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. Naively using the size-change principle to check productivity and termination is straightforward but unsound when inductive and coinductive types are nested. We can however adapt the size-change principle to check "totality" [Hyv25], which corresponds exactly to correctness with respect to the corresponding (co)inductive type.

1. Introduction

One of the goals of strong typing in languages like Caml or Haskell, and the heart of Hindley-Milner type checking / type inference, is to catch a whole class of errors before they actually happen: if a piece of code is accepted (at compile time), evaluation cannot fail (at run time). Of course, the program can be incorrect but functions can only be applied to arguments they are ready to accept. The dreaded results `segmentation fault`, `core dumped` or `NullPointerException` are, in theory, a thing of the past. And even if, in practice, bindings to libraries written in other languages allow such errors to creep back into the language, this additional guarantee is a strong selling point.

In proof assistants based on type theory like Coq [The04] or Agda [Nor08], strong typing is even more important as it implies consistency: no closed element of the empty type can be defined. However, typing alone cannot prevent such ill-formed definitions as

```
val magic = magic
```

which is well-typed but belongs to all types. This recursive definition is for example valid in Haskell and while languages like Caml or SML only allow recursive definitions for explicit functions they accept the following variants

Received by the editors October 20, 2025.
Key words and phrases: coinductive types; nested fixed points; size-change principle; functional programming; recursive definitions.
This work was partially funded by ANR project RECIPROG, project reference ANR-21-CE48-019-01.

```
(* Caml syntax *)
# let rec magic x = magic x;;
val magic : 'a -> 'b = <fun>
```

```
(* SML syntax *)
> val rec magic = fn x => magic x;
val magic = fn: 'a -> 'b
```

Coq and Agda take different approaches to reject such definitions:

(1) Coq restricts a priori the syntax and type system so that only terminating functions can be written,
(2) Agda forbids some definitions a posteriori by using an external termination checker.

Because the halting problem is undecidable [Tur36], the first approach cannot give a Turing complete language. The language `charity` [CF92] takes a similar approach: recursion is only available in the form of a typed combinators that enforce that some argument of the recursive function is structurally decreasing.

In Agda however, unrestricted recursion is syntactically possible and as a programming language, Agda is Turing complete. The termination checker warns about well-typed definitions that may lead to non termination. The size-change principle [LJBA01, Hyv14] is particularly well suited for this task. Note however that undecidability of the halting problem now implies that the termination checker will reject some correct functions. One advantage of this approach is that it is in theory easy to combine several termination checkers and that this doesn't impact the underlying type theory. The disadvantage is that the validity of a recursive definition now depends on some external "oracle".

In the presence of coinductive types like streams or infinite trees, the situation is more complex because we need to prevent infinite computations by adding some laziness to the evaluation. Both approaches can still be used, but the second becomes more complex. In simple cases, checking that a definition is *productive* [Coq94][1] is enough to guarantee termination and validity of the definition. Unfortunately, as shown by T. Altenkirch and N. A. Danielsson [AD12, Section 5], checking termination and productivity independently is not enough to guarantee that a recursive definition involving nested (co)inductive types is valid.

This paper presents a provably correct validity checker for recursive definitions in a first order language. It relies on a characterization of mixed inductive / coinductive types from a previous paper [Hyv25]. I have tried to make this paper as self contained as possible but additional motivations and references to other works can be found there. The resulting totality checker is based on the size-change principle and generalizes both standard termination and productivity. It cannot deal with the full cornucopia of Agda's dependent types but possible extensions are mentioned in the conclusion.

1.1. **`chariot`.** `chariot` is the prototype language developed for experimenting with the principle described in this paper.[2] Simply put, `chariot` is a non-strict, first-order, strongly typed, purely functional language. Like the `Charity` language [CF92, Coc96], `chariot` has both standard inductive types and coinductive types. Implementation details for the language `chariot` are irrelevant and most ideas can be found in standard references [PJ87].

[1] Productivity of a recursive definition means that when unfolding the definition, a coinductive constructor is bound to be output in a finite time. Because of laziness, this implies that no computation can go on forever. The syntactical notion of *guardedness*, i.e. that all recursive call appear directly below a coinductive constructor is easily checked and implies productivity but is very restrictive.

[2] `chariot` is written in Caml and is freely available: `https://github.com/phyver/chariot`

Unlike `Charity`, functions are defined by recursion with no restriction besides standard Hindley-Milner type checking [Mil78]. Writing functions in `chariot` is thus closer to writing functions in ML, Haskell or Agda than it is to writing functions in `Charity` or Coq.

Examples. The syntax was formally described in a previous paper [Hyv25] and won't be repeated. It should be readable by anyone familiar with Caml or Haskell. One thing to remember is that inductive types are given by constructors while coinductive types are given by destructors. Here are some examples that should give a taste of what `chariot` looks like and what to expect from the totality checker. First, a simple example involving only inductive types:

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

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

val length : list(x) -> nat
  | length Nil = Zero
  | length (Cons _ l) = Succ (length l)
```

The `stream` datatype is coinductive with destructors giving access to the head and tail of a stream. We can define a stream using a record notation as in the `nats` definition below.

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

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

The definition of `length` is structurally decreasing and the definition of `nats` is syntactically guarded. It should come as no surprise that they are accepted by the totality checker. The next one is more interesting. The function adds the elements of each list in a stream of lists using an accumulator that is reset for each list. This definition is a little ad hoc but illustrates some strength[3] of the totality checker.

```
val sums : nat -> stream(list(nat)) -> stream(nat)
  | sums acc { Head = Nil ; Tail = s } =
                 { Head = acc ; Tail = sums Zero s }
  | sums acc { Head = Cons(n, l) ; Tail = s } =
                 sums (add acc n) { Head = l ; Tail = s }
```

where `add` is the usual addition of natural numbers. Even though it contains a non guarded recursive call (the second one), the definition is still productive. Since the head of the stream gets structurally smaller (it is a list), there can be no infinite sequence of consecutive calls using only the second clause: the first clause must be used after a finite time, adding a stream constructor to the result. This makes the definition productive.

As a last example, here is a non-total recursive definition that is nevertheless terminating and productive [AD12].

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

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

[3]For example, Agda doesn't detect that this function terminates.

Because the type `stree` is inductive without any base constructor, it is empty. The definition of `bad_s` is however well typed (for Hindley-Milner) and productive. It would unfold to a stream of non well-founded infinitary trees. From there, it is easy to construct a "magic" value having all types by recursing into `bad_s`:

```
val lower_left : stree -> 'x
  | lower_left (Node s) = lower_left (s.Head)
val magic : 'x
  | magic = lower_left bad_s.Head
```

Note the important fact that the problem doesn't come from `lower_left`, which is trivially total (it is structurally decreasing on an inductive type). It comes from the definition of `bad_s`, which will be rejected by the totality checker described in this paper.

Operational semantics. Totality is a property of the *denotational semantics* of a recursive definition. Because of that, the operational semantics of `chariot` is not very important. `chariot` uses a lazy evaluation of records, which is enough to guarantee that all computation involving total functions and values are finite.

Restrictions. To simplify the presentation, we only consider the first order fragment of the `chariot` language, with the following restrictions:[4]

- all functions are fully applied,
- all functions and constructors take exactly one argument,
- there are no mutually defined recursive functions,
- the empty record is forbidden in recursive definitions.

The last one is a little ad hoc but makes for a simpler theory. If we insist that all constructors have a single argument, the empty record becomes the only atomic term. For example, the constructor `0` must have type `unit -> nat` and is used as "`Zero{}`". In recursive definitions however, we can remove it:

- in the pattern-matching of a clause, it can be replaced by a fresh, dummy variable:

  ```
  | f (Zero {}) = Succ ...
  ```

  becomes

  ```
  | f (Zero _x) = Succ ...
  ```

  which has the same semantics;
- in the right hand side of a clause, it can be replaced by one of those dummy variables, and

  ```
  | f (Zero {}) = Succ (Zero {})
  ```

  simply becomes

  ```
  | f (Zero _x) = Succ (Zero _x)
  ```

  If no such dummy variable is present on the pattern matching side, we can rely on an auxiliary ad hoc function

  ```
  | g (Succ x) = Zero (empty_record x)
  ```

  This function `empty_record`, of type `nat -> unit` cannot be written in our fragment but needs to be considered an outside library function that our test cannot inspect. Like all such functions, we'll only assume they are total.

With that in mind, the definition of `length` we actually deal with in this paper is

[4]Dealing with the full `chariot` language is possible at the cost of additional overhead.

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

Because this transformation can easily be automatized, we will write some examples using the unrestricted `chariot` language and rely on a preprocessor (the reader) to translate them to the restricted syntax. This will help keep the example more familiar.

1.2. **Recap of previous work.** Values are built from structures and constructors but because of coinductive types, they can be infinite [Hyv25].

Definition 1.1. The domain $\mathcal{V}$ (for "$\mathcal{V}$alues") is defined *coinductively* by the grammar

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

where

- each $\mathtt{C}$ 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. (Empty records are legal values, they are only forbidden in recursive definitions.)

The order on $\mathcal{V}$ is inductively generated by

(1) $\bot \leq v$ for all values v,
(2) "$\leq$" is contextual: if $u \leq v$ then $C[\mathtt{x} := u] \leq C[\mathtt{x} := v]$ for any value C with variable $\mathtt{x}$, where substitution is defined in the obvious way.

Contextuality, together with the fact that we generate an order, implies that comparing records is done component wise: if $u_1 \leq v_1$ and $u_2 \leq v_2$, then

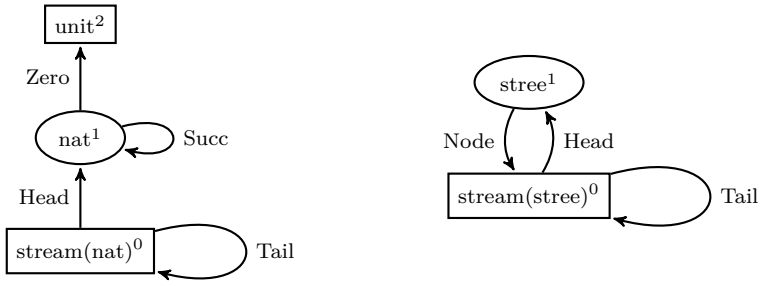
$$\begin{aligned} \{\mathtt{D}_1 = u_1; \mathtt{D}_2 = u_2\} &\leq \{\mathtt{D}_1 = v_1; \mathtt{D}_2 = u_2\} && \text{(contextuality, with } C = \{\mathtt{D}_1 = \mathtt{x}; \mathtt{D}_2 = u_2\}) \\ &\leq \{\mathtt{D}_1 = v_1; \mathtt{D}_2 = v_2\} && \text{(contextuality, with } C = \{\mathtt{D}_1 = v_1; \mathtt{D}_2 = \mathtt{x}\}) \end{aligned}$$

In many cases, compatible structures are simply not comparable, like $\{\mathtt{D}_1 = \bot, \mathtt{D}_2 = \{\}\}$ and $\{\mathtt{D}_1 = \{\}, \mathtt{D}_2 = \bot\}$.

Note that the set of values is defined coinductively but the order is defined inductively: it is the *least* order subject to some conditions. Because of that, reasoning about inequalities is usually done with standard inductive proofs.

Inductive types are interpreted by least fixed points and coinductive types are interpreted by greatest fixed points [Hyv25], but in this case, they coincide in domains! A value in a given type is *total* when it belongs to the appropriate fixed point in Set. For example, the infinite value $\mathsf{Succ}^\infty = \mathsf{Succ}\,\mathsf{Succ}\,\mathsf{Succ}\ldots$ is a valid element of $\mathcal{V}$ but is not total for the type `nat`, whose Set-based interpretation only contains the finite natural numbers.

The main result of the previous paper [Hyv25] is an "untyped" characterization of total values for a given type as winning strategies for a parity game constructed from the type. This is done by tagging each type with a *priority* that is odd for datatypes and even for codatatypes. Those priorities are taken from a parity game computed from the types involved in the definition. For `stream(nat)` and `stree`, they are



Note that the arcs for inductive constructors are reversed: the `Node` transition goes from `stree` to `stream(stree)` whereas its type is `stream(stree)` $\to$ `stree`. The reason is that terms are interpreted as strategies for the corresponding game, which entails making the transitions deconstruct a term into its subterms.

The constraints are that datatypes [resp. codatatypes] have odd [resp. even] priorities and that if type S is a sub-expression of type T, then the priority of S is greater than the priority of T. Here, the priority of `nat` is indeed greater than the priority of `stream(nat)`. Because of that, priorities keep information about the type of fixed point (inductive or coinductive type) *and* their nesting, which is important to distinguish between "inductive of coinductive" and "coinductive of inductive" definitions. Values of type T correspond exactly to strategies starting from T. The important result is the following.

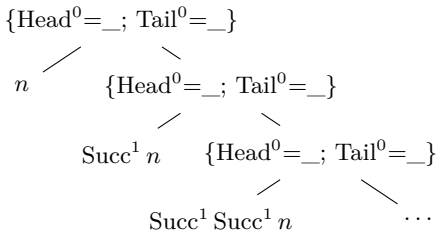
Proposition 1.2 [Hyv25, Corollary 2.11]**.** *A value v of type T is total iff the corresponding strategy is winning for the associated parity game, i.e. if no branch of v contains $\bot$ and along all infinite branches of v, the maximal priority that appears infinitely often is even.*

Constructors and destructors appearing in a definition can be tagged with the corresponding priority during type checking.[5] For example, the above definitions become:

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

val nats : nat¹ -> stream⁰(nat¹)
  | nats x = { Head⁰ = n ; Tail⁰ = nats (Succ¹ x) }

val sums : nat¹ -> stream⁰(list¹(nat¹)) -> stream⁰(nat¹)
  | sums acc { Head⁰ = Nil¹ ; Tail⁰ = s } =
      { Head⁰ = acc ; Tail⁰ = sums (Zero³{}) s }
  | sums acc { Head⁰ = Cons¹ {Fst⁰ = n ; Snd⁰ = l} ; Tail⁰ = s } =
      sums (add acc n) { Head⁰ = l ; Tail⁰ = s }

val bad_s : stream⁰(stree¹)
  | bad_s = { Head⁰ = Node¹ bad_s ; Tail⁰ = bad_s }
```

For the simple case of `nats`, suppose n is a total natural number. The value generated from `nats` n is

[5]Formally speaking, it is the record constructor ("curly bracket") that is tagged with a priority. We usually put the priority on fields for readability.

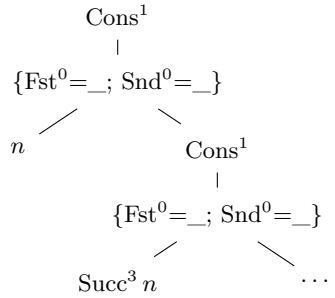


For the infinite branch $\texttt{\{Tail}^0\texttt{=\{Tail}^0\texttt{=\{Tail}^0\texttt{=...\}\}\}}$, the maximal priority appearing infinitely often is even. All other infinite branches would end with an infinite branch in n which is supposed to be total. The value `nats` n is thus total in `stream(nat)`.

Contrast this with the recursive definition using lists instead of streams:

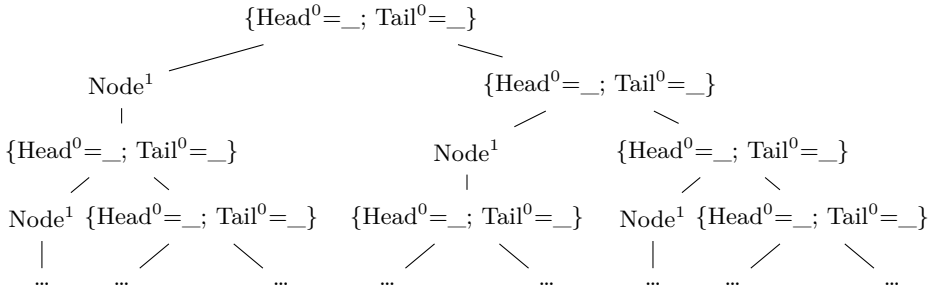
```
val nats_list : nat³ -> list¹(nat³)
  | nats_list x = Cons¹ { Fst⁰ = x ; Snd⁰ = nats_list (Succ³ x) }
```

Here, `nats_list` n gives



which contains the branch $\texttt{Cons}^1\texttt{\{Snd}^0\texttt{=Cons}^1\texttt{\{Snd}^0\texttt{=Cons}^1\texttt{\{Snd}^0\texttt{=...\}\}\}}$ where the maximal priority appearing infinitely often is odd. This value is not total, as expected.

The definition of `bad_s` is non total because it unfolds to



with leftmost infinite branch

$$\texttt{\{Head}^0\texttt{=Node}^1\texttt{\{Head}^0\texttt{=Node}^1\texttt{\{...\}\}\}}$$

which is non-total.

For the rest of this paper, we assume the recursive definitions have been tagged with priorities, i.e. that all constructor [resp. destructor] names come with an odd [resp. even] priority. All other typing information has been removed and is irrelevant for totality. Constructors and destructors used in values (Definition 1.1) also carry a priority.

Definition 1.3. An element $v \in V$ is *total* if it doesn't contain $\bot$ and for all infinite branches of v, the maximal priority appearing infinitely often in the branch is even.

For any given list of recursive definitions, the set of priorities is finite, so that the definition always makes sense. We usually call the "maximal priority appearing infinitely often" the *principal priority*. By Proposition 1.2, those values correspond precisely to the total values of the original type.

The usual semantics of a recursive definition is a function computed using Kleene's fixed point theorem. We call a function *total* if it sends total values to total values. This paper describe a computable totality test taking a recursive definition as argument and answering either "`YES`, the semantics of this definition is total" or "`I DON'T KNOW` whether the semantics of this definition is total or not." Since checking totality is undecidable,[6] nothing more can be expected.

1.3. **Plan of the Paper.** We start by giving, in Section 2, an interpretation of recursive definitions that is mathematically simpler than the ordered lists of clauses used in `chariot`. Thanks to the characterization of mixed inductive / coinductive types [Hyv25] recalled in Section 1.2, we can make this semantics untyped, which means we only have to consider a single domain of values. The two steps necessary to define this interpretation are

- instead of only considering the first matching clause, we consider all of them and take their non-deterministic sum,
- since a clause can now be applied to non matching value, we introduce a notion of error, which is nothing more than the empty non-deterministic sum.

Note that errors (and general sums) are just an artifact produced by the totality checker. They are not part of the `chariot` operational semantics, where Hindley-Milner type checking is precisely meant to prevent their apparition.

Non-deterministic values are an instance the usual Smyth power domain, but interpreting definitions and clauses requires more care and technicalities (Sections 2.3 and 2.4).

The resulting interpretation is still too complex: we thus split each clause of the recursive definition into the sum of its recursive calls (Section 3.1). While going from the standard to the non-deterministic sum left the semantics "mostly" unchanged, splitting a definition into its call-graph results in a very different semantics. That's not a problem because, as shown in Proposition 3.8, this simplification reflects totality.

The last step before applying the size-change principle, detailed in Sections 3.2 and 3.3, is to show how we can collapse the call-graph inside a finitary structure by introducing approximations that forget about parts of the terms. Doing so in a consistent way introduces subtle difficulties, like composition of terms becoming non-associative.

Everything is then in place to apply the size-change principle from C. Lee, N. Jones and A. Ben-Amram [LJBA01]. This is done in Section 4.1.

The paper then gives detailed examples to show how the totality checker reaches its conclusion and some remarks about the actual implementation of the totality checker.

[6] Recall that without coinductive types, totality is just termination.

2. Non-Deterministic Semantics for Definitions

A `chariot` recursive definition is a complex object: an ordered set of clauses accepted by the Hindley-Milner type checking algorithm. We first interpret them in a standard mathematical structure (a domain) with a simple syntactical representation. The key ideas are to use a non-deterministic (commutative) sum of untyped clauses to replace ordered set of clauses and to allow runtime errors in the model.

2.1. Smyth Power Domain.

A *domain* will be an *algebraic DCPO*. Basic definitions and important results about domain theory are recalled in Appendix B. An important tool for constructing domains is the *ideal completion* which transforms any partial order into an algebraic DCPO whose compact elements are exactly the elements of the original partial order. Refer to Appendix B for details.

The Smyth power domain construction [Smy78] adds a binary greatest lower bound operation "+" to any domain. It is similar to adding arbitrary lower bounds for a partial order by considering upper closed sets ordered by reverse inclusion: instead of taking all upper closed sets, only some of them are used. Refer to Appendix C for some details and additional references. The important point is that any element of the Smyth power domain can be seen as a formal sum of elements of the starting domain, ordered by

$$\sum_i u_i \leq \sum_j v_j \quad \text{iff} \quad \forall j, \exists i, u_i \leq v_j \ .$$

Definition 2.1. Let $\mathcal{S}$ (for "$\mathcal{S}$ums", or "$\mathcal{S}$myth") be the domain obtained from $\mathcal{V}$ by:
(1) taking the Smyth power domain construction over $\mathcal{V}$,
(2) adding a greatest element $\mathbf{0}$, which can be identified with the empty sum.

Lemma 2.2. *The greatest element* $\mathbf{0}$ *is neutral for "+".*

Proof. Note that adding $\mathbf{0}$ as a greatest element to a domain is always possible:

- $\mathbf{0}$ will be compact because any limit that doesn't contain it exists in the original domain, and is thus different from $\mathbf{0}$;
- a directed set containing $\mathbf{0}$ has $\mathbf{0}$ as a limit, and a directed set not containing $\mathbf{0}$ has a limit in the original domain;
- any element different from $\mathbf{0}$ is the limit of the compact elements below it (because that's the case in the original domain) and since $\mathbf{0}$ is compact, it is the limit of the compact elements below it.

That $\mathbf{0}$ is neutral for $+$ is a direct consequence of $+$ being the greatest lower bound operation. Refer to Appendix C for details. □

The domain $\mathcal{S}$ contains all the original values from $\mathcal{V}$, which we call *simple values*. All its elements are "formal" sums of elements of $\mathcal{V}$. Those formal sums can be empty ($\mathbf{0}$), unary (simple values), finite (greatest lower bounds) or infinite. Only infinite sums that can be obtained as limits of finite sums exist. For example, the sum of all maximal elements of `nat`, i.e. $\texttt{Zero} + \texttt{Succ Zero} + \texttt{Succ Succ Zero} + \cdots$ is the limit of

$$\bot \quad \leq \quad \texttt{Zero} + \texttt{Succ} \ \bot \quad \leq \quad \texttt{Zero} + \texttt{Succ Zero} + \texttt{Succ Succ} \ \bot \quad \leq \quad \ldots$$

However, the sum of all *total* elements of `nat`, i.e. the same sum *without* $\texttt{Succ}^\infty$ doesn't exist in $\mathcal{V}$. Totality on $\mathcal{S}$ is defined in the expected way.

Definition 2.3. An element $t \in \mathcal{S}$ is total if all its summands are total in $\mathcal{V}$.

The value $\mathbf{0}$ will be used as the semantics for runtime errors. It may be surprising that errors are total but because type checking precisely implies that those error never happen when running actual programs, they can be seen as be artefacts introduced by the totality checking mechanism.

As previously shown (Lemma 1.8, [Hyv25]), total elements of $\mathcal{V}$ are maximal. The next lemma is a direct corollary and implies that totality is compatible with the pre-order on $\mathcal{S}$: if $t_1 \approx t_2$, then t_1 is total iff t_2 is.

Lemma 2.4.

- *If $t_1 \leq t_2$ in $\mathcal{S}$ and if t_1 is total, then so is t_2.*
- *if $f \leq g$ in $\mathcal{S} \to \mathcal{S}$ (for the pointwise order) and f is total, then so is g.*

Proof. Let $t_1 = \sum T_1$ and $t_2 = \sum T_2$. Writing $X^\uparrow$ for the upward closure of X, we have $t_1 \leq t_2$ iff $T_2^\uparrow \subseteq T_1^\uparrow$. If T_1 is total, it only contains total elements and by Lemma 1.8 from [Hyv25], $T_1^\uparrow = T_1$. As a result $T_2^\uparrow$ only contains total elements and T_2 only contains total elements as well. The second point follows directly, as a total function is simply a function sending total elements to total elements. □

2.2. **Recursion and Fixed Points.**

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. This fixed point is equal to (Kleene theorem)

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

We are interested in fixed points of operators from $[\mathcal{S} \to \mathcal{S}]$ to itself and we require that all functions satisfy $f(\mathbf{0}) = \mathbf{0}$, i.e. that errors propagate. We write Ω for

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

All the fixed points we are computing are of the form

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

with $\varphi : [\mathcal{S} \to \mathcal{S}] \to [\mathcal{S} \to \mathcal{S}]$. They will simply be denoted by $\mathsf{fix}(\varphi)$. The following is a direct consequence of Kleene's formula.

Lemma 2.5. *If $\Omega \leq \theta(\Omega)$ and $\theta \leq \phi$ in $[\mathcal{S} \to \mathcal{S}] \to [\mathcal{S} \to \mathcal{S}]$, then $\mathsf{fix}(\theta) \leq \mathsf{fix}(\phi)$ in $\mathcal{S} \to \mathcal{S}$.*

Non-deterministic semantics. We extend the standard semantics of `chariot` functions to accept arbitrary values in $\mathcal{V}$. Overlapping clauses introduce non-determinism and the empty sum $\mathbf{0}$ naturally arises when no clause matches a value.

Recall that *linear patterns* are inductively generated by the following grammar

$$p \quad ::= \quad \mathtt{x} \;\mid\; \mathtt{C}\,p \;\mid\; \{\mathtt{D}_1 = p_1; \ldots; \mathtt{D}_n = p_n\}$$

with the restriction that variables occur at most once. The standard semantics ([Hyv25, Definition 1.10]) for a recursive definition of `f` is the fixed point of the following operator:

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

where "`f p = u`" is the first clause from the definition of `f` where p matches v. When `f` is of type $A \to B$, its standard semantics is a function from the interpretation of A to the interpretation of B.

The fact that we use the first matching clause means in particular that clauses are not independent of each other: for example, the pattern

```
| f Zero = Zero
```

doesn't necessarily say anything about the value of `f` on `Zero`, as it could come after

```
| f x = Succ Zero
```

One way to break this dependency on the order of clauses is to use non-determinism. Rather than take the first matching clause, we take the sum of all clauses. Non matching clause evaluate to $\mathbf{0}$ and do not contribute to the sum and non overlapping clauses do not interfere with each other: at most one term is non $\mathbf{0}$. When there are overlapping clauses, some information is lost. An extreme case would be

```
val f : nat -> nat
  | f x =  Zero
  | f Zero = f Zero
```

The second clause is never used when evaluating `f Zero` because `Zero` matches the first clause. With the non deterministic semantics, `f Zero` would evaluate to `Zero` + `f Zero`, which would loop: the semantics of `f Zero` would thus be $\bot$. Fortunately such example are very rare in practice, and the advantages of this simplification more than compensate for that. The concept of "operators" (Section 2.3) and call-graph (Section 3.1) are based on this.

Formally, the standard semantics of some `f` of type $A \to B$ is extended to the whole of $\mathcal{S}$, which contains all terms: those in the interpretations of A, B and all possible types.

Definition 2.6.

(1) Given a linear pattern p and a simple value v, the unifier $[p := v]$ is the substitution defined inductively with
- $[\mathtt{y} := v] = [\mathtt{y} := v]$ where the RHS is the usual substitution of `y` by v,
- $[\mathtt{C}p := \mathtt{C}v] = [p := v]$,
- $[\{\mathtt{D}_1 = p_1; \ldots; \mathtt{D}_n = p_n\} := \{\mathtt{D}_1 = v_1; \ldots; \mathtt{D}_n = v_n\}] = [p_1 := v_1] \cup \cdots \cup [p_n := v_n]$ (because patterns are linear, the unifiers don't overlap),
- in all other cases, the unifier is the substitution giving $\mathbf{0}$ for all variables. Those cases are:
  - $[\mathtt{C}p := \mathtt{C}'v]$ with $\mathtt{C} \neq \mathtt{C}'$,
  - $[\{\ldots\} := \{\ldots\}]$ when the 2 records have different sets of fields,

– $[\mathtt{C}p := \{\ldots\}]$ and $[\{\ldots\} := \mathtt{C}v]$.

If only the first three cases are encountered when computing $[p := v]$, we say that *the value v matches the pattern p.*

(2) Given a recursive definition for $\mathtt{f}$ and an environment ρ for all other functions, define the **n**on-**det**erministic semantics $\Theta^{\mathrm{ndt}}_{\rho,\mathtt{f}} : [\mathcal{S} \to \mathcal{S}] \to [\mathcal{S} \to \mathcal{S}]$ as follows. Suppose $f : \mathcal{S} \to \mathcal{S}$,
- $\Theta^{\mathrm{ndt}}_{\rho,\mathtt{f}}(f)\big(\sum v\big) = \sum \Theta^{\mathrm{ndt}}_{\rho,\mathtt{f}}(f)(v)$.
- For $v \in \mathcal{V}$ a simple value, define

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

where the sum ranges over all clauses of the definition.

(3) The non-deterministic semantics of the function $\mathtt{f}$ is then $\mathsf{fix}\big(\Theta^{\mathrm{ndt}}_{\rho,\mathtt{f}}\big) : \mathcal{S} \to \mathcal{S}$.

Because there is no guarantee that v matches some clause of the definition, we can have $\Theta^{\mathrm{ndt}}_{\rho,\mathtt{f}}(f)(v) = \mathbf{0}$. Even with matching clauses, this semantics does not necessarily coincide with the standard one. For example, consider the following two definitions of the halving function:

```
val half1 : nat -> nat
  | half1 (Succ (Succ n)) = Succ (half1 n)
  | half1 (Succ Zero)     = Zero
  | half1 Zero            = Zero
```

and

```
val half2 : nat -> nat
  | half2 (Succ (Succ n)) = Succ (half2 n)
  | half2 n               = Zero
```

Because patterns are disjoint in the first definition, the order of clauses is not important. On natural numbers, the standard and non-deterministic semantics coincide. For the second definition however, we get different semantics:

$$\begin{array}{rcl} \Theta^{\mathrm{std}}_{\mathtt{half2}}(f)(\mathtt{Succ\ Succ}\ n) & = & \mathtt{Succ}(f(n)) \\ & \neq & \\ \Theta^{\mathrm{ndt}}_{\mathtt{half2}}(f)(\mathtt{Succ\ Succ}\ n) & = & \mathtt{Zero} + \mathtt{Succ}(f(n)) \end{array}$$

The additional "`Zero`" comes from the interpretation of clause "`half2 n = Zero`" in the definition of `half2`, which can be applied to the argument `Succ Succ` n.

To formally compare the two semantics, we extend typed functions to $\mathcal{S}$.

Lemma 2.7. *If $f : [\![A]\!] \to [\![B]\!]$ is a continuous function between the interpretations of 2 types, define $\widehat{f} : \mathcal{S} \to \mathcal{S}$ by*

- $\widehat{f}\big(\sum v\big) = \sum \widehat{f}(v)$*, where $\sum v$ is a sum of simple terms;*
- $\widehat{f}(v) = \begin{cases} f(v) & \text{if } v \in [\![A]\!] \\ \mathbf{0} & \text{if } v \notin [\![A]\!] \end{cases}$ *where v is any simple term in $\mathcal{S}$.*

We have

- *$\widehat{f}$ is continuous iff f is continuous,*
- *$\widehat{f}$ is total iff f is total.*

Extending functions in this way doesn't change their standard (typed) fixed points or totality:

Lemma 2.8. *For a definition of* $\mathtt{f}$ *of type* $A \to B$*, an environment* ρ *and* $f : [\![A]\!] \to [\![B]\!]$*, let* $\widehat{\Theta^{\mathrm{std}}_{\rho,\mathtt{f}}}$ *be the lifting of the usual semantics of* $\mathtt{f}$*. We have*

(1) $\Theta^{\mathrm{std}}_{\rho,\mathtt{f}}(f) = \widehat{\Theta^{\mathrm{std}}_{\widehat{\rho},\mathtt{f}}}(\widehat{f}) \restriction [\![A]\!]$ *(i.e. "the standard semantics is the restriction of its lifting"),*

(2) $\mathsf{fix}(\Theta^{\mathrm{std}}_{\rho,\mathtt{f}}) = \mathsf{fix}\big(\widehat{\Theta^{\mathrm{std}}_{\widehat{\rho},\mathtt{f}}}\big) \restriction [\![A]\!]$*, ("the standard fixed point is the restriction of the lifted fixed point"),*

(3) *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 $\Theta^{\mathrm{std}\,n}_{\rho,\mathtt{f}}(\Omega)$ is equal to $\widehat{\Theta^{\mathrm{std}}_{\widehat{\rho},\mathtt{f}}}^{n}(\Omega) \restriction [\![A]\!]$, 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. □

Lemma 2.9. *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* $f : \mathcal{S} \to \mathcal{S}$*,*

(3) *If* $\mathsf{fix}(\Theta^{\mathrm{ndt}}_{\rho,\mathtt{f}}) : \mathcal{S} \to \mathcal{S}$ *is total then* $\mathsf{fix}\big(\widehat{\Theta^{\mathrm{std}}_{\rho,\mathtt{f}}}\big) : \mathcal{S} \to \mathcal{S}$ *is total as well.*

Proof. The first point is straightforward and the third point is a direct consequence of Lemma 2.5 and Lemma 2.4. 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 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, we can forget about the standard semantics and show totality of $\Theta^{\mathrm{ndt}}_{\rho,\mathtt{f}}$.

Corollary 2.10. *For a definition of* $\mathtt{f}$ *of type* $A \to B$*, if* $\mathsf{fix}\big(\Theta^{\mathrm{ndt}}_{\rho,\mathtt{f}}\big) : \mathcal{S} \to \mathcal{S}$ *is total, then so is* $\mathsf{fix}\big(\Theta^{\mathrm{std}}_{\rho,\mathtt{f}}\big) : [\![A]\!] \to [\![B]\!]$*.*

2.3. **Operators.**

2.3.1. *Terms.* The operators $\Theta^{\mathrm{ndt}}_{\rho,\mathtt{f}}$ are continuous functions from $[\mathcal{S} \to \mathcal{S}]$ to itself. This section introduces an inductively generated language containing them.

Definition 2.11.

(1) $\mathcal{O}_0$ (for "$\mathcal{O}$perators") is the set of terms inductively generated from

$$\begin{array}{rcl} t & ::= & \mathtt{C}^p t \;\mid\; \{\mathtt{D}_1 = t_1; \dots; \mathtt{D}_n = t_n\}^p \;\mid \\ & & \mathtt{C}^{p^-} t \;\mid\; \mathtt{.D}^p t \;\mid \\ & & \mathtt{f}\ t \;\mid\; \mathtt{x} \;\mid \\ & & \Omega t \;\mid \\ & & t_1 + \dots + t_n \end{array}$$

where $n > 0$, $\mathtt{x}$ is the only possible variable name and each $\mathtt{f}$ belongs to a finite set of function names. Each $\mathtt{C}$ and $\mathtt{D}$ comes from a finite set of constructor and destructor names and each p comes from a finite set of priorities (natural numbers). Those priorities are odd for constructors and even for destructors.

(2) Sums can be empty, in which case they are written $\mathbf{0}$.

(3) Terms are quotiented by associativity, commutativity and idempotence of $+$, together with (multi)linearity of all term constructors ($\mathtt{C}$, $\{\dots; \mathtt{D}{=}\_; \dots\}$, $\mathtt{C}^-$, $\mathtt{.D}$, $\mathtt{f}$, Ω).

(4) An element $t \in \mathcal{O}_0$ is called *simple* if it contains no sum (empty or otherwise).

Since all term constructors are linear, any term containing $\mathbf{0}$ and no other sum is automatically equal to $\mathbf{0}$, which is not simple. The full semantics of terms will be given on page 18, but in the meantime, it is helpful to keep the following in mind.

- Constructors $\mathtt{C}t$ and $\{\mathtt{D}_1 = t_1; \dots; \mathtt{D}_k = t_k\}$ construct values directly, just like in $\mathcal{V}$ or $\mathcal{S}$.
- Destructors $\mathtt{C}^- t$ and $\mathtt{.D}\,t$ deconstruct values (in $\mathcal{V}$) respectively by:
  - doing a pattern matching *which fails* if t doesn't start with constructor $\mathtt{C}$,
  - projecting a structure on field $\mathtt{D}$ which fails if t is not a structure with field $\mathtt{D}$.
- "$+$" is a non deterministic sum and the empty sum $\mathbf{0}$ represents an error.
- $\mathtt{f}$, $\mathtt{g}$, ... are function names and are either the function being recursively defined or a previously defined function meant to be replaced by a real function from the environment.
- Each $t \in \mathcal{O}_0$ represents a function depending on $\mathtt{x}$.
- If we identify $\mathtt{f}$ as the recursive function being defined, each $t \in \mathcal{O}_0$ can be seen as a function on $[\mathcal{S} \to \mathcal{S}] \to [\mathcal{S} \to \mathcal{S}]$, whose fixed point is what interests us.
- Ωt represents the function $v \mapsto \begin{cases} \mathbf{0} & \text{if } t(v) = \mathbf{0} \\ \bot & \text{otherwise} \end{cases}$

Because of the interaction between projections, partial matches, constructors and records, the order on $\mathcal{O}_0$ is more complex than on $\mathcal{S}$.

Definition 2.12. The order $\leq$ on $\mathcal{O}_0$ is inductively generated from

- $\mathbf{0}$ is the greatest element: $\forall t \in \mathcal{O}_0, t \leq \mathbf{0}$,
- contextuality: if $C \in \mathcal{O}_0$ is a context, then $t_1 \leq t_2 \implies C[t_1] \leq C[t_2]$,[7]
- $\Omega\mathtt{x} \leq \mathtt{x}$, and $\Omega\mathtt{x} \leq \mathtt{f}\mathtt{x}$ for each function name $\mathtt{f}$,
- $s + t \leq t$,

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

[7]Contexts are terms possibly containing a special variable $\Box$ and $C[t]$ is the result of substituting $\Box$ by t.

$$(*)\begin{cases}
(1) & \mathtt{C}^-\mathtt{C}t & \approx & t & \\
(1) & .\mathtt{D}_{i_0}\{\ldots;\mathtt{D}_i = t_i;\ldots\} & \geq & t_{i_0} & \\
(2) & \mathtt{C}^-\{\ldots\} & \approx & \mathbf{0} & \\
(2) & .\mathtt{D}\mathtt{C}t & \approx & \mathbf{0} & \\
(2) & .\mathtt{D}\{\ldots\} & \approx & \mathbf{0} & \text{if the record has no field } \mathtt{D} \\
(2) & \mathtt{C}^-\mathtt{C}'t & \approx & \mathbf{0} & \text{if } \mathtt{C} \neq \mathtt{C}' \\
(3) & \mathtt{C}^-\Omega t & \approx & \Omega t & \\
(3) & .\mathtt{D}\Omega t & \approx & \Omega t & \\
(3) & \Omega\mathtt{C}t & \approx & \Omega t & \\
(3) & \Omega\{\mathtt{D}_1 = t_1;\ldots;\mathtt{D}_n = t_n\} & \geq & \Omega t_1 + \cdots + \Omega t_n & \text{if } n > 0 \\
(3) & \Omega\Omega t & \approx & \Omega t &
\end{cases}$$

Groups (1) and (2) correspond to the intended operational semantics of the language. Group (3) contains (in)equalities that hold semantically and will be justified a posteriori by Definition 2.18 and Lemma 2.19.

Note in particular that projecting a structure on one of its fields (second inequality) yields a *smaller* term. This accounts for the fact that projecting may "hide" errors that could have occurred in other fields, as in

$$\begin{array}{lcll}
\mathbf{0} & \approx & .\mathtt{D}_1\{\mathtt{D}_1 = \mathtt{x}; \mathtt{D}_2 = \mathbf{0}\} & \text{(linearity)} \\
& \approx & .\mathtt{D}_1\{\mathtt{D}_1 = \mathtt{x}; \mathtt{D}_2 = \mathtt{C}^-\mathtt{C}'\ldots\} & \text{(fourth rule (2))} \\
& \geq & \mathtt{x} & \text{(second rule (1))}
\end{array}$$

Lemma 2.21 shows this is indeed the only possibility.

That the pre-order is "generated" from the above (in)equalities means that proving properties of the order can be done by induction. Any $s \leq t$ either comes from an (in)equality from the definition, or from reflexivity or transitivity. It is not obvious at first but corollary 2.20 will show that $\leq$ doesn't collapse to a trivial pre-order. Here are some simple consequences of the definition.

Lemma 2.13. *We have:*

(1) *If for all j, there is an i s.t. $s_i \leq t_j$, then $\sum_i s_i \leq \sum_j t_j$,*
(2) *$\Omega s \leq t$ iff $\Omega s \leq \Omega t$,*
(3) *$\Omega t \leq \Omega\mathtt{C}^- t$,*
(4) *$\Omega t \leq \Omega .\mathtt{D} t$,*
(5) *for all t, $\Omega\mathtt{x} \leq t$.*

Proof.

(1) It is a direct consequence of contextuality and the fact that $s + t \leq t$. The special case where $\sum_j t_j$ is the empty sum follows from the fact that $\mathbf{0}$ is the greatest element.
(2) Suppose $\Omega s \leq t$, we have $\Omega\Omega s \leq \Omega t$ by contextuality, and since $\Omega\Omega s \approx \Omega s$, we have $\Omega s \leq \Omega t$. The converse is a consequence of transitivity and the fact that $\Omega t \leq t$.
(3) Because $t \geq \Omega t$, we have $\Omega\mathtt{C}^- t \geq \Omega\mathtt{C}^-\Omega t$ by contextuality; and since $\mathtt{C}^-\Omega t \approx \Omega t$, we have $\Omega\mathtt{C}^- t \geq \Omega\Omega t \approx \Omega t$.
(4) The third point is proved similarly.
(5) The last point is proved by induction on t:
   • This is obvious if $t = \mathtt{x}$.

- If $t = \Omega t'$, we have $\Omega\mathtt{x} \leq t'$ by induction and the result follows from the first point.
- Similarly, if t is a sum, the result follows directly from the induction hypothesis.
- If $t = \mathtt{C}t'$, we have $\Omega\mathtt{x} \leq \Omega t' \leq \Omega\mathtt{C}t'$ by induction hypothesis and by definition. Using the first point, this implies that $\Omega\mathtt{x} \leq \mathtt{C}t'$.
- The same argument works when $t = \mathtt{f}t'$.
- When $t = \mathtt{C}^{-}t'$ [resp. $t = \mathtt{.D}t'$], we can use the same argument, except that $\Omega t' \leq \mathtt{C}^{-}t'$ [resp. $\Omega t' \leq \mathtt{.D}t'$] comes from the second [resp. third] point.
- If $t = \mathtt{\{}\dots\mathtt{;D}_i = t_i\mathtt{;}\dots\mathtt{\}}$, we have $\Omega\mathtt{x} \leq \Omega t_i$ by induction hypothesis. This shows that $\Omega\mathtt{x} \leq \sum_i \Omega t_i \leq \Omega\mathtt{\{}\dots\mathtt{;D}_i = t_i\mathtt{;}\dots\mathtt{\}}$, from which we conclude that $\Omega\mathtt{x} \leq t$.

□

Because of inequality $\sum_i \Omega t_i \leq \Omega\mathtt{\{}\dots\mathtt{;D}_i = t_i\mathtt{;}\dots\mathtt{\}}$, the converse of point (1) doesn't hold in general and the resulting domain (Definition 2.22) is not a Smyth power domain. The next lemma, a kind of dual to contextuality, might look obvious but isn't completely immediate.

Lemma 2.14. *If $s_1 \leq s_2$, then $s_1[\mathtt{x} := t] \leq s_2[\mathtt{x} := t]$.*

Proof. By induction on the proof of $s_1 \leq s_2$. Most cases are trivial:

- If $s_1 \leq s_2$ comes from reflexivity [resp. transitivity], we have $s_1[\mathtt{x} := t] \leq s_2[\mathtt{x} := t]$ by reflexivity [resp. transitivity] using the induction hypothesis.
- If $s_2 = \mathbf{0}$, then $s_2[\mathtt{x} := t] = \mathbf{0}$ as well so that $s_1[\mathtt{x} := t] \leq s_2[\mathtt{x} := t]$ by definition.
- If $s_1 = \Omega\mathtt{x}$ and $s_2 = \mathtt{x}$, we have $\Omega(s_1[\mathtt{x} := t]) \leq s_2[\mathtt{x} := t]$ by contextuality. The case $s_1 = \Omega\mathtt{x}$ and $s_2 = \mathtt{f}\mathtt{x}$ is similar.
- If $s_1 = u + v$ and $s_2 = u$, the result holds because $u[\mathtt{x} := t] + v[\mathtt{x} := t] \leq u[\mathtt{x} := t]$.
- This is trivial for all (in)equalities from (*). For example, if $s_1 = \mathtt{.D}_{i_0}\mathtt{\{}\dots\mathtt{;D}_i = u_i\mathtt{;}\dots\mathtt{\}}$ and $s_2 = u_{i_0}$. By definition, we have $s_1[\mathtt{x} := t] = \mathtt{.D}_{i_0}\mathtt{\{}\dots\mathtt{;D}_i = u_i[\mathtt{x} := t]\mathtt{;}\dots\mathtt{\}}$ and similarly, $s_2[\mathtt{x} := t] = u_{i_0}[\mathtt{x} := t]$. Thus, $s_1[\mathtt{x} := t] \leq s_2[\mathtt{x} := t]$ by definition of $\leq$.
- The only interesting case is contextuality: suppose that $s_1 = C[s'_1]$ and $s_2 = C[s'_2]$ with $s'_1 \leq s'_2$. By induction hypothesis, we know that $s'_1[\mathtt{x} := t] \leq s'_2[\mathtt{x} := t]$. A straightforward induction on C shows that $s'_1[\mathtt{x} := t] \leq s'_2[\mathtt{x} := t]$ implies $C[s'_1][\mathtt{x} := t] \leq C[s'_2][\mathtt{x} := t]$ for all C, s'_1, s'_2, t. □

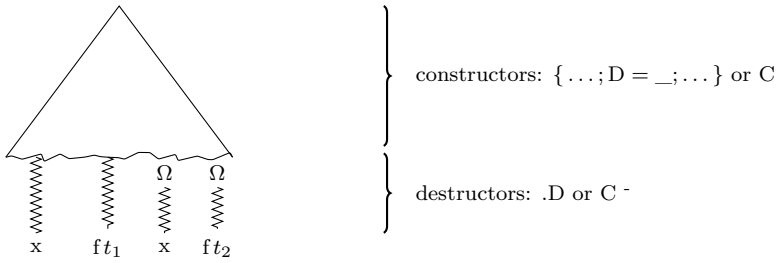
Definition 2.15. The reduction relation $\to$ on terms is the contextual closure of the left-to-right inequalities $(*)$ from Definition 2.12.

Lemma 2.16.

(1) *If $t \to s$ then $s \leq t$.*
(2) *The reduction $\to$ is strongly normalizing.*
(3) *Simple normal forms are given by the grammar*

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

A typical normal form thus looks like, where t_1, t_2, are themselves in normal form



Proof of Lemma 2.16. The first point follows from the definition. Reduction is strongly normalizing because the depth of the term decreases. For the third point, all terms generated by the grammar are obviously in normal form. It is also straightforward to check that all simple normal forms are generated by the grammar because:

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

This reduction isn't confluent because terms of the form "$\mathtt{.D_1\{D_1} = t_1; \mathtt{D_2} = t_2\mathtt{\}}$" can reduce to t_1 or to $\mathbf{0}$ if t_2 reduces to $\mathbf{0}$.[8]

Lemma 2.17. *Write* $\mathsf{nf}(t)$ *for the normal form of a term according to the "rightmost first" reduction strategy. For any context and term,* $\mathsf{nf}(C[t] = \mathsf{nf}(C[\mathsf{nf}(t)])$.

Proof. This is a straightforward induction on the context C.

- If C starts with a constructor $\mathtt{C}$, i.e. C is of the form $\mathtt{C}\,C'$, the result follows from the induction hypothesis: since no reduction involves a constructor on the left, we have $\mathsf{nf}(\mathtt{C}\,C'[t]) = \mathtt{C}\,\mathsf{nf}(C'[t]) = \mathtt{C}\,\mathsf{nf}(C'[\mathsf{nf}(t)]) = \mathsf{nf}(\mathtt{C}\,C'[\mathsf{nf}(t)])$.
- Reasoning is similar if C starts with a function name or a structure.
- The result is obvious if C is the placeholder variable, or $\mathtt{x}$.
- If C starts with a destructor $\mathtt{.D}$, i.e. C is of the form $\mathtt{.D}\,C'$, computing $\mathsf{nf}(C[u])$ is done by first computing $v = \mathsf{nf}(C'[u])$ and then reducing $\mathtt{.D}\,v$. By induction hypothesis, we have $\mathsf{nf}(C'[t]) = \mathsf{nf}(C'[\mathsf{nf}(t)])$, so that reducing $\mathsf{nf}(C[t])$ and $\mathsf{nf}(C[\mathsf{nf}(t)])$ give same results.
- Reasoning is similar if C starts with a destructor $\mathtt{C}^{-}$ or a Ω. □

2.3.2. *Semantics.* Both "$\mathtt{.D}$" and "$\mathtt{C}^{-}$" have natural interpretations as continuous functions:

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

and

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

This allows to define the semantics of any element of $\mathcal{O}_0$ as a function depending on $\mathtt{x}$.

[8] It is however "almost" confluent in that a term can have at most one non-$\mathbf{0}$ normal form. Lemma 2.21 is a weaker version of that fact that is sufficient for our needs.

Definition 2.18. Let ρ be an environment giving, for each functions names, a continuous function on $\mathcal{S}$; let t be a simple term in $\mathcal{O}_0$. We define $\llbracket t \rrbracket_\rho : \mathcal{S} \to \mathcal{S}$ with

(0) $\llbracket t \rrbracket_\rho (\sum_i v_i) = \sum_i \llbracket t \rrbracket_\rho (v_i)$, and in particular $\llbracket t \rrbracket_\rho (\mathbf{0}) = \mathbf{0}$,
(1) $\llbracket \mathtt{C}t \rrbracket_\rho (v) = \mathtt{C}\big(\llbracket t \rrbracket_\rho (v)\big)$,
(2) $\llbracket \{\mathtt{D}_1 = t_1; \dots\} \rrbracket_\rho (v) = \{\mathtt{D}_1 = \llbracket t_1 \rrbracket_\rho (v); \dots\}$,
(3) $\llbracket \Omega t \rrbracket_\rho (v) = \Omega\big(\llbracket t \rrbracket_\rho (v)\big) = \begin{cases} \mathbf{0} & \text{if } \llbracket t \rrbracket_\rho (v) = \mathbf{0} \\ \bot & \text{otherwise,} \end{cases}$
(4) $\llbracket \mathtt{C}^- t \rrbracket_\rho (v) = \mathtt{C}^-\big(\llbracket t \rrbracket_\rho (v)\big)$,
(5) $\llbracket .\mathtt{D}t \rrbracket_\rho (v) = .\mathtt{D}\big(\llbracket t \rrbracket_\rho (v)\big)$,
(6) $\llbracket \mathtt{x} \rrbracket_\rho (v) = v$,
(7) $\llbracket \mathtt{g}\ t \rrbracket_\rho (v) = \rho(\mathtt{g})\big(\llbracket t \rrbracket_\rho (v)\big)$.

$\llbracket \_ \rrbracket$ is extended to all terms in $\mathcal{O}_0$ by linearity.

Because Ω is the semantics of $\Omega\mathtt{x}$ we sometimes write Ω for $\Omega\mathtt{x}$.

Lemma 2.19.
(1) *If $t_1 \leq t_2$, then $\llbracket t_1 \rrbracket_\rho \leq \llbracket t_2 \rrbracket_\rho$; $\llbracket \_ \rrbracket_\rho$ is thus compatible with $\approx$.*
(2) *If $\rho(\mathtt{g})$ is continuous for any $\mathtt{g}$, then $\llbracket t \rrbracket_\rho$ is also continuous.*
(3) *$\llbracket T \rrbracket_\rho \geq \Omega$. provided $\rho(\mathtt{f}) \geq \Omega$ for all function names,*
(4) *For all terms $t_1, t_2 \in \mathcal{O}_0$ and environment ρ, we have $\big\llbracket t_1[\mathtt{x} := t_2] \big\rrbracket_\rho = \llbracket t_1 \rrbracket_\rho \circ \llbracket t_2 \rrbracket_\rho$.*

Proof. Checking the first points amounts to checking that all inequations from Definition 2.11 hold semantically in $[\mathcal{S} \to \mathcal{S}]$. This is straightforward. The functions $\mathtt{C}^-$, $.\mathtt{D}$ and Ω are easily shown continuous, $\llbracket t \rrbracket_\rho$ is continuous as a composition of continuous functions. Points (3) follows from linearity of $\llbracket t \rrbracket_\rho$, and point (4) is proved by immediate induction. □

Corollary 2.20. *The order $\leq$ on $\mathcal{O}_0$ is non trivial.*

Proof. Any equivalence in $\mathcal{O}_0$ gives rise to an equality in $\mathcal{S} \to \mathcal{S}$, which is non trivial. □

Lemma 2.21. *Suppose $t_1 \to t_2$ and let $v \in \mathcal{S}$, we have $\llbracket t_1 \rrbracket_\rho (v) = \llbracket t_2 \rrbracket_\rho (v)$ or $\llbracket t_1 \rrbracket_\rho (v) = \mathbf{0}$.*

Proof. This is straightforward:

- For $\mathtt{C}^-\mathtt{C}t \to t$: if $\llbracket t \rrbracket_\rho (v) = \mathbf{0}$, then $\llbracket \mathtt{C}^-\mathtt{C}t \rrbracket_\rho (v) = \mathbf{0}$, and if $\llbracket t \rrbracket_\rho (v) \neq \mathbf{0}$, then by definition, $\llbracket \mathtt{C}^-\mathtt{C}t \rrbracket_\rho (v) = \llbracket t \rrbracket_\rho (v)$.
- For $.\mathtt{D}_i\{\dots;\mathtt{D}_i = t_i;\dots\} \to t_i$: if some $\llbracket t_j \rrbracket_\rho (v) = \mathbf{0}$ then $\llbracket .\mathtt{D}_i\{\dots;\mathtt{D}_i = t_i;\dots\} \rrbracket_\rho (v) = \mathbf{0}$ as well and we have nothing to prove. Otherwise, $\llbracket .\mathtt{D}_i\{\dots;\mathtt{D}_i = t_i;\dots\} \rrbracket_\rho (v) = \llbracket t_i \rrbracket_\rho (v)$.
- For $\Omega\{\dots;\mathtt{D}_i = t_i;\dots\} \to \sum_i \Omega t_i$, both $\llbracket \Omega\{\dots;\mathtt{D}_i = t_i;\dots\} \rrbracket_\rho (v)$ and $\llbracket \sum_i \Omega t_i \rrbracket_\rho (v)$ can only be equal to $\bot$ or $\mathbf{0}$. The only way to make the lemma false would be by having $\llbracket \sum_i \Omega t_i \rrbracket_\rho (v) = \mathbf{0}$ and $\llbracket \Omega\{\dots;\mathtt{D}_i = t_i;\dots\} \rrbracket_\rho (v) = \bot$. This is impossible by point (1) of the previous lemma.
- The other reduction rules are all treated similarly. □

Definition 2.22. The domain $\mathcal{O}$ is the ideal completion of $\mathcal{O}_0$ quotiented by $\approx$.

This introduces infinite elements like $\mathtt{C}^\infty = \mathtt{CCC}\cdots = \bigsqcup^\uparrow \{\Omega, \mathtt{C}\Omega, \mathtt{CC}\Omega, \dots\}$. However, since $\Omega \approx .\mathtt{D}\Omega \approx .\mathtt{D}.\mathtt{D}\Omega \approx \dots$, we have $.\mathtt{D}^\infty = .\mathtt{D}.\mathtt{D}.\mathtt{D}\cdots = \bigsqcup^\uparrow \{\Omega, .\mathtt{D}\Omega, .\mathtt{D}.\mathtt{D}\Omega, \dots\} \approx \Omega$. Some infinite sums also appears but $\mathcal{O}$ is not a Smyth power domain because of inequalities

of the form $\{\ldots;\mathtt{D}_i = t_i;\ldots\} \geq \Omega\{\ldots;\mathtt{D}_i = t_i;\ldots\} \geq \sum_i \Omega t_i$. We can extend the semantics of $\mathcal{O}_0$ to the whole $\mathcal{O}$.

Definition 2.23. The semantics $\{\!|\_|\!\}$ is extended to $\mathcal{O}$ by continuity.

From now on, we single out a function name $\mathtt{f}$ as the recursive function whose definition we are investigating.

Definition 2.24. Each $T \in \mathcal{O}$ gives rise to an operator $[\![T]\!]$ from $[\mathcal{S} \to \mathcal{S}]$ to itself:

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

The typical environment ρ is constructed inductively from previous recursive definitions and will be omitted in the rest of the paper. In standard logical terminology, $\mathtt{f}$ is a variable, but all other $\mathtt{g}$ are parameters. Note that dealing with mutually recursive definitions would require the introduction of several function variables $\mathtt{f}_1$, $\mathtt{f}_2$, etc.

2.4. **Interpreting Recursive Definitions.**

2.4.1. *Composition.* Elements of $\mathcal{O}_0$ will be used to interpret individual clauses from a recursive definition using Definition 2.24. To use Kleene's formula, we need a notion of composition. Given t_1 and t_2 in $\mathcal{O}_0$, we want to represent the composition $[\![t_1]\!] \circ [\![t_2]\!]$. Using standard λ-calculus notation, $\{\!|t|\!\}$ is the semantics of $\lambda\mathtt{x}.t$, of type $\mathcal{S} \to \mathcal{S}$ and $[\![t]\!]$ the semantics of $\lambda\mathtt{f}\lambda\mathtt{x}.t$, of type $[\mathcal{S} \to \mathcal{S}] \to [\mathcal{S} \to \mathcal{S}]$. The composition of $[\![t_1]\!]$ and $[\![t_2]\!]$ should thus be

$$\begin{array}{rcll}
[\![t_1]\!] \circ [\![t_2]\!] & = & \lambda\mathtt{f}.\,[\![t_1]\!]\,\big([\![t_2]\!]\,(\mathtt{f})\big) & \text{(definition of composition)} \\
& = & \lambda\mathtt{f}.(\lambda\mathtt{f}\lambda\mathtt{x}.t_1)\big([\![t_2]\!]\,(\mathtt{f})\big) & \text{(definition of } [\![t_1]\!]\text{)} \\
& = & \lambda\mathtt{f}.\lambda\mathtt{x}.t_1\big[\mathtt{f} := [\![t_2]\!]\,(\mathtt{f})\big] & (\beta \text{ reduction}) \\
& = & \lambda\mathtt{f}\lambda\mathtt{x}.t_1[\mathtt{f} := \lambda\mathtt{x}.t_2] & \text{(definition of } [\![t_2]\!] \text{ and } \eta \text{ reduction)} \\
& = & \lambda\mathtt{f}\lambda\mathtt{x}.t_1[\mathtt{f}u := (\lambda\mathtt{x}.t_2)u] & \text{(because } \mathtt{f} \text{ is always fully applied)} \\
& = & \lambda\mathtt{f}\lambda\mathtt{x}.t_1\big[\mathtt{f}u := t_2[\mathtt{x} := u]\big] & \\
& = & \big[\!\big[t_1\big[\mathtt{f}u := t_2[\mathtt{x} := u]\big]\big]\!\big] &
\end{array}$$

The composition of t_1 and t_2, as operators, is thus obtained by replacing each $\mathtt{f}u$ in t_1 by $t_2[\mathtt{x} := u]$. The next definition implements that directly.

Definition 2.25. If $t_1, t_2 \in \mathcal{O}_0$, we define $t_1 \circ t_2$ by induction on t_1:

- $\big(\sum t_i\big) \circ t_2 = \sum(t_i \circ t_2)$,
- $(\mathtt{C}t_1) \circ t_2 = \mathtt{C}(t_1 \circ t_2)$,
- $\{\mathtt{D}_1 = t_1;\ldots;\mathtt{D}_k = t_k\} \circ t_2 = \{\mathtt{D}_1 = t_1 \circ t_2;\ldots;\mathtt{D}_k = t_k \circ t_2\}$,
- $(\mathtt{C}^- t_1) \circ t_2 = \mathtt{C}^-(t_1 \circ t_2)$,
- $(.\mathtt{D}\,t_1) \circ t_2 = .\mathtt{D}(t_1 \circ t_2)$,
- $(\Omega t_1) \circ t_2 = \Omega(t_1 \circ t_2)$,
- $\mathtt{x} \circ t_2 = \mathtt{x}$,
- $(\mathtt{g}\,t_1) \circ t_2 = \mathtt{g}(t_1 \circ t_2)$ if $\mathtt{g} \neq \mathtt{f}$,
- $(\mathtt{f}\,t_1) \circ t_2 = t_2[\mathtt{x} := t_1 \circ t_2]$.

The only interesting case is the last one, where we replace $\mathtt{f}$ by t_2 and continue recursively. Because of that, we sometimes abuse the notation and write $t_1[\mathtt{f} := t_2]$.

Recall that to avoid introducing binders, we explicitly assume the argument of an operator is the function with name $\mathtt{f}$. In other words, the only free symbols in the syntax are $\mathtt{f}$ and $\mathtt{x}$. Any other function name $\mathtt{g}$, $\mathtt{h}$ etc. is considered a parameter and is bound to some function taken from an implicit environment.

Lemma 2.26. *For any $t_1, t_2, t_3 \in \mathcal{O}_0$, $t_1 \circ (t_2 \circ t_3) = (t_1 \circ t_2) \circ t_3$.*

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 $t = \mathtt{x}$, 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

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

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

- if $t = \mathtt{x}$, 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 $t = t_2$, $t_1 = t_1' \circ t_2$, and $t_2 = t_3$. □

Lemma 2.27. *For any $t_1, t_2 \in \mathcal{O}_0$, $[\![t_1 \circ t_2]\!] = [\![t_1]\!] \circ [\![t_2]\!]$. If moreover, t_2 doesn't contain $\mathtt{f}$, we have $\{\!\{t_1 \circ t_2\}\!\} = [\![t_1]\!]\left(\{\!\{t_2\}\!\}\right)$. In particular,*

$$\{\!\{\underbrace{t \circ \cdots \circ t}_{n} \circ \Omega\mathtt{x}\}\!\} \quad = \quad [\![t]\!]^n(\Omega)$$

Proof. This is proved by induction. The only non trivial case is $(\mathtt{f}\, t_1) \circ t_2 = t_2[\mathtt{x} := t_1 \circ t_2]$:

$$\begin{array}{lcll}
[\![(\mathtt{f}\, t_1) \circ t_2]\!]_\rho(f) & = & \{\!\{(\mathtt{f}\, t_1) \circ t_2\}\!\}_{\rho, \mathtt{f}=f} & \text{definition of } [\![_]\!] \\
& = & \{\!\{t_2[\mathtt{x} := t_1 \circ t_2]\}\!\}_{\rho, \mathtt{f}=f} & \text{definition of } \circ \\
& = & \{\!\{t_2\}\!\}_{\rho, \mathtt{f}=f} \circ \{\!\{t_1 \circ t_2\}\!\}_{\rho, \mathtt{f}=f} & \text{point (4) of Lemma 2.19} \\
& = & \{\!\{t_2\}\!\}_{\rho, \mathtt{f}=f} \circ \left([\![t_1 \circ t_2]\!]_\rho(f)\right) & \text{definition of } [\![_]\!] \\
& = & \{\!\{t_2\}\!\}_{\rho, \mathtt{f}=f} \circ \left([\![t_1]\!]_\rho \circ [\![t_2]\!]_\rho(f)\right) & \text{induction} \\
& = & \{\!\{t_2\}\!\}_{\rho, \mathtt{f}=f} \circ \left([\![t_1]\!]_\rho(\{\!\{t_2\}\!\}_{\rho, \mathtt{f}=f})\right) & \text{definition of } [\![_]\!] \\
& = & \{\!\{t_2\}\!\}_{\rho, \mathtt{f}=f} \circ (\{\!\{t_1\}\!\}_{\rho, \mathtt{f}=\{\!\{t_2\}\!\}_{\rho, \mathtt{f}=f}}) & \text{definition of } [\![_]\!] \\
& = & \{\!\{\mathtt{f}\, t_1\}\!\}_{\rho, \mathtt{f}=\{\!\{t_2\}\!\}_{\rho, \mathtt{f}=f}} & \text{definition of } \{\!\{_\}\!\} \\
& = & \{\!\{\mathtt{f}\, t_1\}\!\}_{\rho, \mathtt{f}=[\![t_2]\!]_\rho(f)} & \text{definition of } [\![_]\!] \\
& = & [\![\mathtt{f}\, t_1]\!]_\rho \circ [\![t_2]\!]_\rho(f) & \text{definition of } [\![_]\!]
\end{array}$$

The second point is a direct consequence of the first point. □

We also have

Lemma 2.28. *If $t_1 \leq t_2$, then $s \circ t_1 \leq s \circ t_2$ and $t_1 \circ s \leq t_2 \circ s$.*

Proof. The first inequality is proved by induction on s. The only interesting case is when s starts with $\mathtt{f}$.

$$\begin{array}{rcll} (\mathtt{f}\, s) \circ t_1 & = & t_1[\mathtt{x} := s \circ t_1] & \text{(definition)} \\ & \leq & t_2[\mathtt{x} := s \circ t_1] & \text{(because } t_1 \leq t_2\text{, by Lemma 2.14)} \\ & \leq & t_2[\mathtt{x} := s \circ t_2] & \text{(by contextuality, because } s \circ t_1 \leq s \circ t_2 \text{ by induction)} \end{array}$$

The second inequality is proved by induction on $s_1 \leq s_2$. The proof is very similar to the proof of Lemma 2.14 and is omitted. □

2.4.2. *Interpreting Recursive Definitions.* We can interpret the operator $\Theta^{\mathrm{ndt}}_{\mathtt{f}}$ (defined on page 11) by an element of $\mathcal{O}_0$. Consider a single clause "$\mathtt{f}\, p = u$" of the recursive definition of $\mathtt{f}$. The pattern p allows to "extract" some parts of the argument of $\mathtt{f}$ to be used in the right-hand side u. For example, the clause

```
| length (Cons { Fst = e ; Snd = l }) = ...
```

introduces 2 variables: `e` and `l`. If we call the parameter of `length` "`x`", the variable `e` can be obtained as `e` = `.Fst Cons⁻ x`: we remove the leading `Cons` constructor and project on field `Fst`. The variable `l` is obtained similarly with `l` = `.Snd Cons⁻ x`. The following definition formalizes that by defining, for any pattern p, a substitution $[p := \mathtt{x}]$ giving for each variable of p, an element of $\mathcal{O}_0$.

Definition 2.29. Given a linear 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 $\mathtt{y}$ by variable $\mathtt{x}$,
- $[\mathtt{C}p := \mathtt{x}] = [p := \mathtt{x}] \circ \mathtt{C}^-$,
- $[\{\ldots;\mathtt{D}_i = p_i;\ldots\} := \mathtt{x}] = \bigcup_i ([p_i := \mathtt{x}] \circ .\mathtt{D}_i)$ (because patterns are linear, the substitutions don't overlap).

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

As another example, consider the last rule from the `sum` function from page 6

```
| sums _ { Head = Cons {Fst = n ; Snd = l} ; Tail = s } = ...
```

If we call the second argument of `sums` "`x`", the corresponding substitution is

$$[\ \mathtt{n} := .\mathtt{Fst}\ \mathtt{Cons}^-\ .\mathtt{Head}\ \mathtt{x};\quad \mathtt{l} := .\mathtt{Snd}\ \mathtt{Cons}^-\ .\mathtt{Head}\ \mathtt{x};\quad \mathtt{s} := .\mathtt{Tail}\ \mathtt{x};\quad]$$

Lemma 2.30. *If $v \in \mathcal{V}$ matches p (Definition 2.6), then $[p := \mathtt{x}] \circ [\mathtt{x} := v] = [p := v]$, the unifier of p and v.*

Proof. The proof is a simple induction on the pattern.

- When $p = \mathtt{y}$ is a variable, $[p := \mathtt{x}]$ is the substitution $[\mathtt{y} := \mathtt{x}]$, and the unifier $[p := v]$ is the substitution $[\mathtt{y} := v]$. The result is obvious.

- When $p = \mathtt{C}p'$ starts with a constructor, $[\mathtt{C}p := \mathtt{x}] = [p := \mathtt{x}] \circ \mathtt{C}^-$. Because v must match p, it is necessarily of the form $\mathtt{C}v'$, and the unifier $[p := v]$ is equal to $[p' := v']$. We thus have

$$\begin{array}{rcll} [p := \mathtt{x}] \circ [\mathtt{x} := v] & = & [p' := \mathtt{x}] \circ \mathtt{C}^- \circ [\mathtt{x} := \mathtt{C}v'] & \\ & = & [p' := \mathtt{x}] \circ [\mathtt{x} := \mathtt{C}^-\mathtt{C}v'] & \\ & = & [p' := \mathtt{x}] \circ [\mathtt{x} := v'] & \\ & = & [p' := v'] & \text{(induction hypothesis)} \\ & = & [p := v] & \end{array}$$

- Reasoning is similar when p is a structure. □

Any recursive definition can be interpreted by an element of $\mathcal{O}_0$ in the following way:

Definition 2.31. Given a recursive definition of $\mathtt{f}$, define $T_{\mathtt{f}}$ with

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

where the sum ranges over all clauses from the definition of $\mathtt{f}$.

For the `length` function, we get

$$T_{\mathtt{length}} = \mathtt{Succ\ length\ .Snd\ Cons^-\ x} \quad + \quad \mathtt{Zero\ Nil^-\ x}$$

where the first summand corresponds to the first clause

```
| length (Cons Fst = _ ; Snd = l) = Succ (length l)
```

and the second summand corresponds to the second clause

```
| length (Nil _x) = Zero _x
```

Lemma 2.32. *For any environment ρ, we have $[\![T_{\mathtt{f}}]\!]_\rho \leq \Theta^{\mathrm{ndt}}_{\rho,\mathtt{f}}$.*

Proof. Given a value v and a clause $\mathtt{f}\ \ p\ \mathtt{=}\ u$, we have

- if p matches v, then

$$\begin{array}{rcll} [\![u[p := \mathtt{x}]]\!]\,(v) & = & [\![u[p := \mathtt{x}][\mathtt{x} := v]]\!] & \text{(Lemma 2.27)} \\ & = & [\![u[p := v]]\!] & \end{array}$$

- if p doesn't matches v, then

$$\begin{array}{rcll} [\![u[p := \mathtt{x}]]\!]\,(v) & \leq & \mathbf{0} & \text{(Definition 2.6)} \\ & = & [\![u[p := v]]\!] & (p \text{ doesn't match } v) \end{array}$$

So that each summand of $[\![T_{\mathtt{f}}]\!]\,(v)$ is less than a summand of $\Theta^{\mathrm{ndt}}_{\mathtt{f}}(v)$, proving the claim. □

The inequality is strict in general because non-matching clause may introduce non-$\mathbf{0}$ terms: for example, the clause

$$\mathtt{f\ (}\underbrace{\mathtt{Cons\{Fst=Foo\ ;\ Snd\ =\ l\}}}_{p}\mathtt{)\ =\ Foo}$$

doesn't match the value $v = \mathtt{Cons\{Fst = Bar;\ Snd = ...\}}$. The non deterministic $\Theta^{\mathrm{ndt}}_{\mathtt{f}}(f)(v)$ is thus $\mathbf{0}$. But because $[p := \mathtt{x}] = [\mathtt{l} := \mathtt{.Snd\,Cons^-\,x}]$, we have $T_{\mathtt{f}}(v) = \mathtt{Foo}$. The reason is that nothing about the closed pattern "`Foo`" is recorded in $[p := \mathtt{x}]$.

By Lemma 2.9, totality of $\mathsf{fix}([\![T_{\mathtt{f}}]\!])$ implies totality of $\mathsf{fix}(\Theta^{\mathrm{ndt}}_{\mathtt{f}})$. Because of Lemma 2.4, Lemma 2.5 and Lemma 2.27, we have

Corollary 2.33. *To check that* $\mathsf{fix}(\Theta^{\mathrm{ndt}}_{\mathtt{f}})$ *is total, is is enough to check that*

$$\mathsf{fix}([\![T_{\mathtt{f}}]\!]) = \bigsqcup_n^{\uparrow} [\![T_{\mathtt{f}}]\!]^n(\Omega) \quad : \quad \mathcal{S} \to \mathcal{S}$$

is total.

Together with Corollary 2.31, we finally get

Corollary 2.34. *Given a recursive definition for* $\mathtt{f}$*, we have that* $\bigsqcup_n^{\uparrow} [\![T_{\mathtt{f}}]\!]^n(\Omega)$ *is total implies that the usual semantics of* $\mathtt{f}$ *is total.*

Recall that even though we haven't written them, all constructors $\mathtt{C}/\ \{\ldots;\mathtt{D}{=}\_;\ldots\}$ and destructors $\mathtt{C}^{-}/\ .\mathtt{D}$ come with a priority and that totality is defined using those priorities (Definition 1.3).

3. Call-Graphs and the Size-Change Principle

Except for a few minor differences, $T_{\mathtt{f}}$ is a faithful representation of the original recursive definition. We now simplify each $T_{\mathtt{f}}$ into a disjoint sum of independent calls and show that doing so reflects totality.

3.1. Call-Graphs.

3.1.1. *Call Paths.* By definition, composition of operators (Definition 2.25) is linear on the left. When computing $s \circ (t_1 + t_2)$, each occurrence of $\mathtt{f}$ inside s is replaced by $t_1 + t_2$. By linearity, this is a sum of terms where each occurrence of $\mathtt{f}$ inside s is replaced either by t_1 or t_2. For example, with $s = \mathtt{ffx}$ and $t_1 = \mathtt{g}\,\mathtt{x}$, $t_2 = \mathtt{h}\,\mathtt{x}$, we get[9]

$$\begin{aligned}
\mathtt{ffx} \circ (\mathtt{g}\,\mathtt{x} + \mathtt{h}\,\mathtt{x}) &= (\mathtt{g}\,\mathtt{x} + \mathtt{h}\,\mathtt{x})[\mathtt{x} := \mathtt{ffx} \circ (\mathtt{g}\,\mathtt{x} + \mathtt{h}\,\mathtt{x})] \\
&= (\mathtt{g}\,\mathtt{x} + \mathtt{h}\,\mathtt{x})\big[\mathtt{x} := (\mathtt{g}\,\mathtt{x} + \mathtt{h}\,\mathtt{x})[\mathtt{x} := \mathtt{x} \circ (\mathtt{g}\,\mathtt{x} + \mathtt{h}\,\mathtt{x})]\big] \\
&= (\mathtt{g}\,\mathtt{x} + \mathtt{h}\,\mathtt{x})\big[\mathtt{x} := (\mathtt{g}\,\mathtt{x} + \mathtt{h}\,\mathtt{x})[\mathtt{x} := \mathtt{x}]\big] \\
&= (\mathtt{g}\,\mathtt{x} + \mathtt{h}\,\mathtt{x})[\mathtt{x} := (\mathtt{g}\,\mathtt{x} + \mathtt{h}\,\mathtt{x})] \\
&= \mathtt{g}(\mathtt{g}\,\mathtt{x} + \mathtt{h}\,\mathtt{x}) + \mathtt{h}(\mathtt{g}\,\mathtt{x} + \mathtt{h}\,\mathtt{x}) \\
&= \mathtt{g}\,\mathtt{g}\,\mathtt{x} + \mathtt{g}\,\mathtt{h}\,\mathtt{x} + \mathtt{h}\,\mathtt{g}\,\mathtt{x} + \mathtt{h}\,\mathtt{h}\,\mathtt{x}
\end{aligned}$$

To formalize that, we annotate each occurrences of $\mathtt{f}$ with its index and write $\mathtt{f}_1\mathtt{f}_2\mathtt{x}$ for $\mathtt{ffx}$.

Lemma 3.1. *We have*

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

where $\mathsf{occ}(\mathtt{f}, s)$ *represents the set of occurrences of* $\mathtt{f}$ *in* s*, and the substitution occurs at the given occurrences. More precisely,* $s[\sigma] = s\big[\mathtt{f}_i := \sigma(\mathtt{f}_i)\big]$*, which substitutes any* $\mathtt{f}_i\,t$ *appearing in* s *by* $\sigma(\mathtt{f}_i)[\mathtt{x} := t]$*.*

[9] recall that $\mathtt{f}$ is the only free function name. Other names like $\mathtt{g}$ and $\mathtt{h}$, written here in italics, are bound parameters.

Proof. This is a straightforward induction. The most interesting case is when s is the structure $\{\dots;\mathtt{D}_i = s_i;\dots\}$.

$$\begin{aligned}
& s[\mathtt{x} := t_1 + t_2] \\
= \;& \{\dots;\mathtt{D}_i = s_i;\dots\}[\mathtt{x} := t_1 + t_2] \\
= \;& \{\dots;\mathtt{D}_i = s_i[\mathtt{x} := t_1 + t_2];\dots\} \\
= \;& \{\dots;\mathtt{D}_i = \textstyle\sum_{\sigma_i:\mathsf{occ}(\mathtt{f},s_i)\to\{1,2\}} s_i[\sigma_i];\dots\} && \text{(induction hypothesis)} \\
= \;& \textstyle\sum_i \sum_{\sigma_i:\mathsf{occ}(\mathtt{f},s_i)\to\{1,2\}} \{\dots;\mathtt{D}_i = s_i[\sigma_i];\dots\} && \text{(multilinearity)} \\
= \;& \textstyle\sum_{\sigma:\mathsf{occ}(\mathtt{f},s)\to\{1,2\}} \{\dots;\mathtt{D}_i = s_i[\sigma];\dots\}
\end{aligned}$$

where the last equality comes from the fact that $\mathsf{occ}(\mathtt{f}, s)$ is the disjoint sum of all the $\mathsf{occ}(\mathtt{f}, s_i)$. □

In particular, if $t = \sum_i t_i$ is a sum of simple terms, then $t^n = t \circ \cdots \circ t$ has a very specific shape. Each summand of t^n is obtained by taking a summand of t^{n-1} and replacing each occurrence of $\mathtt{f}$ by some t_i. More formally:

Definition 3.2. 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 each occurrence of $\mathtt{f}$ inside s_k by some t_1, …, 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 call such a path *finite*.

We usually omit the substitution and talk about the path "(s_k)".

Lemma 3.3. *For any term t and natural number $k > 0$, we have*

$$t^k = t \circ \cdots \circ t \quad = \sum_{(s)\ \mathit{path\ of}\ t} s_k$$

Proof. Note that because, for any given k, there are only finitely many possible s_k, this sum is actually finite. Suppose that $t = t_1 + \cdots + t_n$; the proof is by induction on k:

- if $k = 1$, $t^1 = t_1 + \cdots + t_n$, and each s_1 in a path is of the form $s_0 = \mathtt{f}\,\mathtt{x}$ where $\mathtt{f}$ is replaced by one t_i, i.e. each s_1 is of the form $\mathtt{f}\,\mathtt{x} \circ t_i = t_i$. Conversely, each t_i appears as some s_1 for some path s.
- By definition, $t^{k+1} = t^k \circ t$, and t^k is the sum of the s_k for all path (s). The term t^{k+1} is thus equal to the sum over all path (s) of the $s_k[\mathtt{f} := t]$. By Lemma 3.1, those are precisely the s_{k+1} of all path (s) of t. □

We can use paths to compute fixed points.

Lemma 3.4. *Suppose $T = t_1 + \cdots + t_n$ is a sum of simple terms, then*

$$\bigsqcup_n{}^{\uparrow} T \circ \cdots \circ T \circ \Omega\mathtt{x} \quad = \sum_{(s)\ \mathit{path\ of}\ T} \ \bigsqcup_{i\geq 0}{}^{\uparrow} s_i(\Omega)$$

Note that the infinite sum makes sense because it is a limit of finite sums: for each i, there are only finitely many possible $s_i(\Omega)$.

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.[10]

[10] That's the order on the Smyth power domain. Refer to Appendix C.

Let s be a simple term in $\bigsqcup^{\uparrow} T^n(\Omega)$. We want to show that s is greater than some $\bigsqcup_{i\geq 0}^{\uparrow} s_i(\Omega)$. For each n, $T \circ \cdots \circ T \circ \Omega\mathtt{x}$ is a finite sum of elements of $\mathcal{O}_0$. Define the following tree:

- nodes of depth i are those summands t in T^i satisfying $t(\Omega) \leq s$,
- a node t' at depth $i+1$ is a child of node t at depth i if t' is the result of substituting all occurrences of $\mathtt{f}$ in t by one of $t_1, \ldots, t_n$.

As there are only finitely many possible substitutions from a given node, this tree is finitely branching. Because $T^n(\Omega) \leq \bigsqcup_i^{\uparrow} T^i(\Omega) = \mathsf{fix}(T) \leq s$, each T^n contains some simple term t such that $t(\Omega) \leq s$. This tree is thus infinite. By König's lemma, it contains an infinite branch $s_0, s_1, \ldots$. This sequence is a path of T and because all $s_i(\Omega)$ are less than s by construction, its limit is less than s. We thus have

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

For the converse, it is enough to show that for each path (s) 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 3.5. *If ρ is a total environment and $\mathsf{fix}(\llbracket T \rrbracket)$ is non-total, then there is a path (s_k) for T such that $\bigsqcup_i^{\uparrow} \llbracket s_i \rrbracket(\Omega)$ is non total.*

Proof. Suppose that $\mathsf{fix}(\llbracket T \rrbracket) = \bigsqcup_n^{\uparrow} \llbracket T \rrbracket^n(\Omega)$ is non total, then by Lemma 2.27 and the previous lemma, there is a path of T that is non total. □

3.1.2. *Call-Graph.* Part of the complexity of checking totality of recursive definitions comes from the fact that clauses can contain nested recursive calls. The Ackerman function is a well known (but useless) example. The call-graph turns a clause into the sum of its recursive calls, making recursive calls "independent" from each other. 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

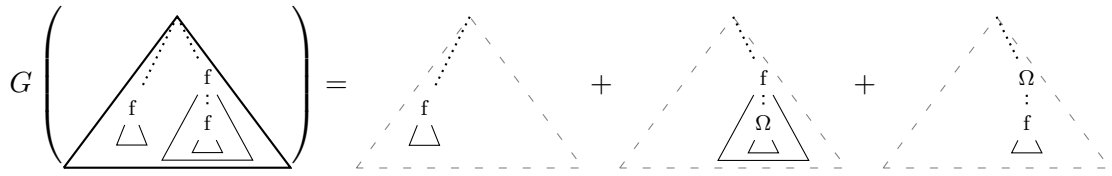
$$\mathtt{C}\big(\underline{\mathtt{f}\big(\underline{\mathtt{f}\ .\mathtt{D}_1\,\mathtt{x}}\big)}\big)$$

and contains 2 recursive calls (underlined).

It is clear that whenever this clause is used, it adds a `C` constructor just above the leftmost recursive call, making it guarded. It is also clear that the rightmost recursive call is structurally decreasing as it uses part of the initial argument `x`. We keep this information and split this clause in two independent calls:

- the leftmost call gives "$\mathtt{C}\mathtt{f}(\Omega\ .\mathtt{D}_1\,\mathtt{x})$", which we write as $\mathtt{f}\ \mathtt{x} \rightsquigarrow \mathtt{C}\ \mathtt{f}\ (\Omega\ .\mathtt{D}_1\,\mathtt{x})$:
  - this call is guarded by `C`,
  - we have no information about the arguments of `f`, except that it is $\mathbf{0}$ when $.\mathtt{D}_1\mathtt{x}$ is.
- the rightmost call gives "$\mathtt{C}\Omega\mathtt{f}(.\mathtt{D}_1\,\mathtt{x})$", which we write $\mathtt{f}\ \mathtt{x} \rightsquigarrow \mathtt{C}\ \Omega\mathtt{f}\ (.\mathtt{D}_1\,\mathtt{x})$:
  - besides a topmost `C`, we have no information about constructors directly above the call,
  - the argument of `f` is built from part of the initial argument.

In general, for each recursive call, we replace all other function calls (recursive or not) by Ω (this is point (2) in the definition below) and we split structures into independent fields (this is point (6) in the definition below). Visually:



Definition 3.6. Let $t \in \mathcal{O}_0$, 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* t *where all function calls have been replaced by* Ω,
(3) $G(\mathtt{g}\, t) = \Omega G(t)$ if $\mathtt{g} \neq \mathtt{f}$,
(4) $G(\mathtt{x}) = \mathbf{0}$,
(5) $G(\mathtt{C}^p\, t) = \mathtt{C}^p\, G(t)$,
(6) $G\big(\{\ldots;\mathtt{D}_i{}^p = t_i;\ldots\}\big) = \sum_i \{\mathtt{D}_i = G(t_i)\}^p$,
(7) $G(\mathtt{C}^{-} t) = \mathtt{C}^{-}\, G(t)$,
(8) $G(.\mathtt{D}\, t) = .\mathtt{D}\, G(t)$.

We write "$\mathtt{f}\ \mathtt{x} \rightsquigarrow u$" whenever u is a simple term in $G(T_{\mathtt{f}})$.

For example, for $t = \mathtt{C\{Fst{=}f_1\ (C^{-}\ x);\ Snd\ =\ f_2\ (C\ (f_3\ x))\}}$, we obtain

$$G(t) \quad = \quad \underbrace{\mathtt{C}\,\{\mathtt{Fst} = \mathtt{f}_1(\mathtt{C}^{-}\,\mathtt{x})\}}_{\text{first call}} + \underbrace{\mathtt{C}\,\{\mathtt{Snd} = \mathtt{f}_2(\mathtt{C}\,\Omega\,\mathtt{x})\}}_{\text{second call}} + \underbrace{\mathtt{C}\,\{\mathtt{Snd} = \Omega\,\mathtt{C}\,\mathtt{f}_3\,\mathtt{x})\}}_{\text{third call}}$$

The following is a direct consequence of the definition.

Lemma 3.7. *For each call* $\mathtt{f}\ \mathtt{x} \rightsquigarrow u$*, there is exactly one occurrence of* $\mathtt{f}$ *inside* u*.*

For mutual recursive definitions (not treated here), this defines an actual graph:

- vertices are the function names,
- arcs from $\mathtt{f}$ to $\mathtt{g}$ are the calls $\mathtt{f}\ \mathtt{x} \rightsquigarrow u$ where u's only function name is $\mathtt{g}$.

In general, the call-graph of T is not comparable to T. However, the construction reflects totality.

Proposition 3.8. *If* $\mathsf{fix}(G(T))$ *is total then so is* $\mathsf{fix}(T)$*.*

Proof. Let $T = \sum t_i$ is a sum of simple terms, and suppose $\mathsf{fix}(T)$ is non-total. By Corollary 3.5, it implies there is a path (s_k) of T and a total element $u \in \mathcal{V}$ such that $\bigsqcup^{\uparrow} s_i(\Omega)(u) \in \mathcal{V}$ is non-total, i.e. contains a non total branch β. In particular, it implies that no $s_i(\Omega)(u)$ reduces to $\mathbf{0}$. This branch β is either an infinite branch with odd principal priority or a finite branch ending with $\bot$.

The path (s_i) is infinite. Otherwise it becomes constant after a finite number of steps and the limit can be obtained by a finite number of applications of total operations (including the non recursive $\mathtt{g}$, ...) on total values (including u).

At each step, we go from s_k to s_{k+1} by replacing all occurrences of $\mathtt{f}$ by some t_i. Suppose the occurrences of $\mathtt{f}$ in T are indexed by natural numbers $1,\ldots,n$; we extend that indexing to occurrences of $\mathtt{f}$ in all the (s_k) using lists of natural numbers in $\{1,\ldots,n\}$. For example, let $T = \mathtt{Cf_1f_2x} + \mathtt{.Df_3f_4x}$ and consider the path that starts with:

$$\mathtt{f\,x}, \quad \mathtt{Cf_1f_2\,x}, \quad \mathtt{CCf_1f_2.Df_3f_4\,x}, \quad \ldots$$

The new indexing is

$$\mathtt{f}_{[]}\,\mathtt{x},\quad \mathtt{C}\mathtt{f}_{[1]}\mathtt{f}_{[2]}\,\mathtt{x},\quad \mathtt{CC}\mathtt{f}_{[1,1]}\mathtt{f}_{[1,2]}.\mathtt{D}\mathtt{f}_{[2,3]}\mathtt{f}_{[2,4]}\,\mathtt{x},\quad \ldots$$

When we replace $\mathtt{f}_{[2]}$ (in $\mathtt{C}\mathtt{f}_{[1]}\mathtt{f}_{[2]}\mathtt{x}$) by $.\mathtt{D}\mathtt{f}_3\mathtt{f}_4\mathtt{x}$, we keep the "[2]" prefix in front of each new occurrence of $\mathtt{f}$, obtaining $.\mathtt{D}\mathtt{f}_{[2,3]}\mathtt{f}_{[2,4]}\mathtt{x}$. Formally,

(0) the only occurrence of $\mathtt{f}$ in $s_0 = \mathtt{f}\,\mathtt{x}$ is indexed by the empty list
(1) given $k \geq 0$, the substitution σ_k replaces each occurrence $\mathtt{f}_L$ in s_k by some t_i. Each occurrence of $\mathtt{f}$ in s_{k+1} comes from a single occurrence $\mathtt{f}_j$ in some summand of T. We index such an occurrence by the list L, j.

An occurrence $\mathtt{f}_L \in s_k$ is called *infinitary* if the $s_{k'}$, for $k' \geq k$, contain infinitely many occurrences of $\mathtt{f}_{L'}$ with L' extending L. Otherwise, an occurrence is called *finitary*.

By the above remark, the occurrence $\mathtt{f}_{[]}$ in s_0 is infinitary. We construct, by induction, an infinite sequence of infinitary occurrences $\mathtt{f}_{[]}, \mathtt{f}_{[n_1]}, \mathtt{f}_{[n_1,n_2]}, \ldots$ in the following way: at each step k, choose $n_k \in \{1, \ldots, n\}$ s.t.

(1) $\mathtt{f}_{[n_1,\ldots,n_k]}$ is infinitary (this is always possible because $\mathtt{f}_{[n_1,\ldots,n_{k-1}]}$ is infinitary),
(2) the branch leading to $\mathtt{f}_{[n_1,\ldots,n_k]}$ in $\mathtt{nf}(s_k)$ starts with a prefix of β of maximal length.

At each step $\mathtt{f}_{[n_1,\ldots,n_k]}$ corresponds to the occurrence $\mathtt{f}_{n_k}$ in T and is thus associated to a single call α_k in the sense of Definition 3.6. The limit $\bigsqcup_k^{\uparrow} \alpha_1 \circ \ldots \circ \alpha_k(\Omega)(u)$ is non-total:

- if for some k_0, the occurrence $\mathtt{f}_{[n_1,\ldots,n_{k_0}]}$ in s_{k_0} appears below a function call (recursive or otherwise), all the compositions $\alpha_1 \circ \cdots \circ \alpha_{k_0} \circ \cdots$ after step k will be of the form $\beta_{k_0}\delta\Omega(\ldots)$ where β_{k_0} is a prefix of β and δ a sequence of destructors $\mathtt{C}\ {}^{-}\!/\mathtt{.D}$, so that their semantics on Ω and u will be equal to $\beta_{k_0}\bot$. (It cannot be equal to $\mathbf{0}$ as it would imply that s_k is also equal to $\mathbf{0}$.) The limits is thus equal to $\beta_{k_0}\bot$, which is non total.
- if for all k, the occurrence $\mathtt{f}_{[n_1,\ldots,n_k]}$ is only below a sequence of constructors $\mathtt{C}/\{\mathtt{D}{=\!\!=}\_\}$ and destructors $\mathtt{C}\ {}^{-}\!/\mathtt{.D}$, this sequence is of the form $\beta_k\delta_k$ where each β_k is a prefix of β, and δ_k a sequence of destructors.
  - If the β_ks are bounded by some β_{k_0}, the limit of the compositions $\alpha_1 \circ \cdots \circ \alpha_k$ will be, like above, equal to $\beta_{k_0}\bot$, which is non total.
  - If the β_ks are unbounded, the limit of the compositions $\alpha_1 \circ \cdots \circ \alpha_k$ will be equal to β, which is non total by hypothesis.

None of the compositions $\mathtt{nf}(\alpha_1 \circ \cdots \circ \alpha_k)(\Omega)(u)$ can be equal to $\mathbf{0}$, as it would imply the corresponding $s_k(\Omega)(u)$ is equal to $\mathbf{0}$ as well. By Lemma 2.21, we have constructed a non-total branch in $\bigsqcup_k^{\uparrow} \alpha_1 \circ \cdots \circ \alpha_k(\Omega)(u)$: this shows that $G(T)$ is non-total. □

3.2. **Weights and Approximations.** To use the size-change principle, compositions of calls need to be bounded. In the definition of the `length` function, the only recursive call is

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

Composing it with itself n times gives

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

which grows arbitrarily large! We introduce *approximations* to deal with that. When a term grows too large, constructors are only counted; and if this counter becomes too big, we stop counting. Everything is parameterized by two natural numbers defining what "too large" and "too big" really mean.

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

Definition 3.9 (Weights)**.** Define the following

(1) $\mathbb{Z}_\infty = \mathbb{Z} \cup \{\infty\}$ where addition is extended with $w + \infty = \infty + w = \infty$ and the order is extended with $w \leq \infty$.
(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:

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

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

Weights can count constructors and destructors (with negative elements of $\mathbb{Z}_\infty$). The special value ∞ is a way to stop counting when those numbers become too big. It does *not* mean that there are infinitely many constructors. The next lemma is straightforward.

Lemma 3.10.

(1) *Addition of weights is commutative, associative and monotonic,*
(2) $\langle 0 \rangle$ *is neutral for addition,*
(3) *any weight can be written (uniquely) as* $\sum_{p \in P} \langle w_p \rangle^p$ *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).*

3.2.2. *Approximations.* An approximation is defined as the sum of all simple terms it is supposed to approximates. Defining that requires the following notions.

Definition 3.11. A "shape" $\Delta \in \mathcal{O}_0$ is a simple normal form (Lemma 2.16) *which contains neither functions names nor* Ω.

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

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

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

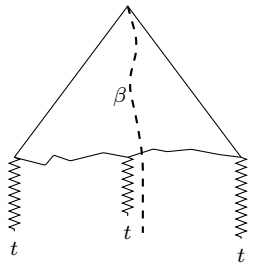
Definition 3.12 (Approximations)**.**

(1) Given some $W \in \mathbb{W}$, we put

$$\langle W\rangle \mathtt{x} \quad = \quad \sum \left\{ \Delta \;\middle|\; \begin{array}{l} \text{all branches } \beta \text{ of } \Delta \\ \text{satisfy } |\beta| \leq \langle W\rangle \end{array} \right\}$$

(2) Given t in $\mathcal{O}$, we write $\langle W\rangle t$ for the corresponding sum of all $\Delta[\mathtt{x} := t]$, for $\Delta \in \langle W\rangle \mathtt{x}$.

The typical summand of $\langle W\rangle t$ looks like:



$|\beta| \leq \langle W\rangle$, etc.

For example, the approximation $\langle\langle 1\rangle^p\rangle t$ contains, among others, the following summands: $\{\mathtt{Foo}^p = t\}$, $\{\mathtt{Foo}^p = \{\mathtt{Foo}^p = t\}\}$ and $\{\mathtt{Fst}^p = t; \mathtt{Snd}^p = \mathtt{C}^{q^-} t\}$.

Lemma 3.13. *Approximations are well defined elements of $\mathcal{O}$.*

Proof. This relies on the fact that there are only finitely many constructor and destructor names.

Let $W \in \mathbb{W}$, we want to show that $\langle W\rangle \mathtt{x}$ can be obtained as the limit of a chain of finite sums of simple elements of $\mathcal{O}$. Given $d \in \mathbb{N}$, define $\langle W\rangle \mathtt{x} \restriction d \subset \mathcal{O}_0$ as the set obtained by truncating summands of $\langle W\rangle \mathtt{x}$ at depth d. Truncating an element Δ is done by replacing subterms of Δ at depth d by $\Omega\mathtt{x}$. For example, "$\mathtt{A\ B\ C}^-\ \mathtt{x}$" truncated at depth 2 gives "$\mathtt{A\ B}\ \Omega\mathtt{x}$". Because there are only finitely many constructors and destructors, each $\langle W\rangle \mathtt{x} \restriction d$ is finite. Moreover, $\langle W\rangle \mathtt{x}$ is the limit of the chain

$$\langle W\rangle \mathtt{x} \restriction 1 \quad \leq \quad \langle W\rangle \mathtt{x} \restriction 2 \quad \leq \quad \cdots$$

Indeed, each element of $\langle W\rangle \mathtt{x} \restriction d+1$ is either in $\langle W\rangle \mathtt{x} \restriction d$ (when its depth is less than d), or greater than an element of $\langle W\rangle \mathtt{x}$ (when its depth is strictly greater than d). This shows that $\langle W\rangle \mathtt{x}$ is a limit of elements of $\mathcal{O}_0$.

This argument works unchanged when $\mathtt{x}$ is replaced by any term t. □

Lemma 3.14. *We have*

(1) *if $W \leq W'$ in $\mathbb{W}$, then $\langle W\rangle t \leq \langle W'\rangle t$,*
(2) $\langle 0\rangle(t) \leq t$,
(3) $\langle W\rangle \mathbf{0} = \mathbf{0}$
(4) $\langle W\rangle \mathtt{C}^p t = \langle W + \langle 1\rangle^p\rangle t$,
(5) $\langle W\rangle \{\mathtt{D}_1 = t_1; \ldots; \mathtt{D}_k = t_k\}^p \geq \Omega t_1 + \cdots + \Omega t_k$,
(6) $\mathtt{C}^{p^-}\langle W\rangle t = \langle W + \langle -1\rangle^p\rangle t$,
(7) $\mathtt{.D}^p\langle W\rangle t \geq \langle W + \langle -1\rangle^p\rangle t$,
(8) $\Omega\langle W\rangle t \geq \Omega t$,
(9) $\langle W\rangle \Omega t \geq \Omega t$,
(10) $\langle V\rangle\langle W\rangle t) \geq \langle V + W\rangle t$.

Proof.

(1) The first point is immediate once you recall the order on $\mathbb{W}$ is the *reverse* order on $\mathbb{Z}^{\mathbb{P}}_{\infty}$.
(2) The second point follows from the fact that $t = \mathtt{x}[t]$ is a summand of $\langle 0\rangle t$.
(3) Because a summand Δ of $\langle W\rangle\mathtt{x}$ cannot contain empty structures by definition, it must contain the variable $\mathtt{x}$. Because of that, all summands $\Delta[\mathbf{0}]$ contain $\mathbf{0}$ and are thus equal to $\mathbf{0}$ by linearity.
(4) Suppose $\Delta[\mathtt{C}^p t]$ is a summand in $\langle W\rangle\mathtt{C}^p t$. By defining $\Delta' = \Delta[\mathtt{C}^p\mathtt{x}]$, we have that $\Delta'[t] = \Delta[\mathtt{C}^p t]$ is a summand of $\langle W + \langle 1\rangle^p\rangle t$. This shows that $\langle W\rangle\mathtt{C}^p t \geq \langle W + \langle 1\rangle^p\rangle t$. For the converse, suppose $\Delta[t]$ is a summand in $\langle W + \langle 1\rangle^p\rangle t$. We put $\Delta' = \Delta[\mathtt{C}^{p^-}\mathtt{x}]$ so that $\Delta'[\mathtt{C}^p t] \geq \Delta[t]$ is a summand in $\langle W\rangle\mathtt{C}^p t$. This shows that $\langle W\rangle\mathtt{C}^p t \leq \langle W + \langle 1\rangle^p\rangle t$.
(5) This follows from points (8) below:

$$\begin{array}{lll} & \langle W\rangle\{\mathtt{D}_1 = t_1; \ldots; \mathtt{D}_k = t_k\}^p & \\ \geq & \Omega\langle W\rangle\{\mathtt{D}_1 = t_1; \ldots; \mathtt{D}_k = t_k\}^p & \text{(by definition of } \leq) \\ \geq & \Omega\{\mathtt{D}_1 = t_1; \ldots; \mathtt{D}_k = t_k\}^p & \text{(by points (9) below)} \\ \geq & \Omega t_1 + \cdots + \Omega t_k & \text{(by definition of } \leq) \end{array}$$

(6) By definition, any summand of $\mathtt{C}^{p^-}\langle W\rangle t$ is also a summand of $\langle W + \langle -1\rangle^p\rangle t$. This implies that $\mathtt{C}^{p^-}\langle W\rangle t \geq \langle W + \langle -1\rangle^p\rangle t$. For the converse, let $s = \Delta[t]$ be a summand in $\langle W + \langle -1\rangle^p\rangle t$. Take $\mathtt{C}^{-p}\mathtt{C}^p\Delta[t]$: this is a summand of $\mathtt{C}^{p^-}\langle W\rangle t$ which is equal to s. We can conclude that s is greater than a summand of t, which implies that $s \geq \langle W + \langle -1\rangle^p\rangle t$.
(7) This is treated similarly, except the second inequality cannot be proved because reducing $\mathtt{.D\{D} = \delta t\mathtt{\}}$ yields δt, a smaller term (second inequality in group (1) in $(*)$ in Definition 2.12).[11]
(8) Let s be a summand in $\Omega\langle W\rangle t$; it is of the form $\Omega\Delta[t]$. By contextuality, we have that $\Omega\Delta[t] \geq \Omega\Delta[\Omega t]$, and because Ω absorbs constructors on its right and destructors on its left, we have that $\Omega\Delta[\Omega t] \to^* \Omega t$. Because terms decrease during reduction, we have that $s \geq \Omega t$, which implies that $\Omega\langle W\rangle t \geq \Omega t$.
(9) We have that $\langle W\rangle\Omega t \geq \Omega\langle W\rangle\Omega t$ by definition of the order, and $\Omega\langle W\rangle\Omega t \geq \Omega t$ by the previous point (and contextuality). The result follows from transitivity.
(10) This is immediate. □

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

Approximations defined above corresponds to the first kind. Rather than having two definitions $\langle W\rangle^{\uparrow}$ and $\langle W\rangle^{\downarrow}$, we simply use negative weights to deal with the second kind. In each call $\mathtt{f\ x} \rightsquigarrow B\ \mathtt{f}\ u$, the branch B used for checking productivity uses "negated" weights, while the term u uses the standard weights as defined in this section. Because calls contain a single occurrence of $\mathtt{f}$, there is no ambiguity as to which weights must be negated.

[11] We could add this equality *for records with a single field*, but it would add yet another case to all the proofs involving the order, without any clear gain.

3.3. **Collapsing.** To avoid unbounded compositions like the one shown on page 27 we need to cut off compositions when they get too big. This is parameterized by 2 natural numbers:

- $D \geq 0$ bounding the depth of terms,
- $B > 0$ bounding the finite weights of approximations.

Both bounds can be chosen independently for each recursive definition.

3.3.1. *Calls.* Approximations are already elements of $\mathcal{O}_0$, but they correspond to infinite sums, which cannot be implemented directly. The following extends the syntax for $\mathcal{O}_0$ with "built-in" approximations.

Definition 3.15. The set $\mathcal{A}_0$ (for "$\mathcal{A}$pproximated terms") is defined inductively by

$$\begin{array}{rcl} t & ::= & \mathtt{C}^p\, t \quad | \quad \{\mathtt{D}_1 = t_1; \ldots; \mathtt{D}_n = t_n\}^p \quad | \\ & & \mathtt{C}^{p^-} t \quad | \quad \mathtt{.D}^p t \quad | \\ & & \mathtt{f}\, t \quad | \quad \mathtt{x} \quad | \\ & & \Omega t \quad | \quad \langle W \rangle t \quad | \\ & & t_1 + \cdots + t_n \end{array}$$

where $n > 0$, $\mathtt{x}$ is a fixed variable name 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.

The order is defined as before, adding inequalities from Lemma 3.14:

Definition 3.16. The order on $\mathcal{A}_0$ is defined inductively using the same rules as the order on $\mathcal{O}_0$ (Definition 2.12) together with some additional rules:

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

and

$$(*) \begin{cases} (4) & \langle W \rangle \mathbf{0} \approx \mathbf{0} \\ (4) & \langle W \rangle \mathtt{C}^p t \approx \langle W + \langle 1 \rangle^p \rangle t \\ (4) & \langle W \rangle \{\mathtt{D}_1 = t_1; \ldots; \mathtt{D}_k = t_k\}^p \geq \Omega t_1 + \cdots + \Omega t_k \\ (4) & \mathtt{C}^{p^-} \langle W \rangle t \approx \langle W + \langle -1 \rangle^p \rangle t \\ (4) & \mathtt{.D}^p \langle W \rangle t \geq \langle W + \langle -1 \rangle^p \rangle t \\ (4) & \Omega \langle W \rangle t \geq \Omega t \\ (4) & \langle W \rangle \Omega t \geq \Omega \langle W \rangle t \\ (4) & \langle V \rangle \langle W \rangle t \geq \langle V + W \rangle t \end{cases}$$

By Lemma 3.14, the order on approximations implies the order on their semantics in $\mathcal{O}$. We extend the reduction relation $\to$ to approximations.

Definition 3.17. Reduction on $\mathcal{A}_0$ extends reduction on $\mathcal{O}_0$ by adding rules, oriented from left to right, for all inequalities in group (4).

Just like before (Lemma 2.16), we have

Lemma 3.18.

(1) *If $t \to t'$ then $t \geq t'$.*
(2) *Reduction on $\mathcal{A}_0$ is strongly normalizing.*

(3) *Normal forms are generated by the following grammar*

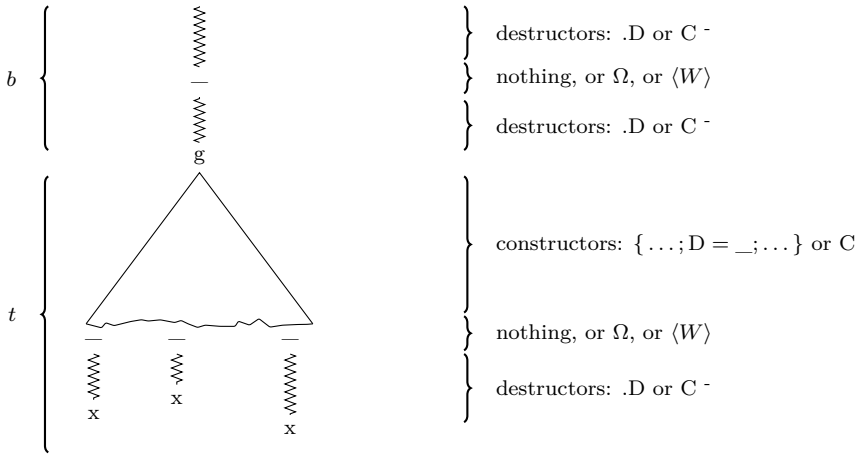
$$\begin{array}{lcl} t & ::= & \mathtt{C}^p t \mid \{\mathtt{D}_1 = t_1; \dots; \mathtt{D}_n = t_n\}^p \mid \\ & & \Omega\delta \mid \langle W\rangle\delta \mid \delta \\ \delta & ::= & \mathtt{C}^{p^-}\delta \mid \mathtt{.D}^p\delta \mid \mathtt{x} \mid \mathtt{f}\, t \end{array}$$

The proof is just a slight extension of the proof of Lemma 2.16.

Recall that $\mathtt{f\ x} \rightsquigarrow u$ is a notation for $u \in G(T_{\mathtt{f}})$. We extend this to approximations u.

Definition 3.19. A *call* from $\mathtt{f}$ to $\mathtt{g}$ is a 3-tuple consisting of

- a calling function name $\mathtt{f}$,
- a called function name $\mathtt{g}$,
- an approximation in normal form with the following shape:



In particular, the approximated term contains exactly one occurrence of a recursive function name, and no other function name. Such a call is written "$\mathtt{f\ x} \rightsquigarrow b\,\mathtt{g}\,t$".[12]

Lemma 3.20. *For any finite term T, $G(T)$ is a finite sum of calls.*

3.3.2. *Collapsing.*

Definition 3.21. Given $B > 0 \in \mathbb{N}$, the *weight collapsing function* $\lceil \_ \rceil_B$ replaces each weight $\sum_p \langle w_p\rangle^p$ (as in point (3) of Lemma 3.10) by $\sum_p \langle \lceil w_p \rceil_B\rangle^p$ where

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

To bound the depth, we add "$\langle 0\rangle$" below D constructors and above D destructors in the calls. When reducing, those weights will absorb the constructors below D and the destructors above D. For example, collapsing $\mathtt{C}_1\mathtt{C}_2\mathtt{C}_3\langle W\rangle\mathtt{C}_4^-\mathtt{C}_5^-\mathtt{C}_6^-\mathtt{C}_7^-\mathtt{x}$ (where all the constructors have the same priority) at depth 2 gives

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

[12]Since we don't deal with mutually recursive definitions, $\mathtt{f}$ and $\mathtt{g}$ are equal in our case.

The actual definition is a little tedious.

Definition 3.22. Let t be in normal form, given a positive bound $D \in \mathbb{N}$, the *depth collapsing function* $\_{\restriction_D}$ absorbs constructors below D and destructors above D into weights:

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

and the following are applied to normal forms

$$\begin{array}{rcll}
\big(\langle W\rangle\delta\mathtt{f}\,t\big)_{\lfloor_i} & \stackrel{\mathsf{def}}{=} & \langle W\rangle\delta_{\lfloor_i}\mathtt{f}\,t_{\restriction_D} & (\dagger)(**) \\
\big(\langle W\rangle\delta\mathtt{x}\big)_{\lfloor_i} & \stackrel{\mathsf{def}}{=} & \langle W\rangle\delta_{\lfloor_i}\mathtt{x} & (**) \\
\big(\delta\mathtt{C}^p\big)_{\lfloor_i} & \stackrel{\mathsf{def}}{=} & \delta_{\lfloor_{i-1}}\mathtt{C}^p & \\
\big(\delta\,.\mathtt{D}^p\big)_{\lfloor_i} & \stackrel{\mathsf{def}}{=} & \delta_{\lfloor_{i-1}}.\mathtt{D}^p & \\
\delta_{\lfloor_0} & \stackrel{\mathsf{def}}{=} & \delta\langle 0\rangle &
\end{array}$$

Note the following.

- The clauses are not disjoint and only the first applicable one is used.
- The innermost normal form in clause $(*)$ ensures that the clauses $(**)$ cover all cases (since weights absorb constructors on their right, $\langle 0\rangle t$ cannot start with constructors). The outermost normal form in clause $(*)$ ensures the result is in normal form.
- Clause $(\dagger)$ 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 3.23. *Given $B > 0$ and $D \geq 0$, the image of $\Big\lceil (\_)_{\restriction_D} \Big\rceil_B$ is finite.*

3.3.3. *Composing calls.*

Definition 3.24. *Collapsed composition* is defined by

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

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

Lemma 3.25. *For any call α, β, 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 $\lceil W\rceil_B \leq W$ in $\mathbb{W}$.

- for $\alpha_{\restriction_D}$, inserting some $\langle 0\rangle$ results in a smaller term by contextuality and the fact that $\langle 0\rangle t \leq t$. Normalizing can only make this smaller. □

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

- $\beta \diamond (\alpha \diamond \alpha) = \texttt{f x} \rightsquigarrow \langle \infty\rangle\ \texttt{f x}$ because $\lceil \lceil 1+1\rceil_1 + (-1)\rceil_1 = \lceil \infty + (-1)\rceil_1 = \infty$,
- $(\beta \diamond \alpha) \diamond \alpha = \texttt{f x} \rightsquigarrow \langle 1\rangle\ \texttt{f x}$. because $\lceil 1 + \lceil 1 + (-1)\rceil_1\rceil_1 = \lceil 1+0\rceil_1 = 1$.

Fortunately, just like in previous work on termination [Hyv14] the next property will be sufficient.

Lemma 3.26. *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 $\tau_1 \frown \tau_2$. 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 3.25. □

4. The Size-Change Principle

4.1. **The Size-Change Principle.** Putting Proposition 3.8, Corollary 3.5 and Lemma 2.33 together, we get

Corollary 4.1. *If all infinite paths in $G(T_{\texttt{f}})$ have a total semantics, then the usual semantics of $\texttt{f}$ is total in every total environment.*

Everything is now in place to apply the size-change principle from C. Lee, N. Jones and A. Ben-Amram [LJBA01], whose goal is precisely to deduce some property of all infinite paths of a graph from some property on its transitive closure. However, because collapsed composition isn't associative, we need a variant of the combinatorial lemma at the heart of the size-change principle. The following lemma is a slight generalization of Lemma 2.1 from previous work on termination [Hyv14].

Lemma 4.2. *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 and that $\diamond$ is a partial, binary, monotonic operation on F satisfying*

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

whenever $o_1 \circ o_2$ is defined.[13] *Then every infinite sequence $o_1, o_2, \ldots$ of elements of F where each finite $o_1 \circ \cdots \circ o_n$ is defined can be decomposed into*

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

where:

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

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

[13] In particular, $o_1 \diamond o_2$ is defined whenever $o_1 \circ o_2$ is.

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 $m < n$:

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

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 $r \in F$. Write $I = \{n_0 < n_1 < \cdots < n_k < \cdots\}$. If $i < j < k \in I$, we have:

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

The first two equalities imply that

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

If $\diamond$ is associative, this implies that $r \diamond r = r$. If not, we only get that both r and $r \diamond r$ are smaller than $o_i \circ \cdots \circ o_{j-1} \circ o_j \circ \cdots \circ o_{k-1}$. □

Definition 4.3. Let G be a call-graph. Start with $G^1 = G$ and define the edges of G^{n+1} to be those of G^n, together with:

> *if σ and ρ are edges from $\mathtt{f}$ to $\mathtt{g}$ and from $\mathtt{g}$ to $\mathtt{h}$ in G^n, then $\rho \diamond \sigma$ is a new edge from $\mathtt{f}$ to $\mathtt{h}$ in G^{n+1}.*

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

We can now state and prove correctness of the size-change principle. We extend the notions of branch and weight (Definition 3.11) with a new clause to deal with approximations:

- $|\langle W \rangle \delta| = \langle W \rangle + |\delta|$.

Theorem 4.4 (size-change principle)**.** *Suppose every loop $\sigma = \mathtt{f}\ \mathtt{x} \rightsquigarrow b\ \mathtt{f}\ u$ in G^* that satisfies $\sigma \subset \sigma \diamond \sigma$ ("σ is coherent") also satisfies one of the following two conditions:*

(1) *either there is an even priority p such that:*
   - *the p component of weight $|b|$ is strictly negative,*
   - *for all $q > p$, the q component of $|b|$ is positive;*

(2) *or there is a branch β of u and an odd priority p such that:*
   - *the p component of weight $|\beta|$ is strictly negative,*
   - *for all $q > p$, the q component of $|\beta|$ is positive;*

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

Proof. By Lemma 3.4, we only need to check that infinite paths are total. Let (s_k) be an 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 4.2: such a path can be decomposed into

$$t_0,\ t_1 \quad \ldots \quad t_{n_0} \quad \ldots \quad t_{n_1} \quad \ldots \quad t_{n_2} \quad \ldots$$

where:

- all the $t_{n_{k+1}-1} \diamond \cdots \diamond t_{n_k}$ are equal to the same $t = \mathtt{f} \rightsquigarrow b\,\mathtt{f}\,u$,
- t is *coherent*: $t \diamond t \subset t$.

In particular, we have $t_{n_{k+1}-1} \circ \cdots \circ t_{n_k} \geq t$.

Suppose that t satisfies the first condition. If we write T_0 for $t_0 \circ \cdots \circ t_{n_0-1}$, we have

$$\begin{aligned}
\bigsqcup_k^\uparrow s_k(\Omega) &= \bigsqcup_k^\uparrow t_0 \circ t_1 \circ \cdots \circ t_k(\Omega) \\
&\geq \bigsqcup_j^\uparrow T_0 \circ \underbrace{t \circ \cdots \circ t}_{j \text{ times}}(\Omega) \\
&= \bigsqcup_j^\uparrow T_0 \circ (b \mathtt{f} u)^j(\Omega) \\
&\geq \bigsqcup_j^\uparrow T_0 \circ (b \mathtt{f} \Omega)^j(\Omega) \\
&= \bigsqcup_j^\uparrow T_0 \circ b^j\Omega \\
&\geq T_0 \circ \bigsqcup_j^\uparrow b^j\Omega
\end{aligned}$$

where b^j is simply $b\,b\,\ldots\,b$.

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.

Dually, if t satisfies the second condition. We have

$$\begin{aligned}
\bigsqcup_k^\uparrow s_k(\Omega) &= \bigsqcup_k^\uparrow t_0 \circ t_1 \circ \cdots \circ t_k(\Omega) \\
&\geq \bigsqcup_j^\uparrow T_0 \circ t^j(\Omega) \\
&= \bigsqcup_j^\uparrow T_0 \circ (b \mathtt{f} u)^j(\Omega) \\
&\geq \bigsqcup_j^\uparrow T_0 \circ (\Omega \mathtt{f} u)^j(\Omega) \\
&= \bigsqcup_j^\uparrow T_0 \circ (\Omega \mathtt{f} u^j)(\Omega) \\
&\geq T_0 \circ \bigsqcup_j^\uparrow \Omega u^j
\end{aligned}$$

where u^j is obtained by replacing all $\mathtt{x}$ in u^{j-1} by u: $u^j = u[\mathtt{x} := u^{j-1}]$. By hypothesis, u contains a branch β and there is an odd p s.t. $|\beta|_p < 0$, so that u^j contains a branch $\beta\beta\ldots\beta$. Such a branch globally removes at least j 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!

Note that in both the first and second case, the branch usually contains approximations, so that while we may not know exactly which constructors are added (in the first case) or removed (in the second case), nor have an exact count, we have a bound of how many are added / removed, which is enough for the argument. □

4.2. **Examples.** Let's apply Theorem 4.4 to the examples from the introduction. With explicit priorities, the definition of `nats` is

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

The call-graph contains a single call $\sigma = \texttt{nats} \rightsquigarrow \texttt{Tail}^0\ \texttt{nats}\ \texttt{Succ}^1\ \texttt{x}$. If we use the bound $D = B = 1$, a single step is necessary to build the transitive closure:

- the first composition $\sigma \circ \sigma = \texttt{nats} \rightsquigarrow \texttt{Tail}^0\ \texttt{Tail}^0\ \texttt{nats}\ (\texttt{Succ}^1\ \texttt{Succ}^1\ \texttt{x})$ collapses[14] to $\rho = \sigma \diamond \sigma = \texttt{nats} \rightsquigarrow \texttt{Tail}^0\ \langle -1\rangle^0\ \texttt{nats}\ (\texttt{Succ}^1\ \langle\infty\rangle^1\ \texttt{x})$,
- after that, all compositions are equal to ρ.

The term ρ satisfies the first property of Theorem 4.4. By Lemma 4.2, all infinite compositions of σ eventually reduce to an infinite composition of ρ. But the only infinite branch resulting from an infinite composition of ρ is "$\texttt{Tail}^0\texttt{Tail}^0\ldots$", which has even principal priority. The recursive definition is total. Note that taking $D = 0, B = 1$ would have worked just as well.

In general, the infinite composition of ρ could also contain infinite branches coming from the argument `x` but since totality of a function only depends on its values on total arguments, we can suppose all infinite branches coming from `x` have even principal priority.

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

$$\sigma = \texttt{length x} \rightsquigarrow \texttt{Succ}^1\ \texttt{length}\ (.\texttt{Snd}^0\ \texttt{Cons}^{1\text{-}}\ \texttt{x})$$

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

$$\rho = \sigma \diamond \sigma = \texttt{length x} \rightsquigarrow \langle -1\rangle^1\ \texttt{length}\ (\langle\langle -1\rangle^0 + \langle -1\rangle^1\rangle\ \texttt{x})$$

The call ρ is coherent. It doesn't satisfy the first property of Theorem 4.4 but the second. By Lemma 4.2, infinite compositions of ρ remove an infinite number of constructors/destructors of priorities 0 and 1 from the argument, as seen in the weight $\langle\langle -1\rangle^0 + \langle -1\rangle^1\rangle$. (Those correspond to $\texttt{Succ}^1$ and $.\texttt{Snd}^0$.) As a result, any argument leading to infinite compositions cannot be total. The recursive definition is total.

The definition of `bad_s` has two recursive calls:

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

Its call-graph contains $\sigma_1 = \texttt{bad\_s} \rightsquigarrow \texttt{Head}^0\ \texttt{Node}^1\ \texttt{bad\_s}$ and $\sigma_2 = \texttt{bad\_s} \rightsquigarrow \texttt{Tail}^0\ \texttt{bad\_s}$. For $B = D = 1$, the transitive closure stabilizes after one step, and it contains 3 calls besides σ_1 and σ_2:

- $\rho_{1,1} = \sigma_1 \diamond \sigma_1 = \sigma_1 \diamond \sigma_2 = \texttt{bad\_s} \rightsquigarrow \texttt{Head}^0\ \langle\langle -1\rangle^0 + \langle -1\rangle^1\rangle\ \texttt{bad\_s}$
- $\rho_{2,2} = \sigma_2 \diamond \sigma_2 = \texttt{bad\_s} \rightsquigarrow \texttt{Tail}^0\ \langle -1\rangle^0\ \texttt{bad\_s}$
- $\rho_{2,1} = \sigma_2 \diamond \sigma_1 = \texttt{bad\_s} \rightsquigarrow \texttt{Tail}^0\ \langle\langle -1\rangle^0 + \langle -1\rangle^1\rangle\ \texttt{bad\_s}$

Those 3 calls are coherent, but while $\rho_{2,2}$ satisfies the first property of Theorem 4.4 neither $\rho_{1,1}$ nor $\rho_{2,1}$ 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. It means an infinite composition of recursive calls will necessarily generate an infinite branch with infinitely many constructors of priority 1 and 0, which is a non-total branch. The recursive definition is thus rejected by the totality checker, as it should. Changing B and D doesn't make a difference.

[14] recall that the branch above the recursive call uses negated weights (Section 3.2.3)

Theorem 4.4 is strong enough to deal with mixed inductive and coinductive types. Recall the definition of `sums` from page 6

```
val sums : nat¹ -> stream⁰(list¹(nat¹)) -> stream⁰(nat¹)
  | sums acc { Head⁰ = Nil¹ ; Tail⁰ = s } =
      { Head⁰ = acc ; Tail⁰ = sums (Zero³{}⁴) s }
  | sums acc { Head⁰ = Cons¹ {Fst⁰ = n ; Snd⁰ = l} ; Tail⁰ = s } =
      sums (add acc n) { Head⁰ = l ; Tail⁰ = s }
```

Because of the second clause, this definition isn't guarded. It is productive because this second clause cannot occur infinitely many times consecutively. Agda doesn't detect this definition as total. 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:

- ρ_1, coming from compositions of the first call with itself:

$$\texttt{sums x}_1\ \texttt{x}_2 \rightsquigarrow \underline{\texttt{Tail}^0\langle -1^0\rangle}\ \texttt{sums}\ (\texttt{Zero}^1)\ (\langle -1\rangle^0.\texttt{Tail}^0\,\texttt{x}_2)$$

  where the $\langle -1\rangle^0$ corresponds to the collapse of $\texttt{Tail}^0$,
- ρ_2, coming from compositions of the second call with itself:

$$\texttt{sums x}_1\ \texttt{x}_2 \rightsquigarrow \texttt{sums}\ (\Omega...)\ \{\ \underline{\texttt{Head}^0{=}\langle -1\rangle^1.\texttt{Head}^0\,\texttt{x}_2}\ ;\ \texttt{Tail}^0{=}.\texttt{Tail}^0\,\texttt{x}_2\ \}$$

  where $\langle -1\rangle^1$ corresponds to the collapse of $\texttt{Cons}^{1\text{-}}\texttt{Cons}^{1\text{-}}$,[15]
- ρ_3, coming from compositions of the first and second call:

$$\texttt{sum x}_1\ \texttt{x}_2 \quad\mapsto\quad \underline{\texttt{Tail}^0\langle -1\rangle^0}\ \texttt{sums}\ (\Omega...)\ \{\ \begin{array}{l}\texttt{Head}^0 = \langle\langle -1\rangle^0{+}\langle -1\rangle^1\rangle.\texttt{Tail}^0\,\texttt{x}_2\ ;\\ \texttt{Tail}^0 = \langle -1\rangle^0.\texttt{Tail}^0\,\texttt{x}_2\ \}\end{array}$$

  where $\langle\langle -1\rangle^0{+}\langle -1\rangle^1\rangle$ comes from the collapse of $\texttt{Cons}^{1\text{-}}.\texttt{Head}^0$ and $\langle -1\rangle^0$ from the collapse of $.\texttt{Tail}^0$.

Both ρ_1 and ρ_3 satisfy the first property of Theorem 4.4, while ρ_2 satisfies the second property. This definition is total.

4.3. **Implementing the Totality Checker.** Implementing the totality checker based on Theorem 4.4 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 [Hyv25].
(3) Annotate all constructors and destructors appearing in the recursive definition with their priorities. The types themselves can be forgotten at this point.
(4) The type of calls in normal forms is a simple first-order inductive type. Define relevant functions on calls, namely composition, and collapsing. Note that composition is implicitly followed by reduction to normal form, which must be done during composition.
(5) 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.
(6) Compute the transitive closure of the call graph using the previously defined functions (composition and collapsing).

[15] The "Ω..." comes from the application `add acc n` and doesn't play any role in this example.

(7) 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 4.4. If all of them do, the definition is total.

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 implementing the order as well. Because calls are kept in normal form for collapsing, we can use the syntax directed inductive relation $\sqsubseteq$ from Appendix A.1 and A.2 instead of $\leq$.

The last part requires checking coherence for loops in the transitive closure of the call-graph. In practice, it looks like checking loops for which $\sigma = \sigma \diamond \sigma$ is enough but I haven't tried very hard to prove this fact. Devising an inductive characterization of coherence is difficult but we can simplify things by using a weaker inductive relation that is only implied by coherence. That means we may end up checking more loop than strictly necessary but the result is provably correct.[16] This is described in Appendix A.3.

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. Hardness comes from computing the transitive closure of the call-graph. 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 examples do not arise naturally. Letting the user choose the bounds B and D (with sane default values) limits the extra complexity cost 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 simple totality checker with low complexity.

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

- 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 $\alpha =$ `f` $\rightsquigarrow$ `f (C1 C2⁻ x)`. With $D = 0$, this call is collapsed to `f` $\rightsquigarrow$ `f (`$\langle 0 \rangle$`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}
```

[16] And I haven't found any definition where this stricter check changes the result of the totality checker.

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

$$\texttt{f} \rightsquigarrow \texttt{f}\big(\Omega(\ \langle\langle\infty\rangle^0 + \langle -1\rangle^1\rangle\ \texttt{x} + \langle\langle\infty\rangle^0 + \langle\infty\rangle^1\rangle\ \texttt{x}\ )\big)$$

which doesn't satisfy the hypothesis of Theorem 4.4.

- Increasing B isn't as useful. It 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 of mutually inductive definitions is accepted with $B = 2$ and $D = 0$ but rejected with $B = 1$ and $D = 0$:

  ```
  val s1 = s2.Tail^0
  and s2 = { hd^0 = Zero^1; Tail^0 = { hd^0=1^1; Tail^0=s1 }}
  ```

  The call-graph has 2 vertices and 2 arcs: `s1` $\rightsquigarrow \langle 1\rangle^0$ `s2` and `s2` $\rightsquigarrow \langle -2\rangle^0$ `s1`. When projecting with $B = 2$, the composition gives `s1` $\rightsquigarrow \langle -1\rangle^0$`s1` (and similarly for `s2`), which passes the totality test. When projecting with $B = 1$, the first arc gives `s1` $\rightsquigarrow \langle\infty\rangle^0$`s2` which gives compositions of `s1` $\rightsquigarrow \langle\infty\rangle^0$`s1` (and similarly for `s2`), which doesn't pass the totality test.

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

Note that none of those examples are detected as correct by the current termination checker in Agda.

Strength of the totality checker. This paper only proves correctness of the totality checker. It doesn't prove anything about its strength. Another provably correct totality checker is the one that always returns "`I DON'T KNOW`". The only argument in favor of the totality checker is of a practical nature: experimenting with `chariot` shows that it is enough for many recursive functions. General results like "all structurally recursive definitions are total", "all syntactically guarded definitions are total", etc. are certainly true but not investigated here.

Higher order types. The implementation of `chariot` deals with some higher order datatypes. With `b`-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` functions passes the totality test. The theory should extend straightforwardly to account for this kind of datatypes.

Dependent Types. This totality checker could deal with dependent types by simply tagging any definition involving dependent types with "`I DON'T KNOW`". Of course, extending it to do something interesting on dependent types would be preferable. Many useful dependent types like "lists of size n" can be embedded in bigger non dependent datatypes like ("lists" in this case). Because the totality checker is essentially untyped, those types, together with dependent sums could be dealt with by using the totality checker unchanged. That, and the extension to some higher order as described above would go a long way to provide a theoretically sound totality checker for a dependent languages like Agda.[17]

References

[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. doi:10.29007/n51d.

[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. URL: http://dl.acm.org/citation.cfm?id=218742.218744.

[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.

[Coc96] Robin Cockett. Charitable thoughts, 1996. draft lecture notes. URL: http://www.cpsc.ucalgary.ca/projects/charity/home.html.

[Coq94] Thierry Coquand. Infinite objects in type theory. In Henk Barendregt and Tobias Nipkow, editors, *Types for Proofs and Programs*, pages 62–78, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg. doi:10.1007/3-540-58085-9_72.

[Hyv14] Pierre Hyvernat. The size-change termination principle for constructor based languages. *Logical Methods in Computer Science*, 10(1), 2014. URL: http://dx.doi.org/10.2168/LMCS-10(1:11)2014, doi:10.2168/LMCS-10(1:11)2014.

[Hyv25] Pierre Hyvernat. Totality for mixed inductive and coinductive types, 2025. to appear Logical Methods in Computer Science. URL: https://arxiv.org/abs/1901.07820, arXiv:1901.07820.

[LJBA01] Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '01, page 81–92, New York, NY, USA, 2001. Association for Computing Machinery. doi:10.1145/360204.360210.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978. doi:10.1016/0022-0000(78)90014-4.

[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. URL: https://www.microsoft.com/en-us/research/publication/the-implementation-of-functional-programming-languages/.

[Smy78] Michael B. Smyth. Power domains. *Journal of Computer and System Sciences*, 16(1):23–36, 1978. doi:10.1016/0022-0000(78)90048-X.

[The04] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. Version 8.0. URL: http://coq.inria.fr.

[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. URL: https://londmathsoc.onlinelibrary.wiley.com/doi/abs/10.1112/plms/s2-42.1.230, doi:10.1112/plms/s2-42.1.230.

[17]Types that cannot be analyzed with this totality checker probably include things like functions types with non constant arity.

Appendix A. Inductive order

This section describes some simple inductive relations that are much easier to implement than the order $\leq$ on $\mathcal{O}_0$ and $\mathcal{A}_0$, and the coherence $\frown$. This is possible because the implementation only needs to deal with terms in normal form.

A.1. Inductive order on normal forms in $\mathcal{O}$.

Definition A.1. The relation $\sqsubseteq$ on normal forms of $\mathcal{O}_0$ is inductively generated by the following rules:

$$\frac{\forall j, \exists i, s_i \sqsubseteq t_j}{\sum_i s_i \sqsubseteq \sum_j t_j}\sqsubseteq_+$$

$$\frac{}{\mathtt{x} \sqsubseteq \mathtt{x}}\sqsubseteq_{\mathtt{x}} \qquad \frac{s \sqsubseteq t}{\mathtt{f}\, s \sqsubseteq \mathtt{f}\, t}\sqsubseteq_{\mathtt{f}}$$

$$\frac{s \sqsubseteq t}{\mathtt{C}\, s \sqsubseteq \mathtt{C}\, t}\sqsubseteq_{\mathtt{C}} \qquad \frac{s_1 \sqsubseteq t_1 \quad \dots \quad s_n \sqsubseteq t_n}{\{\mathtt{D}_1 = s_1; \dots; \mathtt{D}_n = s_n\} \sqsubseteq \{\mathtt{D}_1 = t_1; \dots; \mathtt{D}_n = t_n\}}\sqsubseteq_{\{\mathtt{D}_1;\dots;\mathtt{D}_n\}}$$

$$\frac{s \sqsubseteq t}{\mathtt{C}^{-} s \sqsubseteq \mathtt{C}^{-} t}\sqsubseteq_{\mathtt{C}^{-}} \qquad \frac{s \sqsubseteq t}{.\mathtt{D}\, s \sqsubseteq .\mathtt{D}\, t}\sqsubseteq_{.\mathtt{D}}$$

$$\frac{s \sqsubseteq t}{\Omega s \sqsubseteq \Omega \delta t}\sqsubseteq_{\Omega 1} \qquad \frac{\Omega s \sqsubseteq \mathsf{nf}(\Omega t)}{\Omega s \sqsubseteq t}\sqsubseteq_{\Omega 2} \textit{(provided } t \textit{ doesn't start with } \Omega\textit{)}$$

where in rule $\sqsubseteq_{\Omega 1}$, δ is any sequence of destructors $\mathtt{C}^{-}/.\mathtt{D}$ and function names $\mathtt{f}$, $\mathtt{g}$, etc.

The last two clauses are the most interesting. The rule $\sqsubseteq_{\Omega 2}$ requires normalizing Ωt before the inductive step, and $\sqsubseteq_{\Omega 1}$ requires finding the appropriate prefix δ. This is decidable as there are only finitely many prefixes to check. The first rule is computationally the most complex one, as it potentially requires checking $s_i \sqsubseteq t_j$ for all i, j. Here again, there are only finitely many possibilities. Note for any given s and t, there is at most one rule that can be used to derive $s \sqsubseteq t$, which makes the $\sqsubseteq$ relation straightforward to implement.

Lemma A.2. *$s \sqsubseteq t$ implies $s \leq t$.*

Proof. This is a straightforward induction on $s \sqsubseteq t$. For example, suppose the last rule was $\sqsubseteq_{\mathtt{C}}$, with conclusion $\mathtt{C}s \sqsubseteq \mathtt{C}t$ and premise $s \sqsubseteq t$. By induction, we have $s \leq t$, which implies that $\mathtt{C}s \leq \mathtt{C}t$ by contextuality of $\leq$. All other cases are treated similarly, with an extra bit of reasoning for rules $\sqsubseteq_+$, $\sqsubseteq_{\Omega 1}$ and $\sqsubseteq_{\Omega 2}$:

- For $\sqsubseteq_+$, we use point (1) of Lemma 2.13.
- for $\sqsubseteq_{\Omega 1}$, we have $s \leq t$ by induction. From there, we have

$$\begin{array}{rcll} \Omega s & \leq & \Omega t & \text{(definition of } \leq\text{: contextuality)} \\ & \approx & \Omega\Omega t & \text{(definition of } \leq\text{)} \\ & \leq & \Omega\delta\Omega t & \text{(Lemma 2.13)} \\ & \leq & \Omega\delta t & \text{(definition of } \leq\text{: contextuality)} \end{array}$$

- For $\sqsubseteq_{\Omega 2}$, we have that $\Omega s \leq \mathsf{nf}(\Omega t)$ by induction. Since $\mathsf{nf}(\Omega t) \leq \Omega t \leq t$, we get that $\Omega s \leq t$ by transitivity. □

The rest of this section will show that $\sqsubseteq$ is in fact equivalent to $\leq$ restricted to normal forms

Proposition A.3. *For all terms s and t, $s \leq t$ implies $\mathtt{nf}(s) \sqsubseteq \mathtt{nf}(t)$.*

The proof is a little tedious. We decompose it into several auxiliary lemmas with straightforward proofs.

Lemma A.4.

(1) *If $s \sqsubseteq t$ are simple normal forms, then $\mathtt{nf}(\mathtt{.D}s) \sqsubseteq \mathtt{nf}(\mathtt{.D}t)$.*
(2) *The same holds when $\mathtt{.D}$ is replaced by $\mathtt{C}^-$ or $\mathtt{f}$.*
(3) *The same holds when $\mathtt{.D}$ is replaced by Ω.*

Proof.

(1) We prove the first point by induction on $s \sqsubseteq t$, looking at the last rule.
- Rule $\sqsubseteq_+$: by induction hypothesis, we have that $\forall j, \exists i, \mathtt{nf}(\mathtt{.D}s_i) \leq \mathtt{nf}(\mathtt{.D}t_j)$. This implies that $\mathtt{nf}(\mathtt{.D}s) = \sum_i \mathtt{nf}(\mathtt{.D}s_i) \sqsubseteq \sum_j \mathtt{nf}(\mathtt{.D}t_j) = \mathtt{nf}(\mathtt{.D}t)$.
- Rule $\sqsubseteq_{\mathtt{x}}$: the result holds by definition.
- Rule $\sqsubseteq_{\mathtt{f}}$: because $\mathtt{f}s$ and $\mathtt{f}t$ are in normal form, $\mathtt{nf}(\mathtt{.D}\mathtt{f}s) = \mathtt{.D}\mathtt{f}s$ and $\mathtt{nf}(\mathtt{.D}\mathtt{f}t) = \mathtt{.D}\mathtt{f}t$. The result holds by definition of $\sqsubseteq$. (No induction necessary.)
- Rule $\sqsubseteq_{\mathtt{C}}$, or $\sqsubseteq_{\{\ldots\}}$ when the record doesn't contain the $\mathtt{D}$ field: $\mathtt{nf}(\mathtt{.D}s) = \mathtt{nf}(\mathtt{.D}t) = \mathbf{0}$. The result follows from rule $\sqsubseteq_+$.
- Rule $\sqsubseteq_{\{\ldots;\mathtt{D};\ldots\}}$: because $\mathtt{nf}(\mathtt{.D}\{\ldots;\mathtt{D} = u;\ldots\} = \mathtt{nf}(u)$, the result holds by definition.
- Rule $\sqsubseteq_{\mathtt{.D}'}$: because $\mathtt{.D}'s$ is in normal form by hypothesis, we have $\mathtt{nf}(\mathtt{.D.D}'s) = \mathtt{.D.D}'s$. Similarly, $\mathtt{nf}(\mathtt{.D.D}'t) = \mathtt{.D.D}'t$, and the result holds by definition.
- The same reasoning works for rule $\sqsubseteq_{\mathtt{C}^-}$.
- Rule $\sqsubseteq_{\Omega 1}$: we have $\mathtt{nf}(\mathtt{.D}\Omega s) = \Omega s$ and $\mathtt{nf}(\mathtt{.D}\Omega\delta t) = \Omega\delta t$. The result hold trivially.
- Rule $\sqsubseteq_{\Omega 2}$ is the most complex case. Suppose we have $\Omega s \leq t$ because $\Omega s \sqsubseteq \mathtt{nf}(\Omega t)$. Because Ωs is already in normal form by hypothesis, we have $\mathtt{nf}(\mathtt{.D}\Omega s) = \Omega s$. We thus need to check that $\Omega s \sqsubseteq \mathtt{nf}(\mathtt{.D}t)$. Using $\sqsubseteq_{\Omega 2}$, we need to show that $\Omega s \sqsubseteq \mathtt{nf}(\Omega\mathtt{nf}(\mathtt{.D}t))$. By Lemma 2.17, $\mathtt{nf}(\Omega\mathtt{nf}(\mathtt{.D}t)) = \mathtt{nf}(\Omega\,\mathtt{.D}t)$. We do a case analysis on t:
  - if $t = \mathtt{x}$, $t = \mathtt{f}t'$, $t = \mathtt{C}^-t'$ or $t = \mathtt{.D}'t'$: because t in already in normal form, we have $\mathtt{nf}(\Omega\,\mathtt{.D}t) = \Omega\,\mathtt{.D}t$. By hypothesis, $\Omega s \sqsubseteq \mathtt{nf}(\Omega t)$, which an only be proved using rules $\sqsubseteq_+$ (if $\mathtt{nf}(\Omega t)$ involves sums) followed by $\sqsubseteq_{\Omega 1}$. So all summands of $\mathtt{nf}(\Omega t)$ are of the form $\Omega\delta t'$ with $s \sqsubseteq t'$.
    But when t has one of the above shapes, $\mathtt{nf}(\Omega t) = \Omega t$, so that t itself is of the form $\delta t'$ with $s \sqsubseteq t'$. This implies that $\mathtt{.D}t$ is also of the form $\delta t'$ with $s' \sqsubseteq t'$. From that, we get that $\Omega s \sqsubseteq \Omega\,\mathtt{.D}t$ using rule $\sqsubseteq_{\Omega 1}$ and thus that $\Omega s \sqsubseteq \mathtt{.D}t$ using rule $\sqsubseteq_{\Omega 2}$.
  - If $t = \mathtt{C}t'$, or if t is a record without the $\mathtt{D}$ field, then $\mathtt{nf}(\mathtt{.D}t) = \mathbf{0}$ and the result holds trivially.
  - If $t = \{\ldots;\mathtt{D} = t';\ldots\}$, and because t is in normal form, we have that $\mathtt{nf}(\mathtt{.D}t) = t'$. By hypothesis, we now that $\Omega s \sqsubseteq \mathtt{nf}(\Omega t)$, which can only be proved using rules $\sqsubseteq_+$ and $\sqsubseteq_{\Omega 1}$. This implies that all summands of $\mathtt{nf}(\Omega t) = \sum_j \mathtt{nf}(\Omega t_j)$ are of the form $\delta t''$ with $s \sqsubseteq t''$. Since all summands of $\mathtt{nf}(\Omega t')$ are summands of $\mathtt{nf}(\Omega t)$, we can conclude that $\Omega s \sqsubseteq \mathtt{nf}(\mathtt{.D}t)$.
  - If $t = \Omega t'$, then $\mathtt{nf}(\Omega\,\mathtt{.D}t) = t$. The result holds trivially.

(2) The same reasoning applies to the case where we replace $\mathtt{.D}$ by $\mathtt{C}^-$ or $\mathtt{f}$.

(3) The third point is proved similarly:
- rule $\sqsubseteq_+$: follows directly from the induction hypothesis and linearity of Ω.
- rule $\sqsubseteq_{\mathtt{x}}$: the result holds by definition.

- rule $\sqsubseteq_{\mathtt{f}}$: because $\mathtt{f}s$ and $\mathtt{f}t$ are in normal form, we have $\mathsf{nf}(\Omega\mathtt{f}s) = \Omega\mathtt{f}s$ and $\mathsf{nf}(\Omega\mathtt{f}t) = \Omega\mathtt{f}t$. The result holds by definition of $\sqsubseteq$ (rule $\sqsubseteq_{\Omega 1}$).
- rule $\sqsubseteq_{\mathtt{C}}$: because $\mathsf{nf}(\Omega\mathtt{C}s) = \mathsf{nf}(\Omega s)$ and $\mathsf{nf}(\Omega\mathtt{C}t) = \mathsf{nf}(\Omega t)$, the result holds by induction.
- rule $\sqsubseteq_{\{\dots;\mathtt{D};\dots\}}$: because $\mathsf{nf}(\Omega\{\dots;\mathtt{D}_i = u_i;\dots\} = \sum_i \mathsf{nf}(\Omega u_i)$, the result follows from the induction hypotheses and rule $\sqsubseteq_+$.
- rule $\sqsubseteq_{\mathtt{C}^-}$: because both $\mathtt{C}^- s$ and $\mathtt{C}^- t$ are in normal form, we have $\mathsf{nf}(\Omega\mathtt{C}^- s) = \Omega\mathtt{C}^- s$ and similarly for t. The result hold by definition (rule $\sqsubseteq_{\Omega 1}$).
- The same reasoning works for rule $\sqsubseteq_{.\mathtt{D}}$.
- rule $\sqsubseteq_{\Omega 1}$: both Ωs and Ωt are in normal form, so that we have $\mathsf{nf}(\Omega\Omega s) = \Omega s$ and similarly for t. The result hold by hypothesis.
- rule $\sqsubseteq_{\Omega 2}$: Ωs is in normal, so that $\mathsf{nf}(\Omega\Omega s) = \Omega s$. We need to prove that $\Omega s \sqsubseteq \mathsf{nf}(\Omega t)$, which is precisely the premise of rule $\sqsubseteq_{\Omega 2}$. □

Note that the fact that the terms are in normal form is crucial in the proof.

Lemma A.5. *The relation $\sqsubseteq$ is contextual: if $s \sqsubseteq t$ for some s, t simple normal forms, and if C is a context, then $\mathsf{nf}(C[\mathtt{y} := s]) \sqsubseteq \mathsf{nf}(C[\mathtt{y} := t])$.*

Proof. This is done by induction of the context C, where the difficult inductive steps are taken care of by the preceding lemma.

- If $C = \mathtt{x}$ or $\mathtt{C} = \mathtt{y}$, the result holds trivially.
- If $C = \{\dots;\mathtt{D}_i = C_i;\dots\}$, then we know that each $\mathsf{nf}(C_i[\mathtt{y} := s]) \sqsubseteq \mathsf{nf}(C_i[\mathtt{y} := t])$ by induction. Since $\mathsf{nf}(C[\mathtt{y} := s]) = \{\dots;\mathtt{D}_i = \mathsf{nf}(C_i[\mathtt{y} := s]);\dots\}$ and similarly for $C[\mathtt{y} := t]$, the results holds by definition of $\sqsubseteq$.
- The same reasoning works for the cases $C = \mathtt{f}C'$ or $C = \mathtt{C}C'$.
- If $C = .\mathtt{D}C'$, then we know that $\mathsf{nf}(C'[\mathtt{y} := s]) \sqsubseteq \mathsf{nf}(C'[\mathtt{y} := t])$ by induction. Lemma A.4 implies that $\mathsf{nf}(C[\mathtt{y} := s]) \sqsubseteq \mathsf{nf}(C[\mathtt{y} := t])$.
- The cases $C = \mathtt{C}^- C'$ is treated similarly, with the help of Lemma A.4.
- If $C = \Omega C'$ then we know that $\mathsf{nf}(C'[\mathtt{y} := s]) \sqsubseteq \mathsf{nf}(C'[\mathtt{y} := t])$ by induction. The previous lemma implies that $\mathsf{nf}(\Omega C'[\mathtt{y} := s]) \sqsubseteq \mathsf{nf}(\Omega C'[\mathtt{y} := t])$. □

Lemma A.6. *The relation $\sqsubseteq$ is reflexive and transitive.*

Proof. Reflexivity is an obvious induction. For transitivity, suppose $s \sqsubseteq t$ and $t \sqsubseteq u$. We proceed by induction on $t \sqsubseteq u$ and case inspection on $s \sqsubseteq t$.

- if both $t \sqsubseteq u$ and $s \sqsubseteq t$ come from $\sqsubseteq_+$, each u_j is greater than some t_i which is greater than some s_k. By induction, each u_j is thus greater than some s_k, implying than $\sum_k s_k \sqsubseteq \sum_i u_i$ by rule $\sqsubseteq_+$.
- if only $t \sqsubseteq u$ comes from $\sqsubseteq_+$, then t is not a sum. Each u_j is thus greater than t, and induction implies that each u_j is greater than s. Rule $\sqsubseteq_+$ implies that $s \sqsubseteq \sum_i u_i$.
- rule $\sqsubseteq_{\mathtt{f}}$: we have $t = \mathtt{f}t'$ and $u = \mathtt{f}u'$, together with $t' \sqsubseteq u'$. Because $s \sqsubseteq t$, we also have $s = \mathtt{f}s'$ with $s' \sqsubseteq t'$. By induction hypothesis, we get that $s' \sqsubseteq u'$, and thus that $s \sqsubseteq u$ by rule $\sqsubseteq_{\mathtt{f}}$.
- The rules $\sqsubseteq_{\mathtt{x}}$, $\sqsubseteq_{\mathtt{C}}$, $\sqsubseteq_{\{\dots;\mathtt{D};\dots\}}$, $\sqsubseteq_{\mathtt{C}^-}$ and $\sqsubseteq_{.\mathtt{D}}$ are all treated similarly.
- If $t \sqsubseteq u$ comes from rule $\sqsubseteq_{\Omega 1}$, reasoning is similar, as $s \sqsubseteq t$ also comes from $\sqsubseteq_{\Omega 1}$.
- The last case is when $t \sqsubseteq u$ comes from $\sqsubseteq_{\Omega 2}$, i.e. when $t = \Omega t'$ and u doesn't start with Ω. The premise of this rule is $\Omega t' \sqsubseteq \mathsf{nf}(\Omega u')$. Because $s \sqsubseteq \Omega t'$, s is necessarily of the form $\Omega s'$, and we get that $\Omega s' \sqsubseteq \mathsf{nf}(\Omega u)$ by induction. We conclude that $\Omega s' \sqsubseteq u$ by rule $\sqsubseteq_{\Omega 2}$. □

We can now put everything together to prove the main proposition of this section.

Proof of Proposition A.3. The order $\leq$ is generated inductively by reflexivity, transitivity, commutativity, associativity, idempotence of $+$, (multi) linearity, contextuality, $t \leq \Omega t$, $t \leq \mathbf{0}$, $s + t \leq t$ and the(in)equalities from Definition 2.12. The proof of Proposition A.3 is an induction on $s \leq t$.

- If $s \leq t$ by reflexivity, i.e. t syntactically equal to s, then we have $s \sqsubseteq t$.
- If $s \leq t$ holds by transitivity $s \leq u$ and $u \leq t$. By induction hypothesis, we get that $\mathsf{nf}(s) \sqsubseteq \mathsf{nf}(u)$ and $\mathsf{nf}(u) \sqsubseteq \mathsf{nf}(t)$. We thus get $\mathsf{nf}(s) \sqsubseteq \mathsf{nf}(t)$ by transitivity of $\sqsubseteq$ (Lemma A.6).
- If $s \leq t$ holds by commutativity, associativity, idempotence of $+$ or (multi)-linearity, the result follows from rule $\sqsubseteq_+$.
- Similarly, if $s \leq t$ holds by contextuality, we get that $s \sqsubseteq t$ by induction and contextuality of $\sqsubseteq$ (Lemma A.5).
- If s is equal to Ωt, we need to check that $\mathsf{nf}(\Omega t) \sqsubseteq \mathsf{nf}(t)$. This is obvious if t starts with a Ω. Otherwise, we need to use rule $\sqsubseteq_{\Omega 2}$: it is enough to show that $\mathsf{nf}(\Omega t) \sqsubseteq \mathsf{nf}(\Omega \mathsf{nf}(t))$. Since $\mathsf{nf}(\Omega \mathsf{nf}(t)) = \mathsf{nf}(\Omega t)$, the result holds by reflexivity of $\sqsubseteq$.
- If $t = \mathbf{0}$, we have $\mathsf{nf}(s) \sqsubseteq \mathbf{0}$ using rule $\sqsubseteq_+$.
- If $s = s' + t$, we have $\mathsf{nf}(s) = \mathsf{nf}(s') + \mathsf{nf}(t) \sqsubseteq \mathsf{nf}(t)$ using rule $\sqsubseteq_+$.
- If $s \leq t$ using one (in)equality from Definition 2.12, we have that s reduces to t or that t reduces to s. In either case, $\mathsf{nf}(s) = \mathsf{nf}(t)$ so that $\mathsf{nf}(s) \sqsubseteq \mathsf{nf}(t)$. □

A.2. **Inductive order on normal forms in $\mathcal{A}$.** We can extend $\sqsubseteq$ to approximations:

Definition A.7. We extend Definition A.1 to normal forms of $\mathcal{A}$ by adding the following rules:

$$\frac{s \sqsubseteq t \qquad \mathsf{nf}\big(\langle W\rangle\delta\langle 0\rangle t\big) = \langle W'\rangle t \qquad \langle V\rangle \leq \langle W'\rangle}{\langle V\rangle s \sqsubseteq \langle W\rangle \delta t}\sqsubseteq_{\langle\rangle 1} \qquad \frac{\langle W\rangle s \sqsubseteq \mathsf{nf}\big(\langle 0\rangle t\big)}{\langle W\rangle s \sqsubseteq t}\sqsubseteq_{\langle\rangle 2}{}^{*}$$

where, in rule $\sqsubseteq_{\langle\rangle 2}$, t doesn't start with an approximation; and in rule $\sqsubseteq_{\langle\rangle 1}$, δ is any sequence of destructors $\mathtt{C}^{-}/\mathtt{.D}$.

Note that approximations behave very similarly to Ω. Checking $\sqsubseteq$ is still decidable, as the rules are syntax directed, and we still have

Lemma A.8. *For all approximated terms s and t in normal form, if $s \sqsubseteq t$, then $s \leq t$.*

Proof. All the rules remain valid, and we only have to check that the two new rules are correct.

- For $\sqsubseteq_{\langle\rangle 1}$, suppose that $s \leq t$, $\mathsf{nf}(\langle W\rangle\delta\langle 0\rangle t) = \langle W'\rangle t$ and $\langle V\rangle \leq \langle W'\rangle$ in $\mathbb{W}$. We need to show that $\langle V\rangle s \leq \langle W\rangle \delta t$. We have

$$\begin{array}{rcll} \langle W\rangle\delta t & \geq & \langle W\rangle\delta\langle 0\rangle t & \text{(contextuality, because } \langle 0\rangle t \leq t\text{)} \\ & \geq & \mathsf{nf}(\langle W\rangle\delta\langle 0\rangle t) & \text{(terms decrease along reduction)} \\ & = & \langle W'\rangle t & \text{(hypothesis)} \\ & \geq & \langle W'\rangle s & \text{(contextuality, because } s \leq t\text{)} \\ & \geq & \langle V\rangle s & \text{(because } \langle V\rangle \leq \langle W\rangle\text{)} \end{array}$$

- For $\sqsubseteq_{\langle\rangle 2}$, we have that $\langle W\rangle s \leq \mathsf{nf}(\langle 0\rangle t)$ by hypothesis. We also know that $\mathsf{nf}(\langle 0\rangle t) \leq \langle 0\rangle t \leq t$, so that we have $\langle W\rangle \leq t$ by transitivity. □

A.3. **Weak coherence on normal forms in $\mathcal{A}$.**

Lemma A.9.

(1) *Define an inductive binary relation $\sqsubset\!\!\sqsupset$ with:*
   (a) $\mathtt{x} \sqsubset\!\!\sqsupset \mathtt{x}$,
   (b) $\mathtt{C}u \sqsubset\!\!\sqsupset \mathtt{C}v$ *iff* $\mathtt{C}^{-}u \sqsubset\!\!\sqsupset \mathtt{C}^{-}v$ *iff* $\mathtt{.D}u \sqsubset\!\!\sqsupset \mathtt{.D}v$ *iff* $\mathtt{f}u \sqsubset\!\!\sqsupset \mathtt{f}v$ *iff* $u \sqsubset\!\!\sqsupset v$,
   (c) $\{\mathtt{D}_1 = u_1; \dots; \mathtt{D}_k = u_k\} \sqsubset\!\!\sqsupset \{\mathtt{D}_1 = v_1; \dots; \mathtt{D}_k = v_k\}$ *iff* $\forall i, u_i \sqsubset\!\!\sqsupset v_i$,
   (d) $\Omega u \sqsubset\!\!\sqsupset \Omega v$ *iff*
      - *there is a sequence of destructors δ s.t. $u = \delta u'$ with $u' \sqsubset\!\!\sqsupset v$,*
      - *or there is a sequence of destructors δ s.t. $v = \delta v'$ with $u \sqsubset\!\!\sqsupset v'$,*
   (e) $u \sqsubset\!\!\sqsupset \langle W \rangle v$ *iff* $\langle W \rangle u \sqsubset\!\!\sqsupset v$ *iff* $u \sqsubset\!\!\sqsupset \Omega v$ *iff* $\Omega u \sqsubset\!\!\sqsupset v$ *iff* $\mathsf{nf}(\Omega u) \sqsubset\!\!\sqsupset \mathsf{nf}(\Omega v)$.
   (f) *In all other cases, $u \not\sqsubset\!\!\sqsupset v$.*
(2) *For all terms in normal form u and v, if $u \frown v$, then $u \sqsubset\!\!\sqsupset v$.*

Proof. We prove that $u \sqsubseteq t$ and $v \sqsubseteq t$ implies $u \sqsubset\!\!\sqsupset v$ by induction on $u \sqsubseteq t$ and $v \sqsubseteq t$. Using Proposition A.3, this implies point (2) above.

(a) If $u = v = \mathtt{x}$, then we obviously have $t = \mathtt{x}$ and thus $u \sqsubset\!\!\sqsupset v$.
(b) If $u = \mathtt{C}u'$ and $v = \mathtt{C}v'$, then we necessarily have $t = \mathtt{C}t'$, with $u' \sqsubseteq t'$ and $v' \sqsubseteq t'$, which implies by Lemma A.2 that $u' \frown v'$. By induction, we have $u' \sqsubset\!\!\sqsupset v'$, and thus $\mathtt{C}u' \sqsubset\!\!\sqsupset \mathtt{C}v'$.
   The other cases with $\mathtt{C}^{-}$, $\mathtt{.D}$ and $\mathtt{f}$ are treated similarly.
(c) The case $u = \{\dots; \mathtt{D}_i = u_i; \dots\}$ and $v = \{\dots; \mathtt{D}_i = v_i; \dots\}$ is treated similarly.
(d) If $u = \Omega u'$ and $v = \Omega v'$, we have $\Omega u' \sqsubseteq \mathsf{nf}(\Omega t)$ and $\Omega v' \sqsubseteq \mathsf{nf}(\Omega t)$. It implies that $\mathsf{nf}(\Omega t)$ is of the form $\delta_1\delta_2 t'$ with $u' \sqsubseteq \delta_2 t'$ and $v' \sqsubseteq t'$ (or vice versa).
   This implies that $\Omega u' \sqsubseteq \Omega\delta_2 t'$ and $\Omega v' \sqsubseteq \Omega\delta_2 t'$, and thus that $\Omega u' \sqsubset\!\!\sqsupset \Omega v'$ by induction.
(e) If $v = \Omega v'$, and u not of the form Ω, we have $\Omega v' \leq t$ and $u \leq t$. This implies (with no need of the induction hypothesis) that
(f) No other cases are possible.

□

Appendix B. Basic domain theory

Here are the definitions and basic results on domain theory that are used in this paper. Individual references are given for readers who want additional details / proofs.

Definition B.1.

- If $(O, \leq)$ is a partial order, a subset $X \subset O$ is *directed* if it is non-empty and if every pair of elements of X has an upper bound in X. Important examples of directed sets are increasing chains $x_0 \leq x_1 \leq \dots$.
- A *directed-complete partial order* (*DCPO*) is a partial order $(D, \leq)$ for which every directed set X has a least upper-bound $\bigsqcup^{\uparrow} X$ in D.
- An element k of a DCPO is *compact* if whenever $k \leq \bigsqcup^{\uparrow} X$, then $k \leq x$ for some $x \in X$. Compact elements are usually associated with a notion of "finite approximation".
- A DCPO $(D, \leq, \bigsqcup^{\uparrow})$ is *algebraic* if for all $x \in D$, we have

$$x = \bigsqcup\nolimits^{\uparrow} \{k \mid k \text{ is compact and } k \leq x\}$$

  i.e. if every element is the limit of its finite approximations.

- A function between DCPOs is continuous if it is monotonic and if it commutes with directed least upper bounds.

An important tool in domain theory is the notion of ideal completion.

Definition B.2. An *ideal* for the partial order $(O, \leq)$ is a non empty directed set $I \subset O$ that is downward closed: if $y \in I$ and $x \leq y$ then $x \in I$.

The ideal completion of $(O, \leq)$ is the set of ideals of $(O, \leq)$ ordered by inclusion.

Proposition B.3.

(1) *The ideal completion of a partial order is an algebraic DCPO.* [AJ94, Proposition 2.2.22]
(2) *The ideal completion of the compact elements of an algebraic DCPO* $(D, \leq, \bigsqcup^{\uparrow})$ *is isomorphic to* $(D, \leq, \bigsqcup^{\uparrow})$. [AJ94, Proposition 2.2.25]
(3) *The ideal completion has the following universal property: if* D_1 *and* D_2 *are two DCPOs, then any monotonic function from the compact elements of* D_1 *to* D_2 *can be uniquely extended to a continuous function from* D_1 *to* D_2. [AJ94, Corollary 2.2.26]

The next lemma is straightforward.

Lemma B.4. *If* D_1 *and* D_2 *are DCPOs, then the set of continuous functions from* D_1 *to* D_2 *ordered pointwise is also a DCPO. We write* $[D_1 \to D_2]$.

Appendix C. Smyth power domain

Here are the important facts about the Smyth power domain. More details and proofs of the results given below can be found in [AJ94, Section 6.2.2 and 6.2.3]. If D is an algebraic DCPO, the Smyth power domain can be described in several equivalent ways:

(1) the free algebraic DCPO for the binary operation "+" with
$$x + y = y + x \qquad (x + y) + z = x + (y + z) \qquad x + x = x \qquad x + y \leq x$$
[AJ94, Definition 6.2.7]
(2) the ideal completion of the following order on finite sets of compact elements of D:
$$X \leq Y \qquad \text{iff} \qquad \forall y \in Y, \exists x \in X, x \leq_D y$$
[AJ94, Proposition 6.2.12]
(3) the algebraic DCPO of all compact saturated for the Scott topology on D, ordered by reverse inclusion. [AJ94, Theorem 6.2.14]

The fact that $x + y$ turns out to be the greatest lower bound of x and y ([AJ94, Proposition 6.2.8]) is interesting to note but not important in this paper.

From that, we get the following.

- From point (3): the Smyth power domain of $(D, \leq, \bigsqcup^{\uparrow})$ is a set of formal sums of elements of D.
- From point (1): it contains all the finite sums of elements of D.
- From point (2): it is generated from finite sums of compact elements of D.

In order to show that some infinite sum belongs the Smyth power domain, rather than unfolding the definition of "compact-saturated set for the Scott topology", we can simply show how this infinite sum is a limit of finite sums of compact elements of D.

In practice, most sets from point (3) are infinite, which is difficult to work with. In the Scott topology, a set is saturated precisely when it is upward closed for the order. We thus allow using non saturated sets and instead of reverse inclusion, use the order

$$X \leq_{\text{Smyth}} Y \quad \text{iff} \quad X^{\uparrow} \supseteq Y^{\uparrow}$$

where $X^{\uparrow}$ is the upward closure, i.e. $\{z \mid \exists x \in X x \leq z\}$. Unfolding the definition, we get

$$X \leq_{\text{Smyth}} Y \quad \text{iff} \quad \forall y \in Y, \exists x \in X, x \leq y$$

which will serve as our definition of the order on the Smyth power domain.

To summarize, we have

Corollary C.1. *Given a domain D, the corresponding Smyth power domain is obtained with:*

- *elements are sets of elements of D, seen as formal sums,*
- *two such sets are ordered by $S \leq T$ iff $\forall t \in T, \exists s \in S, s \leq_D t$ (note that is only defines a pre-order),*
- *compact elements are precisely the finite sums of compact elements of D,*
- *all finite sums are in the Smyth power domain,*
- *infinite sums are in the Smyth power domain if they can be obtained as directed limits of finites sums.*