *total* when it belongs to $|D|$.
- (3) A function f from $(D, |D|)$ to $(E, |E|)$ is a function from D to E . It is *total* if $f(|D|) \subseteq |E|$, i.e. if it sends total elements to total elements.
- (4) The category **Tot** has domains with totality as objects and total continuous functions as morphisms.

To interpret (co)datatypes inside the category **Tot**, it is enough to describe the associated totality predicate. The following definition corresponds to the “natural” interpretation of inductive / coinductive types in the category of sets.

Definition 1.7. If T is a type whose parameters are domains with totality, we define $|T|$ by induction

- if $T = X$ then $|T| = |X|$
- if $T = \theta_\mu(T_1, \dots, T_n)$ is a datatype, then $|T| = \mu X. \theta_\mu(X, |T_1|, \dots, |T_n|)$,
- if $T = \theta_\nu(T_1, \dots, T_n)$ is a codatatype, then $|T| = \nu X. \theta_\nu(X, |T_1|, \dots, |T_n|)$,

where

- (1) if $T = \theta_\mu(\mathbf{x}_1, \dots, \mathbf{x}_n)$ is a datatype with constructors $\mathbf{C}_i : T_i \rightarrow T$, θ_μ is the operator

$$X, X_1, \dots, X_n \mapsto \bigcup_{i=1, \dots, k} \left\{ \mathbf{C}_i u \mid u \in |T_i[\sigma]| \right\}$$

- (2) if $T = \theta_\nu(\mathbf{x}_1, \dots, \mathbf{x}_n)$ is a codatatype with destructors $\mathbf{D}_i : T \rightarrow T_i$, θ_ν is the operator

$$X, X_1, \dots, X_n \mapsto \left\{ \{ \mathbf{D}_1 = u_1; \dots; \mathbf{D}_k = u_k \} \mid \text{each } u_i \in |T_i[\sigma]| \right\}$$

In both cases, σ is the substitution $[T := X, \mathbf{x}_1 := X_1, \dots, \mathbf{x}_n := X_n]$.

The least and greatest fixed points exist by Knaster-Tarski theorem: the corresponding operators act on subsets of the set of all values. It is not difficult to see that each element of $|T|$ is in $\llbracket T \rrbracket$ and doesn’t contain $\perp$, i.e. is a maximal element of the domain $\llbracket T \rrbracket$:

Lemma 1.8. *If T is a type with domain parameters, $(\llbracket T \rrbracket, |T|)$ is a domain with totality. Moreover, each $t \in |T|$ is maximal in $\llbracket T \rrbracket$.*

1.5. Recursive Definitions. Like in Haskell, recursive definitions are given by lists of clauses. The Ackermann function was given on page 4 and here is the **map** function on streams:⁹

```
val map : ('a -> 'b) -> stream('a) -> stream('b)
  | map f { Head = x ; Tail = s } = { Head = f x ; Tail = map f s }
```

Formally, a (recursive) definition is introduced by the keyword **val** and consists of several clauses of the form

```
f p1 ... pn = u
```

⁸Intrinsic notions of totality exist [Ber93] but are seemingly unrelated to what is considered below.

⁹This definition isn’t strictly speaking first order as it take a function as argument. We will ignore such arguments and they can be seen as free parameters.

where

- $\mathbf{f}$ is a function name,
- each p_i is a *finite* pattern

$$p ::= \mathbf{x}_i \mid \mathbf{C} p \mid \{\mathbf{D}_1 = p_1; \dots; \mathbf{D}_k = p_k\}$$

where each $\mathbf{x}_i$ is a variable name,

- and u is a *finite* term

$$u ::= \mathbf{x}_i \mid \mathbf{f} \mid \mathbf{C} u \mid \{\mathbf{D}_1 = u_1; \dots; \mathbf{D}_k = u_k\} \mid u_1 u_2$$

where each $\mathbf{x}_i$ is a variable name and each $\mathbf{f}$ is a function name (possibly one of the functions being defined).

Note that it is not possible to directly project a record on one of its field in the syntax of terms. This makes the theory somewhat simpler and doesn't change expressivity of the language. It is always possible to

- remove a projection on a variable by extending the pattern on the left,
- replace a projection on the result of a recursively defined function by several mutually recursive functions for each of the fields,
- replace a projection on a previously defined function by another previously defined function.

Of course, the implementation doesn't enforce this restriction and the theory can be extended accordingly.

There can be many clauses and many different functions defined mutually. The system

- (1) checks some syntactical constraints (linearity of pattern variables, ...),
- (2) performs Hindley-Milner type checking (or type inference if no type annotation was given),
- (3) performs an exhaustivity check ensuring that the patterns cover all the possibilities and that records have all their fields.

Those steps are well-known [PJ87] and not described here. Hindley-Milner type checking guarantees that each list of clauses for functions $\mathbf{f}_1 : T_1, \dots, \mathbf{f}_n : T_n$ (each T_i is an arrow type) gives rise to an operator

$$\theta_{\mathbf{f}_1, \dots, \mathbf{f}_n} : \llbracket T_1 \rrbracket \times \dots \times \llbracket T_n \rrbracket \rightarrow \llbracket T_1 \rrbracket \times \dots \times \llbracket T_n \rrbracket$$

where the semantics of types is extended with $\llbracket T \rightarrow T' \rrbracket = \llbracket \llbracket T \rrbracket \rightarrow \llbracket T' \rrbracket \rrbracket$. The semantics of $\mathbf{f}_1, \dots, \mathbf{f}_n$ is then defined as the fixed point of the operator $\theta_{\mathbf{f}_1, \dots, \mathbf{f}_n}$ which exists by Kleene theorem.

Typing ensures that the definition is well behaved from an operational point of view: the “ $\perp$ ” that appear in the result correspond only to non-termination, not to failure of the evaluation mechanism (projecting on a non-existing field or similar problems). For the definition to be correct from a denotational point of view, we need to check more: that it is *total* with respect to its type. For example, the definition

```
val all_nats : nat -> list(nat)
| all_nats n = Cons n (all_nats (Succ n))
```

is well typed and sends elements of the domain $\llbracket \mathbf{nat} \rrbracket$ to the domain $\llbracket \mathbf{list}(\mathbf{nat}) \rrbracket$ but its result on **Zero** contains all the natural numbers. This definition is not total because totality for $\mathbf{list}(\mathbf{nat})$ contains only the finite lists (it is an inductive type). Similarly, the definition

```

val last_stream : stream(nat) -> nat
  | last_stream {Head=_, Tail=s} = last_stream s

```

sends any stream to $\perp$, which is non total.

A subtle example. Here is a surprising example due to T. Altenkirch and N. A. Danielson [AD12]: we define the inductive type

```

data stree where Node : stream(stree) -> stree

```

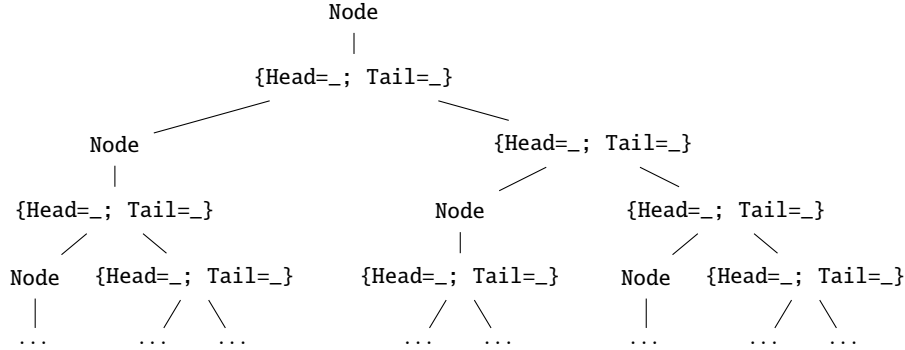
where the type of `stream` was defined on page 7. This type is similar to the usual type of “Rose trees”, but with streams instead of lists. Because streams cannot be empty, there is no way to build such a tree inductively: this type has no total value. Consider however the following definitions:

```

val s : stream(stree)
  | s = { Head = Node s ; Tail = s }
val t : stree
  | t = Node s

```

This is well typed, but because evaluation is lazy, evaluation of `t` or any of its subterms terminates: the semantics of `t` doesn’t contain $\perp$. Unfolding the definition, we obtain



Such a term leads to inconsistencies and shows that a simple termination checker isn’t enough.

The rest of the paper describes a partial totality test on recursive definitions: some definitions are tagged “total” while some other are tagged “unsafe” either because they are indeed not total, or because the argument for totality is too complex.

2. COMBINATORIAL DESCRIPTION OF TOTALITY

The set of total values for a given type can be rather complex when datatypes and co-datatypes are interleaved. Consider the definition

```

val inf = Node { Left = inf; Right = inf }

```

It *is not* total with respect to the type definitions

```

codata pair('x,'y) where Left : pair('x,'y) -> 'x
                        | Snd : pair('x,'y) -> 'y
data tree where Node : pair(tree, tree) -> tree

```

but it *is* total with respect to the type definitions

```

data box('x) where Node : 'x -> box('x)
codata tree2 where Left : tree2 -> box(tree2)
                  | Right : tree2 -> box(tree2)

```

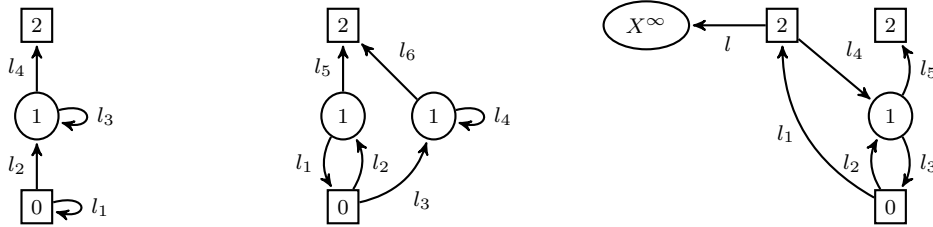
In this case, the value `inf` is of type `box(tree2)`. Analysing totality requires a combinatorial understanding of the least and greatest fixed points involved. Fortunately, there is a close relationship between set theoretic least and greatest fixed points and winning strategies for *parity games*.

2.1. Parity Games. Parity games are a two players games played on a finite transition system where each node is labeled by a *priority* (a natural number). The *height* of such a game is the maximum priority of its nodes. By extension, the priority of a transition is the priority of its target node. When the node has odd priority, *Marie* (or “player”) is required to play. When the node is even, *Nicole* (or “opponent”) is required to play. A move is simply a choice of a transition from the current node and the game continues from the new node. When Nicole (or Marie) cannot move because there is no outgoing transition from the current node, she loses. In case of infinite play, the winning condition is

- (1) if the maximal node visited infinitely often is even, Marie wins,
- (2) if the maximal node visited infinitely often is odd, Nicole wins.

Equivalently, the condition could be stated using the priorities of the transitions taken during the infinite play. We will call a priority *principal* if “it is maximal among the priorities appearing infinitely often”. The winning condition can thus be rephrased as “Marie wins an infinite play if and only if the principal priority of the play is even”.

In order to analyse types with parameters, we add special nodes called *parameters*. Those nodes have no outgoing transition, have priority ∞ ¹⁰ and each of them has an associated set X . On reaching them, Marie is required to choose¹¹ an element of X to finish the game. She wins if she can do it and loses if she cannot (when the set is empty). Here are three examples of such parity games:



Each position p in a parity game G with parameters $X_1, \dots, X_n$ defines a set $\|G_p\|$ depending on $X_1, \dots, X_n$ [San02c]. This set valued function $p \mapsto \|G_p\|$ is defined by induction on the height of G and the number of positions having maximum priority:

- if all the positions are parameters, each position is interpreted by the corresponding parameter $\|G_X\| = X$;
- otherwise, take p to be one of the positions of maximal priority and construct G/p with parameters $Y, X_1, \dots, X_n$ as follows: it is identical to G , except that position p is replaced

¹⁰Those parameter nodes do not count when defining the depth of a parity game.

¹¹because ∞ is obviously odd ☺

by parameter Y and all its outgoing transitions are removed.¹² Compute recursively the interpretations $(G/p)_q$, depending on $Y, X_1, \dots, X_n$ and:

– if p had an odd priority, define

$$\begin{cases} \|G_p\| = \mu Y. ((G/p)_{q_1} + \dots + (G/p)_{q_k}) \\ \|G_q\| = (G/p)_q[Y := \|G_p\|] \end{cases} \quad \text{when } q \neq p$$

where $p \rightarrow q_1, \dots, p \rightarrow q_k$ are all the transitions out of p .

– if p had an even priority, define

$$\begin{cases} \|G_p\| = \nu Y. ((G/p)_{q_1} \times \dots \times (G/p)_{q_k}) \\ \|G_q\| = (G/p)_q[Y := \|G_p\|] \end{cases} \quad \text{when } q \neq p$$

where $p \rightarrow q_1, \dots, p \rightarrow q_k$ are all the transitions out of p .

An important result is:

Proposition 2.1 (L. Santocanale [San02c]).

- For each position p of G , the operation $X_1, \dots, X_n \mapsto \|G(X_1, \dots, X_n)_p\|$ is a functor from $\mathbf{Set}^n$ to $\mathbf{Set}$,
- there is a natural isomorphism $\|G_p\| \cong \mathcal{W}(G)_p$ where $\mathcal{W}(G)_p$ is the set of winning strategies for Marie in game G from position p .

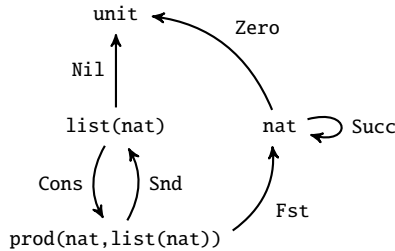
In order to analyse totality, we now construct a parity game G from a type expression T in such a way that $|T| \cong \|G_T\|$, for some distinguished position T in G .

2.2. Parity Games from Types.

Definition 2.2. If T is a type expression (possibly with parameters), the *graph of T* is defined as the subgraph reachable from T in the following (infinite) transition system:

- nodes are type expressions (possibly with parameters),
- transitions are labeled by constructors and destructors: a transition $T_1 \xrightarrow{t} T_2$ is either a destructor t of type $T_1 \rightarrow T_2$ or a constructor t of type $T_2 \rightarrow T_1$ (note the reversal).

Here is for example the graph of `list(nat)`



The orientation of transitions means that

- on data nodes, a transition is a choice of constructor for the origin type,
- on codata nodes, a transition is a choice of field for a record for the origin type.

Because of that, a value of type T can be seen as a strategy for a game on the graph of T where Marie (the player) chooses constructors and Nicole (the opponent) chooses destructors.

¹²This game is called the predecessor of G [San02c].

Lemma 2.3. *The graph of T is finite.*

Proof of Lemma 2.3. We write $T_1 \sqsubseteq T_2$ if T_1 appears in T_2 . More precisely:

- $T \sqsubseteq X$ iff $T = X$,
- $T \sqsubseteq \theta(T_1, \dots, T_n)$ if and only if $T = \theta(T_1, \dots, T_n)$ or $t \sqsubseteq T_1$ or $\dots$ or $t \sqsubseteq T_n$.

To each datatype / codatatype definition, we associate its “definition order”, an integer giving its index in the list of all the type definitions. A (co)datatype may only use parameters and “earlier” type names in its definition and two types of the same order are part of the same mutual definition. The order of a type is the order of its head type constructor.

Suppose that the graph of some type T is infinite, and that T is minimal in the sense that it is of order κ and graphs for types of order less than κ are finite. Since the graph of T has bounded out-degree, by König’s lemma, it contains an infinite simple (without repeated vertex) path $\rho = T \rightarrow T_1 \rightarrow T_2 \rightarrow \dots$. For any n , there is some $l > n$ such that T_l is of order κ . Otherwise, the path $T_{n+1} \rightarrow T_{n+2} \rightarrow \dots$ is infinite and contradicts the minimality of T .

All transitions in the graph of T are of the form $\theta(\overline{T}) \rightarrow \beta$ where β is built using the types in $\overline{T}$ and possibly the $\theta'(\overline{T})$ with the same order as θ . There are three cases:

- (1) In transitions $\theta(\overline{T}) \rightarrow T_i$, i.e., transitions to a parameter, the target is a subexpression of the origin. It is the only way the order a type may strictly increase along a transition. This is the case of **Head : stream(nat) → nat**.
- (2) In transitions $\theta(\overline{T}) \rightarrow \theta'(\overline{T})$, i.e. transitions to a type in the same mutual definition, the order remains constant. An example in **Succ : nat → nat**.
- (3) In all other cases, the transition is of the form $\theta(\overline{T}) \rightarrow \beta$, where β is strictly earlier than θ . In this case however, because of the way β is built, the subexpressions of β with order greater than that of θ are necessarily subexpressions of $\theta(\overline{T})$. This is for example the case of **Cons : list(nat) → prod(nat, list(nat))** (recall that the transition goes in the opposite direction).

The only types of order κ reachable from T (of order κ) are thus:

- subexpressions of T (obtained with transitions of the form (1) and (3)),
- or variants of the above where some types $\theta(\overline{T})$ of order κ have been replaced by $\theta'(\overline{T})$ of same order (obtained with transitions of the form (2)).

Since there are only finitely many of those, the infinite path ρ necessarily contains a cycle! This is a contradiction. $\square$

Definition 2.4. If T is a type expression (possibly with parameters), a *parity game for T* is a parity game G_T on the graph of T satisfying

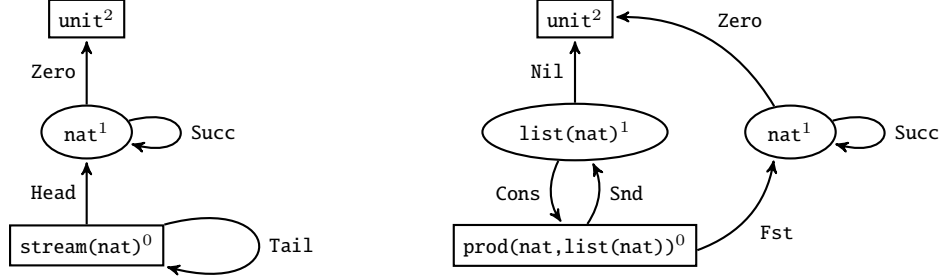
- (1) each parameter of T is a parameter of G_T ,
- (2) if T_0 is a datatype in the graph of T , its priority is odd,
- (3) if T_0 is a codatatype in the graph of T , its priority is even,
- (4) if $T_1 \sqsubseteq T_2$ then the priority of T_1 is greater than the priority of T_2 .

Lemma 2.5. *Each type has a parity game.*

Proof. The relation $\sqsubseteq$ is a strict order and doesn’t contain cycles. Its restriction to the graph of T can be linearized. This gives the relative priorities of the nodes and ensures condition (4) from the definition. Starting from the least priorities (i.e. the larger types), we can now choose a priority odd / even compatible with this linearization. (Note that we

don't actually need to linearize the graph and can instead chose a *normalized parity game*, i.e. one that minimizes gaps in priorities.) $\square$

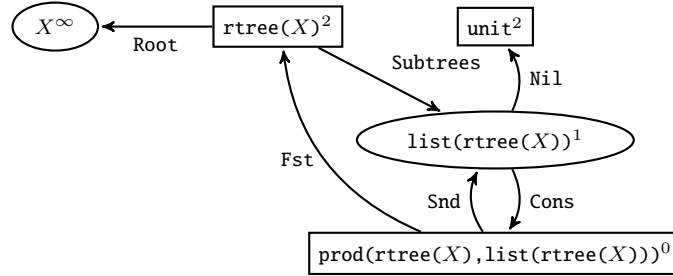
Here are the first two parity games from page 13, seen as parity games for **stream(nat)** and **list(nat)** (the priorities are written as an exponent).



The last example from page 13 corresponds to a coinductive version of Rose trees:

```
codata rtree('x) where
  | Root : rtree('x) -> 'x
  | Subtrees : rtree('x) -> list(rtree('x))
```

with parity game



As those examples show, the priority of T can be anything from minimal to maximal in its parity graph.

Lemma 2.6. *For any type T , if G is a parity game for T and if T_0 is a node of G , we have a natural isomorphism $\|G_{T_0}\| \cong |T_0|$.*

Proof. The proof follows from the following simple fact by a simple induction:

Fact 2.7. If G is a parity game for T and T_0 one of its maximal nodes, then the predecessor game $G/T_0(Y)$ is a parity game for $T[T_0 = Y]$.

$\square$

Corollary 2.8. *If T is a type and G a parity game for T , we have $\mathcal{W}(G)_T \cong |T|$. In particular, $v \in \llbracket T \rrbracket$ is total iff every branch of v has even principal priority.¹³*

¹³Since any $v \in \llbracket T \rrbracket$ gives a strategy in the game of T , priorities can be looked up in the game of T .

2.3. Forgetting Types. A consequence of the previous section is that checking totality doesn't really need types: it needs priorities. We thus annotate each occurrence of constructor / destructor in a definition with its priority (taken from the type's parity game). The refined notion of value is given by

$$v ::= \perp \mid x \mid \mathbf{C}^p v \mid \{\mathbf{D}_1 = v_1; \dots; \mathbf{D}_k = v_k\}^p$$

where

- each x is in one of the X_i ,
- each priority p belong to a finite set of natural numbers,
- each $\mathbf{C}$ belongs to a finite set of *constructors*, and their priority is odd,
- each $\mathbf{D}_i$ belongs to a finite set of *destructors*, and their priority is even,
- k can be 0.

The whole domain of values $\mathcal{V}$ now has a notion of totality.

Definition 2.9. Totality for $\mathcal{V}$ is defined as $v \in |\mathcal{V}|$ iff and only if every branch of v has even principal priority.

Priorities are an artefact used for checking totality. They play no role in definitions or evaluation and are inferred internally:

- (1) each instance of a constructor / destructor is annotated by its type during type checking,
- (2) all the types appearing in the definitions are gathered (and completed) to a parity games,
- (3) each constructor / destructor is then associated with the priority of its type (and the type itself can be dropped).

Because none of this has any impact on evaluation, checking if a definition is total amounts to checking that the annotated definition is total, which can be done without types.

3. CALL-GRAPH AND TOTALITY

3.1. Introduction. A function f between domains with totality is *total* if it sends total elements to total elements. Equivalently, it is total if whenever $f(v) = w$, either v is non total, or w is total. As a result, checking that a recursive definition of $\mathbf{f}$ is total requires looking at the arguments that $\mathbf{f}$ “consumes” and at the result that $\mathbf{f}$ “constructs”. The two are not independent as shown by the following function:

```
val sums : stream(list(nat)) -> stream(nat)
  | sums { Head = [] ; Tail = s } = { Head = 0 ; Tail = sums s }
  | sums { Head = [n] ; Tail = s } = { Head = n ; Tail = sums s }
  | sums { Head = n::m::l ; Tail = s }
    = sums { Head = (add n m)::l ; Tail = s }
```

where we use the following abbreviations:

- `[]` for `Nil`,
- `[a]` for `Cons { Fst = a ; Snd = Nil }`,
- `a::l` for `Cons { Fst = a ; Snd = l }`.

This function maps a stream of lists of natural numbers into a stream of natural numbers by computing the sums of all the lists. It does so by accumulating the partial sums of a given list in its first element. This function is productive (it constructs something) because the third clause cannot occur infinitely many times consecutively (it consumes something).

As we saw in the previous section, a total value in type t is a winning strategy for a parity game of t . The totality checker thus needs to check something like:

for all pairs (v, w) in the graph of the recursive function,

- *either all the infinite branches of w have an even principal priority,*
- *or v contains an infinite branch whose principal priority is odd.*

The analysis is local: it only looks at one mutually recursive definition. The previously defined functions are assumed to be total, but nothing more is known about them. For that reason, we only look for infinite branches that come from the current definition.

The first step of the analysis is to extract some information from the clauses of the definition. The resulting structure is called the *call-graph* (section 3.3). The analysis then looks at infinite path in the call-graph. In order to use the size-change principle [LJBA01, Hyv14], care must be taken to restrict the information kept along calls to a finite set. This is done by introducing a notion of approximation and by collapsing calls to bounded information.

Formalizing and proving that this condition is correct will be done in the last section (starting on page 26). We will for the moment only give a high level description of how one can implement the test, using both the concepts developed in the previous section and the size-change principle.

Simplifying assumptions. In order to reduce the notation overhead, we assume all the functions have a single argument. As far as expressivity is concerned, this is not a real restriction: we can introduce ad-hoc codata (product types) to uncurry all functions. This restriction is of course not enforced in the implementation. Dealing with multiple arguments would require using substitutions instead of terms [Hyv14].

3.2. Interpreting Calls. A single clause may contain several recursive calls. For example, the hypothetical rule

$$| \text{f } (C_1 \{ D_1 = \mathbf{x}; D_2 = C_2 \ y \}) = C_3 \ (\underline{\text{f } (C_2 \ (\underline{\text{f } (C_1 \ y)})}))$$

contains two recursive calls (underlined). It is clear that the final result starts with C_3 , constructed above the leftmost recursive call. It is also clear that the rightmost recursive call uses part of the initial argument. It is however unclear if the rightmost call contributes anything to the final result or if the leftmost call uses part of the initial argument. For each recursive call in a recursive definition, we keep some information about

- the output branch above this recursive call,
- the way the argument of the call is constructed from the initial argument.

The information about the argument of the recursive call uses the same technology that was used when checking termination [Hyv14]. The information about the output is simpler: we only record the number of constructors above the recursive call. A recursive call is guarded simply when it occurs under at least one constructor / record.¹⁴ Each call will thus be interpreted by a rule of the form:

$$\text{f } \mathbf{x} \mapsto \langle W \rangle \text{f } v$$

where W is a *weight* and v is a *generalized pattern* with free variable $\mathbf{x}$. For example, the interpretation of the previous clause will consist of two calls:

¹⁴Refer to page 45 for an idea about what could be done without this simplification.

- $\mathbf{f} \ \mathbf{x} \mapsto \langle -1 \rangle \ \mathbf{f} \ (\mathbf{C}_2 \ \Omega)$ for the leftmost call:
 - $\langle -1 \rangle$: this call is guarded by one constructor ($\mathbf{C}_3$)
 - $\mathbf{C}_2 \ \Omega$: the argument starts with constructor $\mathbf{C}_2$
- $\mathbf{f} \ \mathbf{x} \mapsto \Omega \ \mathbf{f} \ (\mathbf{C}_1 \ \mathbf{C}_2^- \ .\mathbf{D}_2 \ \mathbf{C}_1^- \ \mathbf{x})$ for the rightmost call:
 - Ω : we don't know what this call contributes to the result
 - $(\mathbf{C}_1 \ \mathbf{C}_2^- \ .\mathbf{D}_2 \ \mathbf{C}_1^- \ \mathbf{x})$: the argument starts with $\mathbf{C}_1$ and “ $\mathbf{C}_2^- \ .\mathbf{D}_2 \ \mathbf{C}_1^- \ \mathbf{x}$ ” represents the $\mathbf{y}$ from the definition: it is obtained from the argument $\mathbf{x}$ by: removing $\mathbf{C}_1$ (written $\mathbf{C}_1^-$), projecting on field $\mathbf{D}_2$ (written $.\mathbf{D}_2$) and removing $\mathbf{C}_2$ (written $\mathbf{C}_2^-$).

Since we want to check totality using parity games, counting the constructors is not enough: we also need to remember their priority. The weights are thus more complex than plain integers.

Definition 3.1. Define

- (1) $\mathbb{Z}_\infty = \mathbb{Z} \cup \{\infty\}$ with the obvious addition and order,
- (2) $\mathbb{W}$, the set of *weights* is generated by

$$W ::= \Omega \mid \langle \rangle \mid \langle w \rangle^p \mid W + W$$

where $w \in \mathbb{Z}_\infty$ and each p comes from a finite set $\mathbb{P}$ of natural numbers called priorities. This set is quotiented by

- associativity and commutativity of $+$,
 - $\Omega + W = \Omega$,
 - $\langle w \rangle^p + \langle w' \rangle^p = \langle w + w' \rangle^p$,
 - equivalence generated from the order given below.
- (3) Order $\leq$ on weights is generated from
 - $\Omega \leq W$,
 - $\langle 0 \rangle^p \leq \langle \rangle$,
 - $\langle w_1 \rangle^p \leq \langle w_2 \rangle^p$ whenever $w_1 \geq w_2$ in $\mathbb{Z}_\infty$,
 - $W + W' \leq W$. (In particular, $W + \Omega = \Omega$.)

“ κ_p ” is a synonym for $\langle 1 \rangle^p$ and we usually write $\langle W \rangle$ for an arbitrary weight and $\langle 0 \rangle$ for $\sum_{0 \leq i \leq p} \langle 0 \rangle^i$ (where p is the maximal priority involved locally). The symbols $\langle$ and $\rangle$ are loosely used as grouping for weights...

The intuition is that κ_p represents anything that adds *at most* one constructor of priority p . Similarly, $-\kappa_p$ represents anything that removes at least one constructor of priority p . Note that $\langle 0 \rangle^p$ is different from $\langle \rangle$: the former could add one constructor of priority p and remove another, while the later does nothing. The weight Ω is not very different from the weight $\sum_p \langle \infty \rangle^p$ and can be identified with it in the implementation.

Definition 3.2. Generalized patterns are given by the following grammar

$$t ::= \mathbf{C}^p t \mid \{\mathbf{D}_1 = t_1; \dots; \mathbf{D}_n = t_n\}^p \mid \langle W_1 \rangle t_1 * \dots * \langle W_n \rangle t_n \mid \mathbf{C}^p t \mid .\mathbf{D}^p t \mid \mathbf{x} \mid \mathbf{0} \mid \mathbf{f} t$$

where $n \geq 0$, $\mathbf{x}$ is a *formal parameter*, each $\mathbf{f}$ belongs to a finite set of *function names* and each $\langle W \rangle$ is a weight. As previously, $\mathbf{C}$ and $\mathbf{D}$ come from a finite set of constructor and destructor names, and their priorities come from a finite set of natural numbers. They are respectively odd and even. The product is implicitly commutative, idempotent and associative.

Here are some points worth remembering.

- $\mathbf{x}$ is the parameter (unique in our case) of the definition.
- Priorities are associated to *instances* of constructors: the list constructor may appear in the parity game with different priority!¹⁵ We write $\mathbf{C}^p$ to give a name to the corresponding priority, and just $\mathbf{C}$ when the priority is not important.
- In ML style, the term $\mathbf{C}^-u$ is similar to the partial “`match u with C v -> v`” with a single pattern. This is used to deconstruct a value.
- $\mathbf{0}$ represents runtime errors which we can ignore in our analysis because typing forbids them. It propagates through values.
- The product is used to approximate records: $\{\mathbf{Fst}=\mathbf{Succ}^- \mathbf{x}; \mathbf{Snd}=\mathbf{Succ} \mathbf{x}\}$ can for example be approximated by $\langle 1 \rangle \mathbf{Succ}^- \mathbf{x} * \langle 2 \rangle \mathbf{x}$. All the branches of an element approximated by $\langle W_1 \rangle t_1 * \dots * \langle W_n \rangle t_n$ must be approximated by a $\langle W_i \rangle t_i$.

Definition 3.3. A call from $\mathbf{g}$ to $\mathbf{f}$ is of the form “ $\mathbf{g} \mapsto \langle W \rangle \mathbf{f} v$ ” where $\langle W \rangle \in \mathbb{W}$ and v is a generalized pattern.

For symmetry reasons, *output weights* are counted negatively: *adding* one constructor on the output will be represented as $\langle -1 \rangle$. Just like *removing* some constructor in a recursive argument is “good” (think structural recursion), *adding* a constructor on the result is “good” (think guardedness). The two are thus counted similarly.

The next few pages recall, without proofs, the notions and results that are useful for implementing the totality criterion. The proofs will be given in the next section.

Definition 3.4. We define a reduction relation on generalized patterns:

$$\begin{array}{llll}
(0) & \mathbf{C0}, \mathbf{C}^- \mathbf{0}, \mathbf{.D0} & \rightarrow & \mathbf{0} \\
(0) & \{\dots; \mathbf{D} = \mathbf{0}; \dots\} & \rightarrow & \mathbf{0} \\
(0) & \langle W \rangle \mathbf{0} * P & \rightarrow & P \\
(0) & \langle W \rangle \mathbf{0} & \rightarrow & \mathbf{0} & \text{only for unary product} \\
(1) & \mathbf{C}^- \mathbf{C}t & \rightarrow & t \\
(1) & \mathbf{.D}_j \{\dots; \mathbf{D}_j = t_j; \dots\} & \rightarrow & t_j & \text{if no } t_i \rightarrow^* \mathbf{0} \\
(2) & \mathbf{C}^- \mathbf{C}'t & \rightarrow & \mathbf{0} & \text{if } \mathbf{C} \neq \mathbf{C}' \\
(2) & \mathbf{.D}\{\dots\} & \rightarrow & \mathbf{0} & \text{if the record has no field D} \\
(2) & \mathbf{C}^- \{\dots\} & \rightarrow & \mathbf{0} \\
(2) & \mathbf{.D} \mathbf{C}t & \rightarrow & \mathbf{0} \\
(3) & \langle W \rangle \mathbf{C}^p t & \rightarrow & \langle W + \kappa_p \rangle t \\
(3) & \langle W \rangle \{\mathbf{D}_1 = t_1; \dots\}^p & \rightarrow & \langle W + \kappa_p \rangle t_1 * \dots & \text{if the record is not empty} \\
(3) & \mathbf{C}^{p-} \prod_i \langle W_i \rangle t_i & \rightarrow & \prod_i \langle W_i - \kappa_p \rangle t_i \\
(3) & \mathbf{.D}^p \prod_i \langle W_i \rangle t_i & \rightarrow & \prod_i \langle W_i - \kappa_p \rangle t_i \\
(3) & (\langle W \rangle^p \prod_i \langle W_i \rangle t_i) * P & \rightarrow & (\prod_i \langle W + W_i - \kappa_p \rangle t_i) * P
\end{array}$$

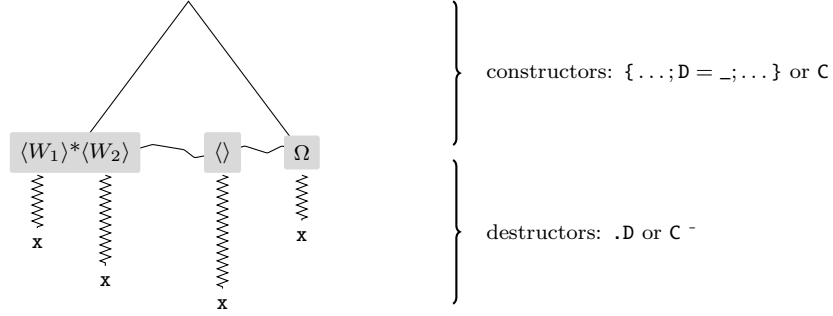
Group (1) of reductions corresponds to the operational semantics of the language, group (3) deals with approximations in a way that is compatible with group (1), and groups (0) and (2) deal with errors. In particular, in group (2)

- the first 3 reductions are forbidden by type checking (we cannot project a constructor, or pattern match on a record),
- the last reduction is forbidden by the operational semantics (if the argument starts with a $\mathbf{C}'$, the reduction doesn’t take the $\mathbf{C}$ clause).

¹⁵for example when dealing with lists of streams of lists

Looking at groups (3) of reductions, it is clear that weights absorb all constructors on their right and all destructors on their left. As a result, non- $\mathbf{0}$ normal forms have a very specific shape:

- a tree of constructors (akin to a value),
- followed by (linear) branches of destructors.



Lemma 3.5.

- (1) *This reduction is confluent and strongly normalizing. We write $\text{nf}(t)$ for the normal form of t .*
- (2) *The normal forms are either $\mathbf{0}$, or generated by the grammar*

$$\begin{aligned}
 p &::= \mathbf{C}p \mid \{ \dots; \mathbf{D}_i = p_i; \dots \} \mid \langle W_1 \rangle \lambda_1 * \dots * \langle W_n \rangle \lambda_n \\
 \lambda &::= \{ \} \mid \delta \mathbf{x} \\
 \delta &::= \delta \mathbf{C}^- \mid \delta . \mathbf{D}
 \end{aligned}$$

There is a notion of approximation: for example, $\mathbf{S}^- \mathbf{S}^- \mathbf{S}^- \mathbf{S}^- \mathbf{x}$ can be approximated by $\mathbf{S}^- \mathbf{S}^- \langle -2 \rangle \mathbf{x}$ where $\langle -2 \rangle$ means that “at least 2 constructors were removed”.

Definition 3.6. The relation “ $u \leq v$ ” (read “ u approximates v ”) is defined with

- $\leq$ is contextual: if $u \leq v$ then $t[\mathbf{x} := u] \leq t[\mathbf{x} := v]$,
- $\leq$ is compatible with reduction: if $u \rightarrow v$ then $v \leq u$, and in particular, $\text{nf}(u) \leq u$,
- $\mathbf{0}$ is a *greatest* element,
- for every weights $\langle V \rangle \leq \langle W \rangle$, we have $\langle V \rangle t \leq \langle W \rangle t$,
- $\langle 0 \rangle t \leq t$

This order is extended to calls with

$$\begin{aligned}
 \mathbf{f} \mathbf{x} \mapsto \langle W \rangle \mathbf{f} u &\leq \mathbf{f} \mathbf{x} \mapsto \langle W' \rangle \mathbf{f} u' \\
 &\text{iff} \\
 \langle W \rangle &\leq \langle W' \rangle \quad \text{and} \quad u \leq u'
 \end{aligned}$$

3.3. Call-Graph and Composition. The next definition is very verbose, but the example following it should make it clearer.

Definition 3.7. From a recursive definition, we construct its (oriented) *call-graph* with

- vertices are the names of the functions mutually defined
- arcs from $\mathbf{f}$ to $\mathbf{g}$ are given by all the “ $\mathbf{f} \mathbf{x} \mapsto \langle W \rangle \mathbf{g} v[\sigma_p]$ ” where “ $\langle W \rangle \mathbf{g} v$ ” $\in \text{calls}(u)$ for some clause $\mathbf{f} p = u$ and σ_p and $\text{calls}(u)$ are defined with:
 - (1) Given a pattern p , define the substitution σ_p as follows:
 - $\sigma_{\mathbf{y}} = [\mathbf{y} := \mathbf{x}]$

- $\sigma_{Cp} = C^- \circ \sigma_p$,
 - $\sigma_{\{...\mathbf{D}_i=p_i;...\}} = \bigcup_i (\cdot \mathbf{D}_i \circ \sigma_{p_i})$.
- where $\circ$ represents composition of substitutions.

(2) From each right-hand side u of a clause $\mathbf{f} p_1 \dots p_k = u$, we define $\text{calls}(u)$:

$$\text{calls}(C^p u) = \langle \kappa_p \rangle . \text{calls}(u) \quad (i)$$

$$\text{calls}(\{ \dots ; \mathbf{D}_i = u_i ; \dots \}^p) = \bigcup_i \langle \kappa_p \rangle . \text{calls}(u_i)$$

$$\text{calls}(\mathbf{f} u_1 \dots u_l) = \{ \langle 0 \rangle \mathbf{f} u_1^\Omega \dots u_l^\Omega \} \cup \bigcup_i \Omega . \text{calls}(u_i) \quad (ii)$$

$$\text{calls}(\mathbf{g} u_1 \dots u_l) = \bigcup_i \Omega . \text{calls}(u_i) \quad (iii)$$

$$\text{calls}(\mathbf{x}) = \emptyset$$

where

- (i) $\langle \kappa_p \rangle . S$ is a notation for $\{ \langle \kappa_p \rangle u \mid u \in S \}$,
- (ii) if $\mathbf{f}$ is one of the functions recursively defined and where u^Ω is obtained from u by replacing each function application $\mathbf{g} u_1 \dots u_l$ (recursive or otherwise) by Ω .
- (iii) the name $\mathbf{g}$ could also be a parameter $\mathbf{x}_j$ coming from the left pattern.

An example is probably more informative. Consider the definition from page 17 with explicit priorities:

```
val sums : stream0(list1(nat1)) -> stream0(nat1)
| sums { Head0 = []1 ; Tail0 = s } = { Head0 =  $\emptyset$  ; Tail0 = sums s }
| sums { Head0 = [n]1 ; Tail0 = s } = { Head0 = n ; Tail0 = sums s }
| sums { Head0 = n::1m::1l ; Tail0 = s }
  = sums { Head0 = (add n1 m1)::1l ; Tail0 = s }
```

The three associated calls will be $\text{sums } \mathbf{x} \mapsto \langle -1 \rangle^0 \text{sums } (\text{Tail}^0 \mathbf{x})$ (twice) and

$$\text{sums } \mathbf{x} \mapsto \langle \rangle \text{sums } \{ \text{Head}^0 = \Omega::^1(\text{Head}^0 . \text{Tail}^{0-} . \text{Snd}^1 . \text{Tail}^{0-} . \text{Snd}^1 \mathbf{x}) ; \\ \text{Tail}^0 = \text{Tail}^0 \mathbf{x} \}$$

(Recall the “ $a::l$ ” is an abbreviation for “ $\text{Cons}\{\text{Fst}=a; \text{Snd}=l\}$ ”.)

A call gives some information about one recursive call: depth of the recursive call, and part of the shape of the original argument. We can compose them:

```
val length : list(x) -> nat
| length Nil = Zero
| length (Cons{Fst=x; Snd=l}) = Succ (length l)
```

has a single call: “ $\text{length } l \mapsto \langle -1 \rangle \text{length } (\text{Snd } \text{Cons}^- l)$ ”. Composing this call with itself will give $\text{length } l \mapsto \langle -2 \rangle \text{length } (\text{Snd } \text{Cons}^- . \text{Snd } \text{Cons}^- l)$.

Definition 3.8. The composition $\beta \circ \alpha$ of the calls $\alpha = \mathbf{f} \mathbf{x} \mapsto \langle W \rangle \mathbf{g} p$ and $\beta = \mathbf{g} \mathbf{x} \mapsto \langle W' \rangle \mathbf{h} q$ is defined as

$$\beta \circ \alpha = \mathbf{f} \mathbf{x} \mapsto \langle W + W' \rangle \mathbf{h} q[\mathbf{x} := p]$$

Some compositions are automatically ignored: “ $\mathbf{f} \mathbf{x} \mapsto \langle \rangle \mathbf{f} C_2 C_1^- \mathbf{x}$ ”, arising from

```
val f (C1 x) = f (C2 x)
| ...
```

gives $\mathbf{0}$ when composed with itself. This is because “ $\mathbf{C}_2 \mathbf{x}$ ” doesn’t match “ $\mathbf{C}_1 \mathbf{y}$ ”.

Totality can sometimes be checked on the call-graph. Since we haven’t yet formalized infinite compositions, this condition is for the moment expressed in a very informal way.

Definition 3.9 (Informal Totality Condition). A call-graph is *total* if, along all infinite path,

- (1) either its output weight *obviously* has an even principal priority,
- (2) or one of the branches in its argument *obviously* has an odd principal priority.

Two such examples are the call-graphs with a single arc:

- (1) $\mathbf{fx} \mapsto \langle \langle -1 \rangle^2 + \langle -2 \rangle^1 \rangle \mathbf{f} \mathbf{x}$: the prefixes of the only infinite path give the compositions
 - $\mathbf{fx} \mapsto \langle \langle -2 \rangle^2 + \langle -4 \rangle^1 \rangle \mathbf{f} \mathbf{x}$
 - $\mathbf{fx} \mapsto \langle \langle -3 \rangle^2 + \langle -6 \rangle^1 \rangle \mathbf{f} \mathbf{x}$
 - ...

The recursive calls are guarded by an increasing number of constructors of priority 2 (coinductive). Some inductive constructors are also added,¹⁶ but those have smaller priority. The limit will construct a term with infinitely many constructors of priority 2 and infinitely many constructors of priority 1: this is a total value.

- (2) $\mathbf{fx} \mapsto \Omega \mathbf{f} \mathbf{Succ}^{1-} \mathbf{x}$: prefixes of the only infinite path in this graph give
 - $\mathbf{fx} \mapsto \Omega \mathbf{f} \mathbf{Succ}^{1-} \mathbf{Succ}^{1-} \mathbf{x}$
 - $\mathbf{fx} \mapsto \Omega \mathbf{f} \mathbf{Succ}^{1-} \mathbf{Succ}^{1-} \mathbf{Succ}^{1-} \mathbf{x}$
 - ...

Such terms only apply to arguments having enough **Succ** constructors (of priority 1) and no other constructors. The limit thus only applies to arguments having infinitely many constructors of priority 1 (and no other): that is not possible of total values!

Theorem 3.10 (informal). *If a call-graph is total, then the original recursive definition defines total functions.*

3.4. Collapsing. The totality condition on call-graphs involves infinite path. It is natural to try using the size-change principle to get a decidable approximation of this condition. For that, we need a *finite* call-graph that is closed under composition (its transitive closure). This is impossible: the example of **length 1** can be composed with itself many times to get

$$\mathbf{length} \ 1 \mapsto \langle -n \rangle \mathbf{length} \ (\mathbf{.Snd} \ \mathbf{Cons}^- \ \dots \ \mathbf{.Snd} \ \mathbf{Cons}^- \ 1)$$

Both the output weight and the recursive argument of the call can grow arbitrarily. To prevent that, we *collapse* the calls to bound their depth and weights [Hyv14].

Definition 3.11. Given a strictly positive bound B , the *weight collapsing function* $\lceil - \rceil_B$ acts on terms by replacing each weight $\langle w \rangle^p$ by $\langle \lceil w \rceil_B \rangle^p$ where

$$\lceil w \rceil_B = \begin{cases} -B & \text{if } w < -B \\ w & \text{if } -B \leq w < B \\ \infty & \text{if } B \leq w \end{cases}$$

¹⁶Recall that constructors are counted negatively for the output.

Ensuring a bounded depth is more complex: we introduce some “ $\langle 0 \rangle$ ” below D constructors and above D destructors. Because of the reduction, those new weights will absorb the constructors with depth greater than D and the destructors with “inverse depth” greater than D . For example, collapsing $C_1 C_2 C_3 \langle W \rangle . D_4 C_3 C_2^- . D_1 \mathbf{x}$ at depth 2 gives

$$C_1 C_2 \langle 0 \rangle C_3 \langle W \rangle C_5^- . D_4 C_3^- \langle 0 \rangle C_2^- . D_1 \mathbf{x} \rightarrow^* C_1 C_2 \langle 1 + W - 3 \rangle C_2^- . D_1 \mathbf{x}$$

Definition 3.12. Given a positive bound D , the *height collapsing function* $-|_D$ acts on terms by integrating constructors below D and destructors above D into weights:

$$\begin{aligned} (C\ t)|_i &\stackrel{\text{def}}{=} C\ (t|_{i-1}) && \text{if } i > 0 \\ \{\dots; D_k = t_k; \dots\}|_i &\stackrel{\text{def}}{=} \{\dots; D_k = t_k|_{i-1}; \dots\} && \text{if } i > 0 \\ (W\beta)|_i &\stackrel{\text{def}}{=} W(\beta|_D) && \text{if } i > 0 \\ \beta|_i &\stackrel{\text{def}}{=} \beta|_D \\ t|_0 &\stackrel{\text{def}}{=} \text{nf}(\langle 0 \rangle t)|_D && (*) \end{aligned}$$

and

$$\begin{aligned} (\langle W_1 \rangle t_1 * \dots * \langle W_n \rangle t_n)|_i &\stackrel{\text{def}}{=} (\langle W_1 \rangle t_1)|_i * \dots * (\langle W_n \rangle t_n)|_i && (**) \\ (\langle W \rangle \{\})|_i &\stackrel{\text{def}}{=} \langle W \rangle \{\} && (**) \\ (\langle W \rangle \beta \mathbf{x})|_i &\stackrel{\text{def}}{=} \langle W \rangle \beta|_i \mathbf{x} && (**) \\ (\beta C^p)|_i &\stackrel{\text{def}}{=} \beta|_{i-1} C^p \\ (\beta . D^p)|_i &\stackrel{\text{def}}{=} \beta|_{i-1} . D^p \\ \beta|_0 &\stackrel{\text{def}}{=} \beta \langle 0 \rangle \end{aligned}$$

- The clauses are not disjoint and only the first appropriate one is used.
- We compute a normal form in clause (*) to ensure that the clauses (**) cover all cases (since weights absorb constructors on their right, $\langle 0 \rangle t$ doesn’t contain constructors),

We can extend collapsing to calls.

Definition 3.13. If α is a call $\mathbf{f}\ \mathbf{x} \mapsto \langle W \rangle\ \mathbf{g}\ u$, we put

- $[\alpha]_B = \mathbf{f}\ \mathbf{x} \mapsto [W]_B\ \mathbf{g}\ [u]_B$
- $\alpha|_D = \mathbf{f}\ \mathbf{x} \mapsto \langle W \rangle\ \mathbf{g}\ u|_D$

Lemma 3.14 ([Hyv14]). *For any call α , we have*

- $[\alpha]_B \leq \alpha$,
- $\alpha|_D \leq \alpha$.

Definition 3.15. Given some bounds B and D , *collapsed composition* is defined by

$$\beta \diamond_{B,D} \alpha \quad := \quad \text{nf} \left(\left[(\beta \circ \alpha)|_D \right]_B \right)$$

Since the bounds are fixed, we usually write $\beta \diamond \alpha$.

Unfortunately, this composition is *not* associative.¹⁷ For example, with bound $B = 2$ if $\alpha = \mathbf{f}\ \mathbf{x} \mapsto \langle 1 \rangle\ \mathbf{f}\ \mathbf{x}$ and $\beta = \mathbf{f}\ \mathbf{x} \mapsto \langle -1 \rangle\ \mathbf{f}\ \mathbf{x}$ we have $\beta \diamond (\alpha \diamond \alpha) = \mathbf{f}\ \mathbf{x} \mapsto \langle \infty \rangle\ \mathbf{f}\ \mathbf{x}$ but $(\beta \diamond \alpha) \diamond \alpha = \mathbf{f}\ \mathbf{x} \mapsto \langle 1 \rangle\ \mathbf{f}\ \mathbf{x}$. The next property can be seen as a kind of weak associativity.

¹⁷except when $B = 1$ and $D = 0$

Lemma 3.16. *If $\sigma_n \circ \dots \circ \sigma_1 \neq \mathbf{0}$, and if τ_1 and τ_2 are the results of computing $\sigma_n \diamond \dots \diamond \sigma_1$ in two different ways, then τ_1 and τ_2 are compatible, written $\tau_1 \supset \tau_2$. This means that there is some $\tau \neq \mathbf{0}$ such that $\tau_1 \leq \tau$ and $\tau_2 \leq \tau$.*

Proof. Take $\tau = \sigma_n \circ \dots \circ \sigma_1$. □

3.5. Size-Change Principle. The initial call-graph G of a definition is finite, and collapsed composition ensures that there exists a finite transitive closure of this initial call-graph: starting with $G_0 = G$, we define the new edges of G_{n+1} with:

if α and β are edges from $\mathbf{f}$ to $\mathbf{g}$ and from $\mathbf{g}$ to $\mathbf{h}$ in G_n , then $\beta \diamond \alpha$ is a new edge from $\mathbf{f}$ to $\mathbf{h}$ in G_{n+1} .

Finiteness of the set of bounded terms guarantees that this sequence stabilizes on some graph, written G^* .

To simplify the statement of the size-change totality principle, we first define the p -norm a finite branch inside a generalized pattern: it counts constructors and destructors of priority p .

Definition 3.17. Given $p \in \mathbb{P}$ and a branch β in a generalized pattern, the p -norm of β , written $|\beta|_p$ is defined with:

- $|\mathbf{C}^p \beta|_p = |\beta|_p + 1$ and $|\mathbf{C}^q \beta|_p = |\beta|_p$ if $p \neq q$,
- $|\{\mathbf{D} = \beta\}^p|_p = |\beta|_p + 1$ and $|\{\mathbf{D} = \beta\}^q|_p = |\beta|_p$ if $p \neq q$,
- $|\mathbf{C}^{p-} \beta|_p = |\beta|_p - 1$ and $|\mathbf{C}^{q-} \beta|_p = |\beta|_p$ if $p \neq q$,
- $|\mathbf{D}^{p-} \beta|_p = |\beta|_p - 1$ and $|\mathbf{D}^{q-} \beta|_p = |\beta|_p$ if $p \neq q$,
- $|\Omega \beta|_p = \infty$,
- $|\langle \rangle \beta|_p = |\beta|_p$,
- $|\langle w \rangle^p \beta|_p = w$ and $|\langle w \rangle^q \beta|_p = |\beta|_p$ if $p \neq q$,
- $|\langle W_1 + \dots + W_n \rangle \beta|_p = \sum_i |\langle W_i \rangle \beta|_p$.

Theorem 3.18 (size-change principle). *Suppose every loop $\gamma = \mathbf{f} \ \mathbf{x} \mapsto \langle W \rangle \ \mathbf{f} \ u$ in G^* that satisfies $\gamma \supset \gamma \diamond \gamma$ also satisfies one of the following two conditions:*

- *either the maximal priority appearing in $\langle W \rangle$ is even, with negative weight,*
- *or there is a subterm $\beta \prod \langle W_i \rangle \lambda_i$ of v where the maximal priority p of each $\beta \langle W_i \rangle \lambda_i$ is odd, with negative weight (i.e. $|\beta \langle W_i \rangle \lambda_i|_p < 0$)*

then the definition is total.

Here is an informal sketch of the proof. The combinatorial lemma¹⁸ at the heart of the size-change principle implies that every infinite path can be decomposed (up to a finite prefix) into an infinite sequence of the same idempotent loop ($\gamma \supset \gamma \diamond \gamma$). The condition of totality on G_0 involves such infinite path, i.e., infinite sequences of identical idempotent loops in G^* .

- If such a loop satisfies the first condition, composing it infinitely many times will surely add¹⁹ an infinite number of constructors of even priority, while not involving constructors of higher priority. Hence the infinite path satisfies the first totality condition from page 23.

¹⁸given in a slightly simplified form in section 4.37, or in greater generality in [Hyv14]

¹⁹recall that on the output, negative weight means adding constructors

- If such a loop satisfies the second condition, composing it infinitely many times will explore, in the original argument, a branch $\beta\langle W_{i_0}\rangle\lambda_{i_0}\beta\langle W_{i_1}\rangle\lambda_{i_1}\dots$ containing an infinite number of constructors of odd priority, and no constructor of higher priority. Thus, there is a branch of the initial argument with odd principal priority (second condition from page 23).

3.6. The Algorithm. All this makes it possible to actually implement a correct (but incomplete) totality test for some mutual recursive definition of $\mathbf{f}_1, \dots, \mathbf{f}_n$.

- (1) Compute the call-graph G_0 of the definition. This step is purely syntactical and linear in the size of the definition.
- (2) Compute the transitive closure G^* of G_0 by iterating collapsed composition. G^* is potentially exponentially bigger than G_0 , but this doesn't happen in practice.
- (3) Check all idempotent loops against the conditions of theorem 3.18. Since there are only finitely many branches, finding β can be done by exploring the argument.

This is what is implemented, with minor variations, in the **chariot** prototype. More details about the actual implementation, including the definition of the order or compatibility relations by induction on terms can be found in [Hyv14].

4. SEMANTICS AND CORRECTNESS OF THE CALL-GRAPH

We can now define the semantics of the call-graph from the previous section, and prove correctness of the size change principle. It roughly proceeds as follows:

- we extend the domain of values with non-deterministic sums,
- we define (generalizing definition 3.2) a domain of syntactical operators,
- totality of such an operator implies totality of the corresponding definition,
- the call-graph is another operator and its totality implies that of the previous one.

Several notions from domain theory will be used in this section but the prerequisites are kept to a minimum. They can be found in any of the several introductions to domain theory [Plo83, SHGL94, AJ94]. One particular result that is worth knowing is that any partial order can be completed to an algebraic DCPO whose compact elements are exactly the elements of the original partial order. This *ideal completion* formally adds limits of all directed sets. Moreover, if the original order is a sup-semi-lattice,²⁰ its ideal completion is a domain. This is one way to easily prove that values (definition 1.1) form a domain.

4.1. Smyth power domain. We first extend the grammar of values with formal sums:

$$v ::= \perp \mid \mathbf{C}^p v \mid \{\mathbf{D}_1 = v_1; \dots; \mathbf{D}_k = v_k\}^p \mid v_1 + v_2$$

together with the order generated from:

- the order on $\mathcal{V}$ (definition 1.2),
- commutativity, associativity and idempotence ($v + v = v$) of “+”,
- (multi)linearity of $\mathbf{C}$ and $\{\mathbf{D} = _; \dots\}$,
- $u + v \leq u$.

²⁰A sup-semi-lattice is a partial order where every finite bounded set has a least upper bound.

This gives rise to a preorder instead of a partial order which we implicitly quotient by the corresponding equivalence. This construction is well known: it is the “Smyth power domain” [Smy78]. The ideal completion of the finite terms generated by this grammar introduces *some* infinite sums: the Smyth power domain contains all “finitely generated” sums.²¹ The following gives a concrete description of the corresponding order [Smy83, AJ94].

Proposition 4.1. *The Smyth power domain on $\mathcal{V}$ satisfies*

- (1) *Compact elements are finite sets of compact elements of $\mathcal{V}$,*
- (2) *Elements are finitely generated subsets $V \subseteq \mathcal{V}$,*
- (3) *$U \leq V$ iff $V^\uparrow \subseteq U^\uparrow$, (where $V^\uparrow = \bigcup_{v \in V} v^\uparrow$ and $v^\uparrow = \{u \mid v \leq u\}$)*
- (4) *directed least upper bounds are given by intersections,*
- (5) *binary greatest lower bounds exist and are given by unions.*

The resulting structure is a preorder rather than a partial order, but it is possible to take $V^\uparrow$ as a representative for the equivalence class of V (if V is finitely generated, so is $V^\uparrow$) in which case the order is reverse inclusion. When working with concrete examples, it is easier to take $\text{Min}(V)$, the set of minimal elements of V , as a representative of V , which will often be finite. This can serve as a representative of $V^\uparrow$ since $V^\uparrow = \text{Min}(V)^\uparrow$ for any subset of $\mathcal{V}$. Note that this gives a way to check that an infinite sum is finitely generated: we need to check that it is the limit of a directed set (or chain) of finite sums of compact elements.

We also need a special term $\mathbf{0}$, greater than all other elements, to denote errors.

Lemma 4.2. *For any domain D , $D \cup \{\top\}$ is a domain.²²*

Applied to a Smyth power domain, adding a greatest element amounts to allowing the empty set, which is explicitly forbidden in the definition of finitely generated set. We write this new empty sum $\mathbf{0}$.

Definition 4.3. The domain $\mathcal{A}$ of values with approximations is obtained from $\mathcal{V}$ by the Smyth power domain construction, allowing the empty sum.

Definition 4.4. An element of $\mathcal{A}$ (a sum of elements of $\mathcal{V}$) is total when all its summands are total. In particular, $\mathbf{0}$ is total.

The following follows directly from lemma 1.8 and proposition 4.1. It implies in particular that totality is compatible with equivalence: if $T_1 \approx T_2$ and T_1 is total, so is T_2 .

Lemma 4.5. *If $T_1 \leq T_2$ in $\mathcal{A}$ and if T_1 is total, then so is T_2 .*

Here is a summary of the properties of $\mathcal{A}$.

Corollary 4.6. *The compact elements of $\mathcal{A}$ are inductively generated by*

$$v \quad ::= \quad \perp \quad | \quad \mathbf{C}^p v \quad | \quad \{\mathbf{D}_1 = v_1; \dots; \mathbf{D}_k = v_k\}^p \quad | \quad v_1 + v_2 \quad | \quad \mathbf{0}$$

The order satisfies

- *if $u \leq v$ in $\mathcal{V}$ then $u \leq v$ in $\mathcal{A}$,*
- *$+$ is commutative, associative and idempotent, with neutral element $\mathbf{0}$,*
- *$u + v$ is the greatest lower bound of u and v and $\mathbf{0}$ is the greatest element,*
- *constructors and records are (multi) linear:*

²¹The sets of limits of the infinite branches of a finitely branching tree of elements of the domain.

²²Domains with a top element are in fact complete lattices and thus, algebraic lattices.

$$\begin{aligned}
- \mathbf{C}(\sum_i v_i) &= \sum_i \mathbf{C}v_i \\
- \{\dots; \mathbf{D} = \sum_i t_i; \dots\} &= \sum_i \{\dots; \mathbf{D} = t_i; \dots\}.
\end{aligned}$$

4.2. Preliminaries: Recursion and Fixed Points.

Simplifying assumption. In order to simplify notation, we will restrict the rest of this section to the case where the recursive definition we are interested in contains a single function (no mutually recursive functions) with a single argument.

A formula for fixed points. Whenever $\varphi : D \rightarrow D$ is a continuous function on a domain and $b \in D$ such that $b \leq \varphi(b)$, it has a least fixed point greater than b (Kleene theorem) defined with

$$\text{fix}(f, b) = \bigsqcup_{n \geq 0}^{\uparrow} f^n(b)$$

In our case, we are interested in the fixed point of an operator from $[\mathcal{A} \rightarrow \mathcal{A}]$ to itself. Moreover, we require that the functions satisfy $f(\mathbf{0}) = \mathbf{0}$, i.e. that errors propagate. There is a least such function, written Ω :

$$v \mapsto \Omega(v) = \begin{cases} \mathbf{0} & \text{if } v = \mathbf{0} \\ \perp & \text{otherwise} \end{cases}$$

The fixed points we are computing are thus of the form

$$\text{fix}(\varphi, \Omega) = \bigsqcup_{n \geq 0}^{\uparrow} \varphi^n(\Omega)$$

From now on, every fixed point is going to be of this form, and we will simply write $\text{fix}(\varphi)$.

A recursive definition for $\mathbf{f} : A \rightarrow B$ gives rise to an operator $\Theta_{\mathbf{f}}$ on $[[A] \rightarrow [B]]$ whose fixed point is called the semantics of $\mathbf{f}$. It is written $[[\mathbf{f}]] : [A] \rightarrow [B]$. The operator $\Theta_{\mathbf{f}}$ is defined as follows: if $f : [A] \rightarrow [B]$ and $v \in [A]$, the value of $\Theta_{\mathbf{f}}(f)(v)$ is obtained by:

- (1) taking the first clause “ $\mathbf{f} \ p = u$ ” in the definition of $\mathbf{f}$ where p matches v ,
- (2) returning $[[u[\sigma]]]_{\rho, \mathbf{f} := f}$ where σ is the most general unifier of u and p .

The unifier matching a pattern p to a value v can be described as $[p := v]$, computed inductively:

- $[Cp := Cv] = [p := v]$,
- $[\{\mathbf{D}_1 = p_1; \dots; \mathbf{D}_n = p_n\} := \{\mathbf{D}_1 = v_1; \dots; \mathbf{D}_n = v_n\}] = [p_1 := v_1] \cup \dots \cup [p_n := v_n]$,
- $[Cp := C'v]$ (when $C \neq C'$), $[\{\dots\} := \{\dots\}]$ (when the records have different sets of fields), $[Cp := \{\dots\}]$ and $[\{\dots\} := Cv]$ are undefined: the patterns don't match the values.

The fact that definitions are well-typed guarantees that there is always a matching clause. It is possible for patterns to overlap, and in that case, the first matching clause is used.

Setup. Our aim is to define a domain that extends the language of the call-graph (definition 3.2). Every element of this domain will represent an *operator* and the call-graph will be a particular one satisfying the condition “if its fixed point is total, then so is the fixed point of the recursive definition”.

A definition of $\mathbf{f}$ naturally gives rise to an “untyped” $\phi_{\mathbf{f}} : [\mathcal{A} \rightarrow \mathcal{A}] \rightarrow [\mathcal{A} \rightarrow \mathcal{A}]$:

- $\phi_{\mathbf{f}}(f)(\sum v) = \sum \phi_{\mathbf{f}}(f)(v)$
- for $\phi_{\mathbf{f}}(f)(v)$, with $v \in \mathcal{V}$, take the first clause “ $\mathbf{f} \mathbf{p} = \mathbf{u}$ ” where $\mathbf{p}$ matches v , and return $\llbracket \mathbf{u}[\sigma] \rrbracket$ where σ is the most general unifier of u and $\mathbf{p}$,
- if no clause matches v , $\phi_{\mathbf{f}}(f)(v) = \mathbf{0}$.

In order for this to make sense, the functions from the environment need to be lifted to accept arbitrary values: if $\mathbf{g}$ is a previously defined function of type $T_1 \rightarrow T_2$, its semantics is extended by giving $\mathbf{0}$ for any value outside of $\llbracket T_1 \rrbracket$.²³

Lemma 4.7.

- (0) $\Omega \leq \phi(\Omega)$.
- (1) $\text{fix}(\phi_{\mathbf{f}})$ restricted to $\llbracket A \rrbracket$ is equal to $\text{fix}(\Theta_{\mathbf{f}})$.
- (2) If $\text{fix}(\phi_{\mathbf{f}})$ is total (on $\mathcal{A}$), then $\text{fix}(\Theta_{\mathbf{f}})$ is total (on $\llbracket A \rrbracket$).

Proof. The first point follows from the definitions. The second follows from the fact that if f satisfies $f(a) \in \llbracket B \rrbracket$ for every $a \in \llbracket A \rrbracket$, then $\phi_{\mathbf{f}}(f)|_{\llbracket A \rrbracket} = \Theta_{\mathbf{f}}(f|_{\llbracket A \rrbracket})$. This follows from the definition of $\Theta_{\mathbf{f}}$ and $\phi_{\mathbf{f}}$. This implies that $\phi_{\mathbf{f}}^n(\Omega)|_{\llbracket A \rrbracket} = \Theta_{\mathbf{f}}^n(\Omega)$. Since $\text{fix}(F) = \bigsqcup_n^{\uparrow} F^n(\Omega)$, we get $\text{fix}(\Theta_{\mathbf{f}}) = \text{fix}(\phi_{\mathbf{f}})|_{\llbracket A \rrbracket}$. The last point is immediate as $b \in \llbracket B \rrbracket$ is total iff $\{b\} \in \mathcal{A}$ is total. $\square$

The following lemma shows how approximations relate to totality.

Lemma 4.8.

- (1) Suppose $\theta \leq \phi$ in $[\mathcal{A} \rightarrow \mathcal{A}] \rightarrow [\mathcal{A} \rightarrow \mathcal{A}]$, then $\text{fix}(\theta) \leq \text{fix}(\phi)$ in $\mathcal{A} \rightarrow \mathcal{A}$.
- (2) If $f \leq g$ in $\mathcal{A} \rightarrow \mathcal{A}$ and f is total, then so is g .

Proof. The first point follows from the fact that $\text{fix}(\phi) = \bigsqcup^{\uparrow} \phi^n(\Omega)$ and that $\theta \leq \phi$ implies that $\theta^n \leq \phi^n$ for any n . The second point follows from the fact that if $U \leq V$ (in $\mathcal{A}$) and U is total, then V is also total (lemma 4.5). $\square$

4.3. A Language for Operators.

4.3.1. Terms.

Definition 4.9. $\mathcal{F}_0$ is the set of terms inductively generated from

$$\begin{array}{lcl}
 t & ::= & \mathbf{C}^p t \mid \{\mathbf{D}_1 = t_1; \dots; \mathbf{D}_n = t_n\}^p \mid \\
 & & \mathbf{C}^{p^-} t \mid \mathbf{D}^p t \mid \mathbf{x} \mid \\
 & & \Omega \theta \mid \mathbf{f} t \mid \\
 & & \mathbf{0}
 \end{array}$$

where $\mathbf{x}$ is a *formal parameter*, each $\mathbf{f}$ belongs to a finite set of *function names* and each $\theta = \{t_1, \dots, t_n\}$ is a finite set (possibly empty) of terms. As previously, $\mathbf{C}$ and $\mathbf{D}$ come from

²³This is monotonic because types are downward closed sets of values.

a finite set of constructor and destructor names, and their priorities come from a finite set of natural numbers. They are respectively odd and even. We usually write “ Ω ” for $\Omega\{\}$ and “ Ωt ” for $\Omega\{t\}$.

We impose the following (in)equalities:

$$(*) \left\{ \begin{array}{ll} (0) & \mathbf{C0} = \mathbf{0} \\ (0) & \mathbf{C}^-\mathbf{0} = \mathbf{0} \\ (0) & \mathbf{.D0} = \mathbf{0} \\ (0) & \{\mathbf{D} = \mathbf{0}; \dots\} = \mathbf{0} \\ (0) & \Omega\{\mathbf{0}, \theta\} = \mathbf{0} \\ (0) & \mathbf{f0} = \mathbf{0} \\ (1) & \mathbf{C}^-\mathbf{C}t = t \\ (1) & \mathbf{.D}_{i_0}\{\dots; \mathbf{D}_i = t_i; \dots\} \geq t_{i_0} \quad \text{if no } t_i = \mathbf{0} \\ (2) & \mathbf{C}^-\mathbf{C}'t = \mathbf{0} \quad \text{if } \mathbf{C} \neq \mathbf{C}' \\ (2) & \mathbf{.D}\{\dots\} = \mathbf{0} \quad \text{if the record has no field } \mathbf{D} \\ (2) & \mathbf{C}^-\{\dots\} = \mathbf{0} \\ (2) & \mathbf{.DC}t = \mathbf{0} \\ (3) & \Omega\{\mathbf{C}t, \theta\} = \Omega\{t, \theta\} \\ (3) & \Omega\{\{\mathbf{D}_1 = t_1; \dots; \mathbf{D}_k = t_k\}, \theta\} = \Omega\{t_1, \dots, t_k, \theta\} \\ (3) & \mathbf{C}^-\Omega\theta = \Omega\theta \\ (3) & \mathbf{.D}\Omega\theta = \Omega\theta \\ (3) & \Omega\{\Omega\{t_1; \dots; t_k\}, \theta\} = \Omega\{t_1, \dots, t_k, \theta\} \end{array} \right.$$

The order $\leq$ on $\mathcal{F}_0$ is furthermore generated from

- $t \leq \mathbf{0}$,
- $\Omega\{t_1\} \leq \Omega\{t_1, t_2\}$ and $\Omega\{\} \leq t$,
- contextuality (if $C[\]$ is a context, then $t_1 \leq t_2 \implies C[t_1] \leq C[t_2]$),
- if $C[\]$ is a context, then $\Omega\{t\} \leq C[t]$, and in particular, $\Omega\{t\} \leq t$,

We call *simple term* any element of $\mathcal{F}_0$ different from $\mathbf{0}$.

The whole group of inequalities (*) naturally gives rise to a notion of reduction (written $\rightarrow$) when oriented from left to right.

Lemma 4.10. *The reduction $\rightarrow$ is strongly normalizing and confluent. We write $\text{nf}(t)$ for the normal form of a term. Non zero normal forms are given by the grammar*

$$\begin{array}{lll} t & ::= & \mathbf{C}^p t \mid \{\mathbf{D}_1 = t_1; \dots; \mathbf{D}_n = t_n\}^p \mid \beta \\ \beta & ::= & \Omega\{\delta_1, \dots, \delta_n\} \mid \delta \\ \delta & ::= & \mathbf{C}^p \delta \mid \mathbf{.D}^p \delta \mid \mathbf{x} \mid \mathbf{f} t \end{array}$$

Sketch of proof. Reduction is strongly normalizing because the size of the term decreases.

Because of the side condition “no $t_i = \mathbf{0}$ ” in the rule $\mathbf{.D}_{i_0}\{\dots; \mathbf{D}_i = t_i; \dots\} \rightarrow t_{i_0}$ reduction is not strictly speaking a rewriting system and we cannot use Newman’s lemma as usual. Because we don’t know (yet) that reduction is confluent, we write $t \rightarrow^* t'$ for “there exists a reduction from t to t' .” First, we prove

Fact 4.11. We have $t = \mathbf{0}$ iff $t \geq \mathbf{0}$ iff $t \rightarrow^* \mathbf{0}$.

The first equivalence follows from the fact that $\mathbf{0}$ is the greatest element, and the right to left implication of the second equivalence follows from fact that $t \rightarrow t'$ implies $t \geq t'$.

For the left to right implication, it is enough to prove that for all the inequalities $t_1 \leq t_2$ (or $t_1 = t_2$) generating the order, we have “ $t_1 \rightarrow^* \mathbf{0}$ implies $t_2 \rightarrow^* \mathbf{0}$ ”. This is obvious for most of the generating inequalities except:

- the inequalities involving Ω ,
- $\mathbf{C}^-\mathbf{C}t \leq t$,
- contextuality,
- $\Omega\{t\} \leq C[t]$.

In order to deal with those inequalities, we prove

- (1) $\mathbf{C}t \rightarrow^* \mathbf{0}$ iff $t \rightarrow^* \mathbf{0}$,
- (2) $\{\dots; \mathbf{D}_i = t_i; \dots\} \rightarrow^* \mathbf{0}$ iff $t_i \rightarrow^* \mathbf{0}$ for some i ,
- (3) $\Omega\{\dots; t_i; \dots\} \rightarrow^* \mathbf{0}$ iff $t_i \rightarrow^* \mathbf{0}$ for some i , (this point requires the previous two).

Point (3) takes care of all inequalities of the form $\Omega\theta \geq \mathbf{0}$ and of $\Omega\{t\} \leq C[t]$, point (1) takes care of $\mathbf{C}^-\mathbf{C}t \leq t$. For contextuality, we need one additional fact.

Fact 4.12. If $\mathbf{C}t_1 \leq t_2$, then either $t_2 \rightarrow^* \mathbf{0}$ or $t_2 \rightarrow^* \mathbf{C}t'_2$ with $t_1 \leq t'_2$; and similarly, if $\{\dots; \mathbf{D}_i = t_{1,i}; \dots\} \leq t_2$, then either $t_2 \rightarrow^* \mathbf{0}$ or $t_2 \rightarrow^* \{\dots; \mathbf{D}_i = t_{2,i}; \dots\}$ with $t_{1,i} \leq t_{2,i}$ for all $i = 1, \dots, n$.

To prove confluence, we need to prove that each t has a single normal form. We distinguish 2 cases:

- $t \neq \mathbf{0}$. Because of fact 4.11, none of the reductions from t can involve $\mathbf{0}$. Those reduction form a confluent rewriting system (the side condition “no $t_i = \mathbf{0}$ ” is always true) which is strongly normalizing. The only critical pairs are

$$\begin{array}{ccc} \mathbf{C}^-\Omega\{\mathbf{C}'t, \theta\} & \mathbf{C}^-\Omega\{\{\mathbf{D}_1 = t_1, \dots\}, \theta\} & \mathbf{C}^-\Omega\{\Omega\{\theta'\}, \theta\} \\ \mathbf{D}\Omega\{\mathbf{C}'t, \theta\} & \mathbf{D}\Omega\{\{\mathbf{D}_1 = t_1, \dots\}, \theta\} & \mathbf{D}\Omega\{\Omega\{\theta'\}, \theta\} \\ \Omega\{\Omega\{\mathbf{C}'t, \theta\}, \theta'\} & \Omega\{\Omega\{\{\mathbf{D}_1 = t_1, \dots\}, \theta\}, \theta'\} & \Omega\{\Omega\{\Omega\{\theta'\}, \theta\}, \theta'\} \end{array}$$

and inspection readily shows that the system is locally confluent. By Newman’s lemma, it is confluent, and each $t \neq \mathbf{0}$ has a unique normal form.

- $t = \mathbf{0}$. In that case, t has one reduction to $\mathbf{0}$ (by the previous lemma) and one reduction to a non $\mathbf{0}$ term. The only reduction that allows to decrease (for the order $\leq$) a term is $\mathbf{D}_{i_0}\{\dots; \mathbf{D}_i = t_i; \dots\} \rightarrow t_{i_0}$. But in this case, the left side of the rule is equal to $\mathbf{0}$, which implies that one of the t_i is also equal to $\mathbf{0}$ by fact 4.12. This is forbidden by the side condition.

Reduction removes all destructors ($\mathbf{C}^-$ and $\mathbf{D}$) that are directly above constructors ($\mathbf{C}$ and $\{\dots\}$). It also remove any constructor directly below Ω and destructors directly above Ω . This is exactly what the given grammar does. $\square$

Because of the property $t = \mathbf{0}$ iff $t \rightarrow^* \mathbf{0}$, the easiest way to compute the normal form of a term is to reduce “bottom-up”, i.e. reduce all the t_i before reducing $\mathbf{D}_{i_0}\{\dots; \mathbf{D}_i = t_i; \dots\}$ $\blacksquare$

Unfortunately, $\mathcal{F}_0$ isn’t a sup semi-lattice, or, since $\mathbf{0}$ is a greatest element, a sup-lattice. We thus need to formally add least upper-bounds before taking the ideal completion.

Definition 4.13. $\mathcal{F}$ is obtained as the Smyth power construction (without empty sums) of the ideal completion of the free sup-lattice generated by $\mathcal{F}_0$.

The free sup-lattice $F(O)$ generated from an order $(O, \leq)$ is obtained as the set of non-empty subsets of the form $U^\downarrow$ ordered by inclusion. The fact that this is a free construction means that defining a (sup preserving) function from $F(\mathcal{F}_0)$ to another sup-lattice L is equivalent

to defining an increasing function from $\mathcal{F}_0$ to L . In particular, by proposition 4.1, to define a linear continuous function from $\mathcal{F}$ to another domain D , it is enough to define an increasing function from $\mathcal{F}_0$ to D (as is done in definitions 4.14, 4.17 and 4.30).

Since $\mathbf{0}$ is a top element, it can implicitly be added to all sums without changing their meaning. Because of that, $\mathbf{0}$ can, a posteriori, be identified with the empty sum and we don't need to add it explicitly as we did when defining $\mathcal{A}$.

The ideal completion introduces infinite elements like “ $\text{CCCC} \cdots = \bigsqcup^\uparrow \{\Omega, \text{C}\Omega, \text{CC}\Omega, \dots\}$ ” but also like “ $\cdots .\text{D}.\text{D}.\text{D}.\text{D} \mathbf{x} = \bigsqcup^\uparrow \{\Omega, \Omega.\text{D} \mathbf{x}, \Omega.\text{D}.\text{D} \mathbf{x}, \dots\}$ ”. Notice however that there is no real infinite element of the form “ $.\text{D}.\text{D}.\text{D} \cdots = \bigsqcup^\uparrow \{\Omega, .\text{D}\Omega, .\text{D}.\text{D}\Omega, \dots\}$ ” because each $.\text{D} \cdots .\text{D}\Omega$ is equal to Ω . Note also that, just like $\mathcal{V}$, $\mathcal{F}$ contains some infinite sums.

The constructions “ D ” and “ C^- ” have natural interpretations as continuous functions:

$$v \mapsto .\text{D}(v) = \begin{cases} \perp & \text{if } v = \perp \\ u & \text{if } v \text{ is of the form } \{\dots; \text{D} = u; \dots\} \\ \mathbf{0} & \text{otherwise} \end{cases}$$

and

$$v \mapsto \text{C}^-(v) = \begin{cases} \perp & \text{if } v = \perp \\ u & \text{if } v \text{ is of the form } \text{C}u \\ \mathbf{0} & \text{otherwise} \end{cases}$$

Definition 4.14. Given an environment ρ associating functions to names, and $t \neq \mathbf{0}$ an element of $\mathcal{F}_0$, we define $\llbracket t \rrbracket_\rho : \mathcal{A} \rightarrow \mathcal{A}$ with

- $\llbracket \text{C}t \rrbracket_\rho(v) = \text{C} \circ \llbracket t \rrbracket_\rho$,
- $\llbracket \{\dots; \text{D}_i = t_i; \dots\} \rrbracket_\rho(v) = v \mapsto \{\dots; \text{D}_i = \llbracket t_i \rrbracket_\rho(v); \dots\}$,
- $\llbracket \Omega\{\dots, t_i, \dots\} \rrbracket_\rho = \max_i (\Omega \circ \llbracket t_i \rrbracket_\rho)$, or more concretely
 - $\llbracket \Omega \rrbracket_\rho = \Omega$,
 - $\llbracket \Omega\{\dots, t_i, \dots\} \rrbracket_\rho(v) = \mathbf{0}$ if one of $\llbracket t_i \rrbracket_\rho(v) = \mathbf{0}$, and $\perp$ otherwise,
- $\llbracket \text{C}^- t \rrbracket_\rho = \text{C}^- \circ \llbracket t \rrbracket_\rho$,
- $\llbracket .\text{D}t \rrbracket_\rho = .\text{D} \circ \llbracket t \rrbracket_\rho$,
- $\llbracket \mathbf{x} \rrbracket_\rho = u \mapsto u$,
- $\llbracket \mathbf{0} \rrbracket_\rho = u \mapsto \mathbf{0}$,
- $\llbracket \mathbf{f}t \rrbracket_\rho = \rho(\mathbf{f}) \circ \llbracket t \rrbracket_\rho$,

We extend $\llbracket _ \rrbracket_\rho$ to the whole of $\mathcal{F}$ with

- $\llbracket \bigvee t \rrbracket_\rho = \bigvee \llbracket t \rrbracket_\rho$ (pointwise),²⁴
- $\llbracket \sum t \rrbracket_\rho = \sum \llbracket t \rrbracket_\rho$ (pointwise), and in particular, $\llbracket \mathbf{0} \rrbracket_\rho(u) = \mathbf{0}$,
- $\llbracket \bigsqcup^\uparrow T \rrbracket_\rho(u) = \bigsqcup^\uparrow \{ \llbracket t \rrbracket_\rho(u) \mid t \in T \}$ (pointwise).

Lemma 4.15.

- (1) If $t_1 \leq t_2$, then $\llbracket t_1 \rrbracket_\rho \leq \llbracket t_2 \rrbracket_\rho$.
- (2) If $\rho(\mathbf{f})$ is continuous for any $\mathbf{f}$, then $\llbracket t \rrbracket_\rho$ is also continuous.
- (3) $T \mapsto \llbracket T \rrbracket$ is continuous as a function from $\mathcal{F}$ to $[\mathcal{A} \rightarrow \mathcal{A}]$.
- (4) If $\rho(\mathbf{f}) \geq \Omega$ (in $[\mathcal{A} \rightarrow \mathcal{A}]$) for all function names, then $\llbracket T \rrbracket_\rho \geq \Omega$.
- (5) For all terms $t_1, t_2 \in \mathcal{F}_0$ and environment ρ , we have $\llbracket t_1[\mathbf{x} := t_2] \rrbracket_\rho = \llbracket t_1 \rrbracket_\rho \circ \llbracket t_2 \rrbracket_\rho$.

²⁴recall that we used the ideal completion of the free sup-lattice generated by $\mathcal{F}_0$

Sketch of proof. Checking the first points amounts to checking that all inequations from definition 4.9 hold semantically in $[\mathcal{A} \rightarrow \mathcal{A}]$. This is immediate. The functions C^- , $\mathsf{.D}$ and Ω are easily shown continuous, $\llbracket t \rrbracket_\rho$ is continuous as a composition of continuous functions. Continuity of $\llbracket _ \rrbracket$ follows from the definition of $\llbracket _ \rrbracket^\uparrow \dots$. The only thing that really needs checking is that $\llbracket _ \rrbracket$ is monotonic. This is an easy induction. Points (4) and (5) are proved by immediate induction. $\square$

Definition 4.16. We identify a special function name “ $\mathbf{f}$ ” for the recursive function being defined and we assume given an environment ρ for all the other names. Then each $T \in \mathcal{A}$ gives rise to an operator $\llbracket T \rrbracket$ from $[\mathcal{A} \rightarrow \mathcal{A}]$ to itself:

$$\llbracket T \rrbracket_\rho : f \mapsto \llbracket T \rrbracket_\rho(f) = \llbracket T \rrbracket_{\rho, \mathbf{f} := f}$$

All other functions will be called $\mathbf{g}$, $\mathbf{h}$ etc.. The typical environment ρ is given inductively, it will be omitted in the rest of the paper. A consequence of point (4) from lemma 4.15 is that if $\rho(\mathbf{g}) \geq \Omega$ for all function names, then $\llbracket T \rrbracket(\Omega) \geq \Omega$, and we can thus use the formula for the least fixed point of $\llbracket T \rrbracket_\rho$ greater than Ω .

4.3.2. Composition. Given two terms t_1 and t_2 , we want to find a term representing the composition $\llbracket t_1 \rrbracket \circ \llbracket t_2 \rrbracket$. The idea is to replace each “ $\mathbf{f} u$ ” inside t_1 by “ $t_2 u$ ”, i.e. by $t_2[\mathbf{x} := u]$.

Definition 4.17. If $t_1 \neq \mathbf{0} \in \mathcal{F}_0$ and $T_2 \in \mathcal{F}$, we define $t_1 \circ T_2$ by induction on t_1 where

- $(\mathsf{C}t_1) \circ T_2 = \mathsf{C}(t_1 \circ T_2)$,
- $\{\dots; \mathsf{D}_i = t_i; \dots\} \circ T_2 = \{\dots; \mathsf{D}_i = t_i \circ T_2; \dots\}$,
- $(\Omega\{\dots, t_i, \dots\}) \circ T_2 = \Omega\{t_i \circ T_2\}$,
- $(\mathsf{C}^-t_1) \circ T_2 = \mathsf{C}^-(t_1 \circ T_2)$,
- $(\mathsf{.D}t_1) \circ T_2 = \mathsf{.D}(t_1 \circ T_2)$,
- $\mathbf{x} \circ T_2 = \mathbf{x}$,
- $(\mathbf{f}t_1) \circ T_2 = T_2[\mathbf{x} := t_1 \circ T_2]$.

This is extended to $T_1 \in \mathcal{F}$ by commutation with $\bigvee$, $\bigsum$ and $\llbracket _ \rrbracket^\uparrow$.

Only the last case is interesting, and because of it, we sometimes use the less formal notation $T_1 \circ T_2 = T_1[\mathbf{f}t := T_2[\mathbf{x} := t]]$, or even $T_1[\mathbf{f} := T_2]$. This definition of $\circ$ generalizes definition 3.8 on page 22. Note that because the definition is by induction on t_1 , composition is automatically linear in its left argument but not necessarily on its right argument.

Lemma 4.18. For any $t_1, t_2, t_3 \in \mathcal{F}_0$, $t_1 \circ (t_2 \circ t_3) = (t_1 \circ t_2) \circ t_3$.

Proof. We first prove that $t[\mathbf{x} := t_1] \circ t_2 = (t \circ t_2)[\mathbf{x} := t_1 \circ t_2]$ by induction on t :

- if $t = \mathbf{x}$, this is immediate,
- if t starts with a constructor, record, destructor or Ω , the result follows by induction,
- if $t = \mathbf{f}t'$, we have

$$\begin{aligned} (\mathbf{f}t')[\mathbf{x} := t_1] \circ t_2 &= (\mathbf{f}t'[\mathbf{x} := t_1]) \circ t_2 \\ &= t_2[\mathbf{x} := t'[\mathbf{x} := t_1] \circ t_2] && \text{definition of } \circ \\ &= t_2[\mathbf{x} := (t' \circ t_2)[\mathbf{x} := t_1 \circ t_2]] && \text{induction} \\ &= t_2[\mathbf{x} := t' \circ t_2][\mathbf{x} := t_1 \circ t_2] && \text{substitution lemma} \\ &= (\mathbf{f}t' \circ t_2)[\mathbf{x} := t_1 \circ t_2] && \text{definition of } \circ \end{aligned}$$

We can now prove that $t_1 \circ (t_2 \circ t_3) = (t_1 \circ t_2) \circ t_3$ by induction on t_1 :

- if $t_1 = \mathbf{x}$, this is immediate,
- if t_1 starts with a constructor, record, destructor or Ω , it follows by induction,
- if $t_1 = \mathbf{f} t'_1$, we need to show that $t_2[\mathbf{x} := t_1 \circ t_2] \circ t_3 = (t_2 \circ t_3)[\mathbf{x} := t_1 \circ (t_2 \circ t_3)]$. By induction, it is enough to show that $t_2[\mathbf{x} := t_1 \circ t_2] \circ t_3 = (t_2 \circ t_3)[\mathbf{x} := (t_1 \circ t_2) \circ t_3]$. This follows from the previous lemma, with $t = t_2$, $t_2 = t_1 \circ t_2$, and $t_2 = t_3$.

□

Lemma 4.19. *For any $T_1, T_2 \in \mathcal{F}$, $\llbracket T_1 \circ T_2 \rrbracket = \llbracket T_1 \rrbracket \circ \llbracket T_2 \rrbracket$. When T_2 doesn't contain $\mathbf{f}$, we also have $\llbracket T_1 \circ T_2 \rrbracket = \llbracket T_1 \rrbracket (\llbracket T_2 \rrbracket)$. In particular,*

$$\llbracket \underbrace{T \circ \dots \circ T}_n \circ \Omega \rrbracket = \llbracket T \rrbracket^n (\Omega)$$

Proof. The first point is proved by induction. The crucial case is $(\mathbf{f} t_1) \circ t_2 = t_2[\mathbf{x} := t_1 \circ t_2]$:

$$\begin{aligned} \llbracket (\mathbf{f} t_1) \circ t_2 \rrbracket_\rho &= f \mapsto \llbracket (\mathbf{f} t_1) \circ t_2 \rrbracket_{\rho, \mathbf{f}=f} && \text{definition of } \llbracket - \rrbracket \\ &= f \mapsto \llbracket t_2[\mathbf{x} := t_1 \circ t_2] \rrbracket_{\rho, \mathbf{f}=f} && \text{definition of } \circ \\ &= f \mapsto \llbracket t_2 \rrbracket_{\rho, \mathbf{f}=f} \circ \llbracket t_1 \circ t_2 \rrbracket_{\rho, \mathbf{f}=f} && \text{point (5) of lemma 4.15} \\ &= f \mapsto \llbracket t_2 \rrbracket_{\rho, \mathbf{f}=f} \circ \llbracket t_1 \rrbracket_{\rho, \mathbf{f}=\llbracket t_2 \rrbracket_{\rho, \mathbf{f}=f}} && \text{induction} \\ &= f \mapsto \llbracket \mathbf{f} t_1 \rrbracket_{\rho, \mathbf{f}=\llbracket t_2 \rrbracket_{\rho, \mathbf{f}=f}} && \text{definition of } \llbracket - \rrbracket \\ &= \llbracket \mathbf{f} t_1 \rrbracket_\rho \circ \llbracket t_2 \rrbracket_\rho && \text{definition of } \llbracket - \rrbracket \end{aligned}$$

□

4.4. Interpreting recursive definitions. We can now replace the operator $\phi_{\mathbf{f}}$ (obtained from a recursive definition) by the interpretation of an element of $\mathcal{F}$. Consider a single clause “ $\mathbf{f} p = u$ ” of the recursive definition of $\mathbf{f}$. The pattern p allows to “extract” some parts of the argument of $\mathbf{f}$ to be used in u . We can mimics that in $\mathcal{F}$:

Definition 4.20. Given a pattern p , define the substitution σ_p (a function from variables to terms) as follows: (also in definition 3.7 on page 21)

- $\sigma_{\mathbf{y}} = [\mathbf{y} := \mathbf{x}]$,
- $\sigma_{\mathbf{C}p} = \mathbf{C}^- \circ \sigma_p$,
- $\sigma_{\{\dots; \mathbf{D}_i = p_i; \dots\}} = \bigcup_i (\cdot \mathbf{D}_i \circ \sigma_{p_i})$ (note that because patterns are linear, the union is disjoint).

where $\circ$ represents composition functions. For example, $\mathbf{C}^- \circ \sigma_p = [\dots, \mathbf{y} := \mathbf{C}^- \sigma_p(\mathbf{y}), \dots]$.

Lemma 4.21. *If $v \in \mathcal{V}$ matches p , then $\sigma_p(\mathbf{y})[x := v] \neq \mathbf{0}$ for all variables $\mathbf{y}$ in p . In that case, $\sigma_p \circ [\mathbf{x} = v]$ is the unifier for p and v .*

Proof. The proof is a simple induction on the pair pattern / value. (Refer to definition of the matching unifier on page 28). □

An example should make that clearer. Consider the pattern **Cons{Fst=n; Snd=1}** (simply “**n::1**” in Caml), we get the substitution

$$\left[\mathbf{n} := \cdot \mathbf{Fst} \text{ Cons}^- \mathbf{x}, \quad \mathbf{1} := \cdot \mathbf{Snd} \text{ Cons}^- \mathbf{x} \right]$$

which precisely describes how to get **n** and **1** from a value $(\mathbf{x})$.

A part of a pattern that doesn't contain variables isn't recorded in σ_p . For example, the pattern **Cons{Fst=Zero; Snd=1}** would only give the substitution $[\mathbf{1} := \cdot \mathbf{Snd} \text{ Cons}^- \mathbf{x}]$ and would thus match any non-empty list.

Definition 4.22. Given a recursive definition of $\mathbf{f}$, define $T_{\mathbf{f}}$ with

$$T_{\mathbf{f}} = \sum_{\mathbf{f} \ p = u} u[\sigma_p]$$

where the sum ranges over all clauses defining $\mathbf{f}$.

Each summand of this sum is a simple term (a non $\mathbf{0}$ element of $\mathcal{F}_0$) and we interpret in this way any recursive definition as a compact element of $\mathcal{F}$. The previous lemma makes it easy to show

Corollary 4.23. *If $T_{\mathbf{f}}$ is defined as above for a definition of $\mathbf{f}$, we have $\llbracket T_{\mathbf{f}} \rrbracket_{\rho} \leq \phi_{\mathbf{f}}$, where the environment ρ associates to each previously defined function, its interpretation (lifted to $\mathcal{A} \rightarrow \mathcal{A}$).*

One reason $T_{\mathbf{f}}$ is generally smaller than $\phi_{\mathbf{f}}$ is that patterns may overlap. In that case, $\phi_{\mathbf{f}}$ only uses the first matching clause while $T_{\mathbf{f}}$ sums over all matching clauses. The other reason is that we forget about the patterns without variables (like **Zero** above). By lemma 4.8, totality of $\text{fix}(\llbracket T_{\mathbf{f}} \rrbracket)$ implies totality of $\text{fix}(\phi_{\mathbf{f}})$. Note that nothing in this corollary requires ρ to be the interpretation of previously defined functions, we only need to interpret each $\mathbf{g}$ by a total function! Because of lemma 4.19, we have

Lemma 4.24. *To check that $\text{fix}(\phi_{\mathbf{f}})$ is total, it is enough to check that*

$$\bigsqcup_n \uparrow \left[\underbrace{T_{\mathbf{f}} \circ \dots \circ T_{\mathbf{f}}}_n \circ \Omega \right]$$

is total.

From now on, we will omit the semantics brackets and write $T_{\mathbf{f}}$ for $\llbracket T_{\mathbf{f}} \rrbracket_{\rho}$. We will also write $T_{\mathbf{f}}^n(\Omega)$ for $T_{\mathbf{f}} \circ \dots \circ T_{\mathbf{f}} \circ \Omega$ and the notation “ $\text{fix}(T_n)$ ” will refer to $\bigsqcup_n \uparrow T_{\mathbf{f}} \circ \dots \circ T_{\mathbf{f}} \circ \Omega$.

4.5. Removing finite path. During evaluation, or when using the formula for $\text{fix}(T)$, some computations are finite. Since the size-change principle only deals with infinite path, such finite computations are ignored. In simple cases, this is justified by the following.

Lemma 4.25. *Suppose $T_{\mathbf{f}}$ doesn't contain Ω and is of the form $T_1 + T_0$, where all summands of T_1 contains a single occurrence of $\mathbf{f}$, and all summands of T_0 contain no occurrence of $\mathbf{f}$.²⁵ Then $\text{fix}(T_1 + T_0)$ is total iff $\text{fix}(T_1)$ is total.*

Sketch of proof. We have $T_1 + T_0 \leq T_1$, which proves the left to right implication. For the other direction, note that $T_1(t + t') = T_1 \circ (t + t')$ by lemma 4.19. Since each summand of T_1 contains a single occurrence of $\mathbf{f}$, its semantics is linear: we have $T_1 \circ (t + t') = T_1(t) + T_1(t')$. This implies that $(T_1 + T_0)^2(t) = T_1^2(t) + T_1(T_0) + T_0$, and more generally:

$$(T_1 + T_0)^n(t) = \sum_{i=0}^{n-1} T_1^i(T_0) + T_1^n(t)$$

²⁵Many everyday recursive function are of this form. The first real counter example that comes to mind is the Ackermann function.

We now have

$$\begin{aligned}
\text{fix}(T_1 + T_0) &= \bigsqcup_{n \geq 0} \uparrow (T_1 + T_0)^n(\Omega) \\
&= \bigsqcup_{n \geq 0} \uparrow \left(\sum_{i=0}^{n-1} T_1^i(T_0) + T_1^n(\Omega) \right) \\
&= \sum_{n > 0} T_1^n(T_0) + \bigsqcup_{n \geq 0} \uparrow T_1^n(\Omega) \quad (*) \\
&= \sum_{n > 0} T_1^n(T_0) + \text{fix}(T_1) \quad (1)
\end{aligned}$$

where $(*)$ follows from basic properties of unions $(+)$ and intersections $(\sqcup^\uparrow)$. Each $T_1^n(T_0)$ is a finite composition of total functionals and is thus total. Since $\text{fix}(T_1)$ is total by hypothesis, we can conclude. $\square$

When T_1 contains nested calls, the previous lemma is false! The following ad-hoc definition provides a counter example:

```

val f : prod(nat, nat) -> nat
  | f {Fst=0; Snd=s} = s
  | f {Fst=n+1; Snd=s} = f {Fst=n; Snd=f {Fst=n+1; Snd=s}}

```

This definition yields a non total T_f : it is undefined whenever the first projection of its input is strictly greater than 0. However, removing the base case, or replacing it with

```

  | f {Snd=0; Snd=s} = 3

```

yields a total function, at least in call-by-name.²⁶ To derive a general formula analogous to (1), we start by indexing the occurrences of $\mathbf{f}$ in T : if T is $\mathbf{ffx}$, we write $\mathbf{f}_1\mathbf{f}_2\mathbf{x}$. Substituting $\mathbf{f}$ by $\mathbf{g} + \mathbf{h}$ gives $\mathbf{ggx} + \mathbf{ghx} + \mathbf{hgx} + \mathbf{hxx}$, i.e. each occurrence of $\mathbf{f}$ is substituted either by $\mathbf{g}$ or $\mathbf{h}$. Since substituting a single occurrence of $\mathbf{f}$ is linear, substituting all of them is multilinear.

Lemma 4.26. *We have*

$$T \circ (t_1 + \dots + t_n) = \sum_{\sigma : \text{occ}(\mathbf{f}, T) \rightarrow \{t_1, \dots, t_n\}} T[\sigma]$$

where $\text{occ}(\mathbf{f}, T)$ represents the set of occurrences of $\mathbf{f}$ in T , and the substitution occurs at the given occurrences. More precisely, $T[\sigma] = T[\mathbf{f}_i t := \sigma(\mathbf{f}_i)[\mathbf{x} := t]]$ as in definition 4.17.

In particular, if $T = \sum_i t_i$ is a sum of simple terms, then T^n is a sum of simple terms obtained in the following way:

- start with a simple term t_{i_0} ,
- replace each occurrence of $\mathbf{f}$ by one of the t_i ,
- repeat $n - 2$ times.

We now extend this to infinite compositions.

Definition 4.27. Given $T = t_1 + \dots + t_n$ a sum of simple terms, a *path* for T is a sequence $(s_k, \sigma_k)_{k \geq 0}$ such that:

²⁶Our semantics corresponds to call-by-name, but the fact that this definition is total can also be checked using the formula for $\text{fix}(T_f)$, i.e. without referring to the operational semantics of the language.

- $s_0 = \mathbf{f} \mathbf{x}$,
- $s_{k+1} = s_k[\sigma_k]$ where σ_k replaces occurrences of $\mathbf{f}$ inside s_k by $t_1, \dots$, or t_n .

If some s_k doesn't contain any occurrence of $\mathbf{f}$, then all latter s_{k+i} are equal to s_k . We call such a path *finite*.

We usually don't write the substitution and talk about the path (s_k) . Note that s_1 is just one of the summands of T . We can now state and prove a general version of the formula (1).

Lemma 4.28. *Suppose $T = t_1 + \dots + t_n$ is a sum of simple terms, then*

$$\text{fix}(T) = \sum_{s \text{ path of } T} \bigsqcup_{i \geq 0}^\uparrow s_i(\Omega)$$

Proof. Let s be a simple term in $\text{fix}(T) = \bigsqcup^\uparrow T^n(\Omega)$. We want to show that s is greater than some $\bigsqcup_{i \geq 0}^\uparrow s_i(\Omega)$. For each i , $T^i(\Omega)$ is a finite sum of elements of $\mathcal{F}_0$. Define the following forest:

- nodes of depth i are those summands t in T^i satisfying $t(\Omega) \leq s$,
- a node s at depth i is related to a node s' at depth $i+1$ if $s' = s[\sigma]$, where σ substitute all occurrences of $\mathbf{f}$ in s by one of $t_1, \dots, t_n$.

This forest is finitely branching (there are only finitely many possible substitutions from a given node) and infinite (because $T^n(\Omega) \leq \text{fix}(T) \leq s$, each $T^n(\Omega)$ contains some term t such that $t(\Omega) \leq s$). It thus contains an infinite branch (König's lemma) $s_0, s_1, \dots$. This sequence satisfies all the properties of definition 4.27 and its limit is less than s (because all $s_i(\Omega)$ are less than s by construction). We thus have

$$\bigsqcup^\uparrow T^n(\Omega) \geq \sum_{s \text{ path of } T} \bigsqcup_{i \geq 0}^\uparrow s_i(\Omega)$$

For the converse, it is enough to show that for each path (s_k) and natural number n , the limit of $s_k(\Omega)$ is greater than $T^n(\Omega)$. This is immediate because each $s_k(\Omega)$ is a summand of $T^k(\Omega)$. $\square$

Corollary 4.29. *If T doesn't contain Ω and $\text{fix}(T)$ is non-total, then there is an infinite path (s_k) for T such that $\bigsqcup^\uparrow s_i(\Omega)$ is non total.*

Proof. If $\text{fix}(T)$ is non total, then by the previous lemma, there is a path of T that is non total. Since every finite path is total as a finite composition of total operators, there necessarily exists a non total infinite path. $\square$

4.6. The Call-Graph. In an infinite non total path (s_k) , each s_k must contain at least one occurrence of $\mathbf{f}$ that leads to non totality. The call-graph allows to keep that occurrence alive. All other occurrences are either removed or replaced by Ω .

Definition 4.30. Let $T = t_1 + \dots + t_n$ be a sum of simple terms that do not contain Ω . Define $G(T) = \sum_{\beta \mathbf{f} t \in T} \beta^\Omega \mathbf{f} u^\Omega$ where

- β is a branch (i.e. a term where records have a single field),
- $\beta \mathbf{f} t \in T$ means that T contains the subterm $\mathbf{f} t$ at position β ,
- β^Ω is equal to β where all function names ($\mathbf{f}$, $\mathbf{g}$, $\mathbf{h}$, etc.) have been replaced by Ω , and similarly for t^Ω .

A more pedestrian inductive definition is

- (1) $G(\mathbf{f} t) = \mathbf{f} t$ if $\mathbf{f}$ doesn't occur in t ,
- (2) $G(\mathbf{f} t) = \mathbf{f}(t^\Omega) + \Omega G(t)$ if $\mathbf{f}$ occurs in t , (note that there is no recursive call to G in the left summand)
- (3) $G(\mathbf{g} t) = \Omega G(t)$,
- (4) $G(\mathbf{x}) = \mathbf{x}$,
- (5) $G(\mathbf{C} t) = \mathbf{C} G(t)$,
- (6) $G(\{\dots; \mathbf{D}_i = t_i; \dots\}) = \sum_i \{\mathbf{D}_i = G(t_i)\}$,
- (7) $G(\mathbf{C}^- t) = \mathbf{C}^- G(t)$,
- (8) $G(\mathbf{D} t) = \mathbf{D} G(t)$,

extended by commutation with $\bigvee$, $\sum$ and $\sqcup^\uparrow$.

In particular, $G(t) = \mathbf{0}$ whenever t doesn't depend on $\mathbf{f}$. In general, T and $G(T)$ are not comparable. However, we have

Proposition 4.31. *If $\text{fix}(G(T))$ is total then so is $\text{fix}(T)$.*

Proof. Suppose $\text{fix}(T)$ is non-total. By corollary 4.29, it implies there is a path (s_k) and an element $u \in \mathcal{V}$ with $\sqcup^\uparrow s_i(\Omega)(u) \in \mathcal{V}$ non-total, i.e. contains a non total branch (either a finite branch ending with $\perp$, or an infinite branch with odd principal priority). In particular, no $s_i(\Omega)(u)$ is equal to $\mathbf{0}$. Denote this branch with β .

We index occurrences of $\mathbf{f}$ in $T_{\mathbf{f}}$ by a natural number and extend that indexing to occurrences of $\mathbf{f}$ in (s_k) . The occurrences of $\mathbf{f}$ in s_k are indexed by lists of length k :

- (1) the only occurrence of $\mathbf{f}$ in $s_0 = \mathbf{f} \mathbf{x}$ is indexed by the empty list
- (2) occurrences of $\mathbf{f}$ in s_1 are indexed using the list with a single number corresponding to the index of this occurrence in $T_{\mathbf{f}}$.
- (3) given $k > 1$, suppose the original substitution σ_k replaces occurrence $\mathbf{f}_i$ in s_k by some summand t of $T_{\mathbf{f}}$. The new substitution replaces occurrence $\mathbf{f}_{L,i}$ by the same t , but where each $\mathbf{f}_j$ is instead indexed by $\mathbf{f}_{L,i,j}$.

An example should make this clearer: suppose $\mathbf{f}_{3,5} t$ occurs in s_2 , corresponding initially to $\mathbf{f}_5 t$ in s_2 . If this occurrence was originally replaced by $\sigma_2(\mathbf{f}_5) = \mathbf{f}_4 \mathbf{C} \mathbf{f}_7 \mathbf{x}$, we now replace it with $\sigma_2(\mathbf{f}_{3,5}) = \mathbf{f}_{3,5,4} \mathbf{C} \mathbf{f}_{3,5,7} \mathbf{x}$. The new s_3 will thus contain $\mathbf{f}_{3,5,4} \mathbf{C} \mathbf{f}_{3,5,7} u$. These lists record a kind of genealogy of each occurrence of $\mathbf{f}$ by keeping track of which previous occurrences introduced it.

An occurrence $\mathbf{f}_L \in s_k$ is called *non-total* if the path

$$s'_0 = \mathbf{f}_L \mathbf{x}, s'_1 = \sigma_k(\mathbf{f}_L), s'_2 = s'_1[\sigma_{k+1}], \dots$$

is non-total. In other words, an occurrence is non-total if it “converges to a non-total term”.

We now construct a path (s'_k) of $G(T)$: suppose we have constructed $\sigma'_0, \dots, \sigma'_{k-1}$ so that each s'_i is of the form $\beta_{\upharpoonright i} \gamma_i^\Omega \mathbf{f}_L t_i^\Omega$ where

- $\beta_{\upharpoonright i} \gamma_i \mathbf{f}_L t_i$ is a subterm of s_i , with $\mathbf{f}_L$ being non-total,
- $\beta_{\upharpoonright i}$ is a prefix of β (it contains only $\mathbf{C}$ and $\{\mathbf{D} = _ \}$),
- γ_i is either empty or starts with a function name ($\mathbf{f}$ or some $\mathbf{g}$) and only contains unary records.

Since s'_k contains a single occurrence of $\mathbf{f}$, the substitution σ'_k only need to act on $\mathbf{f}_L$. Suppose $\mathbf{f}_L$ was replaced (by σ_k) by the summand t of T . Since $\mathbf{f}_L$ was chosen non-total, t necessarily contains non-total occurrences $\mathbf{f}_{L,i}$.

- If γ_k was non empty, we replace $\mathbf{f}_L$ by any $\gamma'^\Omega \mathbf{f}_{L,i} t'^\Omega$ where $\gamma' \mathbf{f}_{L,i} t'$ corresponds to a non-total occurrence of $\mathbf{f}$ in t . This is indeed a summand of $G(T)$, and s'_{k+1} is equal to

$$\begin{aligned} (\beta_{|k} \gamma_k^\Omega \mathbf{f}_L t_k^\Omega) [\mathbf{f} := \gamma'^\Omega \mathbf{f}_{L,i} t'^\Omega] &= \beta_{|k} \gamma_k^\Omega \gamma'^\Omega \mathbf{f}_{L,i} t'^\Omega [\mathbf{x} := t_k^\Omega] \\ &= \beta_{|k} (\gamma_k \gamma')^\Omega \mathbf{f}_{L,i} (t'[\mathbf{x} := t_k])^\Omega \end{aligned}$$

- If γ_k was empty, the summand t starts with part of the branch β : there is a sub-term $\beta_{|k,k+1} \gamma' \mathbf{f}_{L,i} t'$ of t such that
 - $\beta_{|k} \beta_{|k,k+1}$ is a prefix of β ,
 - γ' is either empty or starts with a function name,
 - $\mathbf{f}_{L,i}$ is a non-total occurrence.

In that case, we replace $\mathbf{f}_L$ by $\beta_{|k,k+1} \gamma'^\Omega \mathbf{f}_{L,i} t'^\Omega$, which is indeed a summand in $G(T)$. The term s'_{k+1} is then equal to

$$\begin{aligned} (\beta_{|k} \mathbf{f}_L t_k^\Omega) [\mathbf{f} := \beta_{|k,k+1} \gamma'^\Omega \mathbf{f}_{L,i} t'^\Omega] &= \beta_{|k} \beta_{|k,k+1} \gamma'^\Omega \mathbf{f}_{L,i} t'^\Omega [\mathbf{x} := t_k^\Omega] \\ &= \beta_{|k} \beta_{|k,k+1} \gamma'^\Omega \mathbf{f}_{L,i} (t'[\mathbf{x} := t_k])^\Omega \end{aligned}$$

In both cases, the resulting s'_{k+1} as a shape compatible with the invariant given above.

Fact 4.32. This path (s'_k) of $G(T)$ is non-total.

We started by supposing that $\bigsqcup^\uparrow s_i(\Omega)(u)$ is a non-total element of $\mathcal{V}$ containing the non-total branch β . In particular, it implied that no $s_i(\Omega)(u)$ was equal to $\mathbf{0}$. The limit $\bigsqcup^\uparrow s'_k(\Omega)(u)$ is thus a limit of the form $\bigsqcup^\uparrow \beta_k \gamma_k \Omega t(u)$ where $t(u)$ is never equal to $\mathbf{0}$. This means that $\Omega t(u) = \perp$. Since γ starts with a Ω (or is empty), the limit is of the form $\bigsqcup^\uparrow \beta_k \perp$. Because β is non-total, this limit is also non-total. $\square$

Note that this is only a soundness result and doesn't say anything about the strength of reducing totality for T to totality for $G(T)$. (The same holds trivially of $G'(T) = \Omega$, which is never total!) The only argument presented in this paper about that is of a practical nature: experimenting with **chariot** shows that $G(T)$ is quite powerful. General results like “all structurally recursive definitions are total” or “all syntactically guarded definitions are total” are certainly provable, but they are left as an exercise to the reader.

Note that since summands in $G(T_{\mathbf{f}})$ contain exactly one occurrence of $\mathbf{f}$, an infinite path for $G(t)$, usually given by a sequence of substitutions (σ_n) is simply given by a sequence (t_n) of summands of $G(T_{\mathbf{f}})$. Such a path is total if $\bigsqcup^\uparrow (t_1 \circ \dots \circ t_n)(\Omega)$ is total.

4.7. Weights. Before proving correctness of the size-change principle for recursive definitions, we need to define the terms $\langle W_1 \rangle t_1 * \dots * \langle W_n \rangle t_n$ that were used for collapsing in section 3.4.

Definition 4.33. We write Δ for an element of $\mathcal{F}$ inductively built from

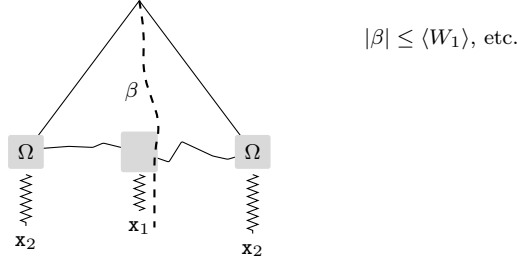
$$\begin{array}{lll} \Delta & ::= & \mathbf{C}^p \Delta \mid \{ \mathbf{D}_1 = \Delta_1; \dots; \mathbf{D}_k = \Delta_k; \dots \}^p \mid \delta \mid \Omega \delta \\ \delta & ::= & \mathbf{x}_i \mid \mathbf{C}^{p^-} \delta \mid . \mathbf{D}^p \delta \end{array}$$

where $k > 0$. Given a branch leading to one $\mathbf{x}_i$, its weight is an element of $\mathbb{W}$ defined inductively:

$$\begin{aligned} |\mathcal{C}^p \beta| &= \langle \kappa_p \rangle + |\beta| \\ |\{\mathcal{D} = \beta\}^p| &= \langle \kappa_p \rangle + |\beta| \\ |\mathbf{x}_i| &= \langle \rangle \\ |\Omega \beta| &= \Omega \\ |\mathcal{C}^{p^-} \beta| &= \langle -\kappa_p \rangle + |\beta| \\ |.\mathcal{D}^p \beta| &= \langle -\kappa_p \rangle + |\beta| \end{aligned}$$

Given some $W_i \in \mathbb{W}$, we put $\langle W_1 \rangle^\downarrow \mathbf{x}_1 * \dots * \langle W_n \rangle^\downarrow \mathbf{x}_n = \sum \Delta$ where for each Δ , the branches leading to $\mathbf{x}_i$ have weight less than $\langle W_i \rangle$. Given $t_1, \dots, t_n$ in $\mathcal{F}$, we write $\langle W_1 \rangle^\downarrow t_1 * \dots * \langle W_n \rangle^\downarrow t_n$ for the corresponding $\sum \Delta[\mathbf{x}_1 := t_1, \dots]$.

A summand of $\langle W_1 \rangle^\downarrow \mathbf{x}_1 * \dots * \langle W_n \rangle^\downarrow \mathbf{x}_n$ looks like:



Lemma 4.34. *The infinite sum from the previous definitions is finitely generated.*

Sketch of proof. This relies on the fact that there only are finitely many constructor and destructor names: given some weights $W_j \in \mathbb{W}$, write Ξ for $\langle W_1 \rangle^\downarrow \mathbf{x}_1 * \dots * \langle W_n \rangle^\downarrow \mathbf{x}_n$. We want to show that Ξ can be obtained as the limit of a chain of finite sums of elements of $\mathcal{F}_0$. Given $d \in \mathbb{N}$, define $\Xi_{\upharpoonright d} \subset \mathcal{F}_0$ as the set of all those Δ obtained as collapses of elements of Ξ of “syntactical depth” d . Collapsing an element Δ is done by introducing some Ω inside Δ and normalizing. For example, $\text{Succ Succ } \Omega \text{ Succ } \mathbf{x}$ has several collapses at depth 3:

- $\text{Succ Succ } \Omega \text{ Succ } \mathbf{x}$,
- $\text{Succ } \Omega \text{ Succ } \mathbf{x}$.

Because there are only finitely many different constructors and destructors, each one of the set $\Xi_{\upharpoonright d}$ is finite. Moreover, Ξ is the limit of the chain

$$\Xi_{\upharpoonright 1} \leq \Xi_{\upharpoonright 2} \leq \dots$$

Indeed, each element of $\Xi_{\upharpoonright d+1}$ is either in $\Xi_{\upharpoonright d}$ (when its syntactical depth is less than d), or greater than an element of $(\langle W_1 \rangle^\downarrow \mathbf{x}_1 * \dots * \langle W_n \rangle^\downarrow \mathbf{x}_n)_i$ (when its syntactical depth is strictly greater than d). This shows that $\langle W_1 \rangle^\downarrow \mathbf{x}_1 * \dots * \langle W_n \rangle^\downarrow \mathbf{x}_n$ is a limit of compact elements of $\mathcal{F}$. $\square$

Those new terms are “compatible” with the reduction of generalized patterns containing approximations (c.f. definition 3.4 in the previous section).

Lemma 4.35. *We have, where each P denotes a product $\langle W_1 \rangle^\downarrow s_1 * \dots * \langle W_n \rangle^\downarrow s_n$:*

- (1) *distribution:* $\langle W \rangle^\downarrow (t_1 + t_2) * P = (\langle W \rangle^\downarrow t_1 * P) + (\langle W \rangle^\downarrow t_2 * P)$,
- (2) $\langle W \rangle^\downarrow \mathbf{0} * P = P$,

- (3) $\langle W \rangle^\downarrow \mathbf{0} = \mathbf{0}$ for a unary product,
- (4) $\langle W \rangle^\downarrow \mathcal{C}^p t * P = \langle W + \kappa_p \rangle^\downarrow t * P$,
- (5) $\langle W \rangle^\downarrow \{\dots; \mathcal{D}_i = t_i; \dots\}^p * P = \prod_i \langle W_i + \kappa_p \rangle^\downarrow t_i * P$ if the record is not empty,
- (6) $\mathcal{C}^{p-} \prod_i \langle W_i \rangle^\downarrow t_i = \prod_i \langle W - \kappa_p \rangle^\downarrow t_i$,
- (7) $\mathcal{D}^p \prod_i \langle W_i \rangle^\downarrow t_i = \prod_i \langle W - \kappa_p \rangle^\downarrow t_i$,
- (8) $(\langle V \rangle^\downarrow (\prod_i \langle W_i \rangle^\downarrow t_i)) * P = (\prod_i \langle V + W_i \rangle^\downarrow t_i) * P$.

We also have:

- $\langle 0 \rangle^\downarrow(t) \leq t$,
- if $W \leq W'$ in $\mathbb{W}$, then $\langle W' \rangle^\downarrow t \leq u \langle W \rangle^\downarrow t$.

Proof. We only look at one case, the other being treated similarly.

- (4) Suppose $\Delta[\mathbf{x} := \mathcal{C}^p t_0, \dots]$ is a summand in $\langle W \rangle^\downarrow \mathcal{C}^p t_0 * P$. We construct Δ' by replacing each occurrence of $\mathbf{x}_0$ in Δ by $\mathcal{C}^p \mathbf{x}_0$. We have that $\Delta'[\mathbf{x}_0 := t_0, \dots] = \Delta[\mathbf{x} := \mathcal{C}^p t_0, \dots]$ is a summand in $\langle W + \kappa_p \rangle^\downarrow t_0 * P$. This shows that $\langle W \rangle^\downarrow \mathcal{C}^p t_0 * P \geq \langle W + \kappa_p \rangle^\downarrow t_0 * P$.

For the converse, let $\Delta[\mathbf{x} := t_0, \dots]$ be a summand in $\langle W + \kappa_p \rangle^\downarrow t_0 * P$. We construct Δ' by replacing each $\mathbf{x}_0$ by $\mathcal{C}^{p-} \mathbf{x}_0$. The term $\Delta'[\mathbf{x}_0 := \mathcal{C}^p t_0, \dots] \geq \Delta[\mathbf{x} := t_0, \dots]$ is then a summand in $\langle W \rangle^\downarrow \mathcal{C}^p t_0 * P$.

□

Formally speaking, we also need to introduce terms $\langle W_1 \rangle^\uparrow t_1 * \dots * \langle W_n \rangle^\uparrow t_n$ to approximate the output. This is done in exactly the same way, except we start from $\mathbb{Z} \cup \{-\infty\}$ and the order on weights (definition 3.1 on page 19) satisfies

- $\langle w_1 \rangle^{p\uparrow} \leq \langle w_2 \rangle^{p\uparrow}$ whenever $w_1 \leq w_2$ in $\mathbb{Z}_\infty$

In other words, while $\langle 1 \rangle^\downarrow$ removes at least one constructor (good one an argument), $\langle 1 \rangle^\uparrow$ adds at least one constructor (good on top of a recursive call). We then have the same properties (lemma 4.35).

4.8. The Size-Change Principle. Putting everything together (proposition 4.31, corollary 4.29 and lemma 4.24), we get

Corollary 4.36. *If all infinite path in $G(T_{\mathbf{f}})$ are total, then $\llbracket \mathbf{f} \rrbracket_\rho$ is total for every total environment ρ .*

This is where the size-change principle comes into play. We first prove a variant of combinatorial lemma at the heart of the size-change principle.

Lemma 4.37. *Suppose $(O, \leq)$ is a partial order, and $F \subseteq O$ is a finite subset. Suppose moreover that $\circ$ is a binary, associative and monotonic operation on $O \times O$ and that $\diamond$ is a binary, monotonic operation on $F \times F$ satisfying*

$$\forall o_1, o_2 \in F, (o_1 \diamond o_2) \leq (o_1 \circ o_2)$$

then every infinite sequence $o_1, o_2, \dots$ of elements of F can be subdivided into

$$\underbrace{o_1, \dots, o_{n_0-1}}_{\text{initial prefix}}, \underbrace{o_{n_0}, \dots, o_{n_1-1}}_r, \underbrace{o_{n_1}, \dots, o_{n_2-1}}_r, \dots$$

where:

- all the $(\dots (o_{n_k} \diamond o_{n_{k+1}}) \diamond \dots) \diamond o_{n_{k+1}-1}$ are equal to the same $r \in F$,
- r is coherent: there is some $o \in O$ such that $r, (r \diamond r) \leq o$.

In particular,

$$\left(o_{n_0} \circ \cdots \circ o_{n_1-1} \circ o_{n_1} \circ \cdots \circ o_{n_2-1} \circ \cdots \circ o_{n_{k-1}} \circ \cdots \circ o_{n_k-1} \right) \geq \underbrace{o \circ o \circ \cdots \circ o}_{k \text{ times}}$$

Proof. This is a consequence of the infinite Ramsey theorem. Let $(o_n)_{n \geq 0}$ be an infinite sequence of elements of F . We associate a “color” $c(m, n)$ to each pair (m, n) of natural numbers where $m < n$:

$$c(m, n) \stackrel{\text{def}}{=} (\dots(o_m \diamond o_{m+1}) \diamond \cdots) \diamond o_{n-1}$$

Since F is finite, the number of possible colors is finite. By the infinite Ramsey theorem, there is an infinite set $I \subseteq \mathbb{N}$ such all the (i, j) for $i < j \in I$ have the same color $o \in F$. Write $I = \{n_0 < n_1 < \cdots < n_k < \cdots\}$. If $i < j < k \in I$, we have:

$$\begin{aligned} o &= (\dots(o_i \diamond o_{i+1}) \diamond \cdots) \diamond o_{j-1} \\ &= (\dots(o_j \diamond o_{j+1}) \diamond \cdots) \diamond o_{k-1} \\ &= (\dots((\dots(o_i \diamond o_{i+1}) \diamond \cdots) \diamond o_j) \diamond \cdots) \diamond o_{k-1} \end{aligned}$$

The first two equalities imply that

$$o \diamond o = ((\dots(o_i \diamond o_{i+1}) \diamond \cdots) \diamond o_{j-1}) \diamond ((\dots(o_j \diamond o_{j+1}) \diamond \cdots) \diamond o_{k-1})$$

If $\diamond$ is associative, this implies that $o \diamond o = o$. If not, we only get that both o and $o \diamond o$ are smaller than

$$o_i \circ \cdots \circ o_{j-1} \circ o_j \circ \cdots \circ o_{k-1}$$

□

We can now mimic what was done in section 3:

- (1) terms with weights are elements of $\mathcal{F}$ (definition 4.33) and combine just like in section 3 (lemma 4.35)
- (2) collapsing decreases information, almost by definition (collapsing weights decreases the weight, and collapsing depth inserts $\langle 0 \rangle$ at some strategic places)

There is one subtlety when collapsing the output: terms in $G(T_f)$ are of the form $\beta \mathbf{f} t$ where $\mathbf{f}$ doesn't appear in u . We want to have a lower bound on the number of constructors in β , and an upper bound on the number of constructors in u (or equivalently, a lower bound on the number of destructors). For that reason, collapsing on the u part is done (as in section 3) by introducing $\langle W \rangle^\downarrow$, while collapsing on the β part is done by introducing $\langle W \rangle^\uparrow$. The implementation used $\langle -W \rangle^\downarrow$ instead, which behaves similarly.

With all that, we can define the transitive closure G^* of G as in the previous section.²⁷ By construction, it satisfies:

Lemma 4.38. *For every finite sequence $s_0, s_1=s_0[\mathbf{f}:=t_0], \dots, s_k=s_{k-1}[\mathbf{f}:=t_{k-1}]$ in G , we have $t \leq t_0 \circ \cdots \circ t_{k-1}$ for some simple term t in G^* .*

We can now state and prove correctness of the size-change principle here in the case of a single function. (The definition of p -norm for a branch is given on page 25.)

Theorem 4.39 (size-change principle). *Suppose every simple term $t = \beta \mathbf{f} u$ in G^* that satisfies $t \supset t \diamond t$ also satisfies one of the following two conditions:*

- either the maximal priority p appearing in β is even and positive ($|\beta|_p > 0$)

²⁷Note that G^* is not an element of $\mathcal{F}_0$ as it may contain weights, which are not compact in $\mathcal{F}$.

- or there is a subterm $\beta \prod \langle W_i \rangle^\downarrow \lambda_i$ of v where the maximal priority p of each $\beta \langle W_i \rangle^\downarrow \lambda_i$ is odd, with negative weight (i.e. $|\beta \langle W_i \rangle^\downarrow \lambda_i|_p < 0$)

then $\text{fix}(G)$ is total.

The proof is nothing more than a rephrasing of the intuition given on page 25.

Proof. By lemma 4.28, we only need to check that infinite paths are total. Let (s_k) be an infinite path of G . By the lemma 4.37, we know that such a path can be decomposed into

$$s_0 \quad \dots \quad s_{n_0} = s_0[t_0 \circ \dots \circ t_{n_0-1}] \quad \dots \quad s_{n_1} = s_{n_0}[t_{n_0} \circ \dots \circ t_{n_1-1}] \quad \dots \quad s_{n_2} \quad \dots$$

where:

- all the $t_{n_{k+1}-1} \diamond \dots \diamond t_{n_k}$ are equal to the same t ,
- t is *coherent*: $t \diamond t \subset t$.

Suppose that t satisfies the first condition. If we write init for $t_0 \circ \dots \circ t_{n_0-1}$, we have

$$\begin{aligned} \bigsqcup_k^\uparrow s_k(\Omega) &= \bigsqcup_k^\uparrow t_0 \circ t_1 \circ \dots \circ t_k(\Omega) \\ &\geq \bigsqcup_j^\uparrow t_0 \circ \dots \circ t_{n_0-1} \circ t^j(\Omega) \\ &= \bigsqcup_j^\uparrow \text{init} \circ \beta^j \mathbf{f}u(\Omega) \\ &\geq \text{init} \circ \bigsqcup_j^\uparrow \beta^j \Omega \end{aligned}$$

Now, for any simple value v , $\beta^k \Omega(v)$ is either $\mathbf{0}$ or has at least k constructors of priority $p = 2q$ coming from β^k above any constructor coming from v . At the limit, there will be infinitely many constructors of priority $p = 2q$, all coming from β . Because β doesn't involve constructors of priority greater than $p = 2q$, the limit will be total.

Similarly, if t satisfies the second condition. We have

$$\begin{aligned} \bigsqcup_k^\uparrow s_k(\Omega) &= \bigsqcup_k^\uparrow t_0 \circ t_1 \circ \dots \circ t_k(\Omega) \\ &\geq \bigsqcup_j^\uparrow \text{init} \circ t^j(\Omega) \\ &\geq \text{init} \circ \bigsqcup_j^\uparrow \Omega u^j \end{aligned}$$

By hypothesis, $u^k = u[\mathbf{x} := u^{k-1}]$ contains a subterm $\beta \prod \langle W_i \rangle^\downarrow \lambda_i$ with $|\beta \langle W_i \rangle^\downarrow \lambda_i|_p < 0$ for all i , p being the maximal priority appearing in $\beta \prod \langle W_i \rangle^\downarrow \lambda_i$. Since u contains approximations, it is in fact an infinite sum of elements of $\mathcal{F}_0$. By definition of approximations, each summand of u^k necessarily has a branch of the form

$$\beta \beta_{i_1} \lambda_{i_1} \beta \beta_{i_2} \lambda_{i_2} \dots \beta \beta_{i_k} \lambda_{i_k}$$

where, by hypothesis, each $|\beta \beta_{i_j} \lambda_{i_j}|_p < 0$. Such a branch globally removes at least k constructors of priority $p = 2q + 1$ and doesn't involve greater priorities. If v is a total value, then each $u^k(v)$ can only be non- $\mathbf{0}$ if v contains at least k constructors of priority $p = 2q + 1$ and no constructors of greater priority. At the limit, the only values such that $\bigsqcup_k^\uparrow u^k(v)$ are non- $\mathbf{0}$ are values that contain a branch with an infinite number of constructors of

priority $p = 2q + 1$ and no constructor of priority greater than p . This is impossible for total values! $\square$

4.9. Back to the Previous Section. We can now recast all of section 3:

- generalized patterns (definition 3.2) are just elements of $\mathcal{F}$ incorporating weights (definition 4.33),
- generalized patterns decrease along reduction (definition 3.4) because it consists of reductions in $\mathcal{F}$ and equalities in $\mathcal{F}$ (lemma 4.35),
- the call-graph of section 3.3 is a smaller than $G(T_f)$ of section 4.6: comparing the definitions, we see that the call graph can be obtained from $G(T_f)$ by
 - replacing the output branch β by its weight, for example by taking $\langle 0 \rangle \beta \langle 0 \rangle$, which is smaller than β ,
 - in the arguments, replacing each Ωu by Ω , which is smaller than Ωu .
 None of these simplifications is really necessary in the implementation.
- Composition of calls (definition 3.8) is an instance of composition on $\mathcal{F}$ (definition 4.17).
- Collapsing and the rest of the criterion is then copied verbatim from sections 3.4 and 3.5 to section 4.8.

CONCLUDING REMARKS

Complexity. Since this totality test extends the termination test described in [Hyv14] and thus the usual size-change termination principle, its complexity is at least as bad: P-space hard. The extensions presented here do not make it any harder. As a result, the complexity of this totality test is P-space complete. However, it seems to work really well in practice. Letting the user choose the bounds B and D (with sane default values²⁸) allows to limit the combinatorial explosion to the definitions that really need it. Several other implementation tricks described in previous works [Hyv14] also contribute to a more than reasonable practical complexity.

Rewrite rules vs pattern matching. The choice of presenting the language in Haskell style with rewriting rules rather than using a “**match**” construction as in ML isn’t very important. The approach taken here is mixed:

- the language itself uses rewriting rules,
- the semantics (the domain $\mathcal{F}$) uses C^- and $.D$ which are closer to (partial) **match** expressions.

The reason is that the prototype was developed with clauses, but the theory is simpler with pattern matching: the semantics of C^- and $.D$ is very simple, while the semantics of rewriting rules would require introducing formally unification on sums of values and make for an even more verbose definition of $\mathcal{F}$.

The advantage pattern matching is that it allows direct analysis of definitions like

```
val f x = ...
    ... match f v with
        C y -> u
```

²⁸ $B = 2$ and $D = 2$ is a good choice

by producing a term $u[y := C^{-}fv]$. With this approach, the function

```
val f x = Succ (match f x with
                  Zero -> Zero
                  | Succ n -> Succ Zero)
```

could be seen as total. For such a definition, rewriting rules would hide the match with an external function doing the pattern matching, and by doing so, would lose all the information about this external function:

```
val m Zero = Zero
  | m (Succ _) = Succ Zero
val f x = Succ (m (f x))
```

(Recall that the analysis is local and that nothing is assumed of previously defined functions but their totality).

Such definitions are rare since their results depend on the evaluation mechanism: the Caml version doesn't terminate, but the corresponding Haskell version does, with the (expected) value `Succ (Succ Zero)`.²⁹

On the other hand, keeping rewriting rules allows the test to see that

```
val f Zero = Succ Zero
  | f (Succ n) = f Zero
```

is total. In this paper, this definition is interpreted by `f Zero + Succ Zero` which will be tagged “non-total”. The reason is that the recursive call `f Zero` doesn't use the `n` variable and the interpretation thus forgets about the corresponding left pattern.

Those two examples are rather ad-hoc and seldom appear in practice. The choice is thus mostly a matter of taste.

Copatterns. A. Abel has advocated the use of “copatterns” while writing coinductive definitions. This syntactical “trick” amounts to replacing the definition

```
val all_nats : nat -> stream(nat)
  | all_nats n = { Head=n; Tail=all_nats (Succ n) }
```

by³⁰

```
val all_nats : nat -> stream(nat)
  | (all_nats n).Head = n
  | (all_nats n).Tail = all_nats (Succ n)
```

This doesn't change the semantics of the program in any way and has the nice side effect that as rewriting rules the definition is terminating. The drawback is that writing functions such as `sums` (page 17) involves additional functions and is very tedious.

Operational Semantics. We have voluntarily refrained from giving the operational semantics of the language. The idea is that totality is a semantical property that is checked syntactically. The operational semantics has to guarantee that evaluating a total function on a total value is well defined, in particular that it should terminate. For example, head reduction that stops on records guarantees that a total value has a normal form: it cannot contain $\perp$ and cannot start with infinitely many inductive constructors (their priority is odd). Evaluation must reach a record (coinductive) at some point.

²⁹Simpler definitions where, for example, the matched call has a decreasing argument would still be accepted in their rewriting rules form.

³⁰This syntax is in fact supported by the `chariot` prototype.

Of course, a real programming language could introduce two kinds of records: coinductive ones and finite ones. The later could be evaluated during head reduction. Even better, destructors themselves could be coinductive (like **Tail** for streams) or finite (like **Head** for streams.)

In a similar vein, the language could have coinductive constructors to deal with coinductive types like finite or infinite lists. At the moment, the only way to introduce this type is with

```
data list_aux('a, 'b) where
  Nil : unit -> list_aux('a, 'b)
  | Cons : prod('a, 'b) -> list_aux('a, 'b)

codata inf_list('a) where
  unfold : inf_list('a) -> list_aux('a, inf_list('a))
```

Needless to say, using this quickly gets tiring.

Call-by-Value. Our semantics is “lazy” in the sense that values containing $\perp$ are not necessarily equal to $\perp$ themselves. The fact that value constructors are lazy is crucial when applying the size-change principle. This is what guarantees that a C^p corresponds exactly to one constructor of priority p . With the strict version, a C^p corresponds either to one constructor (when its argument is different from $\perp$), to **0** (which is total), or *nothing* (when its argument is $\perp$).

However, because the call-graph contains all the recursive calls appearing in the definition, it seems like it cannot see the difference between call-by-name and call-by-value. For example, examples like

```
val f {Fst=0; Snd=s} = 3
  | f {Fst=n+1; Snd=s} = f {Fst=n; Snd=f {Fst=n+1; Snd=s}}
```

from page 36 are always rejected as the call-graph contains the call

$$f \{Fst=n+1; Snd=s\} \rightarrow \Omega f \{Fst=n+1; Snd=s\}$$

The way the call-graph is constructed is reminiscent of the concept of *dependency pairs* [AG00] from term rewriting, where we replace a rule $f l_1 \dots l_k \rightarrow r$ by the $f l_1 \dots l_k \rightarrow g u_1 \dots u_l$ where $g u_1 \dots u_l$ are subterms of r . This new rewriting system is terminating if and only if the original one is terminating. Note that “terminating” for a rewriting system is independent of the reduction strategy.

If the analogy holds, it would make the criterion as described a criterion that would work unchanged for call-by-value.

Higher order types. The theory should extend to account for some higher order datatypes. Defining T -branching trees as (coinductive)

```
codata tree('b, 'n) where
  child : tree('b, 'n) -> ('b -> tree('b, 'n))
```

or (inductive)

```
data tree('b, 'n) where
  root : unit -> tree('b, 'n)
  | fork : ('b -> tree('b, 'n)) -> tree('b, 'n)
```

should for example make the corresponding **map** function pass the totality test.

Dependent Types. Dealing with dependent types is easy: tag dependent functions as “non-total”. While this sounds like a joke, it illustrates the fact that even without any theory for dependent types, this totality checker can be used for dependently typed languages. Of course, the idea would be to extend it to actually do something interesting on dependent types.

Many useful dependent types like “lists of size n ” can in fact be embedded in bigger non dependent datatypes like (“lists” in this case). The totality checker can, at least in principle, be used for those types. That, and the extension to some higher order as described above would go a long way to provide a theoretically sound totality checker for dependent languages like Agda or Coq.

Sized types. Just like in Agda, it seems using sized-types for termination and totality is mostly orthogonal to this totality checker. Considering the type of sizes to be a datatype is probably enough to make the totality checker “size aware”. Adding sized types would help dealing with examples that the current totality test ignores, like

```
val nats = {Head = 0; Tail = map_stream Succ nats}
```

We could have the information that `Succ` as type $\text{nat}^\alpha \rightarrow \text{nat}^{\alpha+1}$. Even purely inductive functions like

```
val sum_list : list(nat) -> nat
| sum_list [] = 0
| sum_list n::l = n + sum_list(l)
```

```
data rose_tree('a) where
  Node : list(rose_tree('a)) -> rose_tree('a)
```

```
val sum_rose_tree : rose_tree(nat) -> nat
| sum_rose_tree (Node(l)) = sum_list (map_list sum_rose_tree l)
```

would benefit from sized types.

REFERENCES

- [AA02] Andreas Abel and Thorsten Altenkirch. A predicative analysis of structural recursion. *Journal of Functional Programming*, 12:1–41, January 2002.
- [Abe10] Andreas Abel. Miniagda: Integrating sized and dependent types. In *In Partiality and Recursion (PAR 2010)*, 2010. arXiv:1012.4896.
- [Abe12] Andreas Abel. Type-based termination, inflationary fixed-points, and mixed inductive-coinductive types. In *Proceedings 8th Workshop on Fixed Points in Computer Science, FICS 2012, Tallinn, Estonia, 24th March 2012.*, pages 1–11, 2012.
- [AD12] Thorsten Altenkirch and Nils Anders Danielsson. Termination checking in the presence of nested inductive and coinductive types. In Ekaterina Komendantskaya, Ana Bove, and Milad Niqui, editors, *PAR-10. Partiality and Recursion in Interactive Theorem Provers*, volume 5 of *EasyChair Proceedings in Computing*, pages 101–106. EasyChair, 2012.
- [AG00] Thomas Arts and Jürgen Giesl. Termination of term rewriting using dependency pairs. *Theor. Comput. Sci.*, 236(1-2):133–178, 2000.
- [Ahn14] Ki Yung Ahn. *The Nax Language: Unifying Functional Programming and Logical Reasoning in a Language based on Mendler-style Recursion Schemes and Term-indexed Types*. PhD, Portland State University, December 2014.
- [AJ94] Samson Abramsky and Achim Jung. Domain theory. In Samson Abramsky, Dov M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science (Vol. 3)*, chapter Domain Theory, pages 1–168. Oxford University Press, Inc., New York, NY, USA, 1994.

- [Bar92] Michael Barr. Algebraically compact functors. *J. Pure Appl. Algebra*, 82(3):211–231, 1992.
- [Ber93] Ulrich Berger. Total sets and objects in domain theory. *Annals of Pure and Applied Logic*, 60(2):91–117, 1993.
- [CBGB16] Ranald Clouston, Ales Bizjak, Hans Bugge Grathwohl, and Lars Birkedal. The guarded lambda-calculus: Programming and reasoning with guarded recursion for coinductive types. *Logical Methods in Computer Science*, 12(3), 2016.
- [CF92] Robin Cockett and Tom Fukushima. About Charity. Yellow Series Report No. 92/480/18, Department of Computer Science, The University of Calgary, June 1992.
- [Cla13] Pierre Clairambault. Strong functors and interleaving fixpoints in game semantics. *RAIRO - Theor. Inf. and Applic.*, 47(1):25–68, 2013.
- [Coc96] Robin Cockett. Charitable thoughts, 1996. (draft lecture notes, <http://www.cpsc.ucalgary.ca/projects/charity/home.html>).
- [Coq93] Thierry Coquand. Infinite objects in type theory. In H. Barendregt and T. Nipkow, editors, *Types for Proofs and Programs*, pages 62–78. Springer, Berlin, Heidelberg, 1993.
- [Dou17a] Amina Doumane. Constructive completeness for the linear-time μ -calculus. In *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017*, pages 1–12, 2017.
- [Dou17b] Amina Doumane. *On the infinitary proof theory of logics with fixed points. (Théorie de la démonstration infinitaire pour les logiques à points fixes)*. PhD thesis, Paris Diderot University, France, 2017.
- [For14] Jérôme Fortier. *Expressive Power of Circular Proofs*. PhD, Aix Marseille Université ; Université du Québec à Montréal, December 2014.
- [FS14] Jérôme Fortier and Luigi Santocanale. Cuts for circular proofs. In Nikolaos Galatos, Alexander Kurz, and Constantine Tsinakis, editors, *TACL 2013. Sixth International Conference on Topology, Algebra and Categories in Logic*, volume 25 of *EasyChair Proceedings in Computing*, pages 72–75. EasyChair, 2014.
- [Gua18] Adrien Guatto. A generalized modality for recursion. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, pages 482–491, 2018.
- [Hyv14] Pierre Hyvernât. The size-change termination principle for constructor based languages. *Logical Methods in Computer Science*, 10(1), 2014.
- [LJBA01] Chin Soon Lee, Neil D. Jones, and Amir Ben-Amram. The size-change principle for program termination. In *Symposium on Principles of Programming Languages*, volume 28, pages 81–92. ACM press, january 2001.
- [LR18] Rodolphe Lepigre and Christophe Raffalli. Practical subtyping for curry-style languages, 2018. accepted for publication in *ACM Transactions on Programming Languages and Systems (TOPLAS)*.
- [Men91] Nax Paul Mendler. Inductive types and type constraints in the second-order lambda calculus. *Annals of Pure and Applied Logic*, 51(1-2):159–172, 1991.
- [Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.
- [Nak00] Hiroshi Nakano. A modality for recursion. In *15th Annual IEEE Symposium on Logic in Computer Science, Santa Barbara, California, USA, June 26-29, 2000*, pages 255–266, 2000.
- [Nor08] Ulf Norell. Dependently typed programming in agda. In *In Lecture Notes from the Summer School in Advanced Functional Programming*, 2008.
- [PJ87] Simon Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, January 1987.
- [Plo83] Gordon Plotkin. Domains, 1983. Pisa Notes.
- [San02a] Luigi Santocanale. A calculus of circular proofs and its categorical semantics. In Mogens Nielsen and Uffe Engberg, editors, *FoSSaCS*, volume 2303 of *Lecture Notes in Computer Science*, pages 357–371. Springer, 2002.
- [San02b] Luigi Santocanale. From parity games to circular proofs. *Electr. Notes Theor. Comput. Sci.*, 65(1):305–316, 2002.
- [San02c] Luigi Santocanale. μ -bicomplete categories and parity games. *Theoretical Informatics and Applications*, 36:195–227, 2002.

- [SHGL94] Viggo. Stoltenberg-Hansen, Edward R. Griffor, and Ingrid. Lindstrom. *Mathematical Theory of Domains*. Cambridge University Press Cambridge ; New York, 1994.
- [Smy78] Michael B. Smyth. Power domains. *Journal of Computer and System Sciences*, 16(1):23–36, 1978.
- [Smy83] Michael B. Smyth. Power domains and predicate transformers: A topological view. In *ICALP*, volume 154 of *Lecture Notes in Computer Science*, pages 662–675. Springer, 1983.
- [The04] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004.
- [Tur36] Alan Mathison Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265, 1936.