The Size-Change Principle for Mixed Inductive and Coinductive types

Pierre Hyvernat

▶ To cite this version:

Pierre Hyvernat. The Size-Change Principle for Mixed Inductive and Coinductive types. 2022. hal-01989688v2

HAL Id: hal-01989688
https://hal.science/hal-01989688v2

Preprint submitted on 21 Jul 2022 (v2), last revised 13 Jul 2023 (v3)

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 Savoie Mont Blanc, CNRS, LAMA, 73000 Chambéry, France.
e-mail address: pierre.hyvernat@univ-smb.fr
URL: http://lama.univ-savoie.fr/~hyvernat/

Abstract. This paper shows 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 with inductive and coinductive types. The size-change principle is used to check both termination and productivity, and the resulting principle is sound even if inductive and coinductive types are arbitrarily nested. A 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 early work on circular proofs. The proof of correctness of the criterion relies on an extension of the language's denotational semantics to a domain of untyped values with non-deterministic sums.

Contents

Introduction 2
Related Works 3
Plan of the Paper 5
1. The Language and its Semantics 5
1.1. Values 5
1.2. Type Definitions 6
1.3. Semantics in Domains 8
1.4. Semantics in Domains with Totality 9
1.5. Recursive Definitions 10
2. Combinatorial Description of Totality 13
2.1. Parity Games 14
2.2. Parity Games from Types 15
2.3. Forgetting Types 18
3. Non-Deterministic Semantics for Definitions 18
3.1. Smyth Power Domain 19
3.2. Recursion and Fixed Points 20
3.3. Operators 22
3.4. Interpreting Recursive Definitions 28

Received by the editors July 21, 2022.
Key words and phrases: coinductive types; nested fixed points; size-change principle; functional programming; recursive definitions.

© Pierre Hyvernat
Creative Commons

4. Call-Graphs and the Size-Change Principle 29
4.1. Call-Graph 29
4.2. Weights and Approximations 32
4.3. Collapsing 36
4.4. The Size-Change Principle 40
4.5. Implementing the Totality Checker 43
Concluding Remarks 44
References 47

Introduction

Inductive types (also called algebraic datatypes) are a cornerstone of typed functional programming: Haskell and Caml both rely heavily on them. One mismatch between the two languages is that Haskell is *lazy* while Caml is *strict*. A definition[1] like

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

is valid but useless in Caml because the evaluation mechanism will loop trying to evaluate it completely (call-by-value evaluation). In Haskell, because evaluation is lazy (call-by-need), such a definition isn't unfolded until strictly necessary and asking for its third element will only unfold the definition three times. 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] which distinguishes between least and greatest fixed points and where the user can nest them arbitrarily to define new datatypes. To offer a familiar programming experience, definitions are not restricted and 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)          -- f(1) => f(f(0)) => f(1) => ...
```

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

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

When no coinductive type is involved, totality amount to termination and this works is a generalization of the termination checker previously developed by the author [Hyv14]. It is important to keep in mind that any definition that passes this test is guaranteed to be correct but that some correct definitions are rejected.[3] In a programming context, the result of this second step can be ignored when the programmer (thinks she) knows better. In a proof-assistant context however, it cannot be ignored as non total definitions lead to inconsistencies, the most obvious example being

```
val magic_proof = magic_proof
```

[1] The examples are given using the syntax of `chariot` which is briefly described in sections 1.2 and 1.5. They should nevertheless be readable by anyone with a modicum of experience in functional programming.

[2] A prototype implementation in Caml is available from `https://github.com/phyver/chariot`.

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

which is non-terminating but belongs to all types. There are subtler examples of definitions that normalize to values but still lead to inconsistencies (c.f. example on page 13).

In Coq [The04], the productivity condition for coinductive definitions is ensured by a 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. Agda's checker extends the termination checker developed by A. Abel [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]. Currently, Agda's checker is patched to deal with known counter examples like the one described in Section 1.5, but no proof of correctness is available. This paper provides a first step toward a provably correct totality checker.

Related Works.

Circular proofs. The main inspiration for this work comes from 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.

The criterion developed in this paper uses a strong combinatorial principle (the size-change principle) to check a sanity condition on a circular proof. This is strictly stronger than the criterion L. Santocanale and G. Fortier used in their work, which corresponds to the syntactical structurally decreasing / guardedness condition on recursive definitions.

While circular proofs were a primary inspiration, this work cannot be reduced to a circular proof system. The main problem is that all existing circular proof systems are linear and do not have a simple cut-elimination procedure, i.e. an evaluation mechanism. Cuts and exponentials would be needed to interpret the full `chariot` language and while cuts can be added [FS14, For14], adding exponentials looks difficult and hasn't been done.

More recent works in circular proof theory replace L. Santocanale's criterion by a much stronger combinatorial condition [Dou17b, Dou17a]. It involves checking that some infinite words are recognized by a parity automata, which is a decidable problem. 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 simple termination. 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 definition

```
val inf = Node { Left = inf; Right = inf }       -- infinite binary tree
```

yields an infinite binary tree and depending on the types of `Node`, `Fst` and `Snd`, the definition may be correct or incorrect (page 13)!

Charity. The closest ancestor to `chariot` is the language `charity`[4] [CF92, Coc96], developed by R. Cockett and T. Fukushima. It lets the programmer 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.

It means that recursive functions can only be defined via eliminators. By construction, they are either "trivially" structurally decreasing on their argument, or "trivially" guarded. The advantage is that *all* functions are total by construction and the disadvantage is that the language is not Turing complete.

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", written "$\triangleright$", is introduced. The 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 constructors for types and terms. Finally, these works only consider greatest fixed points (as in Haskell) and are thus of limited interest for systems like Agda or Coq.

Sized types. This approach extends type theory with a notion of "size" that annotate types. It has been successful and is implemented in Agda [Abe10, Abe12]. It is possible to specify that the `map` function on list has type $\forall n, \mathtt{list}^n(T) \to \mathtt{list}^n(T)$, where $\mathtt{list}^n(T)$ is the type of lists with n elements of type T. These extra parameters give 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 the libraries' implementors job: in many cases, sizes are invisible to the casual user. Note however that sizes only help showing termination and productivity. Developing a totality checker is orthogonal to designing an appropriate notion of size, and the totality checker described in this paper can probably work hand in hand with standard size notions.

Fixed points in game semantics. An important tool in this paper is the notion of *parity game*. P. Clairambault [Cla13] explored a category of games 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 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 usual semantics of recursion.

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

SubML. C. Raffalli and R. Lepigre used the size-change principle 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`, and usual recursive definitions are not allowed. Since no implementation of Nax is available, it is however difficult to experiment with it.

Plan of the Paper. We start by introducing the language `chariot` and its denotational semantics in Section 1. We assume the reader is familiar with functional programming, recursive definitions and their semantics, Hindley-Milner type checking, algebraic datatypes, pattern matching, etc. The notion of totality is also given there. Briefly, it generalizes termination in a way that accounts for inductive and coinductive types. 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 gives an interpretation of recursive definitions that is mathematically better behaved and easier to work with than the lists of clauses used in `chariot`. Section 4 introduces the notion of *call-graph* of a recursive definition and shows it reflects totality. The notion of approximations, necessary for the size-change principle is also defined there. Finally, this section applies the size-change principle to totality checking and briefly describes how to implement it.

1. The Language and its Semantics

1.1. Values.

Given a recursive definition, we are interested in the "healthiness" of its semantics. Such considerations take place in the realm of semantics values, and while every reader will have her favorite programming language and reduction strategy, those are mostly irrelevant to the rest of the paper.

Any finite list of recursive definitions only involves a finite number of types, with a finite number of constructors and destructors. We thus fix, once and for all, a finite set of constructor and destructor names. Because we deal with semantically infinite values, the next definition is coinductive.

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 coinductively by the grammar

$$v \quad ::= \quad \bot \quad | \quad x \quad | \quad \mathtt{C}\,v \quad | \quad \{\mathtt{D}_1 = v_1; \dots; \mathtt{D}_k = v_k\}$$

where

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

To make the theory slightly less verbose, constructors always have a single argument. Expressivity doesn't suffer because we can always use a tuple $\{\mathtt{Fst} = t_1; \mathtt{Snd} = t_2\}$ as argument. Of course, the implementation of `chariot` allows constructors of arbitrary arity.

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

(1) $\bot \leq v$ for all values v,
(2) if $x \leq x'$ in X_i, then $x \leq x'$ in $\mathcal{V}(X_1,\dots,X_n)$,
(3) "$\leq$" is contextual: if $u \leq v$ then $C[x := u] \leq C[x := v]$ for any value C, where substitution is defined in the obvious way.

1.2. **Type Definitions.** The approach described in this paper is 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.[5]

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.

Definition 1.3. Datatypes are introduced by the keywords "`data`" or "`codata`" and may have parameters. Types parameters are written with a quote as in Caml. The syntax is:

```
data new_type('x, ...) where
    | C₁ : T₁ -> new_type('x, ...)
    ...
    | Cₖ : Tₖ -> new_type('x, ...)
```

```
codata new_type('x, ...) where
    | D₁ : new_type('x, ...) -> T₁
    ...
    | Dₖ : new_type('x, ...) -> Tₖ
```

[5]Note that can't formally 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        --non terminating
```

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.

where each T_i is built from earlier types, parameters and *new_type*`('x, ...)`. Note that type definitions are *uniform* in that the parameters of *new_type* are the same everywhere in the definition.

Mutually recursive types are possible, but they need to be of the same polarity (all `data` or all `codata`) and all of them needs to have exactly the same parameters "`('x, ...)`".

Here are some examples:

```
codata unit where                                         -- unit type: no destructor

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

data nat where  Zero : unit -> nat                        -- unary natural numbers
              | Succ : nat  -> nat

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

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

The examples given in the paper extend this syntax by allowing n-ary constructors. For example, `Zero` will have type `nat` (instead of `unit -> nat`) and `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 codatatype 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 }
```

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 a sound operational semantics, codata should not be fully evaluated. The easiest way to ensure that is to stop evaluation on records: evaluating "`zeros`" will give "`{Head = ???; Tail = ???}`" where the "`???`" are not evaluated. The copattern view [APTS13] is natural here. The definition of `zeros` using copatterns (allowed in `chariot`) looks like

```
val zeros : stream(nat)
  | zeros.Head = Zero
  | zeros.Tail = zeros
```

We can interpret the clauses as a terminating rewriting system. In particular, the term `zeros` doesn't reduce by itself. Because this paper is only interested in the denotational semantics of definitions, the details of the evaluation mechanism are fortunately irrelevant.

We will use the following conventions:

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

1.3. **Semantics in Domains.** Values are naturally interpreted in Scott domains, but our main construction isn't quite a Scott domain. We will thus use the weaker notion of algebraic DCPO. It is an order with the following properties:

- every directed set has a least upper bound (DCPO),
- it has a basis of compact elements (algebraic).

What is missing to have a Scott domain is that the order is bounded complete, i.e. that finite bounded subsets have a least upper bound. Scott domains have the advantage of forming a cartesian closed category, which is not the case of algebraic DCPOs. We use the notation $[\mathcal{D} \to \mathcal{E}]$ for the Scott domain of continuous functions from $\mathcal{D}$ to $\mathcal{E}$.

There is a natural interpretation of types in the category of Scott-domains / where morphisms are continuous functions that are *not* required to preserve the least element. The category theory aspect is not relevant because all the types are 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 1.4. *If the X_is are Scott-domains, then $\big(\mathcal{V}(X_1,\dots,X_n), \leq\big)$ is a Scott-domain.*

Type expressions with parameters are generated by the grammar

$$T \quad ::= \quad X \quad | \quad \mathtt{x} \quad | \quad \theta_\mu(T_1,\dots,T_k) \quad | \quad \theta_\nu(T_1,\dots,T_k)$$

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 k and θ_ν is the name of a codatatype of arity k. A type is *closed* if it doesn't contain variables. It may contains parameters though.

Definition 1.5. The interpretation of a closed type $T\big(\overline{X}\big)$ with domain parameters is defined *coinductively* from the following typing rules:

(1) $\dfrac{}{\bot : T}$ for any type T,

(2) $\dfrac{u \in X}{u : X}$ for any parameter X,

(3) $\dfrac{u : T[\sigma]}{\mathtt{C}\, u : \theta_\mu(\sigma)}$ where $\mathtt{C} : T \to \theta_\mu(\sigma)$ is a constructor of θ_μ,

(4) $\dfrac{u_1 : T_1[\sigma] \quad \dots \quad u_k : T_k[\sigma]}{\{\mathtt{D}_1 = u_1;\ \dots;\ \mathtt{D}_k = u_k\} : \theta_\nu(\sigma)}$ where $\mathtt{D}_i : \theta_\nu(\sigma) \to T_i$, $i = 1,\dots,k$ are all the destructors for type θ_ν.

In the third and fourth rules, σ denotes a substitution $[\mathtt{x}_1 := T_1,\dots,\mathtt{x}_n := T_n]$ and $T[\sigma]$ denotes the type T where each variable $\mathtt{x}_i$ has been replaced by T_i.

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

All the $\bot$ coming from the parameters are identified. There are thus several ways to prove that $\bot$ belongs to the interpretation of a type: either with rule (1) or rules (2). The following is easily proved by induction on the type expression T.

Proposition 1.6. *Let X_1,..., X_n be domains, if T is a type then*

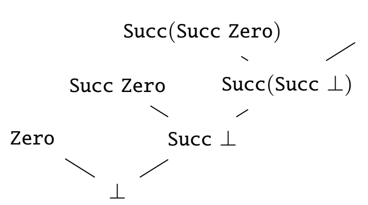
(1) *with the order inherited from the* X_i*s (Definition 1.2),* $[\![T]\!](X_1, \dots, X_n)$ *is a domain,*
(2) $X_1, \dots, X_n \mapsto [\![T]\!](X_1, \dots, X_n)$ *is functorial.*
(3) *if* $T = \theta_\mu(\mathtt{x}_1, \dots, \mathtt{x}_n)$ *is a datatype with constructors* $\mathtt{C}_i : T_i \to T$*, we have*

$$\begin{array}{rcl} [\![T]\!]\left(\overline{X}\right) & = & \left\{\mathtt{C}_i u_i \mid i = 1, \dots, n \text{ and } u_i \in [\![T_i]\!]\right\} \cup \{\bot\} \\ & \cong & \left([\![T_1]\!]\left(\overline{X}\right) + \cdots + [\![T_k]\!]\left(\overline{X}\right)\right)_\bot \end{array}$$

(4) *if* $T = \theta_\nu(\mathtt{x}_1, \dots, \mathtt{x}_n)$ *is a codatatype with destructors* $\mathtt{D}_i : T_i \to T$*, we have*

$$\begin{array}{rcl} [\![T]\!]\left(\overline{X}\right) & = & \left\{\{\dots; \mathtt{D}_i = u_i; \dots\} \mid i = 1, \dots, n \text{ and } u_i \in [\![T_i]\!]\right\} \cup \{\bot\} \\ & \cong & \left([\![T_1]\!]\left(\overline{X}\right) \times \cdots \times [\![T_k]\!]\left(\overline{X}\right)\right)_\bot \end{array}$$

The operations $+$ and $\times$ are the set theoretic operations (disjoint union and cartesian product), and $S_\bot$ is the usual notation for $S \cup \{\bot\}$. This shows that the semantics of types are fixed points of standard operators. For example, $[\![\mathtt{nat}]\!]$ is the domain of "lazy natural numbers":



and the following are different elements of $[\![\mathtt{stream(nat)}]\!]$:

- $\bot$,
- $\{\mathtt{Head} = \mathtt{Succ}\,\bot; \mathtt{Tail} = \bot\}$
- $\{\mathtt{Head} = \mathtt{Zero}; \mathtt{Tail} = \{\mathtt{Head} = \mathtt{Zero}; \mathtt{Tail} = \{\mathtt{Head} = \mathtt{Zero}; \dots\}\}\}$

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, $\mathtt{Succ}(\mathtt{Succ}(\mathtt{Succ}(\dots)))$ is an element of $[\![\mathtt{nat}]\!]$ as the limit of the chain $\bot \leq \mathtt{Succ}\,\bot \leq \mathtt{Succ}(\mathtt{Succ}\,\bot) \leq \cdots$. In order to distinguish between inductive and coinductive types, we add a notion of *totality*[6] to the domains.

Definition 1.7.
(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.

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

Definition 1.8. 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 $|T| = \mu X.\widehat{\theta_\mu}(X, |T_1|, \dots, |T_n|)$ (least fixed point),
- if $T = \theta_\nu(T_1, \dots, T_n)$ is a codatatype, then $|T| = \nu X.\widehat{\theta_\nu}(X, |T_1|, \dots, |T_n|)$ (greatest fixed point),

where

(1) if $T = \theta_\mu(\mathtt{x}_1, \dots, \mathtt{x}_n)$ is a datatype with constructors $\mathtt{C}_i : T_i \to T$, $\widehat{\theta_\mu}$ is the operator
$$X, X_1, \dots, X_n \quad \mapsto \quad \bigcup_{i=1,\dots,k} \Big\{ \mathtt{C}_i u \;\Big|\; u \in \big|T_i[\sigma]\big| \Big\}$$

(2) if $T = \theta_\nu(\mathtt{x}_1, \dots, \mathtt{x}_n)$ is a codatatype with destructors $\mathtt{D}_i : T \to T_i$, $\widehat{\theta_\nu}$ is the operator
$$X, X_1, \dots, X_n \quad \mapsto \quad \Big\{ \{\mathtt{D}_1 = u_1; \dots; \mathtt{D}_k = u_k\} \;\Big|\; \text{each } u_i \in \big|T_i[\sigma]\big| \Big\}$$

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

Because these operators act on subsets of the set of all values and are monotonic, the least and greatest fixed points exist by the Knaster-Tarski theorem. It is not difficult to see that each element of $|T|$ is in $[\![T]\!]$ and since no element of $|T|$ contains $\perp$, $|T|$ contains only maximal element of $[\![T]\!]$:

Lemma 1.9. *If T is a type with domain parameters, $\big([\![T]\!], |T|\big)$ is a domain with totality. Moreover, if T is closed, each $t \in |T|$ is maximal in $[\![T]\!]$.*

1.5. **Recursive Definitions.** Like in Haskell, recursive definitions are given by lists of clauses. Here are two examples: the Ackermann function (using some syntactic sugar for the constructors `Zero` and `Succ`)

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

and the `map` function on streams:[7]

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

Definition 1.10. A recursive definition is introduced by the keyword `val` and consists of a finite list of clauses of the form

`| f` p_1 `...` p_n `=` u

where

- `f` is one of the function names being mutually defined,
- each p_i is a *finite* pattern
$$p \quad ::= \quad \mathtt{x}_i \;\mid\; \mathtt{C}\ p \;\mid\; \{\mathtt{D}_1 = p_1; \dots; \mathtt{D}_k = p_k\}$$
  where each $\mathtt{x}_i$ is a variable name,

[7]This 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.

- and u is a *finite* term

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

  where k can be equal to 0, each $\mathtt{x}_i$ is a variable name, and each $\mathtt{f}$ is function name (recursive or otherwise).

Moreover, for any clause, the patterns $p_1, \dots, p_k$ are *linear*: variables can only appear at most once in the p_is,

We assume the definitions are validated using standard Hindley-Milner type inference / type checking . This includes in particular checking that clauses of the definition cover all values of the appropriate type, and that no record is missing any field. Those steps are not described here [PJ87].

Standard semantics of a recursive definition. Hindley-Milner type checking guarantees that each list of clauses for functions $\mathtt{f}_1 : T_1, \dots, \mathtt{f}_n : T_n$ (each T_i is a function type) gives rise to an operator

$$\Theta^{\mathrm{std}}_{\mathtt{f}_1,\dots,\mathtt{f}_n} : [\![T_1]\!] \times \cdots \times [\![T_n]\!] \to [\![T_1]\!] \times \cdots \times [\![T_n]\!]$$

where the semantics of types is extended with $[\![T \to T']\!] = \big[\, [\![T]\!] \to [\![T']\!] \,\big]$. The semantics of $\mathtt{f}_1, \dots, \mathtt{f}_n$ is then defined as the fixed point of the operator $\Theta^{\mathrm{std}}_{\mathtt{f}_1,\dots,\mathtt{f}_n}$ which exists by Kleene theorem.

Because this will be central to the paper, let's describe more precisely the standard semantics of the definition in the simple case of a single recursive function $\mathtt{f}$ taking a single argument. Given an environment ρ for functions other than $\mathtt{f}$, the recursive definition for $\mathtt{f} : A \to B$ gives rise to an operator $\Theta^{\mathrm{std}}_{\rho,\mathtt{f}}$ on $[[\![A]\!] \to [\![B]\!]]$ whose fixed point is the semantics of $\mathtt{f}$, written $[\![\mathtt{f}]\!]_\rho : [\![A]\!] \to [\![B]\!]$. The operator $\Theta^{\mathrm{std}}_{\rho,\mathtt{f}}$ is defined as follows.

Definition 1.11.

(1) Given a linear pattern p and a value p, the unifier $[p := v]$ is the substitution defined inductively with
  - $[\mathtt{y} := v] = [\mathtt{y} := v]$ where the RHS is the usual substitution of $\mathtt{y}$ by v,
  - $[\mathtt{C}p := \mathtt{C}v] = [p := v]$,
  - $[\{\mathtt{D}_1 = p_1; \dots; \mathtt{D}_n = p_n\} := \{\mathtt{D}_1 = v_1; \dots; \mathtt{D}_n = v_n\}] \ = \ [p_1 := v_1] \cup \cdots \cup [p_n := v_n]$ (note that because patterns are linear, the unifiers don't overlap),
  - in all other cases, the unifier is undefined. Those cases are:
    - $[\mathtt{C}p := \mathtt{C}'v]$ with $\mathtt{C} \neq \mathtt{C}'$,
    - $[\{\dots\} := \{\dots\}]$ when the 2 records have different sets of fields,
    - $[\mathtt{C}p := \{\dots\}]$ and $[\{\dots\} := \mathtt{C}v]$.

  When the unifier $[p := v]$ is defined, we say that *the value v matches the pattern p.*

(2) Given $f : [\![A]\!] \to [\![B]\!]$ and $v \in [\![A]\!]$, $\Theta^{\mathrm{std}}_{\rho,\mathtt{f}}(f)\big(v\big)$ can now be defined by:
  - taking the first clause "$\mathtt{f}\ p\ =\ u$" in the definition of $\mathtt{f}$ where p matches v,
  - returning $[\![u[p := v]]\!]_{\rho,\mathtt{f}:=f}$.

An important property of Hindley-Milner type checking is that it ensures a definition has a well defined semantics: there always is a matching clause. Because of that, the value "$\bot$" corresponds only to non-termination, not to failure of the evaluation mechanism, like projecting on a non-existing field. However, it doesn't mean the definition is correct from a denotational point of view. For that, we need to 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 (n+1))
```

is well typed and sends elements of the domain $[\![$`nat`$]\!]$ to the domain $[\![$`list`(`nat`)$]\!]$ but the image of `Zero` contains all the natural numbers. This is not total because totality for `list`(`nat`) contains only the finite lists. 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. The aim of this paper is to implement a test that will detect such problems.

A note on projections. The syntax of definitions given in Definition 1.10 doesn't allow projecting a record on one of its field. This makes the theory somewhat simpler and doesn't change expressivity of the language because it is always possible to rewrite a projection using one of the following ways:

- remove a projection on a previously defined function by introducing another function, as in
  ```
  | f x = ... (g u).Fst ...
  ```
  being replaced by
  ```
  | f x = ... projectFst (g u) ...
  ```
  where `projectFst` is defined with
  ```
  val projectFst { Fst = x; Snd = y } = x
  ```
- remove a projection on a variable by extending the pattern on the left, as in
  ```
  | f x = ... x.Head ...
  ```
  being replaced by
  ```
  | f { Head = h; Tail = t } = ... h ...
  ```
- remove a projection on the result of a recursively defined function by splitting the function into several mutually recursive functions, as in
  ```
  | f : prod(A, B) -> prod(A, B)
  | f p = ... (f u).Fst ...
  ```
  being replaced by
  ```
  | f1 : prod(A, B) -> A
  | f1 x = ... (f1 u1) ...
  | f2 : prod(A, B) -> B
  ...
  ```

The first point is the simplest and most general but shouldn't be used to remove projections on variables or recursive functions. The checker sees each external function as a black box about which nothing is known. Introducing external functions in a recursive definition hides information and makes totality checking much less powerful.

Of course, the implementation of `chariot` doesn't enforce this restriction and the theory can be modified accordingly.

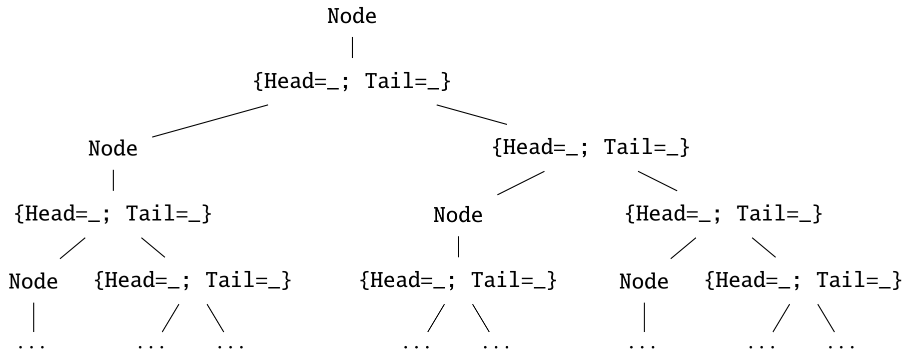
A subtle example. Here is a surprising example due to T. Altenkirch and N. A. Danielsson [AD12]: we define the inductive type

```
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 bad_s : stream(stree)
  | bad_s = { Head = Node bad_s ; Tail = bad_s }
val bad_t : stree
  | bad_t = Node bad_s
```

This is well typed and if evaluation is lazy, evaluation of `bad_t` or any of its subterms terminates. The semantics of `bad_t` doesn't contain $\perp$ and unfolding the definition gives



Such a term clearly leads to inconsistencies. For example, the following structurally decreasing function doesn't terminate when applied to `bad_t`:

```
val lower_left : stree -> empty
  | lower_left (Node { Head = t; Tail = s }) = f t
```

It is important to understand that `lower_left` is a total function and that non termination of `lower_left bad_t` is a result of `bad_t` being non total.

2. Combinatorial Description of Totality

The set of total values for a given type can be rather complex when datatypes and codatatypes 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
                        | Right : pair('x,'y) -> 'y
data tree where Node : pair(tree, tree) -> tree  -- well-founded binary trees
              | Leaf : unit -> tree
```

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

```
data box('x) where Node : 'x -> box('x)
                 | Leaf : unit -> tree
codata tree where Left : tree -> box(tree)  -- non-well founded binary trees
                 | Right : tree -> box(tree)
```

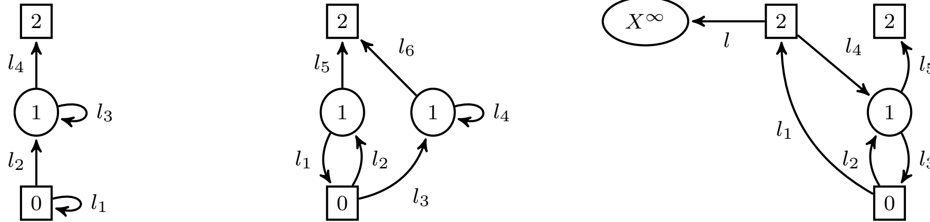
In this case, the value `inf` is of type `box(tree)`. Fortunately, there is a close relationship between set theoretic least and greatest fixed points and winning strategies for *parity games*.

2.1. **Parity Games.** Parity games are a two players games played on a finite transition system where each node is labeled by a natural number called its *priority*. When the node has odd priority, *Marie* (or "μ", or "player") is required to play. When the node is even,[8] *Nicole* (or "ν", 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 looses. 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.

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* to the games. Those nodes have no outgoing transition, have priority ∞ and each of them has an associated set X. On reaching them, Marie is required to choose an element of X to finish the game. She wins if she can do it and looses if the set is empty. Here are three examples of parity games:



Definition 2.1. Each position p in a parity game G with parameters $X_1, \ldots, X_n$ defines a set $||G_p||$ depending on $X_1,\ldots,X_n$ [San02c]. This set valued function $p \mapsto ||G_p||$ is defined by induction on the maximal finite priority of G and the number of positions with this 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 $Y, X_1, \ldots, X_n$ as follows: it is identical to G, except that position p is replaced by parameter Y and all its outgoing transitions are removed.[9] Compute recursively the interpretations $(G/p)_q$, depending on $Y, X_1, \ldots X_n$ and:
  - if p had an odd priority, define

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

  where $p \to q_1, \ldots p \to q_k$ are all the transitions out of p.

[8]Assigning odd to one player and even to the other is just a convention.

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

– if p had an even priority, define

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

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

An important result is:

Proposition 2.2 (L. Santocanale [San02c])**.**

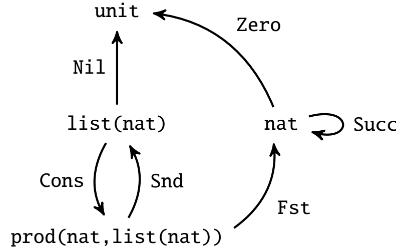
(1) *For each position p of G, the operation $X_1, \dots, X_n \mapsto ||G(X_1, \dots, X_n)_p||$ is a functor from Set^n to Set,*
(2) *there is a natural isomorphism $||G_p|| \cong \mathcal{W}(G)_p$ where $\mathcal{W}(G)_p$ is the set of winning strategies for Marie in game G from position p.*

2.2. **Parity Games from Types.** We can construct a parity game G from any type T in such a way that $|T| \cong ||G_T||$, for some distinguished position T in G.

Definition 2.3. 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 transitions $T_1 \xrightarrow{t} T_2$ is either a destructor t of type $T_1 \to T_2$ or a constructor t of type $T_2 \to T_1$ (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.

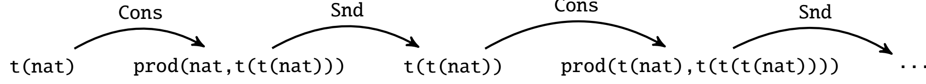
Because of that, Hindley-Milner type checking guarantees that a value of type T gives a strategy for a game on the graph of T where Marie (the player) chooses constructors and Nicole (the opponent) chooses destructors.

Lemma 2.4. *The graph of T is finite.*

This relies on the fact that recursive types are uniform: their parameters are constant in their definition. It becomes false if we were to allow more general types like

```
data t('x) where
  | Empty : unit               -> t('x)
  | Cons : prod('x, t(t('x))) -> t('x)   -- !!! not uniform
```

The graph of `t(nat)` would contain the following infinite chain:



Definition 2.5. 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 ... or $t \sqsubseteq T_n$.

Proof of Lemma 2.4. 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. Moreover, 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 type T is infinite of minimal order. Since the graph of T has bounded out-degree, König's lemma implies it contains an infinite path $\rho = T \to T_1 \to T_2 \to \cdots$ without repeated vertex. For any n, there is some $l > n$ such that T_l is of order at least κ. Otherwise, the path $T_{n+1} \to T_{n+2} \to \cdots$ is infinite and contradicts the minimality of T.

By definition, all transitions in the graph of T are of the form $\theta(\overline{T}) \to \nabla$ where ∇ is built using the type parameters in $\overline{T}$, the recursive types $\theta'(\overline{T})$ from the current (co)inductive type definition, and earlier types. There are thus three kinds of transitions.

(1) Transitions to a parameter $\theta(\overline{T}) \to T_i$. In this case, the target is a subexpression of the origin. This is the case of `Head` : `stream(nat)` $\to$ `nat`.
(2) Transitions $\theta(\overline{T}) \to \theta'(\overline{T})$, i.e. transitions to a type in the same mutual definition, *with the same parameters*. This kind of transitions can only be used a finite number of times because ρ doesn't contain repeated vertices. An example is `Succ` : `nat` $\to$ `nat`.
(3) In all other cases, the transition is of the form $\theta(\overline{T}) \to \nabla$, where ∇ is strictly earlier than θ. This is for example the case of `Cons` : `list(nat)` $\to$ `prod(nat, list(nat))` (recall that the transition goes in the opposite direction).

The order can only strictly increase in case (1). In cases (3), the target may contain types with order κ, but those may only come from within the parameter $\overline{T}$. The only types of order κ reachable from T (of order κ) are thus: subexpressions of some T_is. Since there are only finitely many of those, the infinite path ρ necessarily contains a cycle! This is a contradiction. $\square$

Definition 2.6. 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

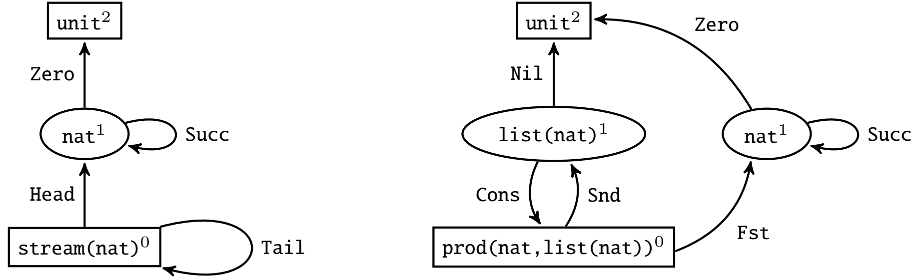
(1) each parameter of T is a parameter of G_T,
(2) if T_0 is a datatype in the graph of T, its priority is odd,
(3) if T_0 is a codatatype in the graph of T, its priority is even,
(4) if $T_1 \sqsubseteq T_2$ then the priority of T_1 is greater than the priority of T_2.

Lemma 2.7. *Each type has a parity game.*

Proof. The relation $\sqsubset$ 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. $\square$

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. Here are the first two parity games from page 14, seen as parity games for `stream(nat)` and `list(nat)`. The priorities are written as an exponent and the

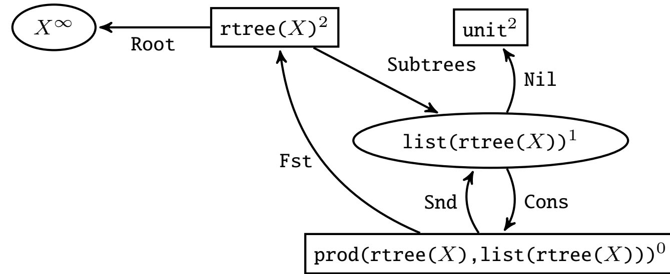
parity can be seen in the shape (square or round) of nodes.



The last example from page 14 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 the examples show, the priority of a type can be minimal (`stream(nat)`^{0}), maximal (`rtree(`X`)`^{2}) or somewhere in between (`list(nat)`^{1}) in its parity game.

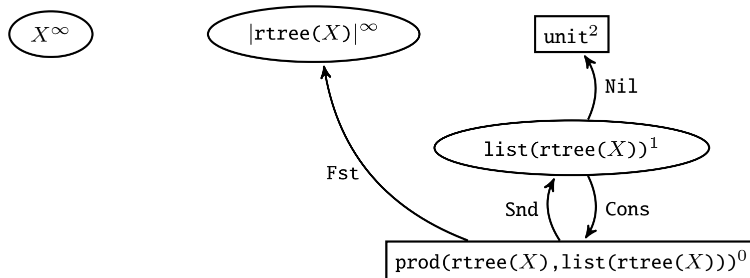
Proposition 2.8. *For any type T, if G is a parity game for T and if T_0 is a node of G, we have a natural isomorphism $||G_{T_0}|| \cong |T_0|$.*

Proof. The proof follows from the following simple lemma by induction.

Lemma 2.9. *If G is a parity game for type T, with parameters $\overline{X}$; and if T_0 one of its maximal nodes, then the predecessor game G/T_0 is a parity game for type T where T_0 as been replaced by its semantics and is a new parameter.*

This is a straightforward consequence of Definitions 1.8 and 2.1. □

For example, the predecessor for the above parity game is



Putting together Proposition 2.8 and Proposition 2.2, we get

Corollary 2.10. *If T is a type and G a parity game for T, we have $\mathcal{W}(G)_T \cong |T|$. In particular, $v \in [\![T]\!]$ is total iff every branch of v has even principal priority.*[10]

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

Definition 2.11. The set of *values with leaves in* $X_1,\dots,\ X_n$, written $\mathcal{V}(X_1,\dots,X_n)$ is defined coinductively by the grammar

$$v \quad ::= \quad \bot \quad | \quad x \quad | \quad \mathtt{C}^p\, v \quad | \quad \{\mathtt{D}_1 = v_1; \dots; \mathtt{D}_k = v_k\}^p$$

where

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

Corollary 2.10 gives an intrinsic notion of totality on $\mathcal{V}$.

Definition 2.12. 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 only used while checking totality. They are never shown to the user and are inferred automatically during Hindley-Milner type checking:

- each instance of a constructor / destructor is annotated by its type during type checking,
- all the types appearing in the definitions are gathered (and completed) to a parity game,
- each constructor / destructor is then associated with the priority of its type,
- the type can be forgotten, and only its priority remains.

3. Non-Deterministic Semantics for Definitions

Several steps will be necessary before get a totality test that is computationaly interesting.

(1) We first extend the domain of values with non-deterministic sums.
(2) This makes it possible to give a "uniform" domain interpretation of recursive definitions. This interpretation both generalizes the standard semantics because it allows untyped definitions with run-time errors, and simplifies it because the order of recursive clauses becomes irrelevant. Simply put, a recursive definition is interpreted by the non-deterministic sum of its clauses.
(3) This interpretation is modified into a "syntactical" domain, replacing clauses by special terms representing functions.

Up until then, one can argue that the interpretation is faithful to the original recursive definition. The next step, called the call-graph of the definition, profoundly alters the semantics of the definition but reflects totality. Thus, showing totality of the simplified interpretation using the size-change principle will imply totality of the original definition. This will be described in the next section.

[10]Since any $v \in [\![T]\!]$ gives a strategy in the game of T, priorities for constructors can be looked up in the game of T.

3.1. **Smyth Power Domain.** Several notions from domain theory will be used but the prerequisites are kept to a minimum. They can be found in any introductory text on domain theory [Plo83, SHGL94, AJ94]. One particular result that is worth recalling 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 starting partial order is a sup semi-lattice, the resulting algebraic DCPO is a Scott-domain. Without the prefix "Scott", the term "domain" will be used as a synomym for "algebraic DCPO".

We will extend the grammar of values with non-deterministic sums

$$v \quad ::= \quad \bot \quad | \quad \mathtt{C}^p\, v \quad | \quad \{\mathtt{D}_1 = v_1; \dots; \mathtt{D}_k = v_k\}^p \quad | \quad v_1 + v_2$$

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 $\mathtt{C}$ and $\{\mathtt{D}{=}_; \dots\}$,
- $u + v \leq u$.

This gives rise to a preorder which we implicitly quotient by the corresponding equivalence. This construction is known as the "Smyth power domain" [Smy78]. The ideal completion of the finite terms generated by above grammar introduces all values of infinite depth (as in Definition 1.1) and *some* infinite sums. Only sums that can be obtained as limits of finite sums of compact elements are allowed. The following gives a concrete description of the corresponding order [Smy83, AJ94].

Proposition 3.1. *The elements of the Smyth power domain on $\mathcal{V}$ are subsets of $\mathcal{V}$. They are written additively and satisfy*

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. *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 simply reverse inclusion. We can always add a greatest element to a domain. In our case, it will be used to denote errors.

Lemma 3.2.

1. *For any domain D, $D \cup \{\top\}$, with $\forall x, x \leq \top$, is a domain.*
2. *If D is a Smyth power domain, the domain $D \cup \{\top\}$ satisfies $V + \top = V$ for any element e. Because of that, $\top$ can be identified a posteriori with the empty sum. In that case, we denote the top element by $\mathbf{0}$.*

Sketch of proof.

1. Every directed subset of $D \cup \{\top\}$ either contains $\top$, in which case its limit is $\top$, or it doesn't contain $\top$, in which case it has a limit in D. Proving bounded completeness is similar. It is also easy to prove that $\top$ is a compact element and that the order is algebraic.
2. It follows from point (3) of Proposition 3.1: we have $\big(V \cup \{\top\}\big)^\uparrow = V^\uparrow$.

Definition 3.3.

1. The domain of non deterministic values $\mathcal{A}$ is obtained from $\mathcal{V}$ by the Smyth power domain construction and the addition of a greatest element $\mathbf{0}$.
2. We extend the sum operation by allowing it to appear under constructors and quotienting by (multi) linearity:
   - $\mathtt{C}\big(\sum_i v_i\big) = \sum_i \mathtt{C} v_i$
   - $\{\dots; \mathtt{D} = \sum_i t_i; \dots\} = \sum_i \{\dots; \mathtt{D} = t_i; \dots\}$.

(3) An element of $\mathcal{A}$ (a sum of elements of $\mathcal{V}$) is total when all its summands are total (cf. Definition 2.12 and Lemma 1.9). In particular, $\mathbf{0}$, the empty sum, is total.

The following follows directly from Lemma 1.9 and Proposition 3.1. 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 3.4. *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 important properties of $\mathcal{A}$. They are a direct consequence of Proposition 3.1.

Corollary 3.5. *The compact elements of $\mathcal{A}$ are inductively generated by*

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

The order satisfies

(1) *if $u \leq v$ in $\mathcal{V}$ then $u \leq v$ in $\mathcal{A}$,*
(2) *$\sum_i u_i \leq \sum_j v_j$ iff $\forall j, \exists i, u_i \leq v_j$,*
(3) *$+$ is commutative, associative and idempotent, with neutral element $\mathbf{0}$,*
(4) *$u + v$ is the greatest lower bound of u and v and $\mathbf{0}$ is the greatest element,*
(5) *constructors are (multi) linear.*

3.2. **Recursion and Fixed Points.**

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

A formula for fixed points. Whenever $\varphi : D \to D$ 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 equal to (Kleene theorem)

$$\mathsf{fix}(\varphi, b) \quad = \quad \bigsqcup_{n \geq 0}^{\uparrow} \varphi^n(b)$$

In our case, we are interested in the fixed point of an operator from $[\mathcal{A} \to \mathcal{A}]$ 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:

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

The fixed points we are computing are thus of the form

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

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

We start by interpreting recursive definitions as the untyped sum of their clauses. When clauses are disjoint, this is exactly the same as the standard interpretation but since overlapping clauses are common, this usually introduces non-determinism. A notion of error appears naturally: it is used when trying to apply a clause to a non-matching value.

Definition 3.6. Given a recursive definition for $\mathtt{f}$ and an environment ρ for all other functions, define the **n**on-**d**e**t**erministic semantics $\Theta^{\mathrm{ndt}}_{\rho,\mathtt{f}} : [\mathcal{A} \to \mathcal{A}] \to [\mathcal{A} \to \mathcal{A}]$ as follows. Suppose $f : \mathcal{A} \to \mathcal{A}$,

- $\Theta^{\mathrm{ndt}}_{\rho,\mathtt{f}}(f)\big(\sum v\big) = \sum \Theta^{\mathrm{ndt}}_{\rho,\mathtt{f}}(f)(v)$.

- For $v \in \mathcal{V}$, i.e. a value without sum, define (the unifier $[p := v]$ was defined on page 11)

$$\Theta^{\mathrm{ndt}}_{\rho,\mathtt{f}}(f)(v) \quad = \sum_{\mathtt{f}\ p\ =\ u} [\![u[p := v]]\!]_{\rho,\mathtt{f}:=f}$$

  where the sum ranges over all clauses of the definition. Because v can be *any* value, $u[p := v]$ is extended to give $\mathbf{0}$ when $[p := v]$ is undefined. In particular, if no clause matches v, we have $\Theta^{\mathrm{ndt}}_{\rho,\mathtt{f}}(f)(v) = \mathbf{0}$.

To compare the standard semantics $\Theta^{\mathrm{std}}_{\rho,\mathtt{f}}$ with the untyped, non deterministic semantics $\Theta^{\mathrm{ndt}}_{\rho,\mathtt{f}}$, we need to lift typed functions from an environment to untyped ones. This is done by making typed functions:

- linear with respect to non-deterministic sums,
- return $\mathbf{0}$ when applied outside of their domain.

Lemma 3.7. *For a definition of* $\mathtt{f}$ *of type* $A \to B$*, and an environment* ρ*, let* $\widehat{\Theta^{\mathrm{std}}_{\rho,\mathtt{f}}}$ *be the lifting of the usual semantics of* $\mathtt{f}$*.*

(1) *if* ρ *is a typed environment and* $f : [\![A]\!] \to [\![B]\!]$*, we have* $\Theta^{\mathrm{std}}_{\rho,\mathtt{f}}(f) = \widehat{\Theta^{\mathrm{std}}_{\widehat{\rho},\mathtt{f}}}(\widehat{f}) \upharpoonright [\![A]\!]$*,*
(2) *if* ρ *is a typed environment, we have* $\mathsf{fix}(\Theta^{\mathrm{std}}_{\rho,\mathtt{f}}) = \mathsf{fix}\big(\widehat{\Theta^{\mathrm{std}}_{\widehat{\rho},\mathtt{f}}}\big) \upharpoonright [\![A]\!]$*,*
(3) *if* ρ *is a typed environment and if* $\mathsf{fix}\big(\widehat{\Theta^{\mathrm{std}}_{\widehat{\rho},\mathtt{f}}}\big)$ *is total, then so is* $\mathsf{fix}(\Theta^{\mathrm{std}}_{\rho,\mathtt{f}})$*.*

Proof. The first point is straightforward as the lifting of a function gives the same (typed) result as the unlifted function on typed values. The second point follows from Kleene's formula for computing the fixed point: each ${\Theta^{\mathrm{std}}_{\rho,\mathtt{f}}}^n(\Omega)$ is equal to $\widehat{\Theta^{\mathrm{std}}_{\widehat{\rho},\mathtt{f}}}^n(\Omega) \upharpoonright A$, and their limits are thus equal. The third point follows from the fact that outside their types, lifting take the value $\mathbf{0}$, which is total. ☐

To compare the two semantics, we need an auxiliary lemma.

Lemma 3.8.
(1) *if* $\Omega \leq \theta(\Omega)$ *and* $\theta \leq \phi$ *in* $[\mathcal{A} \to \mathcal{A}] \to [\mathcal{A} \to \mathcal{A}]$*, then* $\mathsf{fix}(\theta) \leq \mathsf{fix}(\phi)$ *in* $\mathcal{A} \to \mathcal{A}$*;*
(2) *if* $f \leq g$ *in* $\mathcal{A} \to \mathcal{A}$ *and* f *is total, then so is* g*.*

Proof. The first point follows from Kleene's formula for the fixed point, and the second point follows from Lemma 3.4. ☐

Lemma 3.9. *Given a recursive definition for* $\mathtt{f}$ *and environment* ρ *satisfying* $\rho(\mathtt{g}) \geq \Omega$ *for all function names* $\mathtt{g}$*, we have*
(1) $\Omega \leq \Theta^{\mathrm{ndt}}_{\rho,\mathtt{f}}(\Omega)$*,*
(2) $\Theta^{\mathrm{ndt}}_{\rho,\mathtt{f}}(f) \leq \widehat{\Theta^{\mathrm{std}}_{\rho,\mathtt{f}}}(f)$ *for any function* $f : \mathcal{A} \to \mathcal{A}$*,*
(3) *If* $\mathsf{fix}(\Theta^{\mathrm{ndt}}_{\rho,\mathtt{f}})$ *is total on* $[\mathcal{A} \to \mathcal{A}]$ *then* $\mathsf{fix}\big(\widehat{\Theta^{\mathrm{std}}_{\rho,\mathtt{f}}}\big)$ *is total on* $[\mathcal{A} \to \mathcal{A}]$*.*

Proof. The first point is straightforward and the third point is a direct consequence of Lemma 3.8. For the second point, the only places where $\widehat{\Theta^{\mathrm{std}}_{\rho,\mathtt{f}}}$ and $\Theta^{\mathrm{ndt}}_{\rho,\mathtt{f}}$ differ are

- for values of the appropriate type, $\widehat{\Theta^{\mathrm{std}}_{\rho,\mathtt{f}}}$ only takes uses the first matching clause while $\Theta^{\mathrm{ndt}}_{\rho,\mathtt{f}}$ takes the sum over all clauses,
- for values outside the appropriate type, $\widehat{\Theta^{\mathrm{std}}_{\rho,\mathtt{f}}}$ returns $\mathbf{0}$.

In both cases, $\widehat{\Theta^{\mathrm{std}}_{\rho,\mathtt{f}}}$ is greater than $\Theta^{\mathrm{ndt}}_{\rho,\mathtt{f}}$. ☐

As a corollary of the previous lemmas, we can forget about typing and the order of clauses in a definition and simply try to show totality of $\Theta^{\mathrm{ndt}}_{\rho,\mathtt{f}}$.

Corollary 3.10. *If* $\mathsf{fix}(\Theta^{\mathrm{ndt}}_{\rho,\mathtt{f}})$ *is total, then so is* $\mathsf{fix}(\Theta^{\mathrm{std}}_{\rho,\mathtt{f}})$*.*

3.3. **Operators.**

3.3.1. *Terms.* The operators Θ^{ndt} just defined belong to the big Scott-domain $[\mathcal{A} \to \mathcal{A}] \to [\mathcal{A} \to \mathcal{A}]$. This section introduces a simple inductively generated language containing all the $\Theta^{\mathrm{ndt}}_{\rho,\mathtt{f}}$ from the previous section. The idea is to enrich the language of non-deterministic terms with projections "$\mathtt{.D}$" and partial match "$\mathtt{C}^{-}$".

Definition 3.11. $\mathcal{F}_0$ is the set of terms inductively generated from

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

where $\{t_1, \dots, t_n\}$ is a finite set of terms, each $\mathtt{x}$ is a *parameter name* and each $\mathtt{f}$ belongs to a finite set of *function names*. As previously, $\mathtt{C}$ and $\mathtt{D}$ 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. We usually write Ω for $\Omega\{\}$, Ωt for $\Omega\{t\}$ and $\Omega\{T\}$ for arbitrary finite sets.

Because of the interaction projections, partial matches, constructors and records, the order on $\mathcal{F}_0$ is far from trivial.

Definition 3.12. The order $\leq$ on $\mathcal{F}_0$ is generated from
- contextuality: if $C \in \mathcal{F}_0$ is a context, then $t_1 \leq t_2 \implies C[\mathtt{y} := t_1] \leq C[\mathtt{y} := t_2]$,
- $\mathbf{0}$ is the greatest element: $\forall t \in \mathcal{F}_0, t \leq \mathbf{0}$,
- $\Omega t \leq t$,
- $\Omega\{S\} \leq \Omega\{T\}$ if $S \subseteq T$,

together with the following inequalities ("$u \approx v$" means "$u \leq v$ and $v \leq u$"):

$$
(*)\left\{
\begin{array}{lrcll}
(0) & \mathtt{C}\mathbf{0} & \approx & \mathbf{0} & \\
(0) & \mathtt{C}^{-}\mathbf{0} & \approx & \mathbf{0} & \\
(0) & \mathtt{.D}\mathbf{0} & \approx & \mathbf{0} & \\
(0) & \{\dots; \mathtt{D} \approx \mathbf{0}; \dots\} & \approx & \mathbf{0} & \\
(0) & \mathtt{f}\,\mathbf{0} & \approx & \mathbf{0} & \\
(1) & \mathtt{C}^{-}\mathtt{C}t & \approx & t & \\
(1) & \mathtt{.D}_{i_0}\{\dots; \mathtt{D}_i = t_i; \dots\} & \geq & t_{i_0} & \\
(2) & \mathtt{C}^{-}\{\dots\} & \approx & \mathbf{0} & \\
(2) & \mathtt{.DC}t & \approx & \mathbf{0} & \\
(2) & \mathtt{.D}\{\dots\} & \approx & \mathbf{0} & \text{if the record has no field } \mathtt{D} \\
(2) & \mathtt{C}^{-}\mathtt{C}'t & \approx & \mathbf{0} & \text{if } \mathtt{C} \neq \mathtt{C}' \\
(3) & \mathtt{C}^{-}\Omega\{T\} & \approx & \Omega\{T\} & \\
(3) & \mathtt{.D}\Omega\{T\} & \approx & \Omega\{T\} & \\
(3) & \Omega\{\mathtt{C}t, T\} & \approx & \Omega\{t, T\} & \\
(3) & \Omega\{\{\mathtt{D}_1 = t_1; \dots; \mathtt{D}_k = t_k\}, T\} & \approx & \Omega\{t_1, \dots, t_k, T\} & \\
(3) & \Omega\{\mathbf{0}, T\} & \approx & \mathbf{0} & \\
(3) & \Omega\{\mathtt{x}, T\} & \approx & \Omega\{T\} & \\
(3) & \Omega\{\Omega\{S\}, T\} & \approx & \Omega\{S, T\} &
\end{array}
\right.
$$

Any $t \in \mathcal{F}_0$ different from $\mathbf{0}$ is called a *simple term.*

Group (0) of inequalities deals with propagation of errors and groups (1) and (2) correspond to the intended operational semantics of the language. Even in the case of typed language like `chariot`,

errors introduced in group (2) are necessary as splitting a definition into the sum of its clauses makes it necessary to deal with applying a clause to a non-matching value. Group (3) is more complex and will ultimately by justified by Definition 3.18 and Lemma 3.19. To understand those, it helps to keep in mind the following.

- A term t is a value depending on
  - the variable $\mathtt{x}$,
  - the recursive function $\mathtt{f}$ being defined,
  - some other functions $\mathtt{g}$, $\mathtt{h}$ that should be given by the environment.

  Such a term will be interpreted (Definition 3.22) by an operator from $[\mathcal{A} \to \mathcal{A}]$ to itself.
- Constructors $\mathtt{C}t$ and $\{\mathtt{D}_1 = t_1; \dots; \mathtt{D}_k = t_k\}$ construct values directly.
- Destructors $\mathtt{C}^{-}t$ and $.\mathtt{D}t$ destruct values by:
  - doing a partial match "**match** t **with** $\mathtt{C}\ u$ => u" *that may fail*,
  - projecting a structure on a field.
- $\mathbf{0}$ represents an error.
- $\Omega\{T\}$ is equal to $\mathbf{0}$ whenever *some* $t \in T$ is equal to $\mathbf{0}$, or if $\mathtt{x}$ itself is equal to $\mathbf{0}$.

Before showing that this doesn't collapse to a trivial pre-order, here are some simple consequences.

Lemma 3.13. *We have:*

- $\Omega\{T\} \leq t$ *iff* $\Omega\{T\} \leq \Omega t$,
- Ω *is the smallest element,*
- $\Omega\{\mathtt{C}^{-}t, T\} \geq \Omega\{t, T\}$,
- $\Omega\{.\mathtt{D}t, T\} \geq \Omega\{t, T\}$,

Proof.

- Suppose $\Omega\{T\} \leq t$, we have $\Omega\Omega\{T\} \leq \Omega t$ by contextuality, and since $\Omega\Omega\{T\} \approx \Omega\{T\}$, we have $\Omega\{T\} \leq \Omega t$. The converse is a consequence of transitivity and the fact that $\Omega t \leq t$.
- By the previous point, proving that $\Omega \leq t$ is equivalent to proving that $\Omega \leq \Omega t$, which follows from the fact that $\{\} \subseteq \{t\}$.
- $\Omega\{\mathtt{C}^{-}t, T\} \geq \Omega\{\mathtt{C}^{-}\Omega t, T\}$ by contextuality, and since $\mathtt{C}^{-}\Omega t \approx \Omega t$, we have $\Omega\{\mathtt{C}^{-}t, T\} \geq \Omega\{\Omega t, T\}$.
- The last point is proved similarly. □

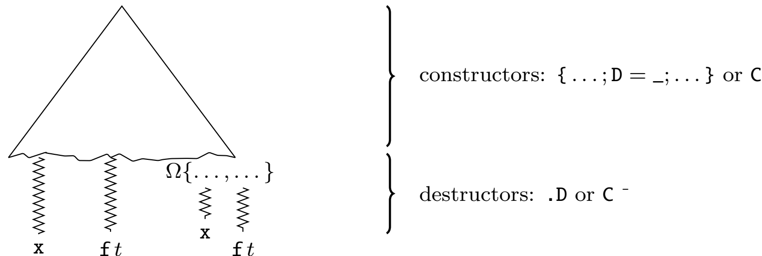
The (in)equalities $(*)$ give rise to a notion of reduction, written $\to$, when oriented from left to right.

Lemma 3.14.

(1) *The reduction* $\to$ *is strongly normalizing.*
(2) *Normal forms different from* $\mathbf{0}$ *are given by the grammar*

$$
\begin{array}{lcl}
t & ::= & \mathtt{C}^p t \quad | \quad \{\mathtt{D}_1 = t_1; \dots; \mathtt{D}_n = t_n\}^p \quad | \quad \Omega\{\delta_1, \dots, \delta_n\} \quad | \quad \delta \\
\delta & ::= & \mathtt{C}^{p^-}\delta \quad | \quad .\mathtt{D}^p\delta \quad | \quad \mathtt{x} \quad | \quad \mathtt{f}\,t
\end{array}
$$

A normal form thus looks like, where each t, t_1, …, t_n is itself in normal form



Proof of Lemma 3.14. Reduction is strongly normalizing because the size of the term decreases. For the second point, all terms generated by the grammar are obviously in normal form. It is also straightforward to check that all non-$\mathbf{0}$ normal forms are generated by the grammar because:

- they cannot contain $\mathbf{0}$,
- there cannot be a destructor ($\mathtt{C}^-$ or $\mathtt{.D}$) directly above a constructor ($\mathtt{C}$ or $\{\dots\}$),
- there cannot be a destructor ($\mathtt{C}^-$ or $\mathtt{.D}$) directly above Ω, nor a constructor ($\mathtt{C}$ or $\{\dots\}$) directly below Ω. □

This reduction isn't confluent because of the critical pairs of the form "$\mathtt{.D}_1\{\mathtt{D}_1 = t; \mathtt{D}_2 = \mathbf{0}; \dots\}$" that reduces both to $\mathbf{0}$ and to t. It is however "almost" confluent in that a term can have at most one non-$\mathbf{0}$ normal form. The "innermost" reduction strategy will always pick the $\mathbf{0}$ normal form if it exists, and thus gives the greatest normal form of a term.

Definition 3.15. We write $\mathsf{nf}(t)$ for the normal form of a term according to the "innermost / leftmost" reduction strategy.

Lemma 3.16.

(1) *$t \approx \mathbf{0}$ iff $t \geq \mathbf{0}$ iff there is a reduction $t \to^* \mathbf{0}$.*
(2) *Reduction is confluent on terms different (using $\not\approx$) from $\mathbf{0}$.*
(3) *If $t \approx \mathbf{0}$, innermost reduction will give $\mathbf{0}$.*

Proof sketch. 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 $t \to t'$ implies $t \geq t'$. For the left to right implication, it is enough to check that for all the inequalities $t_1 \leq t_2$ (or $t_1 \approx t_2$) generating the order, we have "$t_1 = \mathbf{0}$ implies $t_2 \to^* \mathbf{0}$". This is obvious for all the generating (in)equalities except contextuality. Suppose $t_1 \leq t_2$ with $t = \mathtt{C}[\mathtt{x} := t_2] \geq \mathbf{0} = C[\mathtt{x} := t_1]$. The only possible context giving $C[\mathtt{x} := t_2] = \mathbf{0}$ is is $C = \mathbf{0}$, which implies that $t = \mathbf{0}$.

Because of the first point, terms different from $\mathbf{0}$ are closed under reduction. Reduction on those terms has the following critical pairs:

$$\begin{array}{ccc}
\mathtt{C}^-\Omega\{\mathtt{C}'t, T\} & \mathtt{C}^-\Omega\{\{\mathtt{D}_1 = t_1; \dots\}, T\} & \mathtt{C}^-\Omega\{\Omega\{T\}, T'\} \\
\mathtt{.D}\Omega\{\mathtt{C}t, T\} & \mathtt{.D}\Omega\{\{\mathtt{D}_1 = t_1; \dots\}, T\} & \mathtt{.D}\Omega\{\Omega\{T\}, T'\} \\
\Omega\{\Omega\{\mathtt{C}t, T\}, T'\} & \Omega\{\Omega\{\{\mathtt{D}_1 = t_1; \dots\}, T\}, T'\} & \Omega\{\Omega\{\Omega\{T_1\}, T_2\}, T_3\}
\end{array}$$

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

The third point isn't used in the paper, so the proof is skipped. □

The following lemma characterizes the order on normal forms.

Lemma 3.17. *Given s in normal form, we have $s \leq t < \mathbf{0} \implies s \leq \mathsf{nf}(t)$. More precisely:*

(1) *If $s = \mathtt{f}\,s'$, then $\mathsf{nf}(t) = \mathtt{f}\,t'$ with $s' \leq t'$.*
(2) *If $s = \mathtt{C}\,s'$, then $\mathsf{nf}(t) = \mathtt{C}\,t'$ with $s' \leq t'$.*
(3) *If $s = \{\mathtt{D}_1 = s_1; \dots\}$, then $\mathsf{nf}(t) = \{\mathtt{D}_1 = t_1; \dots\}$ with $\forall i, s_i \leq t_i$.*
(4) *If $s = \mathtt{.D}\,s'$ (resp. $s = \mathtt{C}^-s'$), then $\mathsf{nf}(t) = \mathtt{.D}\,t'$ (resp. $t = \mathtt{C}^-t'$) with $s' \leq t'$.*
(5) *If $s = \mathtt{x}$, then $\mathsf{nf}(t) = \mathtt{x}$.*
(6) *If $s = \Omega\{S\}$, then $\mathsf{nf}(\Omega\,t) = \Omega\{T\}$ with for all $s \in S$, there is a sequence of destructors δ s.t. $s \leq t$ for some $\delta\,t \in T$.*

Proof. The proof is by induction on the term s and the proof that $s \leq t$. The order is generated from reflexivity, transitivity and the inequalities given in Definition 3.12. Reflexivity can be ignored: if $s = t$, then everything holds trivially. Inequalities from group (*) can be ignored as well because if s is in normal form, they can only generate inequalities of the form $s \leq t$ with $t \to s$. If means that $\mathsf{nf}(t) = s$ and everything holds trivially. Because we suppose $s \leq t < \mathbf{0}$, inequality $t \leq \mathbf{0}$ can be ignored as well. The proof thus only needs to look at transitivity, contextuality, $\Omega\,t \leq t$ and $\Omega\{S\} \leq \Omega\{T\}$ if $S \subseteq T$.

(1) If $s = \mathtt{f}\,s'$, we inspect the proof that $\mathtt{f}\,s' \leq t$.
   - If $\mathtt{f}\,s' \leq t$ comes from transitivity, we have some u s.t. $\mathtt{f}\,s' \leq u$ and $u \leq t$. Since $u \approx \mathbf{0}$ implies that $t \approx \mathbf{0}$, we can suppose that $u \not\approx \mathbf{0}$. We can thus apply the induction hypothesis

on $\mathtt{f}\,s' \leq u$. We thus have that $\mathsf{nf}(u) = \mathtt{f}\,u'$ with $s' \leq u'$. Since $\mathtt{f}\,u' = \mathsf{nf}(u) \leq u$, we also have that $\mathtt{f}\,u' \leq t$, and we can apply the induction hypothesis: $\mathsf{nf}(t)$ is of the form $\mathtt{f}\,t'$ with $u' \leq t'$. By transitivity, $s' \leq t'$.

- If $\mathtt{f}\,s' \leq t$ was proved by contextuality, there is a context $\mathtt{f}\,C$ and some terms $s'' \leq t''$ s.t. $\mathtt{f}\,s' = \mathtt{f}\,C[\mathtt{y} := s'']$ and $t = \mathtt{f}\,C[\mathtt{y} := t'']$. We have $s' = C[\mathtt{y} := s''] \leq C[\mathtt{y} := t'']$. Writing $t' = C[\mathtt{y} := t'']$, we get that $s' \leq \mathsf{nf}(t')$ by induction. Because $\mathsf{nf}(\mathtt{f}\,t') = \mathtt{f}\,\mathsf{nf}(t')$, we conclude that $\mathsf{nf}(t) = \mathtt{f}\,t'$ with $s' \leq t'$.

(2) The case $s = \mathtt{C}\,s'$ is treated in exactly the same way.
(3) The case of $s = \{\mathtt{D}_1 = s_1; \dots\}$ is treated in exactly the same way.
(4) The case of $s = \mathtt{.D}\,s'$ is very similar. The only difference is that when dealing with contextuality, we need to prove that $\mathsf{nf}(\mathtt{.D}\,t') = \mathtt{.D}\,\mathsf{nf}(t')$. (This is false in general.) Since s was in normal form, s' can only start with $\mathtt{f}$, a destructor or a variable. By induction, it implies that t' can only start with $\mathtt{f}$, a destructor or a variable. In those cases, we do have $\mathsf{nf}(\mathtt{.D}\,t') = \mathtt{.D}\,\mathsf{nf}(t')$.
(5) The case $s = \mathtt{x}$ is straightforward.
(6) If $s = \Omega\{S\} \leq t$:
- If $\Omega\{S\} \leq t$ comes from transitivity $\Omega\{S\} \leq u \leq t$. We can apply the induction hypothesis to $\Omega\{S\} \leq u$: $\mathsf{nf}(\Omega\,u)$ is of the form $\Omega\{U\}$ as in the lemma. Because $\Omega\{U\} \leq \Omega u \leq u \leq t$, we can apply the induction hypothesis: $\mathsf{nf}(\Omega\,t)$ is of the form $\Omega\{T\}$ as in the lemma and we can conclude by transitivity of $\leq$.
- If $\Omega\{S\} \leq t$ comes from contextuality, the context is the form $\Omega\{S_\mathtt{y}\}$ and there are $s'' \leq t''$ s.t. $s = \Omega\{S_\mathtt{y}\}[\mathtt{y} := s'']$ and $t = \Omega\{S_\mathtt{y}\}[\mathtt{y} := t'']$. For each $s' \in S_\mathtt{y}$, $s'[\mathtt{y} := s''] \leq s'[\mathtt{y} := t'']$. Because s is in normal form and because $s'[\mathtt{y} := s'']$ is a subterm of s, it is also in normal form and can only start with a destructor, a function or a variable. By induction hypothesis, we thus have $s'[\mathtt{y} := s''] \leq \mathsf{nf}(s'[\mathtt{y} := t''])$ and $\mathsf{nf}(s'[\mathtt{y} := t''])$ can only start with a destructor, a function or a variable. We've just shown that each $s' \in S$ is less than some $t' \in T$.
- When $s \leq t$ comes from $\Omega\,s \leq s$, because s is in normal form, the result is obvious.
- If $\Omega\{S\} \leq t$ comes $\Omega\,t \leq t$ or $\Omega\{S\} \leq \Omega\{T\}$ with $S \subseteq T$, we can conclude directly. □

When both s and t are in normal form, all implications (1)–(6) are in fact equivalences and this gives a simple recursive way to check that $s \leq t$, which is important when implementing the totality checker. The only case that might not be obvious is how to check that $\Omega\{S\} \leq t$ but this follows from Lemma 3.13.

The constructions "$\mathtt{.D}$" and "$\mathtt{C}^-$" have natural interpretations as continuous functions:

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

and

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

This makes it easy to define the semantics of any element of $\mathcal{F}_0$ as a function depending on $\mathtt{x}$.

Definition 3.18. Given an environment ρ associating functions to names, and $t \neq \mathbf{0}$ an element of $\mathcal{F}_0$, we define $\{\!|t|\!\}_\rho : \mathcal{A} \to \mathcal{A}$ with

(1) $\{\!|\mathtt{C}t|\!\}_\rho(v) = \mathtt{C}\big(\{\!|t|\!\}_\rho(v)\big)$,

(2) $\{\!|\{\dots;\mathtt{D}_i = t_i;\dots\}|\!\}_\rho(v) = \begin{cases} \mathbf{0} & \text{if } v = \mathbf{0}, \text{ or if } \{\!|t|\!\}_\rho = \mathbf{0} \text{ for some } t \in T \\ \{\dots;\mathtt{D}_i = \{\!|t_i|\!\}_\rho(v);\dots\} & \text{otherwise,} \end{cases}$

(3) $\{\!\!\{\Omega\{T\}\}\!\!\}_\rho(v) = \Omega(v)\begin{cases}\mathbf{0} & \text{if } v=\mathbf{0}\text{, or if } \{\!\!\{t\}\!\!\}_\rho = \mathbf{0} \text{ for some } t\in T\\ \bot & \text{otherwise,}\end{cases}$
(4) $\{\!\!\{\mathtt{C}^- t\}\!\!\}_\rho(v) = \mathtt{C}^-(\{\!\!\{t\}\!\!\}_\rho(v))$,
(5) $\{\!\!\{\mathtt{.D}t\}\!\!\}_\rho(v) = \mathtt{.D}\big(\{\!\!\{t\}\!\!\}_\rho(v)\big)$,
(6) $\{\!\!\{\mathtt{x}\}\!\!\}_\rho(v) = v$,
(7) $\{\!\!\{\mathbf{0}\}\!\!\}_\rho(v) = \mathbf{0}$,
(8) $\{\!\!\{\mathtt{g}\ t\}\!\!\}_\rho(v) = \rho(\mathtt{g})\big(\{\!\!\{t\}\!\!\}_\rho(v)\big)$.

The interpretation of $t \in \mathcal{F}_0$ needs to be bigger than Ω (Section 3.2). Because of that, we need to make sure it returns $\mathbf{0}$ on $\mathbf{0}$, *even when* $\mathtt{x}$ *doesn't appear in* t. This explains why clauses (2) and (3) are defined the way they are.

The order on $\mathcal{F}_0$ is compatible with the pointwise order on $\mathcal{A} \to \mathcal{A}$.

Lemma 3.19.
(1) *If* $t_1 \le t_2$*, then* $\{\!\!\{t_1\}\!\!\}_\rho \le \{\!\!\{t_2\}\!\!\}_\rho$*, and* $\{\!\!\{_\}\!\!\}$ *is compatible with* $\approx$.
(2) *If* $\rho(\mathtt{f})$ *is continuous for any* $\mathtt{f}$*, then* $\{\!\!\{t\}\!\!\}_\rho$ *is also continuous.*
(3) *If* $\rho(\mathtt{f}) \ge \Omega$ *(in* $[\mathcal{A} \to \mathcal{A}]$*) for all function names, then* $\{\!\!\{T\}\!\!\}_\rho \ge \Omega$.
(4) *For all terms* $t_1, t_2 \in \mathcal{F}_0$ *and environment* ρ*, we have* $\big\{\!\!\big\{t_1[\mathtt{x} := t_2]\big\}\!\!\big\}_\rho = \{\!\!\{t_1\}\!\!\}_\rho \circ \{\!\!\{t_2\}\!\!\}_\rho$.

Sketch of proof. Checking the first points amounts to checking that all inequations from Definition 3.11 hold semantically in $[\mathcal{A} \to \mathcal{A}]$. This is immediate. The functions $\mathtt{C}^-$, $\mathtt{.D}$ and Ω are easily shown continuous, $[\![t]\!]_\rho$ is continuous as a composition of continuous functions. Points (3) and (4) are proved by immediate induction. □

Definition 3.20.
(1) We quotient $\mathcal{F}_0$ by $\approx$,
(2) $\mathcal{F}$ is defined as the Smyth power domain on the ideal completion of $\mathcal{F}_0$.

Like in $\mathcal{A}$, the term $\mathbf{0}$ ends up being a greatest element in $\mathcal{F}$, and like in Lemma 3.2, it can be identified *a posteriori* with the empty sum. The ideal completion of $\mathcal{F}_0$ introduces infinite elements like $\mathtt{CCCC}\cdots = \bigsqcup^\uparrow\{\Omega, \mathtt{C}\Omega, \mathtt{CC}\Omega, \dots\}$ but infinite branches of destructors like $\mathtt{.D.D.D}\cdots = \bigsqcup^\uparrow\{\Omega, \mathtt{.D}\Omega, \mathtt{.D.D}\Omega, \dots\}$ are really finite values equal to Ω.

We can now extend the semantics of $\mathcal{F}_0$ to the whole $\mathcal{F}$.

Definition 3.21. The semantics $\{\!\!\{_\}\!\!\}$ is extended to $\mathcal{F}$ by continuity and linearity.

From now on, we identify a special function name "$\mathtt{f}$" for the recursive function being defined and we assume given an environment ρ for all the other functions, which will be called $\mathtt{g}$, $\mathtt{h}$ etc..

Definition 3.22. Each $T \in \mathcal{A}$ gives rise to an operator $[\![T]\!]$ from $[\mathcal{A} \to \mathcal{A}]$ to itself:

$$[\![T]\!]_\rho \quad : \quad f \quad \mapsto \quad [\![T]\!]_\rho(f) = \{\!\!\{T\}\!\!\}_{\rho,\mathtt{f}:=f}$$

The typical environment ρ is constructed inductively from previous recursive definitions and will be omitted in the rest of the paper. A consequence of point (3) from Lemma 3.19 is that if $\rho(\mathtt{g}) \ge \Omega$ for all function names, then $[\![T]\!](\Omega) \ge \Omega$, and we can thus use the formula for the least fixed point of $[\![T]\!]_\rho$ greater than Ω.

3.3.2. *Composition.* Given T_1 and T_2, we can find a term representing the composition $[\![T_1]\!] \circ [\![T_2]\!]$. This is done by replacing each "$\mathtt{f}\,u$" inside T_1 by "$T_2[\mathtt{x} := u]$".

Definition 3.23. If $T_1, T_2 \in \mathcal{F}$, we define $T_1 \circ T_2$ by induction on T_1 where
- $(\mathtt{C}T_1) \circ T_2 = \mathtt{C}(T_1 \circ T_2)$,
- $\{\mathtt{D}_1 = t_1; \dots; \mathtt{D}_k = t_k\} \circ T_2 = \{\mathtt{D}_1 = t_1 \circ T_2; \dots; \mathtt{D}_k = t_k \circ T_2\}$,

- $(\mathtt{C}^{-}T_1) \circ T_2 = \mathtt{C}^{-}(T_1 \circ T_2)$,
- $(\mathtt{.D}\,T_1) \circ T_2 = \mathtt{.D}(T_1 \circ T_2)$,
- $\Omega\{\theta_1\} \circ T_2 = \Omega\{t \circ T_2 \mid t \in \theta_1\}$,
- $\mathtt{x} \circ T_2 = \mathtt{x}$,
- $(\mathtt{g}\,T_1) \circ T_2 = \mathtt{g}(T_1 \circ T_2)$,
- $(\mathtt{f}\,T_1) \circ T_2 = T_2[\mathtt{x} := T_1 \circ T_2]$.

This is extended by linearity and continuity (only on the left):

- $\big(\sum t_i\big) \circ T_2 = \sum (t_i \circ T_2)$,
- $\big(\bigsqcup_i^{\uparrow} t_i\big) \circ T_2 = \bigsqcup_i^{\uparrow} (t_i \circ T_2)$.

Only the "$\mathtt{f}$" case is interesting, and because of it, we sometimes use the less formal notation $T_1 \circ T_2 = T_1\big[\mathtt{f}\,t := T_2[\mathtt{x} := t]\big]$, or even $T_1[\mathtt{f} := T_2]$.

Lemma 3.24. *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 $t[\mathtt{x} := t_1] \circ t_2 = (t \circ t_2)[\mathtt{x} := t_1 \circ t_2]$ by induction on t:

- if $t = \mathbf{0}$ or $t = \mathtt{x}$, this is immediate,
- if t starts with a constructor, record, destructor, non-recursive function $\mathtt{g}$, or Ω, the result follows by induction,
- if $t = \mathtt{f}\,t'$, we have

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

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 $t = \mathbf{0}$ or $t = \mathtt{x}$, this is immediate,
- if t_1 starts with Ω, a constructor, record or destructor, it follows by induction,
- if $t_1 = \mathtt{f}\,t_1'$, we need to show that $t_2[\mathtt{x} := t_1 \circ t_2] \circ t_3 = (t_2 \circ t_3)[\mathtt{x} := t_1 \circ (t_2 \circ t_3)]$. By induction, it is enough to show that $t_2[\mathtt{x} := t_1 \circ t_2] \circ t_3 = (t_2 \circ t_3)[\mathtt{x} := (t_1 \circ t_2) \circ t_3)]$. This follows from the previous remark, with $t = t_2$, $t_2 = t_1 \circ t_2$, and $t_2 = t_3$. ∎

Lemma 3.25. *For any $T_1, T_2 \in \mathcal{F}$, $[\![T_1 \circ T_2]\!] = [\![T_1]\!] \circ [\![T_2]\!]$. If moreover, T_2 doesn't contain $\mathtt{f}$, we have $\{\!|T_1 \circ T_2|\!\} = [\![T_1]\!]\big(\{\!|T_2|\!\}\big)$. In particular,*

$$\{\!|\underbrace{T \circ \cdots \circ T}_{n} \circ \Omega|\!\} \quad = \quad [\![T]\!]^n\,(\Omega)$$

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

$$\begin{array}{rcll}
[\![(\mathtt{f}\,t_1) \circ t_2]\!]_\rho & = & f \mapsto \{\!|(\mathtt{f}\,t_1) \circ t_2|\!\}_{\rho,\mathtt{f}=f} & \text{definition of } [\![_]\!] \\
& = & f \mapsto \{\!|t_2[\mathtt{x} := t_1 \circ t_2]|\!\}_{\rho,\mathtt{f}=f} & \text{definition of } \circ \\
& = & f \mapsto \{\!|t_2|\!\}_{\rho,\mathtt{f}=f} \circ \{\!|t_1 \circ t_2|\!\}_{\rho,\mathtt{f}=f} & \text{point (4) of Lemma 3.19} \\
& = & f \mapsto \{\!|t_2|\!\}_{\rho,\mathtt{f}=f} \circ \{\!|t_1|\!\}_{\rho,\mathtt{f}=f} \circ \{\!|t_2|\!\}_{\rho,\mathtt{f}=f} & \text{induction} \\
& = & f \mapsto \{\!|t_2|\!\}_{\rho,\mathtt{f}=f} \circ \{\!|t_1|\!\}_{\rho,\mathtt{f}=\{\!|t_2|\!\}_{\rho,\mathtt{f}=f}} & \text{definition: replace } \mathtt{f} \text{ by } \{\!|t_2|\!\}_{\rho,\mathtt{f}=f} \\
& = & f \mapsto \{\!|\mathtt{f}\,t_1|\!\}_{\rho,\mathtt{f}=\{\!|t_2|\!\}_{\rho,\mathtt{f}=f}} & \text{definition of } \{\!|_|\!\} \\
& = & [\![\mathtt{f}\,t_1]\!]_\rho \circ [\![t_2]\!]_\rho & \text{definition of } [\![_]\!]
\end{array}$$

∎

3.4. **Interpreting Recursive Definitions.** We can now replace the operator $\Theta^{\mathrm{ndt}}_{\mathtt{f}}$ (defined on page 20) by an element of $\mathcal{F}$. Consider a single clause "$\mathtt{f}\,p = u$" of the recursive definition of `f`. The pattern p allows to "extract" some parts of the argument of `f` to be used in u. The clause

```
f {Fst = Node(n) ; Snd = Fork {Fst = t1 ; Snd = t2}} = ...
```

introduces 3 variables: `n`, `t1` and `t2`. If we call the parameter of `f` "`x`", the variable `n` can be obtained as $\mathtt{n} = \mathtt{Node}^{-}\ \mathtt{.Fst}\ \mathtt{x}$: we project `x` on field `Fst` and then remove the leading `Node` constructor. The other variables are obtained in similar ways:

- $\mathtt{t1} = \mathtt{.Fst}^{-}\ \mathtt{Fork}^{-}\ \mathtt{.Snd}\ \mathtt{x}$,
- $\mathtt{t2} = \mathtt{.Snd}^{-}\ \mathtt{Fork}^{-}\ \mathtt{.Snd}\ \mathtt{x}$.

Definition 3.26. Given a pattern p, define the substitution $[p := \mathtt{x}]$ as follows:

- $[\mathtt{y} := \mathtt{x}] = [\mathtt{y} := \mathtt{x}]$ where the substitution on the right is the usual substitution of variable `y` by variable `x`,
- $[\mathtt{C}p := \mathtt{x}] = \mathtt{C}^{-} \circ [p := \mathtt{x}]$,
- $[\{\dots; \mathtt{D}_i = p_i; \dots\} := \mathtt{x}] = \bigcup_i (\mathtt{.D}_i \circ [p_i := \mathtt{x}])$ (note that because patterns are linear, the substitutions don't overlap).

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

Lemma 3.27. *If $v \in \mathcal{V}$ matches p (Definition 1.11), then $[p := \mathtt{x}] \circ [\mathtt{x} := v](\mathtt{y}) \neq \mathbf{0}$ for all variables `y` in p. In that case, $[p := \mathtt{x}] \circ [\mathtt{x} := v] = [p := v]$, the unifier for p and v.*

Proof. The proof is a simple induction on the pair pattern / value. □

Any recursive definition can now be interpreted by an element of the domain $\mathcal{F}$ in the following way.

Definition 3.28. Given a recursive definition of `f`, define $T_{\mathtt{f}}$ with

$$T_{\mathtt{f}} = \sum_{\mathtt{f}\ p\ =\ u} u[p := \mathtt{x}]$$

where the sum ranges over all clauses defining `f`.

Lemma 3.27 doesn't say anything about the case when v doesn't match the pattern p. The reason is that any part of p that doesn't contain variables isn't recorded in $[p := \mathtt{x}]$. For example, the pattern $p = $ `Cons{Fst=Zero; Snd=y}` gives the substitution $[\mathtt{y} := \mathtt{.Snd}\ \mathtt{Cons}^{-}\ \mathtt{x}]$, and while p *doesn't* match the value $v = $ `Cons{Fst=Succ Zero; Snd=y}`, $[p := \mathtt{x}] \circ [\mathtt{x} := v]$ is the substitution $[\mathtt{x} := \mathtt{y}]$ which gives a non-$\mathbf{0}$ result on any non-empty list. Because of that, $[\![T_{\mathtt{f}}]\!]$ is slightly different from the non-deterministic semantics from Section 3.2.

Corollary 3.29. *For any environment ρ, we have $[\![T_{\mathtt{f}}]\!]_\rho \leq \Theta^{\mathrm{ndt}}_{\rho,\mathtt{f}}$.*

By Lemma 3.9, totality of $\mathsf{fix}([\![T_{\mathtt{f}}]\!])$ implies totality of $\mathsf{fix}(\Theta^{\mathrm{ndt}}_{\mathtt{f}})$. Because of Lemma 3.25 and Lemma 3.8, we have, using Kleene's formula

Corollary 3.30. *To check that $\mathsf{fix}(\Theta^{\mathrm{ndt}}_{\mathtt{f}})$ is total, is is enough to check that*

$$\bigsqcup_n^{\uparrow} \left[\!\!\left[\underbrace{T_{\mathtt{f}} \circ \dots T_{\mathtt{f}}}_{n} \circ \Omega \right]\!\!\right]$$

is total.

From now on, we will omit the semantics brackets and write $T_{\mathtt{f}}$ for $[\![T_{\mathtt{f}}]\!]_\rho$. We will also write $T^n_{\mathtt{f}}(\Omega)$ for $T_{\mathtt{f}} \circ \dots \circ T_{\mathtt{f}} \circ \Omega$ and the notation "$\mathsf{fix}(T_n)$" will refer to $\bigsqcup_n^{\uparrow} T_{\mathtt{f}} \circ \dots T_{\mathtt{f}} \circ \Omega$.

4. Call-Graphs and the Size-Change Principle

Except for a few minor differences, $T_{\mathtt{f}}$ can be thought of as a faithful representation of the original recursive definition. We now want to take an element of $\mathcal{F}$ and split it into simpler components in such a way that:

(1) Doing so reflects totality: showing totality of the simplification implies totality of $T_{\mathtt{f}}$, and thus of $\mathtt{f}$.
(2) Inspecting $\bigsqcup_n^{\uparrow} T_{\mathtt{f}}^n(\Omega)$ can be done in a computable way.

A lot of information about the function will be lost along the way, but since this isn't used to compute actual values of the function, this isn't a problem.

4.1. **Call-Graph.**

4.1.1. *Call Paths.* We index all the occurrences of $\mathtt{f}$ in T: if T is $\mathtt{ffx}$, we write $\mathtt{f}_1\mathtt{f}_2\mathtt{x}$. Substituting $\mathtt{f}$ by $\mathtt{g}+\mathtt{h}$ gives $\mathtt{ggx}+\mathtt{ghx}+\mathtt{hgx}+\mathtt{hhx}$, i.e. each occurrence of $\mathtt{f}$ is substituted either by $\mathtt{g}$ or $\mathtt{h}$. Since substituting a single occurrence of $\mathtt{f}$ is linear, substituting all of them is multilinear.

Lemma 4.1. *We have*

$$T \circ (t_1 + \cdots + t_n) \quad = \sum_{\sigma\,:\,\mathsf{occ}(\mathtt{f},T)\to\{t_1,\dots,t_n\}} T[\sigma]$$

where $\mathsf{occ}(\mathtt{f},T)$ *represents the set of occurrences of* $\mathtt{f}$ *in* T*, and the substitution occurs at the given occurrences. More precisely,* $T[\sigma] = T\big[\mathtt{f}_i\, t := \sigma(\mathtt{f}_i)[\mathtt{x} := t]\big]$ *as in Definition 3.23.*

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 $\mathtt{f}$ by one of the t_i,
- repeat $n-2$ times.

We extend this to infinite compositions.

Definition 4.2. Given $T = t_1+\cdots+t_n$ a sum of simple terms, a *path* for T is a sequence $(s_k,\sigma_k)_{k\geq 0}$ such that:

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

If some s_k doesn't contain any occurrence of $\mathtt{f}$, then all later s_{k+i} are equal to s_k.

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.

Lemma 4.3. *Suppose* $T = t_1 + \cdots + t_n$ *is a sum of simple terms, then*

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

Proof. We start by showing that the left-hand side is greater than the right-hand side, i.e. by showing that any simple term in the LHS is greater than some simple term on the RHS.[11] Let s be a simple term in $\mathsf{fix}(T) = \bigsqcup^{\uparrow} T^n(\Omega)$. 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 substitutes all occurrences of $\mathtt{f}$ in s by one of $t_1, \dots, t_n$.

[11] That's point (3) of Corollary 3.5.

As there are only finitely many possible substitutions from a given node, this forest is finitely branching. Because $T^n(\Omega) \leq \mathsf{fix}(T) \leq s$, each $T^n(\Omega)$ contains some term t such that $t(\Omega) \leq s$, this forest is also infinite . By König's lemma, it thus contains an infinite branch $s_0, s_1, \dots$. Because all $s_i(\Omega)$ are less than s by construction, this sequence satisfies all the properties of Definition 4.2 and its limit is less than s. We thus have

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

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 $T^k(\Omega)$. □

Corollary 4.4. *If ρ is a total environment and $\mathsf{fix}(T)$ is non-total, then there is a path (s_k) for T such that $\bigsqcup^{\uparrow} s_i(\Omega)$ is non total.*

Proof. If $\mathsf{fix}(T)$ is non total, then by the previous lemma, there is a path of T that is non total. □

4.1.2. *Call-Graph.* We will now make a drastic simplification of $T_{\mathtt{f}}$ by splitting each summand t of $T_{\mathtt{f}}$ into the sum of its recursive calls. By definition, `f` is total is, whenever $f(v) = w$, either w is total, or v isn't total. For each recursive call, we keep information about what is constructed above the call, and about the argument below the call. All other function calls, recursive or not, are replaced by Ω. Each recursive call occurs *from* a recursive function *to* a recursive function. Calls are thus labels inside a graph whose vertices are function names. As an illustration, consider the following ad-hoc clause,

```
| f { D₁ = y; D₂ = z }  =  C (f (f y)
```

As described in the previous section, it is interpreted by

$$\mathtt{C}\big(\underline{\mathtt{f}\big(\underline{\mathtt{f}\,.\mathtt{D}_1\,\mathtt{x}}\big)}\big)$$

and contains 2 recursive calls (underlined).

It is clear that this clause adds a `C` constructor to the result, and that this occurs just above the leftmost recursive call. It is also clear that the rightmost recursive call uses part of the initial argument. The rightmost call isnt' guarded and the argument to the leftmost call isn't directly constructed from parts of the original argument. This clause gives rise to 2 calls:

- $\mathtt{f}\ \mathtt{x} \mapsto \mathtt{C}\ \mathtt{f}\ (\Omega\,.\mathtt{D}_1\,\mathtt{x})$ for the leftmost call:
  - this call is guarded by `C`,
  - we know nothing on the arguments of `f`, except that it returns $\mathbf{0}$ when $.\mathtt{D}_1, \mathtt{x}$ is $\mathbf{0}$.
- $\mathtt{f}\ \mathtt{x} \mapsto \mathtt{C}\ \Omega\ \mathtt{f}\ (.\mathtt{D}_1\,\mathtt{x})$ for the rightmost call:
  - besides a topmost `C`, nothing is known about the result above the call,
  - the argument of `f` is built from part of the initial argument.

Definition 4.5. Let $T \in \mathcal{F}$, the *call-graph of T*, $G(T)$, is defined inductively as follows:

(1) $G\big(\sum t\big) = \sum G(t)$,
(2) $G(\mathtt{f}\,t) = \mathtt{f}\big(t^\Omega\big) + \Omega G(t)$ *where t^Ω is equal to t where all function calls have been replaced by Ω,*
(3) $G(\mathtt{x}) = \mathbf{0}$,
(4) $G(\mathtt{g}\,t) = \Omega G(t)$,
(5) $G(\mathtt{C}^p\,t) = \mathtt{C}^p\,G(t)$,
(6) $G\big(\{\dots; \mathtt{D}_i{}^p = t_i; \dots\}\big) = \sum_i \{\mathtt{D}_i = G(t_i)\}^p$,
(7) $G(\mathtt{C}^{-}t) = \mathtt{C}^{-}\,G(t)$,
(8) $G(.\mathtt{D}\,t) = .\mathtt{D}\,G(t)$.

For example, for $t = $ `C{Fst=f (C- x); Snd = f (C (f x))}`, we obtain

$$G(t) \quad = \quad \mathtt{C\,\{Fst = f(C^- x)\}} \; + \; \mathtt{C\,\{Snd = f(C\,\Omega\,x)\}} \; + \; \mathtt{C\,\{Snd = \Omega\,C\,f\,x)\}}$$

In general, T and $G(T)$ are not comparable but the construction reflects totality.

Proposition 4.6. *If* $\mathsf{fix}(G(T))$ *is total then so is* $\mathsf{fix}(T)$.

Proof. Suppose $\mathsf{fix}(T)$ is non-total. By Corollary 4.4, it implies there is a path (s_k) of T and a total element $u \in \mathcal{V}$ with $\bigsqcup^{\uparrow} s_i(\Omega)(u) \in \mathcal{V}$ 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 $\mathbf{0}$. Call this branch β.

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

(1) the only occurrence of $\mathtt{f}$ in $s_0 = \mathtt{f\,x}$ is indexed by the empty list
(2) occurrences of $\mathtt{f}$ in s_1 are indexed using the list containing the index of this occurrence in $T_{\mathtt{f}}$.
(3) given $k > 1$, we replace the substitution σ_k by $\widehat{\sigma}_k$, defined with

$$\widehat{\sigma}_k(\mathtt{f}_{L,i}) \quad = \quad \sigma_k(\mathtt{f}_i) \circ [\ldots, \mathtt{f}_j := \mathtt{f}_{L,i,j}, \ldots]$$

In other words, we replace each $\mathtt{f}_{L,i}$ by $\sigma_k(\mathtt{f}_i)$, *while keeping* L, i *in front of all the occurrences of* $\mathtt{f}$ *that are introduced.*

As an example, let $T = \mathtt{C}_1\mathtt{f}_1\mathtt{x} + \mathtt{C}_2\mathtt{f}_2\mathtt{x}$, and consider the path that alternates between those:

$$\mathtt{f\,x}, \quad \mathtt{C}_1\mathtt{f\,x}, \quad \mathtt{C}_1\mathtt{C}_2\mathtt{f\,x}, \quad \mathtt{C}_1\mathtt{C}_2\mathtt{C}_1\mathtt{f\,x}, \quad \ldots$$

We obtain the path

$$\mathtt{f}_{[]}\,\mathtt{x}, \quad \mathtt{C}_1\mathtt{f}_{[1]}\,\mathtt{x}, \quad \mathtt{C}_1\mathtt{C}_2\mathtt{f}_{[1,2]}\,\mathtt{x}, \quad \mathtt{C}_1\mathtt{C}_2\mathtt{C}_1\mathtt{f}_{[1,2,1]}\,\mathtt{x}, \quad \ldots$$

These lists record the "genealogy" of each occurrence of $\mathtt{f}$ by keeping track of which previous occurrences introduced it.

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

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

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 $\sigma'_0, \ldots \sigma'_{k-1}$ so that each s'_i is of the form $\beta_i\ \gamma_i^\Omega\ \mathtt{f}_L\, t_i^\Omega$ where

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

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

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

$$\begin{aligned} \left(\beta_k\ \gamma_k^\Omega\ \mathtt{f}_L\, t_k^\Omega\right)\left[\mathtt{f} := \gamma'^\Omega \mathtt{f}_{L,i} t'^\Omega\right] \quad &= \quad \beta_k\ \gamma_k^\Omega\ \gamma'^\Omega\ \mathtt{f}_{L,i}\, t'^\Omega[\mathtt{x} := t_k^\Omega] \\ &= \quad \beta_k\ (\gamma_k \gamma')^\Omega\ \mathtt{f}_{L,i}\, (t'[\mathtt{x} := t_k])^\Omega \end{aligned}$$

- If γ_k was empty, the summand t starts with part of the branch β: there is a subterm $\beta_{k,k+1}\gamma' \mathtt{f}_{L,i} t'$ 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,
  - $\mathtt{f}_{L,i}$ is a non-total occurrence.

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

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

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

Lemma 4.7. *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 $\mathbf{0}$. By definition of the semantics of Ω, this means that $\Omega t(u) = \bot$. Since γ^Ω starts with a Ω (or is empty), the limit is of the form $\bigsqcup^\uparrow \beta_k\bot$. Because β is non-total, this limit is also non-total. □

Note that this is a soundness result and doesn't say anything about the strength of reducing totality for T to totality for $G(T)$. The only argument in favor of $G(T)$ presented in this paper is of a practical nature: experimenting with `chariot` shows that $G(T)$ is enough for many recursive functions. General results like "all structurally recursive definitions are total" or "all syntactically guarded definitions are total" are certainly provable, but are left open for now.

4.2. **Weights and Approximations.** In order to use the size-change principle, we need arbitrary compositions of clauses to be bounded, which is not the case in general. In the following recursive definition,

```
val length : list¹(x) -> nat¹
  | length Nil¹ = Zero¹
  | length (Cons¹{Fst⁰=_; Snd⁰=l}) = Succ¹ (length l)
```

the only recursive call is

$$\texttt{length l} \mapsto \texttt{Succ}^1\ \texttt{length (.Snd}^0\ \texttt{Cons}^{1\text{-}}\ \texttt{l)}$$

Composing it with itself n times gives

$$\texttt{length l} \mapsto \underbrace{\texttt{Succ}^1\ \ldots\ \texttt{Succ}^1}_{n\ \texttt{repetitions}}\ \texttt{length (}\underbrace{\texttt{.Snd}^0\ \texttt{Cons}^{1\text{-}}\ \ldots\ \texttt{.Snd}^0\ \texttt{Cons}^{1\text{-}}}_{n\ \texttt{repetitions of .Snd}^0\ \texttt{Cons}^{1\text{-}}}\ \texttt{l)}$$

which grows arbitrarily large!

To deal with this problem, we introduce *approximations*: for large terms, we only count constructors (`C`, `{...}`, `C -` and `.D`); and we stop counting if there are too many of them. The totality checker will be parametrized by two natural numbers defining what "large" and "too many" really mean.

4.2.1. *Weights.* Simply counting constructors isn't enough because we also need to keep track of their priorities.

Definition 4.8 (Weights)**.** Define the following

(1) $\mathbb{Z}_\infty = \mathbb{Z} \cup \{\infty\}$ with the usual order and addition extended with $w \leq \infty$ and $w + \infty = \infty + w = \infty$ for all w.
(2) *Weights* are tuples of elements of $\mathbb{Z}_\infty$: $\mathbb{W} = \mathbb{Z}_\infty^{\mathbb{P}}$ where $\mathbb{P}$ is the finite non-empty set of priorities. This set is ordered pointwise with the *reverse* order of $\mathbb{Z}_\infty$. Addition on $\mathbb{W}$ is defined pointwise.

We define the following abbreviations:

- $\langle 0 \rangle = (0, \ldots, 0)$,
- $\langle w \rangle^p$ for the weight $(w_q)_{q \in \mathbb{P}}$ with $w_p = w$ and $w_q = 0$ if $q \neq p$,

We enclose weights with the symbols "$\langle$" and "$\rangle$", as in "$\langle W\rangle$" or "$\langle W_1 + W_2\rangle$".

The next lemma is straightforward.

Lemma 4.9. *Weights have the following properties:*

(1) *addition is commutative and associative,*
(2) $\langle 0\rangle$ *is neutral for addition,*
(3) *every* $\langle W\rangle \in \mathbb{W}$ *can be written (uniquely) as a sum* $\sum_{p\in P} \langle w_p\rangle^p$ *where* $P \subseteq \mathbb{P}$ *and each* $w_p \in \mathbb{Z}_\infty$,
(4) *whenever* $w_1 \leq w_2$ *in* $\mathbb{Z}_\infty$, *then* $\langle w_2\rangle^p \leq \langle w_1\rangle^p$ *in* $\mathbb{W}$ *(note the reversal).*

Weights will be used to count constructors and destructors (with negative elements of $\mathbb{Z}_\infty$). The special value ∞ will be a way to stop counting when those numbers become too big. It should not be interpreted as saying there are infinitely many constructors.

4.2.2. *Approximations.*

Definition 4.10. Let $\Delta \in \mathcal{F}_0$ be a normal form (Lemma 3.14) *which contains neither functions names, empty structures, nor* Ω,

(1) the set of *branches of* Δ is defined inductively

$$\begin{aligned}
\mathsf{branches}(\mathtt{x}) &= \{\mathtt{x}\} \\
\mathsf{branches}\big(\mathtt{C}^p\Delta\big) &= \mathtt{C}^p.\mathsf{branches}(\Delta) \\
\mathsf{branches}\big(\mathtt{.D}^p\Delta\big) &= \mathtt{.D}^p.\mathsf{branches}(\Delta) \\
\mathsf{branches}\big(\mathtt{C}^{\text{-}p}\Delta\big) &= \mathtt{C}^{\text{-}p}.\mathsf{branches}(\Delta) \\
\mathsf{branches}\big(\{\dots; \mathtt{D}_i = \Delta_i; \dots\}^p\big) &= \bigcup_i \{\mathtt{D}_i = _\}.\mathsf{branches}(\Delta_i)
\end{aligned}$$

(2) If B is a branch of Δ, the weight of B, written $|B| \in \mathbb{W}$ is defined with:
- $|\mathtt{x}| = \langle 0\rangle$,
- $|\mathtt{C}^p B| = \langle\langle 1\rangle^p\rangle + |B|$,
- $|\mathtt{.D}^p B| = \langle\langle -1\rangle^p\rangle + |B|$,
- $|\mathtt{C}^{\text{-}p} B| = \langle\langle -1\rangle^p\rangle + |B|$,
- $|\{\mathtt{D} = B\}^p| = \langle\langle 1\rangle^p\rangle + |B|$.

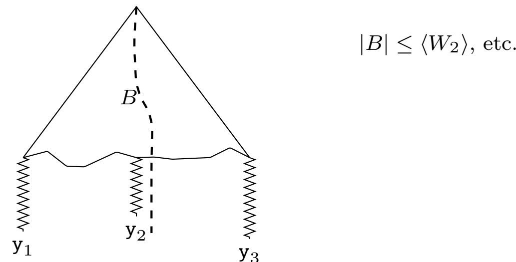
Everything is now in place to define approximations.

Definition 4.11. (1) Given some $W_i \in \mathbb{W}$, we put

$$\langle W_1\rangle \mathtt{y}_1 * \dots * \langle W_n\rangle \mathtt{y}_n \quad = \quad \sum \left\{ \Delta \;\middle|\; \begin{array}{l}\text{all branches } B \text{ of } \Delta \text{ leading} \\ \text{to } \mathtt{y}_i \text{ satisfy } |B| \leq \langle W_i\rangle\end{array} \right\}$$

(2) Given $t_1, \dots, t_n$ in $\mathcal{F}$, we write $\langle W_1\rangle t_1 * \dots * \langle W_n\rangle t_n$ for the corresponding $\sum \Delta[\mathtt{y}_1 := t_1, \dots]$.

The typical summand of $\langle W_1\rangle \mathtt{y}_1 * \dots * \langle W_n\rangle \mathtt{y}_n$ looks like:



$|B| \leq \langle W_2\rangle$, etc.

The guiding intuition is that a product gives information about *all* possible branches of somes elements of $\mathcal{F}$. For example, both $\texttt{\{Fst=y}_1\texttt{;Snd=y}_2\texttt{\}}$ and $\texttt{\{Foo=y}_1\texttt{;Bar=y}_2\texttt{\}}$ are approximated by $\langle 1\rangle\mathtt{y}_1 * \langle 1\rangle\mathtt{y}_2$. Not all branches need to be present, and $\mathtt{C}\,\mathtt{y}_2$ is also approximated by $\langle 1\rangle\mathtt{y}_1 * \langle 1\rangle\mathtt{y}_2$. Because they don't keep field names, projecting a product does not remove branches: it only decreases the weigts. As we'll see with Lemma 4.13, $\mathtt{.Fst}(\langle 1\rangle\mathtt{y}_1 * \langle 1\rangle\mathtt{y}_2) = \langle 0\rangle\mathtt{y}_1 * \langle 0\rangle\mathtt{y}_2$. On the other hand, a term containing $\mathtt{y}_3$ cannot be approximated by $\langle 1\rangle\mathtt{y}_1 * \langle 1\rangle\mathtt{y}_2$.

Lemma 4.12. *Each product $\langle W_1\rangle t_1 * \cdots * \langle W_n\rangle t_n$ is finitely generated.*

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 $\langle W_1\rangle\mathtt{y}_1 * \cdots * \langle W_n\rangle\mathtt{y}_n$. 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 $\Xi_{\restriction d} \subset \mathcal{F}_0$ as the set of all those Δ obtained as truncations of elements of Ξ of "syntactical depth" d. Truncating an element Δ is done by replacing subterms of Δ at syntactical depth d by Ω and normalizing. For example,$\mathtt{Succ\ Succ\ Succ^{-}\ x}$ at depth 2 gives $\mathtt{Succ\ Succ}\ \Omega\ \mathtt{x}$.

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

$$\Xi_{\restriction 1} \quad\leq\quad \Xi_{\restriction 2} \quad\leq\quad \cdots$$

Indeed, each element of $\Xi_{\restriction d+1}$ is either in $\Xi_{\restriction d}$ (when its syntactical depth is less than d), or greater than an element of $\big(\langle W_1\rangle\mathtt{y}_1 * \cdots * \langle W_n\rangle\mathtt{y}_n\big)_i$ (when its syntactical depth is strictly greater than d). This shows that $\langle W_1\rangle\mathtt{y}_1 * \cdots * \langle W_n\rangle\mathtt{y}_n$ is a limit of elements of $\mathcal{F}_0$. □

Approximations interact with the order on $\mathcal{F}$ in a way that will make it possible to compute directly with approximations. In fact, the totality checker will see approximations as syntactical constructions and need to reduce them in way that extends reduction on $\mathcal{F}_0$.

Lemma 4.13.
- *$P * Q \leq Q$ for all products P and Q,*
- *if $W \leq W'$ in $\mathbb{W}$, then $\langle W'\rangle t \leq u\langle W\rangle t$,*
- *$\langle 0\rangle(t) \leq t$.*

Moreover
1. *$\langle W\rangle\mathbf{0} = \mathbf{0}$ for a unary product,*
2. *$\langle W\rangle\mathbf{0} * P = P$,*
3. *$\langle W\rangle\mathtt{C}^p t * P = \langle W + \langle 1\rangle^p\rangle t * P$,*
4. *$\langle W\rangle\mathtt{\{}\dots;\mathtt{D}_i = t_i;\dots\mathtt{\}}^p * P = \prod_i \langle W_i + \langle 1\rangle^p\rangle t_i * P$ if the record is not empty,*
5. *$\mathtt{C}^{p^-}\prod_i \langle W_i\rangle t_i = \prod_i \langle W + \langle -1\rangle^p\rangle t_i$,*
6. *$\mathtt{.D}^p\prod_i \langle W_i\rangle t_i \geq \prod_i \langle W + \langle -1\rangle^p\rangle t_i$,*
7. *$\Omega\Big\{\prod_i \langle W_i\rangle t_i, T\Big\} = \sum_i \Omega\{t_i, T\}$,*
8. *$\langle W\rangle\Omega\{T\} * P \geq \Omega\{T\} + P$.*
9. *$\big(\langle V\rangle(\prod_i \langle W_i\rangle t_i)\big) * P \geq \big(\prod_i \langle V + W_i\rangle t_i\big) * P$.*

Proof.

TODO:

The first three points are immediate. For the rest, we have
1. For a unary product, $\langle W\rangle\mathtt{y}_1$ must contain a branch leading to y_1, so that $\{\langle W\rangle\mathbf{0}\}$ contains a branch leading to $\mathbf{0}$. Because $\mathbf{0}$ propagates everywhere, all summands in $\langle W\rangle\mathbf{0}$ are equal to $\mathbf{0}$, and the sum is thus equal to $\mathbf{0}$.
2. When the product contains several factor, any summand of $\langle W\rangle\mathbf{0} * P$ that contains $\mathbf{0}$ is equal to $\mathbf{0}$ and can be simplified from the sum. There remains the summands comming from the Δ not containing $\mathtt{x}_1$, i.e. those corresponding to the product P.

(3) Suppose $\Delta[\mathtt{y}_1 := \mathtt{C}^p t_1, \dots]$ is a summand in $\langle W\rangle \mathtt{C}^p t_1 * P$. We put $\Delta' = \Delta[\mathtt{y}_1 := \mathtt{C}^p \mathtt{y}_1]$ so that we have $\Delta'[\mathtt{y}_1 := t_1, \dots] = \Delta[\mathtt{y}_1 := \mathtt{C}^p t_1, \dots]$ is a summand in $\langle W + \langle 1\rangle^p\rangle t_1 * P$. This shows that $\langle W\rangle \mathtt{C}^p t_1 * P \geq \langle W + \langle 1\rangle^p\rangle t_1 * P$. For the converse, suppose $\Delta[\mathtt{y}_1 := t_1, \dots]$ is a summand in $\langle W + \langle 1\rangle^p\rangle t_1 * P$. We put $\Delta' = \Delta[\mathtt{y}_1 := \mathtt{C}^{p^-} \mathtt{y}_1]$ so that $\Delta'[\mathtt{y}_1 := \mathtt{C}^p t_1, \dots] \geq \Delta[\mathtt{y}_1 := t_1, \dots]$ is a summand in $\langle W\rangle \mathtt{C}^p t_1 * P$. This shows that $\langle W\rangle \mathtt{C}^p t_1 * P \leq \langle W + \langle 1\rangle^p\rangle t_1 * P$.

(4) This is treated similarly.

(5) Let $t = \mathtt{C}^{-}\Delta[\mathtt{y}_1 := t_1, \dots]$ be a summand in $\mathtt{C}^{p^-} \prod_i \langle W_i\rangle t_i$.
   - If $\mathtt{C}^{-p}\Delta$ reduces to $\mathbf{0}$, then $\mathtt{C}^{-p}\Delta[\mathtt{y}_1 := t_1, \dots] \approx \mathbf{0} \geq \prod_i \langle W + \langle -1\rangle^p\rangle t_i$.
   - If not, the normal form of $\mathtt{C}^{-p}\Delta$ belongs to $\prod_i \langle W + \langle -1\rangle^p\rangle \mathtt{y}_i$, and t is greater than a summand of $\prod_i \langle W + \langle -1\rangle^p\rangle t_i$.

   In both cases $t \geq \prod_i \langle W + \langle -1\rangle^p\rangle t_i$. For the converse, let $t = \Delta[\mathtt{y}_1 := t_1, \dots]$ be a summand in $\prod_i \langle W + \langle -1\rangle^p\rangle t_i$. Take $\Delta' = \mathtt{C}^p\Delta$: this is an element of $\prod_i \langle W_i\rangle \mathtt{y}_i$, so that $\mathtt{C}^{-p}\mathtt{C}^p\Delta \approx \Delta$ is an element of $\mathtt{C}^{p^-} \prod_i \langle W_i\rangle \mathtt{y}_i$. We can conclude that t is a summand of $\prod_i \langle W + \langle -1\rangle^p\rangle t_i$,

(6) This is treated similarly, except the second inequality cannot be proved as projecting on a record makes a term smaller.[12]

(7) Let t be a summand in $\Omega\big\{\prod_i \langle W_i\rangle t_i, T\big\}$ It is of the form $\Omega\{\Delta[\mathtt{y}_1 := t_1, \dots], T\}$. Because Ω absorbs constructors on its right, we can reduce Δ so that $t \to^* t' = \Omega\{T', T\}$, where T' contains elements of the form $\delta\mathtt{x}$, where δ is a sequence of destructors. The destructors on the right of δ come from the t_i, the ones on the left come from Δ. Since $\Omega\{\mathtt{C}^{-}t, T\} \geq \Omega\{\mathtt{C}^{-}\Omega t, T\} \approx \Omega\{t, T\}$, and similarly for $\mathtt{.D}$, we can remove the destructors coming from Δ and obtain $t'' \leq t'$ of the form $\Omega\{T'', T\}$ where T'' contains all the $\delta\mathtt{x}$ appearing in the t_is appearing in Δ. There is at least one such t_i, to that $t'' \geq \Omega\{t_i, T\}$.

   For the converse, any $\Omega\{t_i, T\}$ is a summand in $\Omega\big\{\prod_i \langle W_i\rangle t_i, T\big\}$. This is because we can always find a sequence of destructors γ so that
$$\Omega\{t_i, T\} \quad\approx\quad \Omega\{\Omega\, t_i, T\} \quad\approx\quad \Omega\{\gamma\,\Omega\, t_i, T\} \quad\leq\quad \Omega\{\gamma\, t_i, T\} \quad\in\quad \Omega\Big\{\prod_i \langle W_i\rangle t_i, T\Big\}$$

(8) Let $t = \Delta[\mathtt{y}_1 := \Omega\{T\}, \dots]$ be a summand in $\langle W\rangle\Omega\{T\} * P$. If Δ doesn't contain $\mathtt{y}_1$, t is a summand of P, and is thus greater than $\Omega\{T\} + P$. If Δ does contain $\mathtt{y}_1$, we will prove that $t \geq \Omega\{T\}$. Since $t \geq \Omega\, t$, it is enough to prove that $\Omega\, t = \Omega\,\Delta[\mathtt{y}_1 := \Omega\{T\}] \geq \Omega\{T\}$. The topmost Ω will absorb constructors on top of Δ, and the Ω's on the bottom will absorb destructors on the bottom of Δ. We obtain $\Omega\{\Omega T, \dots\} \approx \Omega\{T, \dots\} \geq \Omega\{T\}$.

(9) This is immediate.

□

4.2.3. *Dual approximations.* Inductive and coinductive types are dual to each other. Formally, approximations should come in 2 dual flavors:

- one that guarantees that at least some constructors have been *removed*, used to detect that inductive argument to a recursive function gets smaller,
- one that guarantees that at least some constructors have been *added*, used to detect that a recursive function is productive.

The notion of approximation defined above corresponds to the first kind, and rather than duplicating the definition, we will simply negate the weights to deal with the second kind.

[12] It would be possible to make projecting on a record with a single field equal to the value of that field, but that adds yet another case to the definition of the order.

4.3. **Collapsing.** Because the first step when using the size-change principle is constructing the transitive closure of the call-graph, we need compositions of calls to be bounded. As shown on page 32, this is not the case in general and we need to replace "large" compositions by smaller approximations. This is parametrized by 2 natural numbers:

- $D \geq 0$ giving the maximum depth of terms,
- $B > 0$ giving a bound for the number of constructors we keep. Any number more than this bound will be replaced by ∞.

Both bounds can be chosen for each recursive definition. The larger they are, the more precise the algorithm will be, but the slower it will be.

4.3.1. *Calls.* Summands in the call-graph (Section 4.1) contain exactly one recursive call and may contain approximations. They can be described with the following syntax.

Definition 4.14. *General calls* are defined inductively by

$$
\begin{array}{lcll}
t & ::= & \mathtt{C}^p\, t \quad | \quad \{\mathtt{D}_1 = t_1; \dots ; \mathtt{D}_n = t_n\}^p \quad | & \\
 & & \mathtt{C}^{p^-} t \quad | \quad .\mathtt{D}^p t \quad | & \\
 & & \Omega\{t_1, \cdots, t_n\} \quad | & \\
 & & \big\{\langle W_1\rangle t_1, \cdots, \langle W_n\rangle t_n\big\} \quad | & \text{(a finite non-empty set)} \\
 & & \mathbf{0} \quad | \quad \mathtt{x} & \\
 & & \mathtt{f}\, t &
\end{array}
$$

where $\mathtt{x}$ is a *formal parameter* and each $\langle W\rangle$ is a weight. As previously, $\mathtt{C}$ and $\mathtt{D}$ 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.

Finite sets of weighted calls are written with a product notation as in "$\langle W_1\rangle t_1 * \cdots * \langle W_n\rangle t_n$", or simply "$\langle W\rangle t$" for the case of a unary product. This product is thus commutative and idempotent by construction.

Definition 4.15. The order on general calls is defined inductively using the same rules as the order on $\mathcal{F}_0$ (Definition 3.12) together with some additional rules:

- $P * Q \leq P$,
- if $W \leq W'$ in $\mathbb{W}$, then $\langle W'\rangle t \leq \langle W\rangle t$,
- $\langle 0\rangle t \leq t$.

and

$$
(*)\left\{
\begin{array}{lrcll}
(4) & \langle W\rangle \mathbf{0} & \approx & \mathbf{0} & \text{for a unary product} \\
(4) & \langle W\rangle \mathbf{0} * P & \approx & P & \\
(4) & \langle W\rangle \mathtt{C}^p t * P & \approx & \langle W + \langle 1\rangle^p\rangle t * P & \\
(4) & \langle W\rangle \{\dots ; \mathtt{D}_i = t_i; \dots\}^p * P & \approx & \prod_i \langle W_i + \langle 1\rangle^p\rangle t_i * P & \text{if the record is not empty} \\
(4) & \langle W\rangle \{\}^p * P & \geq & \Omega & \\
(4) & \mathtt{C}^{p^-} \prod_i \langle W_i\rangle t_i & \approx & \prod_i \langle W + \langle -1\rangle^p\rangle t_i & \\
(4) & .\mathtt{D}^p \prod_i \langle W_i\rangle t_i & \geq & \prod_i \langle W + \langle -1\rangle^p\rangle t_i & \\
(4) & \Omega\Big\{ \prod_i \langle W_i\rangle t_i, T\Big\} & = & \sum_i \Omega\{t_i, T\} & \\
(4) & \langle W\rangle \Omega\{T\} * P & \geq & \Omega\{T\} + P & \\
(4) & \big(\langle V\rangle (\prod_i \langle W_i\rangle t_i)\big) * P & \geq & \big(\prod_i \langle V + W_i\rangle t_i\big) * P &
\end{array}
\right.
$$

Because of Lemma 4.13, the order on general call implies the order on their semantics in $\mathcal{F}$. We can extend the reduction relation $\to$ to general calls.

Definition 4.16. Reduction on general calls extends reduction on $\mathcal{F}_0$ by adding rules, oriented from left to right, for all inequalities in group (4).

Lemma 4.17.
(1) *If $t \to t'$ then $t \geq t'$.*
(2) *Reduction on general calls is strongly normalizing.*
(3) *Normal forms are generated by the following grammar*

$$
\begin{array}{lll}
t & ::= & \mathtt{C}^p t \mid \{\mathtt{D}_1 = t_1; \dots; \mathtt{D}_n = t_n\}^p \mid \\
 & & \Omega\{\delta_1, \dots, \delta_n\} \mid \langle W_1 \rangle \varepsilon_1 * \dots * \langle W_n \rangle \varepsilon_n\} \mid \delta \\
\varepsilon & ::= & \delta \mid \{\} \\
\delta & ::= & \mathtt{C}^{p^-} \delta \mid .\mathtt{D}^p \delta \mid \mathtt{x} \mid \mathtt{f}\, t
\end{array}
$$

(4) *Given s in normal form, we have $s \leq t < \mathbf{0} \implies s \leq \mathsf{nf}(t)$. More precisely:*
   (a) *If $s = \mathtt{f}\, s'$, then $\mathsf{nf}(t) = \mathtt{f}\, t'$ with $s' \leq t'$.*
   (b) *If $s = \mathtt{C}\, s'$, then $\mathsf{nf}(t) = \mathtt{C}\, t'$ with $s' \leq t'$.*
   (c) *If $s = \{\mathtt{D}_1 = s_1; \dots\}$, then $\mathsf{nf}(t) = \{\mathtt{D}_1 = t_1; \dots\}$ with $\forall i, s_i \leq t_i$.*
   (d) *If $s = .\mathtt{D}\, s'$ (resp. $s = \mathtt{C}^- s'$), then $\mathsf{nf}(t) = .\mathtt{D}\, t1$ (resp. $t = \mathtt{C}^- t'$) with $s' \leq t'$.*
   (e) *If $s = \mathtt{x}$, then $\mathsf{nf}(t) = \mathtt{x}$.*
   (f) *If $s = \Omega\{S\}$, then $\mathsf{nf}(\Omega\, t) = \Omega\{T\}$ with for all $s \in S$, there is a sequence of destructors δ s.t. $s \leq t$ for some $\delta\, t \in T$.*
   (g) *If $s = \prod_i \langle V_i \rangle \varepsilon_i$ is a product, then $\mathsf{nf}(\langle 0 \rangle\, t) = \prod_j \langle W_j \rangle \gamma_j$ where for all j, there is some i and a sequence of destructors δ s.t.*
   - *γ_j is of the form $\delta.\gamma$,*
   - *$\mathsf{nf}(\langle W_j \rangle \delta \langle 0 \rangle \gamma) = \langle W \rangle \gamma$ with $\langle V_i \rangle \geq \langle W \rangle$ in $\mathbb{W}$,*
   - *and $\varepsilon_i \leq \gamma$.*

Point (g) deserves some explanation. It can be decomposed in two part: for an arbitrary product P, it says that $P \leq t$ implies $\mathsf{nf}(t) = Q$ where each term in Q, seen as a unary product is greater than a term in P. For unary products in normal forms, we have $\langle V \rangle \varepsilon \leq \langle W \rangle \gamma$ if we can find a sequence of destructors δ s.t.
- $\gamma = \delta.\gamma'$,
- $\varepsilon \leq \gamma'$,
- $\langle V \rangle \geq |\delta| + \langle W \rangle$.

A generic example would be $\langle 0 \rangle\, \mathtt{f}\, \Omega \leq \langle 1 \rangle\, \mathtt{C}^- \mathtt{f}\, \mathtt{x}$.

Proof of Lemma 4.17.

TODO:

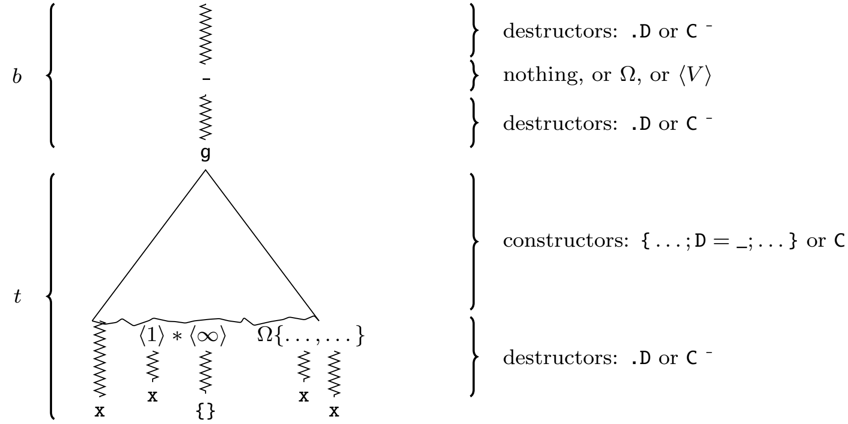
The first three points are straightforward. For the fourth, the proof extends that of Lemma 3.17 (page 24): for points (a)–(f), the proof is exactly the same as points (1)–(6). For point (g):
- If $\prod_i \langle V_i \rangle \varepsilon_i \leq u \leq t$ comes from transitivity. By induction, $\mathsf{nf}(\langle 0 \rangle\, u)$ is a product $\prod_k \langle U_k \rangle \xi_k$ satisfying for all k, there is some i and a sequence of destructors δ s.t.
  – ξ_k is of the form $\delta.\xi$,
  – $\mathsf{nf}(\langle U_k \rangle \delta \langle 0 \rangle \xi) = \langle W \rangle \xi$ with $\langle V_i \rangle \geq \langle W \rangle$ in $\mathbb{W}$,
  – and $\varepsilon_i \leq \xi$.

  Because $\mathsf{nf}(u) \leq u$, we can also use induction on $\mathsf{nf}(u) \leq t$ to get that $\mathsf{nf}(\langle 0 \rangle t)$ is a product of the form $\prod_j \langle W_j \rangle \gamma_j$ for all j, there is some k and a sequence of destructors δ s.t.
  – γ_j is of the form $\delta.\gamma$,
  – $\mathsf{nf}(\langle W_j \rangle \delta \langle 0 \rangle \gamma) = \langle W \rangle \gamma$ with $\langle U_k \rangle \geq \langle W \rangle$ in $\mathbb{W}$,
  – and $\xi_{\leq} \gamma$.

  Combining the two, it is easy to check that the property of the lemma is satisfied as well.
- If $s = \prod_i \langle V_i \rangle \varepsilon_i \leq t$ comes from contextuality for some context C, $s' \leq t'$. The context C is necessarily of the form $\prod_i \langle V_i \rangle \varepsilon_{i,\mathtt{y}}$ where each $\varepsilon_i = \varepsilon_{i,\mathtt{y}}[\mathtt{y} := t']$. Because s is in normal form,

each $\varepsilon_{i,\mathtt{y}}$ can only start with a destructor, function, variable or the empty record. By induction hypothesis, each $\varepsilon_{i,\mathtt{y}}[\mathtt{y} := s'] \leq \mathsf{nf}(\varepsilon_{i,\mathtt{y}}[\mathtt{y} := t'])$, and each $\mathsf{nf}(\varepsilon_{i,\mathtt{y}}[\mathtt{y} := t'])$ can only start with a destructor, function, variable or the empty record. Because of that $\mathsf{nf}(C[\mathtt{y} := t']) = C[\mathtt{y} := \mathsf{nf}(t')]$. To summarize, $s = \prod_i \langle V_i \rangle \varepsilon_{i,\mathtt{y}}[\mathtt{y} := s']$ and $\mathsf{nf}(t) = \prod_i \langle V_i \rangle \varepsilon_{i,\mathtt{y}}[\mathtt{y} := \mathsf{nf}(t')]$ with $s' \leq \mathsf{nf}(t')$. In that case, $\mathsf{nf}(\langle 0 \rangle t) = \mathsf{nf}(t)$ and this implies the condition from point (g).
- If $s = P * Q \leq P$, we can conclude directly.
- If $s = \langle V \rangle \varepsilon \leq t = \langle W \rangle \varepsilon$ with $W \leq V$ in $\mathbb{W}$, we can conclude directly.
- If $s = \langle 0 \rangle \varepsilon \leq \varepsilon$, we can conclude directly. □

Definition 4.18. A *call* is defined as a 3-tuple consisting of
- a calling function name,
- a called function name,
- a general call in normal form of the following shape:



In particular, the general call contains exactly one occurence of $\mathtt{g}$ and no other function name. We write "$\mathtt{f}\ \mathtt{x} \mapsto b\,\mathtt{f}\,t$" to separate the branch b above the call and the argument t below the call.

4.3.2. *Collapsing.*

Definition 4.19. Given $B > 0 \in \mathbb{N}$, the *weight collapsing function* $\lceil _ \rceil_B$ acts on terms by replacing each weight of the form $\sum_p \langle w_p \rangle^p$ (as in point (3) of Lemma 4.9) by $\sum_p \langle \lceil w_p \rceil_B \rangle^p$ where

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

To bound the depth, we introduce "$\langle 0 \rangle$" below D constructors and above D destructors in the calls. Because of the reduction, those weights will absorb the constructors below D and the destructors above D. For example, collapsing $\mathtt{C}_1\mathtt{C}_2\mathtt{C}_3\langle W \rangle \mathtt{C}_4^-\mathtt{C}_5^-\mathtt{C}_6^-\mathtt{C}_7^-\mathtt{x}$ at depth 2 gives

$$\underbrace{\mathtt{C}_1\ \mathtt{C}_2}_{D=2}\ \langle 0 \rangle\ \mathtt{C}_3\ \langle W \rangle\ \mathtt{C}_4^-\ \mathtt{C}_5^-\ \langle 0 \rangle\ \underbrace{\mathtt{C}_6^-\ \mathtt{C}_7^-}_{D=2}\ \mathtt{x} \quad \to^* \quad \underbrace{\mathtt{C}_1\ \mathtt{C}_2}_{D=2}\ \langle 1 + W - 2 \rangle\ \underbrace{\mathtt{C}_6^-\ \mathtt{C}_7^-}_{D=2}\ \mathtt{x}$$

where we assume all the constructors have the same priority.

Definition 4.20. Suppose a call t is in normal form (Lemma 4.17). Given a positive bound $D \in \mathbb{N}$, the *depth collapsing function* $_\!\restriction_D$ acts on t by integrating constructors below D and destructors

above D into weights:

$$
\begin{array}{rcll}
\big(\mathtt{C}\ t\big)_{\restriction_i} & \stackrel{\text{def}}{=} & \mathtt{C}\ \big(t_{\restriction_{i-1}}\big) & \text{if } i > 0 \\
\{\dots;\mathtt{D}_k = t_k;\dots\}_{\restriction_i} & \stackrel{\text{def}}{=} & \{\dots;\mathtt{D}_k = t_{k\restriction_{i-1}};\dots\} & \text{if } i > 0 \\
\big(\prod \langle W\rangle\delta\big)_{\restriction_i} & \stackrel{\text{def}}{=} & \prod \langle W\rangle\big(\delta_{\downharpoonright_D}\big) & \text{if } i > 0 \\
\delta_{\restriction_i} & \stackrel{\text{def}}{=} & \delta_{\downharpoonright_D} & \\
t_{\restriction_0} & \stackrel{\text{def}}{=} & \mathsf{nf}\big(\langle 0\rangle t\big)_{\downharpoonright_D} & (*)
\end{array}
$$

and the following are applied to all terms of a product termwise

$$
\begin{array}{rcll}
\big(\langle W\rangle\mathtt{\{\}}\big)_{\downharpoonright_i} & \stackrel{\text{def}}{=} & \langle W\rangle\mathtt{\{\}} & (**) \\
\big(\langle W\rangle\delta\,\mathtt{f}\,t\big)_{\downharpoonright_i} & \stackrel{\text{def}}{=} & \langle W\rangle\delta_{\downharpoonright_i}\,\mathtt{f}\,t_{\restriction_D} & (+)(**) \\
\big(\langle W\rangle\delta\mathtt{x}\big)_{\downharpoonright_i} & \stackrel{\text{def}}{=} & \langle W\rangle\delta_{\downharpoonright_i}\mathtt{x} & (**) \\
\big(\delta\mathtt{C}^p\big)_{\downharpoonright_i} & \stackrel{\text{def}}{=} & \delta_{\downharpoonright_{i-1}}\mathtt{C}^p & \\
\big(\delta\,\mathtt{.D}^p\big)_{\downharpoonright_i} & \stackrel{\text{def}}{=} & \delta_{\downharpoonright_{i-1}}\mathtt{.D}^p & \\
\delta_{\downharpoonright_0} & \stackrel{\text{def}}{=} & \delta\langle 0\rangle &
\end{array}
$$

Note the following.

- The clauses are not disjoint and only the first applicable 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).
- Clause $(+)$ allows to collapse both the branch above the call to $\mathtt{f}$ and the argument of $\mathtt{f}$. Because calls contain exactly one function name, this clause is used exactly once.

The following is obvious but depends on the fact that there are only finitely many constructors / destructors.

Lemma 4.21. *GivenB > 0 and $D \geq 0$, the image of* $\mathsf{nf}\Big(\big\lceil(_)_{\restriction_D}\big\rceil_B\Big)$ *is finite.*

4.3.3. *Composing calls.*

Definition 4.22. *Collapsed composition* is defined by

$$
\beta \diamond_{B,D} \alpha \quad := \quad \mathsf{nf}\Big(\big\lceil(\beta\circ\alpha)_{\restriction_D}\big\rceil_B\Big)
$$

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

Lemma 4.23. *For any call α, we have*

- $\lceil\alpha\rceil_B \leq \alpha$,
- $\alpha_{\restriction_D} \leq \alpha$,
- $\beta\diamond\alpha \leq \beta\circ\alpha$.

Proof.

- for $\lceil\alpha\rceil_B$, replacing $\langle W\rangle$ by $\langle\lceil W\rceil_B\rangle$ results in a smaller term by contextuality and the fact that $\lceil W\rceil_B \geq W$ in $\mathbb{W}$.
- for $\alpha_{\restriction_D}$, inserting some $\langle 0\rangle$ results in a smaller term by contextuality and the fact that $\langle 0\rangle t \leq t$, and normalizing makes this potentially smaller. □

Unfortunately, unless $B = 1$ and $D = 0$, collapsed composition is *not* associative. For example, using $B = 2$, the calls $\alpha = \mathtt{f\ x} \mapsto \langle 1\rangle\ \mathtt{f\ x}$ and $\beta = \mathtt{f\ x} \mapsto \langle -1\rangle\ \mathtt{f\ x}$ give

- $\beta\diamond(\alpha\diamond\alpha) = \mathtt{f\ x} \mapsto \langle\infty\rangle\ \mathtt{f\ x}$
- $(\beta\diamond\alpha)\diamond\alpha = \mathtt{f\ x} \mapsto \langle 1\rangle\ \mathtt{f\ x}$.

The next property, a kind of weak associativity, will be sufficient for our needs.

Lemma 4.24. *If $\sigma_n \circ \cdots \circ \sigma_1 \neq \mathbf{0}$, and if τ_1 and τ_2 are the results of computing $\sigma_n \diamond \cdots \diamond \sigma_1$ in two different ways, then τ_1 and τ_2 are* compatible, *written $\tau_1 \supset\!\!\!\subset \tau_2$. This means that there is some $\tau \neq \mathbf{0}$ such that $\tau_1 \leq \tau$ and $\tau_2 \leq \tau$.*

Proof. Taking $\tau = \sigma_n \circ \cdots \circ \sigma_1$ works, by repeated use of Lemma 4.23. □

4.4. **The Size-Change Principle.** Putting Proposition 4.6, Corollary 4.4 and Lemma 3.30 together, we get

Corollary 4.25. *If all infinite paths in $G(T_{\mathtt{f}})$ are total, then $[\![\mathtt{f}]\!]_\rho$ is total for every total environment ρ.*

Since we are now interested in a property of all infinite paths in the call-graph, the size-change principle comes to mind. However, because collapsed composition isn't associative, we first need to prove a variant of combinatorial lemma at the heart of the size-change principle.

Lemma 4.26. *Suppose $(O, \leq)$ is a partial order, and $F \subseteq O$ is a finite subset. Suppose moreover that $\circ$ is a partial, binary, associative and monotonic operation on $O \times O$ and that $\diamond$ is a partial, binary, monotonic operation on $F \times F$ satisfying*

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

whenever $o_1 \diamond o_2$ is defined. Then every infinite sequence $o_1, o_2, \dots$ of elements of F where each finite $o_1 \circ \cdots \circ o_n$ is defined can be subdivided into

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

where:

- *all the $(\dots(o_{n_k} \diamond o_{n_k+1}) \diamond \cdots) \diamond o_{n_{k+1}-1}$ 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,

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

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 $m < n$:

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

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:

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

The first two equalities imply that

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

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 define the transitive closure of a call-graph.

Definition 4.27. Let G be a call-graph. Start with $G_0 = G$ and define the edges of G_{n+1} to be those of G_n, together with:

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

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

We can now state and prove correctness of the size-change principle here in the case of a single function. We extend the notions of branch and weight (Definition 4.10) to deal with products:

$$\mathsf{branches}(P) \quad = \quad \{P\} \qquad \text{for any product } P$$

and

- $|\prod_i \langle W_i \rangle \delta_i| = \begin{cases} \sup_i \langle W_i \rangle + |\delta_i| & \text{if no } \delta_i = \texttt{\{\}} \\ \text{undefined} & \text{otherwise} \end{cases}$

where the suppremum is taken in $\mathbb{Z}_\infty$, i.e. corresponds to taking the least informative weight.

Theorem 4.28 (size-change principle)**.** *Suppose every loop $\alpha =$ `f x` $\mapsto$ b `f` u in G^* that satisfies $\alpha \supset \alpha \diamond \alpha$ also satisfies one of the following two conditions:*

- *either there is an even priority p with strictly negative $|b|_p$ such that $|b|_q \geq 0$ for all priorities $q > p$,*
- *or there is a branch B of u and an odd priority p with strictly negative $|B|_p$ such that $|B|_q \geq 0$ for all priorities $q > p$,*

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

Before proving the theorem, let's apply it to some simple recursive definitions. The following generates the stream of natural numbers starting at some `x`:

```
val nats : nat -> stream(nat)
  | nats x = { Head = n ; Tail = nats(x+1) }
```

If we replace "`+1`" by the `Succ` constructor and add explicit priorities, we get

```
val nats : nat -> stream(nat)
  | nats x = { Head⁰ = n ; Tail⁰ = nats (Succ¹ x) }
```

The call-graph contains a single call $\alpha =$ `nats` $\mapsto$ `Tail`^{0} `nats Succ`^{1} `x`. With bound $D = B = 1$, two steps are necessary to build the transitive closure:

- for the first step, we have $\alpha \circ \alpha =$ `nats` $\mapsto$ `Tail`^{0} `Tail`^{0} `nats (Succ`^{1} `Succ`^{1} `x)` which collapses[13] to $\beta = \alpha \diamond \alpha =$ `nats` $\mapsto$ `Tail`^{0} $\langle -1 \rangle^0$ `nats (Succ`^{1} $\langle 1 \rangle^1$ `x)`,
- the second step adds $\gamma = \alpha \diamond \beta = \beta \diamond \alpha = \beta \diamond \beta =$ `nats` $\mapsto$ `Tail`^{0} $\langle -1 \rangle^0$ `nats (Succ`^{1} $\langle \infty \rangle^1$ `x)`.

Only γ is coherent, and it satisfies the first property of Theorem4.28. Infinite compositions of γ build the infinite branch "`Tail`^{0}`Tail`^{0}…", which has even principal priority. The recusive definition is total.

The `length` function from page 4.2 has a call-graph with a single call:

$$\alpha = \texttt{length x} \mapsto \texttt{Succ}^1 \ \texttt{length (.Snd}^0 \ \texttt{Cons}^{1-} \ \texttt{x)}$$

with $B = 1$ and $D = 0$, the transitive closure is reached after one step. Besides α, it contains

$$\beta = \alpha \diamond \alpha = \texttt{length x} \mapsto \langle -1 \rangle^1 \ \texttt{length (}\langle \langle -1 \rangle^0 + \langle -1 \rangle^1 \rangle \ \texttt{x)}$$

The call β is coherent. It doesn't satisfy the first property of Theorem 4.28, but the second. Infinite compositions of β remove an infinite number of `Succ`^{1} from their argument. As a result, any argument leading to infinite compositions cannot be total. The recusive definition is total.

[13] recall that collapsing the branch above the recursive call uses negative weights (refer to page 35 and 39)

The definition of bad_s from page 13 has priorities

```
val bad_s : stream(stree)
  | bad_s = { Head⁰ = Node¹ bad_s ; Tail⁰ = bad_s }
```

and its call-graph has calls $\alpha_1 = \texttt{bad_s} \mapsto \texttt{Head}^0\ \texttt{Node}^1\ \texttt{bad_s}$ and $\alpha_2 = \texttt{bad_s} \mapsto \texttt{Tail}^0\ \texttt{bad_s}$. For $B = D = 1$, the transitive closure stabilizes after one step, and it contains 3 calls besides α_1 and α_2:

- $\beta_{1,1} = \alpha_1 \diamond \alpha_1 = \texttt{bad_s} \mapsto \texttt{Head}^0\ \langle\langle -1\rangle^0 + \langle -1\rangle^1\rangle\ \texttt{bad_s}$
- $\beta_{2,2} = \alpha_2 \diamond \alpha_2 = \texttt{bad_s} \mapsto \texttt{Tail}^0\ \langle -1\rangle^0\ \texttt{bad_s}$
- $\beta_{2,1} = \alpha_2 \diamond \alpha_1 = \texttt{bad_s} \mapsto \texttt{Tail}^0\ \langle\langle -1\rangle^0 + \langle -1\rangle^1\rangle\ \texttt{bad_s}$

The missing composition $\alpha_1 \diamond \alpha_2$ is equal to $\beta_{1,1}$. Those 3 calls are coherent, but while $\beta_{2,2}$ satisfies the first property of Theorem 4.28 neither $\beta_{1,1}$ nor $\beta_{2,1}$ do because the maximal priority comes from $\langle -1\rangle^1$. Because there is no argument to bad_s, they don't satisfy the second property either. The recursive definition is rejected by the totality checker, as it should.

Theorem 4.28 is strong enough to deal with mixed inductive and coinductive types. The following function takes a stream of lists of natural numbers and returns the stream of their sums. It does so by accumulating partial sums in its first argument.[14]

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

Because of the second clause, this definition isn't guarded. It is productive because this second clause cannot occur infinitely many times consecutively.

Provided $D > 0$, this will be detected by the totality checker and this definition will thus be accepted as total. With $B = D = 1$, the transitive closure of the call-graph will contains the following coherent loops:

- β_1, coming from compositions of the first call with itself:
$$\texttt{sums x}_1\ \texttt{x}_2 \mapsto \underline{\texttt{Tail}^0\langle -1^0\rangle}\ \texttt{sums}\ (\texttt{Zero}^1)\ (\langle -1\rangle^0.\texttt{Tail}^0\,\texttt{x}_2)$$
where the $\langle -1\rangle^0$ corresponds to the collapse of $\texttt{Tail}^0$,
- β_2, coming from compositions of the second call with itself:
$$\texttt{sums x}_1\ \texttt{x}_2 \mapsto \texttt{sums}\ \Omega\{\dots\}\ \{\ \underline{\texttt{Head}^0{=}\langle -1\rangle^1.\texttt{Head}^0\,\texttt{x}_2}\ ;\ \texttt{Tail}^0{=}.\texttt{Tail}^0\,\texttt{x}_2\ \}$$
where $\langle -1\rangle^1$ corresponds to the collapse of $\texttt{Cons}^{1^-}\texttt{Cons}^{1^-}$,
- β_3, coming from compositions of the first and second call:
$$\texttt{sum x}_1\ \texttt{x}_2 \quad\mapsto\quad \underline{\texttt{Tail}^0\langle -1\rangle^0}\ \texttt{sums}\ \Omega\{\dots\}\ \{\ \ \texttt{Head}^0 = \langle\langle -1\rangle^0{+}\langle -1\rangle^1\rangle.\texttt{Tail}^0\,\texttt{x}_2\ ;\ \ \texttt{Tail}^0 = \langle -1\rangle^0.\texttt{Tail}^0\,\texttt{x}_2\ \ \}$$
where $\langle\langle -1\rangle^0{+}\langle -1\rangle^1\rangle$ comes from the collapse of $\texttt{Cons}^{1^-}.\texttt{Head}^0$ and $\langle -1\rangle^0$ from the collapse of $.\texttt{Tail}^0$.

Both β_1 and β_3 satisfy the first property of Theorem 4.28, while β_2 satisfies the second property. This definition is total.

[14]Note that because sums has 2 arguments, the following doesn't fit exactly in what is detailed in the paper.

Proof of Theorem 4.28. By Lemma 4.3, we only need to check that infinite paths are total. Let (s_k) be infinite path of G. If any prefix composes to $\mathbf{0}$, the corresponding path is total. If no prefix composes to $\mathbf{0}$, we can use Lemma 4.26: such a path can be decomposed into

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

where:

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

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

$$\begin{aligned}
\bigsqcup_k^{\uparrow} s_k(\Omega) &= \bigsqcup_k^{\uparrow} t_0 \circ t_1 \circ \cdots \circ t_k(\Omega) \\
&\geq \bigsqcup_j^{\uparrow} t_0 \circ \cdots \circ t_{n_0-1} \circ t^j(\Omega) \\
&= \bigsqcup_j^{\uparrow} \mathit{init} \circ b^j \,\mathtt{f}\, u(\Omega) \\
&\geq \mathit{init} \circ \bigsqcup_j^{\uparrow} b^j \Omega
\end{aligned}$$

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

Similarly, if t satisfies the second condition. We have

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

By hypothesis, $u^k = u[\mathtt{x} := u^{k-1}]$ contains a branch $\beta \prod \langle W_i \rangle \delta_i$ and there is an odd p s.t. $|\beta \langle W_i \rangle \delta_i|_p < 0$ for all 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

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

where, by hypothesis, each $|\beta \beta_{i_j} \delta_{i_j}|_p < 0$. Such a branch globally removes at least k constructors of priority $p = 2q + 1$ and doesn't remove constructors of greater priority. 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! □

4.5. **Implementing the Totality Checker.** Implementing the totality checker based on Theorem 4.28 for a first-order functional programming language like `chariot` is relatively straightforward.

(1) During type checking / type inference, annotate all constructors appearing in the recursive definition with their type.
(2) Construct the parity game containing all these types:
    - start with the types of constructors / destructors and add the transitions coming from the types definitions until no transition can be added,
    - add priorities as in the proof of Lemma 2.7.

(3) Annotate all constructors and destructors appearing in the recursive definition with their priorities. The types themselves can be forgotten at this point.
(4) Compute the call-graph of the definition. This is easy for a language like `chariot` because each clause can be treated independently and each call will be in normal form by construction. The type of call is a simple first-order inductive type.
(5) Define inductive functions on calls, namely composition, reduction and collapsing. That makes it possible to compute the transitive closure of the call-graph.
(6) Loop over all loops of the transitive closure of the call-graph. If a loop is coherent, check that it satisfies one of the properties of Theorem 4.28. If all of them do, the definition is total.

Because parallel arcs in the call-graph correspond to non-deterministic sums and because $u + v = u$ whenever $u \leq v$, not all calls need to be added to the call-graph. If a call is greater than some existing call, it can be ignored. Doing so requires an inductive definition of the order $\leq$. Because calls need to be kept in normal form for collapsing, Lemma 4.13 can be transformed into an inductive definition of $\leq$.

The only part that hasn't been described in the paper is checking for coherent loops. It is possible to give an inductive characterization of coherence for normal forms, similar in spirit to Lemma 4.13, but the characterization is quite lengthy. `chariot` simplifies this by replacing each $\Omega\{U\}$ in the arguments of the recursive calls by Ω. Since Ω is the least element, this reflects totality. One can then use the following to check for coherent loops.

Lemma 4.29. *For generalized patterns in normal form where each $\Omega\{U\}$ has $U = \emptyset$, the reflexive closure of the following relation characterizes coherence.*

(1) $\mathtt{C}u \supset \mathtt{C}v$ *iff* $u \supset v$,
(2) $\{\mathtt{D}_1 = u_1; \dots; \mathtt{D}_k = u_k\} \supset \{\mathtt{D}_1 = v_1; \dots; \mathtt{D}_k = v_k\}$ *iff* $\forall i, u_i \supset v_i$,
(3) $\delta \supset \delta$ *when δ is a sequence of destructors followed by a variable,*
(4) $\Omega \supset v$,
(5) *for a product P, $P \supset \mathtt{C}v$ iff $P \supset v$,*
(6) *for a product P, when $k > 0$, $P \supset \{\mathtt{D}_1 = v_1; \dots; \mathtt{D}_k = v_k\}$* if $\forall i, P \supset v_i$,
(7) *for a product P, and if v is of the form $\{\}$ or δ, $P \supset v$ iff $P \leq v$,*
(8) $\prod_i \langle W_i \rangle \delta_i \supset \prod_j \langle W'_j \rangle \delta'_j$ *iff there are some i and j s.t. δ_i is a suffix of δ'_j or δ'_j is a suffix of δ_i.*

Concluding Remarks

Complexity. Since this totality test extends the termination test described in [Hyv14] and thus the usual size-change termination principle, it is at least 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. It seems to work well in all the examples we tried, but there are ad-hoc examples of very short definitions that lead to exponential totality checking.

We think (hope) that such example do not arise naturally and letting the user choose the bounds B and D (with sane default values) allows to limit the combinatorial explosion to the definitions that really need it. It is nevertheless difficult to know how this will scale for very big definitions. The situation is thus not too different from Agda, where the termination checker can become very slow on big definitions.

This should be contrasted to Coq, where the design choice has always been to have a very simple totality checker with linear complexity.

Choosing the bounds. The totality test is parametrized by the bounds $B > 0$ and $D \geq 0$. In many simple cases, $B = 1$ and $D = 0$ are enough but increasing the bounds locally is interesting in the following cases.

- Increasing B helps detect totality when some calls increase the size of their argument (or dually, remove some output constructor). For example, the following ad-hoc example is accepted with $B = 2$ and $D = 0$ but rejected with $B = 1$ and $D = 0$:

```
val s1 = s2.Tail⁰
and s2 = { hd⁰ = Zero¹; Tail⁰ = { hd⁰=1¹; Tail⁰=s1 }}
```

  The call-graph has 2 vertices and 2 arcs: $\texttt{s1} \mapsto \langle 1\rangle^0\ \texttt{s2}$ and $\texttt{s2} \mapsto \langle -2\rangle^0\ \texttt{s1}$. When projecting with $B = 2$, the composition gives $\texttt{s1} \mapsto \langle -1\rangle^0\texttt{s1}$ (and similarly for `s2`), which passes the totality test. When projecting with $B = 1$, the first arc gives $\texttt{s1} \mapsto \langle\infty\rangle^0\texttt{s2}$ which gives compositions of $\texttt{s1} \mapsto \langle\infty\rangle^0\texttt{s1}$ (and similarly for `s2`), which doesn't pass the totality test.
- Increasing D helps detecting "incompatible" calls. For example, the following ad-hoc example is accepted with $D = 1$ but rejected with $D = 0$:

```
val f (C1 x) = 0
  | f (C2 x) = f (C1 x)
```

  The call-graph has a single vertex with a single call $\alpha = \texttt{f} \mapsto \texttt{f (C1 C2}^{\texttt{-}}\texttt{ x)}$. With $D = 0$, this call is collapsed to $\texttt{f} \mapsto \texttt{f (}\langle 0\rangle\texttt{x)}$ which doesn't pass the totality test because this loop is idempotent but doesn't decrease. With $D = 1$, this call is unchanged but is not idempotent: $\alpha \diamond \alpha = \mathbf{0}$, and it passes the totality test.

  Similarly, if some parts of the argument increase while other parts decrease, too small a D can hide totality:

```
val f {Fst⁰=0 ; Snd⁰=x} = x
  | f {Fst⁰=Succ¹ x1 ; Snd⁰=x2} = f {Fst⁰=x1 ; Snd⁰=Succ¹ x2}
```

  requires $D > 0$ to pass the totality test. With $B = D = 0$, we get the coherent loop

$$\texttt{f} \mapsto \texttt{f}\ \Big(\ \langle\langle\infty\rangle^0 + \langle -1\rangle^1\rangle\ \texttt{x * } \langle\langle\infty\rangle^0 + \langle\infty\rangle^1\rangle\ \texttt{x}\ \Big)$$

  which doesn't satisfy the hypothesis of Theorem 4.28. With $B = 0$ and $D = 1$, we get

$$\texttt{f} \mapsto \texttt{f \{ Fst = } \langle -1\rangle^1\texttt{ x ; Snd = } \langle\infty\rangle^1\texttt{ x \}}$$

  which does satisfy the second property of Theorem 4.28.

In practice, we've found that $B = 2$ and $D = 2$ is enough for most cases. In the few situation where increasing B or D is helpful, the programmer can change those bounds locally.

Clauses vs match expressions. 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:

- `chariot` uses rewriting rules,
- its semantics (the domain $\mathcal{F}$) uses `C⁻` which is closer to (partial) `match` expressions.

The totality checker for `chariot` was implemented internally with clauses but the theory of approximations is much simpler with partial match and projections. The advantage of partial match is that it allows direct analysis of definitions like

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

by producing a term $u[\texttt{y} := \texttt{C}^{\texttt{-}}\texttt{f}v]$. On the other hand, keeping rewriting rules make the totality checker detect that

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

is total. Using partial match expressions as done 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 the choice is thus mostly a matter of taste. Of course, combining the 2 approaches is possible!

Operational Semantics. We have voluntarily refrained from giving the operational semantics of the language. The idea is that totality is a semantic property. The operational semantics has be compatible with the standard semantics of recursive definitions The operational semantics must also 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 head 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.

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.[15] 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.

Higher order types. The implementation of `chariot` does deal with some higher order datatypes. With T-branching trees (coinductive) defined as

```
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)
```

the corresponding `map` function passes the totality test. The theory should extend to account for this kind of datatypes.

[15]The interaction between such coinductive constructors and dependent types is however very subtle as they can break subject reduction! `https://github.com/coq/coq/issues/6768`.

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 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.

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.

[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.

[APTS13] Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. Copatterns: Programming infinite structures by observations. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, page 27–38, New York, NY, USA, 2013. Association for Computing Machinery.

[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] Jéôme 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 Hyvernat. 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.

This work is licensed under the Creative Commons Attribution License. To view a copy of this license, visit https://creativecommons.org/licenses/by/4.0/ or send a letter to Creative Commons, 171 Second St, Suite 300, San Francisco, CA 94105, USA, or Eisenacher Strasse 2, 10777 Berlin, Germany