Totality for Mixed Inductive and Coinductive Types

Pierre Hyvernat

TOTALITY 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 introduces an ML / Haskell like programming language with nested inductive and coinductive algebraic datatypes called `chariot`. Functions are defined by arbitrary recursive definitions and can thus lead to non-termination and other "bad" behavior. `chariot` comes with a *totality checker* that tags possibly ill-behaved definitions. Such a totality checker is mandatory in the context of proof assistants based on type theory like Agda.

Proving correctness of this checker is far from trivial and relies on
(1) an interpretation of types as parity games,
(2) an interpretation of correct values as winning strategies for those games,
(3) the Lee, Jones and Ben Amram's size-change principle, used to check that the strategies induced by recursive definitions are winning.

This paper develops the first two points, the last step being the subject of an upcoming paper.

A prototype has been implemented and can be used to experiment with the resulting totality checker, giving a practical argument in favor of this principle.

Introduction

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

```
let rec nats : nat -> nat list
           = fun n -> n::(nats (n+1))
```

is valid but useless in Caml because the evaluation mechanism will loop trying to evaluate it completely (call-by-value evaluation), resulting in a stack overflow exception. In Haskell, because evaluation is lazy (call-by-need), such a definition isn't unfolded until strictly necessary and asking for its third element will only unfold the definition three times. Naively, it seems that types in Caml correspond to "least fixed points" while they correspond to "greatest fixed points" in Haskell.

Received by the editors June 25, 2024.

Key words and phrases: coinductive types; nested fixed points; functional programming; recursive definitions; parity games; circular proofs.

This work was partially funded by by ANR project RECIPROG, project reference ANR-21-CE48-019-01.

The aim of this paper is to introduce a language, called `chariot`,[1] which distinguishes between least and greatest fixed points and where the user can nest them arbitrarily to define new datatypes. To offer a familiar programming experience, definitions are not restricted and any well-typed recursive definition is allowed. In particular, it is possible to write badly behaved definitions like

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

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

(1) Hindley-Milner type-checking [Mil78] to guarantee that evaluation doesn't provoke runtime errors,
(2) a *totality test* to check that the definition is valid with respect to the coinductive / inductive types it involves.

If only inductive types are used, a typed value is total when it is finite, and a function is total if is sends finite values to finite values, i.e. it terminates. In the presence of coinductive types however, values can be infinite. Then, totality roughly means that the only infinite parts of a value come from coinduction. As the examples will show, this quickly becomes very subtle when inductive and coinductive types are interleaved.

The totality test generalizes an older termination checker [Hyv14]. Like before, any definition that passes this test is guaranteed to be correct but because the halting problem is undecidable, some correct definitions are rejected. In a programming context, the programmer may choose to ignore the warning if she (thinks she) knows better. In a proof-assistant context however, it cannot be ignored as non total definitions lead to inconsistencies, the most obvious example being

```
val magic_proof = magic_proof
```

which is non-terminating but belongs to all types. There are subtler examples of definitions that normalize to values but still lead to inconsistencies (Section 1.5).

In Coq [The04], the productivity condition for coinductive definitions is ensured by a strict syntactic condition (guardedness [Coq93]) similar to the condition that inductive definitions need to have one structurally decreasing argument. In Agda [Nor08], the user can write arbitrary recursive definitions and the productivity condition is ensured by the termination checker. Agda's checker extends the termination checker developed by A. Abel [AA02] to deal with coinductive types, but while this is sound for simple types like streams, it is known to be unsound for nested coinductive and inductive types [AD12, Section 5].[2] This paper provides a combinatorial characterization of totality. An upcoming paper will describe how it can be used to implement a totality checker.

Related Works.

[1]All the examples will now be given using the syntax of `chariot` which is described in sections 1.1 and 1.5. They should be readable by anyone with a modicum of experience in functional programming. A prototype implementation in Caml is available from `https://github.com/phyver/chariot` for anyone wishing to experiment with it.

[2]Their counterexample is described in Section 1.5. Simply put, the problem of checking termination and productivity independently is that we cannot distinguish between $\mu_X \nu_Y . F(X,Y)$ and $\nu_Y \mu_X . F(X,Y)$, which have different semantics. Agda's checker is patched to deal with such counter examples but as far as I know, no proof of correctness is available.

Circular proofs. The main inspiration for this work comes from ideas developed by L. Santocanale in his work on circular proofs [San02c, San02a, San02b]. Circular proofs are defined for a linear proof system and are interpreted in categories with products, coproducts and enough initial algebras / terminal coalgebras. In fine, the criterion implemented in `chariot` uses a strong combinatorial principle (the size-change principle) to check a sanity condition on a kind of circular proof (a program). This is strictly stronger than the initial criterion used by L. Santocanale and G. Fortier, which corresponds to the syntactical structurally decreasing / guardedness condition on recursive definitions.

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

More recent works in circular proof theory replace L. Santocanale's criterion by a much stronger combinatorial condition [Dou17b, Dou17a]. It involves checking that some infinite words are recognized by a parity automata, which is a decidable problem. The presence of parity automata points to a relation between this work and the present paper, but the different contexts make it all but obvious.

Size-change principle. The second idea will be developed in an upcoming paper and consists of adapting the *size-change principle* (SCP) from C. S. Lee, N. D. Jones and A. M. Ben-Amram [LJBA01] to the task of checking general totality. This problem is subtle as totality is strictly more than termination and productivity. Moreover, while the principle used to check termination of ML-like recursive definitions [Hyv14] was inherently untyped, totality checking needs to be somewhat type aware. For example, in `chariot`, records are lazy and are used to define coinductive types. The definition

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

yields an infinite binary tree and depending on the types of `Node`, `Left` and `Right`, the definition may be correct or incorrect (c.f. page 13)!

Charity. The closest ancestor to `chariot` is the language `charity`[3] [CF92, Coc96], developed by R. Cockett and T. Fukushima. It lets the programmer define types with arbitrary nesting of induction and coinduction. Values in these types are defined using categorical principles.

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

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

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

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

Sized types. This approach extends type theory with a notion of "size" that annotate types. It has been successful and is implemented in Agda [Abe10, Abe12]. It is possible to specify that the `map` function on list has type $\forall n, \texttt{list}^n(T) \to \texttt{list}^n(T)$, where $\texttt{list}^n(T)$ is the type of lists with n elements of type T. These extra parameters give information about recursive functions and make it easier to check termination. A drawback is that functions on sized-types must take extra size parameters. This complexity is balanced by the fact that most of them can be inferred automatically and are thus mostly the libraries' implementors job: in many cases, sizes are invisible to the casual user. Note however that sizes only help showing termination and productivity. Developing a totality checker is orthogonal to designing an appropriate notion of size and the totality checker described in this paper can probably work hand in hand with standard size notions.

Fixed points in game semantics. An important tool in this paper is the notion of *parity game*. P. Clairambault [Cla13] explored a category of games enriched with winning conditions for infinite plays. The way the winning condition is defined for least and greatest fixed points is reminiscent of L. Santocanale's work on circular proofs and the corresponding category is cartesian closed. Because this work is done in a more complex setting and aims for generality, it seems difficult to extract a practical test for totality from it. The present paper aims for specificity and practicality by devising a totality test for the usual semantics of recursion.

SubML. C. Raffalli and R. Lepigre used the size-change principle to check correctness of recursive definitions in the language SubML [LR18]. Their approach uses a powerful but non-standard type theory with many features: subtyping, polymorphism, sized-types, control operators, some kind of dependent types, etc. On the downside, it makes their type theory more difficult to compare with other approaches. Note that like in Agda or `chariot`, they do allow arbitrary definitions that are checked by an incomplete totality checker. One interesting point of their work is that the size-change termination is only used to check that some object (a proof tree) is well-founded: even coinductive types are justified with well-founded proofs.

Plan of the Paper. We start by introducing the language `chariot` and its denotational semantics in Section 1. We assume the reader is familiar with functional programming, recursive definitions and their semantics, Hindley-Milner type checking, algebraic datatypes, pattern matching, etc. The notion of totality is also given there. We then describe, in Section 2, a combinatorial approach to totality that comes from L. Santocanale's work on circular proofs. This reduces checking totality of a definition to checking that the definitions gives a winning strategy in a parity game associated to the type of the definition.

The actual totality checker using the size-change principle will be described in an upcoming paper [Hyv].

1. The Language and its Semantics

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

- an inductive type comes with a list of *constructors* whose *codomain* is the type being defined,
- a coinductive type comes with a list of *destructors* whose *domain* is the type being defined.

Datatypes are introduced by the keywords "`data`" or "`codata`" and may have parameters. Type parameters are written with a quote as in Caml. Here are some examples:

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

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

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

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

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

Examples will sometimes use shortcuts, allowed in the implementation, and write `Zero` (instead of `Zero{}`) or `Cons(x,xs)` (instead of `Cons{Fst=x;Snd=xs}`).

Formally:

Definition 1.1. the sets $\mathcal{D}_k$ of *definitions of rank k* and $\mathcal{T}_k(P_1, \ldots, P_n)$ of *type expressions of rank k with parameters among* $P_1, \ldots, P_n$ are defined by mutual induction:

(1) For $k > 0$, the set $\mathcal{D}_k$ contains all definitions of the form

```
data S('x, 'y, ...) where                    and    codata S('x, 'y, ...) where
    | C₁ : T₁ -> S('x, 'y, ...)                         | D₁ : S('x, 'y, ...) -> T₁
    ...                                                 ...
    | Cₖ : Tₖ -> S('x, 'y, ...)                         | Dₖ : S('x, 'y, ...) -> Tₖ
```

where

- `S` is the name of the type being defined,

- each $\mathtt{C}_i$ is the name of a constructor,
- each $\mathtt{D}_i$ is the name of a destructor,
- each T_i is an element of $\mathcal{T}_{k-1}(\mathtt{S('x,\ 'y,\ ...)}, \mathtt{'x}, \mathtt{'y}, \dots)$.

(2) The set $\mathcal{T}_k(P_1, \dots, P_n)$ where the P_i are syntactical parameters is defined by the grammar

$$T \in \mathcal{T}_k(P_1, \dots, P_n) \qquad ::= \qquad P_i \quad | \quad \mathtt{T}(T_1, \dots, T_m)$$

where
- $\mathtt{T}$ is the name of a type in some $\mathcal{D}_{k'}$ of arity m, with $k' \leq k$,
- each T_i belongs to $\mathcal{T}_k(P_1, \dots, P_n)$.

(3) The rank of a definition [resp. type expression] is the smallest k such that the definition is in $\mathcal{D}_k$ [resp. the type expression is in some $\mathcal{T}_k(\dots)$].

(4) The head rank of a type expression is the rank of the definition of its head type constructor; or 0 if the type expression is a parameter.

Strictly speaking, each name `T` is indexed by its definition and each constructor / destructor name is indexed by its corresponding type name. In practice, `chariot` forbids the reuse of names so that there is no ambiguity about which definition defines a given type, or about which type corresponds to a given constructor / destructor.[4]

Note that to make the theory slightly simpler, constructors always have a single argument. Of course, the implementation of `chariot` allows constructors of arbitrary arity and the theory can be extended to deal with this.

Destructors act as projections and because of the universal property of terminal coalgebras, we think about elements of a codatatype as records. This is reflected in the syntax of terms. For example, the following defines (recursively) the stream with infinitely many 0s. (The syntax for recursive definitions will be formally given in Definition 1.9.)

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

The denotational semantics of a codata is going to be an *coinductive* type (greatest fixed-point) while the semantics of a data is going to be an *inductive* type (least fixed-point). In order to have a sound operational semantics, codata should not be fully evaluated. The easiest way to ensure that is to stop evaluation on records: evaluating "`zeros`" will result in "`{Head = □; Tail = □}`" where the "□" are non evaluated chunks. The copattern view [APTS13] is natural here. The definition of `zeros` using copatterns (allowed in `chariot`) looks like

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

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

We will use the following conventions:

- outside of actual type definitions (given using `chariot`'s syntax), type parameters will be written without quote: `x`, $\mathtt{x}_1$, ...

[4]Extending the definition to mutual type definitions is left as an exercise for the diligent reader! It is straightforward if one keeps in mind that all types in a mutual definition are `data` or `codata` and that they all have the same parameters "`('x, 'y, ...)`".

- an unknown datatype will be written $\mathtt{T}_\mu(\mathtt{x}_1, \ldots, \mathtt{x}_k)$ and an unknown codatatype will be written $\mathtt{T}_\nu(\mathtt{x}_1, \ldots, \mathtt{x}_k)$,
- an unknown type of unspecified polarity will be written $\mathtt{T}(\mathtt{x}_1, \ldots, \mathtt{x}_k)$.

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

Definition 1.2.

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

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

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

(2) If the X_i are ordered sets, the order on $\mathcal{V}(X_1, \ldots, X_n)$ is generated by
- $\bot \leq v$ for all values v,
- if $x \leq x'$ in X_i, then $x \leq x'$ in $\mathcal{V}(X_1, \ldots, X_n)$,
- "$\leq$" is contextual: if $u \leq v$ then $C[\mathtt{x} := u] \leq C[\mathtt{x} := v]$ for any value C containing a formal variable $\mathtt{x}$ (substitution is defined in the obvious way).

The set of values (first point in Definition 1.2) is defined coinductively but the order on values (second point in Definition 1.2) is defined inductively. Reasoning about about the order is usually done using simple inductive proofs.

1.3. **Semantics in Domains.** There is a natural interpretation of types in the category of algebraic DCPOs where morphisms are continuous functions that are *not* required to preserve the least element. An algebraic DCPO is an order with the following properties:

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

Unless specified otherwise, "domain" will always refer to an algebraic DCPO. Recall that any partial order can be completed to a DCPO whose compact elements are exactly the element of the partial order. This *ideal completion* formally adds limits of all directed sets. The following can be proved directly but is also a direct consequence of this general construction.

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

Given a type definition of arity n in $\mathcal{D}_k$, its semantics sends the sets $X_1, \ldots, X_n$ to a subset of values in $\mathcal{V}(X_1, \ldots, X_n)$.

Definition 1.4. We define, by mutual induction, the interpretation of type definitions and type expressions:

- Let $\mathtt{T} \in \mathcal{D}_k$ be a type definition of arity n. The interpretation of $\mathtt{T}$, written $[\![\mathtt{T}]\!]$, sends any sequence $\overline{X} = X_1, \ldots, X_n$ of n sets to a subset of $\mathcal{V}(\overline{X})$. It is defined *coinductively* by the following "typing" rules:

  (1) $\dfrac{}{\bot : [\![\mathtt{T}]\!]\,(\overline{X})}$,

  (2) $\dfrac{u \in X_i}{u : [\![\mathtt{T}]\!]\,(\overline{X})}$,

  (3) $\dfrac{u : [\![S]\!]\,\big([\![\mathtt{T}_\mu]\!]\,(\overline{X}), \overline{X}\big)}{\mathtt{C}\,u : [\![\mathtt{T}_\mu]\!]\,(\overline{X})}$ when $\mathtt{C} : S \to \mathtt{T}_\mu(\overline{\mathtt{x}})$ is a constructor of $\mathtt{T}_\mu$,[5]

  (4) $\dfrac{u_1 : [\![S_1]\!]\,\big([\![\mathtt{T}_\nu]\!]\,(\overline{X}), \overline{X}\big) \quad \ldots \quad u_k : [\![S_k]\!]\,\big([\![\mathtt{T}_\nu]\!]\,(\overline{X}), \overline{X}\big)}{\{\mathtt{D}_1 = u_1;\ \ldots;\ \mathtt{D}_k = u_k\} : [\![\mathtt{T}_\nu]\!]\,(\overline{X})}$ if the destructors of $\mathtt{T}_\nu$ are exactly the $\mathtt{D}_i : \mathtt{T}_\nu(\overline{\mathtt{x}}) \to S_i$, for $1 \leq i \leq k$.

- the interpretation of a type expression $T \in \mathcal{T}_k(\overline{\mathtt{x}})$ is defined by induction:
  – if $T = \mathtt{x}_i$, then $[\![T]\!]\,(\overline{X}) = X_i$,
  – if $T = \mathtt{T}(S_1, \ldots, S_m)$ then $[\![T]\!]\,(\overline{X}) = [\![\mathtt{T}]\!]\,\big([\![S_1]\!]\,(\overline{X}), \ldots, [\![S_m]\!]\,(\overline{X})\big)$.

The following is easily proved by induction on T.

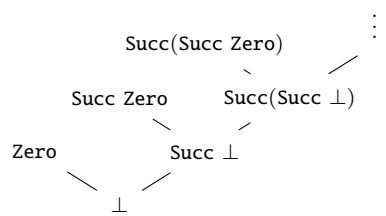
Proposition 1.5. *Let $\overline{X} = X_1, \ldots, X_n$ be domains, if $\mathtt{T}$ is a type then*
(1) *with the order inherited from $\overline{X}$ (point (2) in Definition 1.2), $[\![\mathtt{T}]\!]\,(\overline{X})$ is a domain,*
(2) *$\overline{X} \mapsto [\![\mathtt{T}]\!]\,(\overline{X})$ is functorial.*
(3) *for a datatype $\mathtt{T}_\mu(\overline{\mathtt{x}})$ with constructors $\mathtt{C}_i : S_i \to \mathtt{T}_\mu(\overline{\mathtt{x}})$, we have*

$$\begin{aligned}[\![\mathtt{T}_\mu]\!]\,(\overline{X}) &= \{\bot\} \cup \bigcup_{i=1,\ldots,k} \Big\{\mathtt{C}_i u_i \mid u_i \in [\![S_i]\!]\,\big([\![\mathtt{T}_\mu]\!]\,(\overline{X}), \overline{X}\big)\Big\} \\ &\cong \Big([\![S_1]\!]\,\big([\![\mathtt{T}_\mu]\!]\,(\overline{X}), \overline{X}\big) + \cdots + [\![S_k]\!]\,\big([\![\mathtt{T}_\mu]\!]\,(\overline{X}), \overline{X}\big)\Big)_\bot\end{aligned}$$

(4) *for a codatatype $\mathtt{T}_\nu(\overline{\mathtt{x}})$ with destructors $\mathtt{D}_i : S_i \to \mathtt{T}_\nu(\overline{\mathtt{x}})$, we have*

$$\begin{aligned}[\![\mathtt{T}_\nu]\!]\,(\overline{X}) &= \{\bot\} \cup \Big\{\{\ldots; \mathtt{D}_i = u_i; \ldots\} \mid i = 1, \ldots, k \text{ and } u_i \in [\![S_i]\!]\,\big([\![\mathtt{T}_\nu]\!]\,(\overline{X}), \overline{X}\big)\Big\} \\ &\cong \Big([\![S_1]\!]\,\big([\![\mathtt{T}_\nu]\!]\,(\overline{X}), \overline{X}\big) \times \cdots \times [\![S_k]\!]\,\big([\![\mathtt{T}_\nu]\!]\,(\overline{X}), \overline{X}\big)\Big)_\bot\end{aligned}$$

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



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

[5]There and in point (4), the interpretation $[\![T]\!]$ is already defined as its rank is strictly smaller than k.

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

1.4. **Semantics in Domains with Totality.** We use domains to be able to use Kleene's formula to define the interpretation of recursive definitions (next section). Unfortunately, those domain cannot distinguish greatest and least fixed point: the functors defined by types are *algebraically compact* [Bar92], i.e. their initial algebras and terminal coalgebras are isomorphic. For example, the interpretation of $[\![\mathtt{nat}]\!]$ automatically contains the infinite value $\mathtt{Succ}(\mathtt{Succ}(\mathtt{Succ}(\dots)))$: it is the limit of $\bot \leq \mathtt{Succ}\,\bot \leq \mathtt{Succ}(\mathtt{Succ}\,\bot) \leq \cdots$. We thus add a notion of *totality*[6] on top of the domains.

Definition 1.6.

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

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

Definition 1.7. Totality is defined by mutual induction on type definitions and type expressions.

- If $\mathtt{T} \in \mathcal{D}_k$ is a type definition of arity n, and if $\overline{X}$ is a sequence of n sets,
  - totality for a datatype $\mathtt{T}_\mu$ is defined with (μ is the least fixed point operator)
$$|T|(\overline{X}) \quad = \quad \mu X \,.\, \bigcup_{i=1,\dots,k} \Big\{ \mathtt{C}_i u \;\Big|\; u \in |S_i|(X, \overline{X}) \Big\}$$
    where $\mathtt{C}_i : S_i \to T$ for $i = 1, \dots, k$ are the constructors for $\mathtt{T}_\mu$,
  - totality for a codata type $\mathtt{T}_\nu$ is defined with (ν is the greatest fixed point operator)
$$|T|(\overline{X}) \quad = \quad \nu X. \Big\{ \{\mathtt{D}_1 = u_1; \dots; \mathtt{D}_k = u_k\} \;\Big|\; i = 1, \dots, k \text{ and } u_i \in |S_i|(X, \overline{X}) \Big\}$$
    where $\mathtt{D}_i : T \to S_i$, $i = 1, \dots, k$ are the destructors for $\mathtt{T}_\nu$.
- If T is a type expression,
  - If $T = \mathtt{x}_i$ then $|T|(\overline{X}) = X_i$
  - If $T = \mathtt{T}(\overline{S})$, then $|T|(\overline{X}) = |\mathtt{T}|\big(|S_1|(\overline{X}), \dots, |S_k|(\overline{X})\big)$.

Because these operators act on subsets of the set of all values and are monotonic, the least and greatest fixed points exist by the Knaster-Tarski theorem.

Lemma 1.8. *If $T \in \mathcal{T}_k()$ is a closed type expression, then $\big([\![T]\!], |T|\big)$ is a domain with totality. Moreover, each $t \in |T|$ is maximal in $[\![T]\!]$.*

Proof. Let $v \in |T|(\overline{X})$, we show that we can (coinductively) type it using the rules from Definition 1.4.

[6] Our notion seems unrelated to intrinsic notions of totality that exist in effective domain theory. [Ber93]

- If $T = \mathtt{x}_i$, then $v \in |T|(\overline{X}) = X_i$, so that we can use rule (2) to show that $v \in [\![T]\!](\overline{X})$.
- If $T = \mathtt{T}_\mu(S_1, \dots)$, then $|T|(\overline{X}) = \mu X. \bigcup_i \{\mathtt{C}_i u \mid u \in |S_i|(X, \overline{X})\}$, so that v is of the form $\mathtt{C}_i u$ with $u \in |S_i|(\mathtt{T}_\mu(\overline{T}), \overline{X})$. We can use rule (3) and continue coinductively.
- If $T = \mathtt{T}_\nu(S_1, \dots)$, then $|T|(\overline{X}) = \nu X. \{\mathtt{\{D}_1 = u_1; \dots; \mathtt{D}_k = u_k\mathtt{\}} \mid \forall i, u_i \in |S_i|(X, \overline{X})\}$ so that v is of the form $\mathtt{\{D}_1 = u_1; \dots; \mathtt{D}_k = u_k\mathtt{\}}$ where each $u_i \in |S_i|(|T|(\overline{X}), \overline{X})$. We can use rule (4) and continue coinductively.

Note that we don't use the fact that $|\mathtt{T}_\mu|$ is a least fixed point or that $|\mathtt{T}_\nu|$ is a greatest fixed point.

To show that elements of $|T|$ are maximal, we need to show "if $u \leq v$ and u doesn't contain $\bot$, then $u = v$". This is a trivial proof by induction on $u \leq v$. □

1.5. **Recursive Definitions.** The approach described in this paper is first order: we are only interested in the way values in (co)datatypes are constructed and destructed. Higher order parameters are allowed in the implementation but they are ignored by the totality checker. Some examples in the paper will use such higher order parameters but for simplicity's sake, they are not formalized.[7] Like in Haskell, recursive definitions are given by lists of clauses. Here are two examples: the Ackermann function (using some syntactic sugar for the constructors `Zero` and `Succ`)

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

and the `map` function on streams:

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

Definition 1.9. A recursive definition is introduced by the keyword `val` and consists of a finite list of clauses of the form

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

where

- `f` is one of the function names being mutually defined,
- each p_i is a *finite* pattern

$$p \quad ::= \quad \mathtt{x}_i \quad | \quad \mathtt{C}\ p \quad | \quad \mathtt{\{D}_1 = p_1; \dots; \mathtt{D}_k = p_k\mathtt{\}}$$

  where each $\mathtt{x}_i$ is a variable name,
- and u is a *finite* term

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

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

[7] Note that we can't simply ignore higher order parameters as they can hide some recursive calls:

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

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

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

We assume the definitions are validated using standard Hindley-Milner type inference / type checking . This includes in particular checking that clauses of the definition cover all values of the appropriate type and that no record is missing any field. Those steps are not described here [PJ87, e.g. Section 8 and 9].

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

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

where the semantics of types is extended with $[\![T \to T']\!] = \big[\, [\![T]\!] \to [\![T']\!] \,\big]$. The semantics of $\mathtt{f}_1, \ldots, \mathtt{f}_n$ is then defined as the fixed point of the operator $\Theta^{\mathrm{std}}_{\mathtt{f}_1,\ldots,\mathtt{f}_n}$ which exists by Kleene theorem.[8] Let's describe more precisely the standard semantics of the definition in the simple case of a single recursive function $\mathtt{f}$ taking a single argument. Given an environment ρ for functions other than $\mathtt{f}$, the recursive definition for $\mathtt{f} : A \to B$ gives rise to an operator $\Theta^{\mathrm{std}}_{\rho,\mathtt{f}}$ on $[[\![A]\!] \to [\![B]\!]]$ called the "standard semantics". Its fixed point is the semantics of $\mathtt{f}$, written $[\![\mathtt{f}]\!]_\rho : [\![A]\!] \to [\![B]\!]$. The operator $\Theta^{\mathrm{std}}_{\rho,\mathtt{f}}$ is defined as follows.

Definition 1.10.

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

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

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

An important property of Hindley-Milner type checking is that it ensures a definition has a well defined semantics. In particular, there always is a matching clause and the value "$\bot$" corresponds to non-termination, not to failure of the evaluation mechanism (like projecting on a non-existing field). However, it doesn't mean the definition is correct from a denotational point of view. For that, we need it to be *total* with respect to its type. For example, the definition

```
val all_nats : nat -> list(nat)
  | all_nats n = Cons n (all_nats (n+1))
```

[8]The fixed point exists, but since domains contain the $\bot$ value, the result can be a partial function.

is well typed and sends elements of the domain $[\![\texttt{nat}]\!]$ to the domain $[\![\texttt{list(nat)}]\!]$ but the image of `Zero` is the infinite list containing all the natural numbers. This is not total because totality for $[\![\texttt{list(nat)}]\!]$ contains only the finite lists. Similarly, the definition

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

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

A note on projections. The syntax of definitions given in Definition 1.9 doesn't allow projecting a record on one of its field. This makes the theory somewhat simpler and doesn't change expressivity of the language because it is always possible to rewrite a projection using one of the following tricks:

- remove a projection on a previously defined function by introducing another function, as in

  ```
  | f x = ... (g u).Fst ...
  ```

  being replaced by

  ```
  | f x = ... projectFst (g u) ...
  ```

  where `projectFst` is defined with

  ```
  val projectFst { Fst = x; Snd = y } = x
  ```

- remove a projection on a variable by extending the pattern on the left, as in

  ```
  | f x = ... x.Head ...
  ```

  being replaced by

  ```
  | f { Head = h; Tail = t } = ... h ...
  ```

- remove a projection on the result of a recursively defined function by splitting the function into several mutually recursive functions, as in

  ```
  | f : prod(A, B) -> prod(A, B)
  | f p = ... (f u).Fst ...
  ```

  being replaced by

  ```
  | f1 : prod(A, B) -> A
  | f1 x = ... (f1 u1) ...
  | f2 : prod(A, B) -> B
  ...
  ```

The first point is the simplest and most general but shouldn't be used to remove projections on variables or recursive functions. Since the checker will see each external function as a black box about which nothing is known, introducing external functions in a recursive definition can hide information and makes totality checking much less powerful. Of course, the implementation of `chariot` doesn't enforce this restriction and the theory can be modified accordingly.

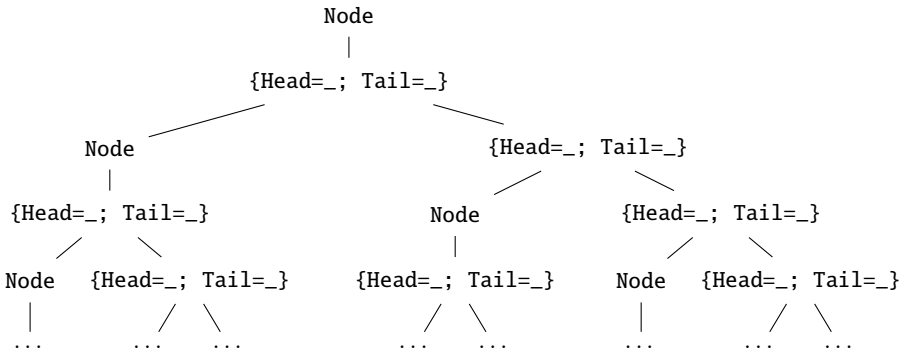
A subtle example. Here is an example showing that productivity and termination are not enough to check validity of a recursive definition [AD12]. We define the inductive type

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

where the type of `stream` was defined on page 5. This type is similar to the usual type of "Rose trees", but with streams instead of lists. Because streams cannot be empty, there is no way to build such a tree inductively: this type has no total value. Consider however the following definitions:

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

This is well typed and productive. Lazy evaluation of `bad_t` or any of its subterms terminates. The semantics of `bad_t` doesn't contain $\bot$ and unfolding the definition gives



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

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

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

2. Combinatorial Description of Totality

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

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

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

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

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

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

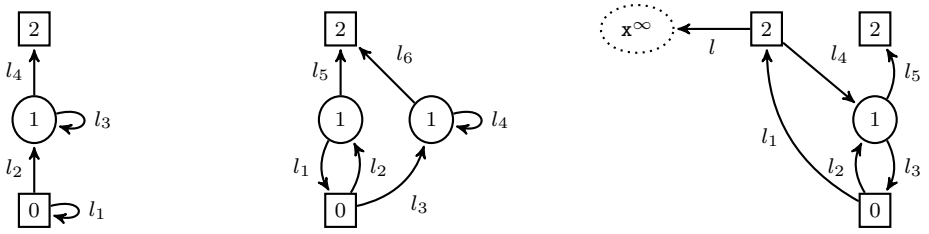
In this case, the value `inf` is of type `option(tree)`.

2.1. **Parity Games.** A parity games is a two players game played on a finite transition system where each node is labeled by a natural number called its *priority*. When the node has odd priority, *Marie* (or "μ", or "player") is required to play. When the node is even,[9] *Nicole* (or "ν", or "opponent") is required to play. A move is simply a choice of a transition from the current node and the game continues from the new node. When Nicole (or Marie) cannot move because there is no outgoing transition from the current node, she looses. In case of infinite play, the winning condition is

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

We will call a priority *principal* if "it is maximal among the priorities appearing infinitely often". The winning condition can thus be rephrased as "Marie wins an infinite play if and only if the principal priority of the play is even".

In order to analyze types with parameters, we add parameter nodes $\mathtt{x}_1$, $\mathtt{x}_2$, ... to the games. Those nodes have no outgoing transition and their parity is, by convention, ∞. On reaching them, Marie will be required to choose an element of some corresponding set X to finish the game. She'll win if she can do it and loose if the set is empty. Here are three examples of parity games:



Definition 2.1. Each position p in a parity game G with parameters $\mathtt{x}_1$, ..., $\mathtt{x}_n$ defines a functor $||G||_p$ from Set^n to Set [San02c]. $||G||_p(\overline{X})$ is defined by induction on the maximal finite priority of G and the number of positions with this priority:

- if all the positions are parameters, each position is interpreted by the corresponding parameter $||G||_{\mathtt{x}_i}(\overline{X}) = X_i$;
- otherwise, take p to be one of the positions of maximal priority and construct G/p with a new parameter $\mathtt{x}_0$ as follows: it is identical to G, except that position p is replaced by $\mathtt{x}_0$ and all its outgoing transitions are removed.[10] We define
  – if p had an odd priority,

$$||G||_p(\overline{X}) = \mu X.\big(||G/p||_{q_1}(X, \overline{X}) + \cdots + ||G/p||_{q_k}(X, \overline{X})\big)$$

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

[9] Assigning odd to one player and even to the other is just a convention.
[10] This game is called the predecessor of G [San02c].

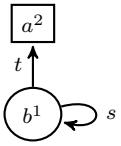
– if p had an even priority,

$$||G||_p(\overline{X}) = \nu X.\big(||G/p||_{q_1}(X, \overline{X}) \times \cdots \times ||G/p||_{q_k}(X, \overline{X})\big)$$

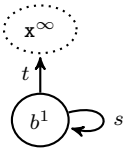
where $p \to q_1, \ldots p \to q_k$ are all the transitions out of p.
– when $p \neq q$,

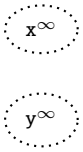
$$||G||_q(\overline{X}) = ||G/p||_q\big(\overline{X}, ||G||_p(\overline{X})\big)$$

Here is a small example to illustrate this construction. Consider the following parity game:

$G \quad = \quad$ 

To compute $||G||_a$ and $||G||_b$, we need to compute $||G/a||$, and thus $||G/a/b||$, given by:

$G/a \quad = \quad$ 

$G/a/b \quad = \quad$ 

By definition, $||G/a/b||_{\mathtt{y}}(X, Y) = Y$ and $||G/a/b||_{\mathtt{x}}(X, Y) = X$. We thus get the following

- $B(X) := ||G/a||_b(X) = \mu_Y.\big(||G/a/b||_{\mathtt{x}}(X, Y) + ||G/a/b||_Y(X, Y)\big) = \mu_Y.(X + Y)$,
- $||G/a||_{\mathtt{x}}(X) = ||G/a/b||_{\mathtt{x}}(X, B(X)) = X$.

From that, we obtain

- $A := ||G||_a = \nu_X.(\text{“empty product”})$ as there is no outgoing transition from a. This set is isomorphic to $\mathbf{1}$, the one element set.
- $||G||_b = ||G/a||_b(A) = B(A) = \mu_Y.(Y + \mathbf{1}) \cong \mathbf{N}$. This set is isomorphic to the natural numbers.

There is a strong link between the set $||G||_p$ and the set of *winning strategies* for Marie in game G with initial position p.

Definition 2.2 (Winning strategies)**.** Suppose G is a parity game with parameters $\mathtt{x}_1, \ldots, \mathtt{x}_n$ and p a position in G. Let $\overline{X} = X_1, \ldots, X_n$ be sets.

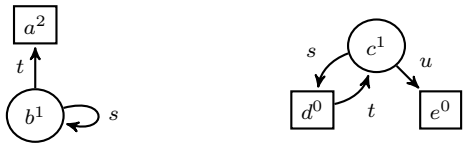
(1) Write $\mathcal{P}_p$ for the set of *finite paths* starting from p, equipped with the prefix order $\sqsubseteq$.
(2) A subset $S \subseteq \mathcal{P}_p$ is a *strategy* if:
   (a) it is downward closed: $\sigma_1 \sqsubseteq \sigma_2 \in S \implies \sigma_1 \in S$,
   (b) it is deterministic on odd positions: if the last position reached by $\sigma \in S$ has odd priority, there is a unique transition l such that $\sigma \cdot l \in S$,
   (c) it is complete on even positions: if the last position reached by $\sigma \in S$ has even priority, for all transition l, the path $\sigma \cdot l$ is in S,
   (d) it is defined on parameters: if the last position reached by $\sigma \in S$ is parameter $\mathtt{x}_i$ (i.e. it has priority ∞), then there is an element $x \in X_i$ such that $\sigma \cdot x$ is in S.

   Note that the “last position reached by σ” is taken to be p if σ is the empty sequence. Each σ in a strategy is a sequence of transitions, *except possibly for the last element of a finite sequence*, which can be an element of a parameter by point(d).

(3) A strategy S is *winning* if all infinite branches of S are winning: for any infinite branch σ, the position of maximal priority that is visited infinitely often by σ is even.[11]

We write $\mathcal{W}(G)_p(\overline{X})$ for the set of such winning strategies for G, from p, with parameters $\overline{X}$.

For examples, strategies from position b in the left game



consists of all finite strategies $(s^n t)^{\downarrow} = \{\varepsilon, s, ss, \dots, s^n, s^n t\}$ together with the infinite strategy $s^{\infty\downarrow} = \{\varepsilon, s, ss, \dots\}$. The finite strategies are obviously winning but the infinite strategy isn't.

There is only one infinite strategy for the right game from position c: $(st)^{\infty\downarrow} = \{\varepsilon, s, st, sts, stst, ststs, \dots\}$, which is winning. The finite strategies are all the $((st)^n u)^{\downarrow}$.

An important result is:

Proposition 2.3 ([San02c, Theorem 5.4])**.** *There is a natural isomorphism*

$$||G||_p \cong \mathcal{W}(G)_p \ .$$

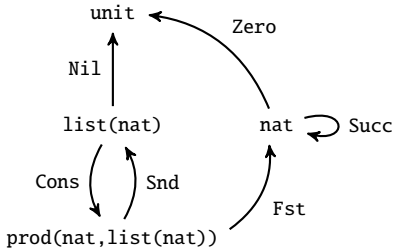
2.2. **Parity Games from Types.** We can construct a parity game G from any type T in such a way that $|T| \cong ||G||_T$, for some distinguished position T in G.

Definition 2.4.

(1) Given a (fixed) list of type definitions, we consider the following transition system:
- nodes are type expressions, possibly with parameters,
- transitions are labeled by constructors and destructors: a transitions $T_1 \xrightarrow{t} T_2$ is either a destructor t of type $T_1 \to T_2$ or a constructor t of type $T_2 \to T_1$ (note the reversal).

(2) If T is a type expression, the *graph of* T is defined as the part of the above transition system that is reachable from T.

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



The transition system is set up so that

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

[11] A branch in S is an increasing sequence of elements of S. It can therefore be infinite even though all its elements are themselves finite.

Because of that, Hindley-Milner type checking will ensure that a value of type T gives a strategy for a game on the graph of T where Marie (the player) chooses constructors and Nicole (the opponent) chooses destructors (that will be Lemma 2.10). We will, in Definition 2.7, add priorities so that

- datatype nodes are odd and codatatype nodes are even,
- the order of priorities correspond in a precise way to the interleaving of least and greatest fixed points.

Checking that this strategy is *winning* will be the goal of the totality checker.

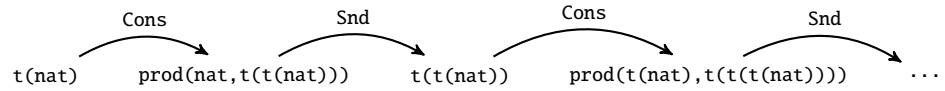
Note that when some of the types have parameters, the transition system is infinite: it will for example contain `list('x)`, `list(list('x))`, `list(list(list('x)))`, etc. However, we have

Lemma 2.5. *For any type T, the graph of T is finite.*

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

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

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



Before proving the lemma, the following definition will be useful.

Definition 2.6. Write $T_1 \sqsubseteq T_2$ if T_1 is a subexpression of T_2. More precisely:

- $T \sqsubseteq X$ iff $T = X$,
- $T \sqsubseteq \mathtt{T}(T_1, \ldots, T_n)$ if and only if $T = \mathtt{T}(T_1, \ldots, T_n)$ or $T \sqsubseteq T_1$ or $\ldots$ or $T \sqsubseteq T_n$.

Proof of Lemma 2.5. From Definition 1.1, we know that the definition of a (co)datatype only uses parameters and type expressions of strictly smaller rank. Moreover, in the presence of mutual type definitions, two types of the same order are part of the same mutual definition.

Suppose by contradiction that T is a type expression of minimal head rank (c.f. Definition 1.1) with an infinite graph. Since the graph of T has bounded out-degree, König's lemma implies it contains an infinite path $\rho = T \to T_1 \to T_2 \to \cdots$ without repeated vertex.

There must be infinitely many T_i with head rank equal to κ. Otherwise, we could construct an infinite path with a strictly smaller head rank: take the first T_n s.t. all later T_m have head rank $\lambda < \kappa$ and let T'_m, for $m > n$ be obtained from T_m by replacing all subexpressions of rank κ by a new parameter $\mathtt{x}$.

Because no T_m has a head rank κ, any transition $T_m \to T_{m+1}$ is also a transition $T'_m \to T'_{m+1}$ and we obtain an infinite path $T'_n \to T'_{n+1} \to \cdots$ with rank strictly smaller than κ.

But if T has rank κ, its subexpressions of head rank κ have fixed arguments by uniformity. So all subexpressions of head rank κ along the infinite path must be subexpressions of T_0. There are only finitely many of those, contradicting the previous remark. □

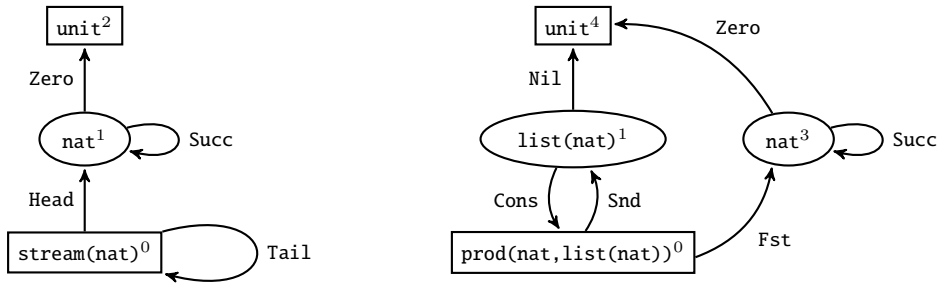
Definition 2.7. If T is a type expression, a *parity game for T* is a parity game on the graph of T (Definition 2.4) which satisfies the following conditions:

(1) if T_0 is a datatype, its priority is odd,
(2) if T_0 is a codatatype, its priority is even,
(3) if $T_1 \sqsubseteq T_2$, then the priority of T_1 is greater than the priority of T_2.

Lemma 2.8. *Each type expression has a parity game.*

Proof. The relation $\sqsubseteq$ is a strict order and doesn't contain cycles. Its restriction to the graph of T can be linearized. This gives the relative priorities of the nodes and ensures condition (5) from the definition. Starting from the least priorities, we can now choose a priority odd / even compatible with this linearization. □

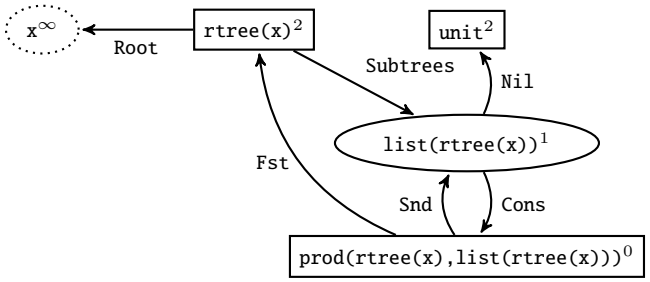
We don't actually need to linearize the graph and can instead chose a *normalized parity game*, i.e. one that minimizes gaps in priorities. Here are the first two parity games from page 14, seen as parity games for `stream(nat)` and `list(nat)`. Priorities are written as exponents and their parity can be seen in the shape (square or round) of nodes.



The last example from page 14 corresponds to a coinductive version of Rose trees:

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

with parity game



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

The semantics of a parity game (Definition 2.1) and that of the totality semantics of a type (Definition 1.7) are similar in that they interleave greatest and least fixed points. Parity games of types are designed to get the following.

Proposition 2.9. *For any type parity game G and for any node T in G, we have $||G||_T \cong |T|$.*

Proof. We prove "*for any type expression T and parity game G containing a parity game for T as a simply connected component, we have $||G||_T \cong |T|$*" by induction on G.

To simplify the argument, we assume that all non-recursive constructors [resp. destructors] for type $\mathtt{T}(\overline{\mathtt{x}})$ are of type $\mathtt{x}_i \to \mathtt{T}(\overline{\mathtt{x}})$ [resp. $\mathtt{T}(\overline{\mathtt{x}}) \to \mathtt{x}_i$]. Every type definition can be transformed into one such by adding additional parameters. For example, we can replace `list(x)` by

```
data list2('x, 'y) where  Nil  : 'y                      -> list2('x, 'y)
                        | Cons : prod('x, list2('x, 'y) -> list2('x, 'y)
```

and the old "`list(x)`" is the same as the new "`list2(x, unit)`": it has the same semantics, totality and parity game.

- The result is obvious when all the nodes of G have priority ∞: T is necessarily a parameter $\mathtt{x}_i$ and we have $||G||_{\mathtt{x}_i}(\overline{X}) = X_i = |\mathtt{x}_i|$.
- If T is a position of maximal finite priority in G and $T = \mathtt{T}_\mu(\overline{T})$ starts with a datatype with constructors $\mathtt{C}_i : S_i \to T$, $i = 1, \ldots, k$. We have

$$\begin{array}{rcll} ||G||_T(\overline{X}) & = & \mu X.\big(||G/T||_{S_1}(X, \overline{X}) + \cdots + ||G/T||_{S_k}(X, \overline{X})\big) & \text{(Definition 2.1)} \\ & \cong & \mu X.\big(|S_1|(X, \overline{X}) + \cdots + |S_k|(X, \overline{X})\big) & \text{(induction hypothesis)} \\ & \cong & \mu X.\big(\mathtt{C}_1|S_1|(X, \overline{X}) \cup \cdots \cup \mathtt{C}_k|S_k|(X, \overline{X})\big) & \\ & = & |T| & \text{(Definition 1.7)} \end{array}$$

  The reason we can apply the induction hypothesis on G/T is that it contains the parity games for each S_i: because types have a special form, transition can only be of 2 forms:
  – non recursive constructor / destructor: $\mathtt{S}(\overline{S}) \to S_i$ where the priority increases,
  – recursive constructor / destructor: $\mathtt{S}(\overline{S}) \to R$ where $\mathtt{S}(\overline{S})$ is a subexpression of R, i.e. the priority decreases.

  Because T is of maximal finite priority, the only possible transitions to T are of the first kind. In G/T, which contains a new parameter $\mathtt{x}$, they become transitions $\mathtt{S}(\overline{\mathtt{x}}) \to \mathtt{x}_i$.[12] This implies that G/T contains a type parity game for S_i.
- The reasoning is similar if $T = \mathtt{T}_\nu(\overline{X})$ is a codatatype.
- if the priority of T is not maximal, write M for the position of maximal finite priority. We have

$$\begin{array}{rcll} ||G||_T(\overline{X}) & = & ||G/M||_T\big(||G||_M(\overline{X}), \overline{X}\big) & \text{(Definition 2.1)} \\ & \cong & ||G/M||_T\big(|M|(\overline{X}), \overline{X}\big) & \text{(previous point)} \\ & \cong & |T|(\overline{X}) & \text{(induction hypothesis, (*))} \end{array}$$

  The last isomorphism is the most subtle: G/M contains a new parameter $\mathtt{x}$ corresponding to M. Like above, any transition to M in G is transformed into a transition to the new parameter M in G/M. By induction hypothesis, we get that $||G/M||_T(\mathtt{x}, \overline{X}) \cong |T'|(\mathtt{x}, \overline{X})$ where T' is the type expression T where every occurrence of M has been replaced by $\mathtt{x}$. The last step decomposes into

$$\begin{array}{rcll} ||G/M||_T\big(|M|(\overline{X}), \overline{X}\big) & \cong & |T'|\big(|M|(\overline{X}), \overline{X}\big) & \\ & \cong & |T|(\overline{X}) & \text{(definition of totality)} \end{array}$$

  Indeed, on reaching $\mathtt{x}$ when computing $|T'|(|M|, \overline{X})$, we return M; just like on reaching M when computing $|T|(\overline{X})$. □

[12] Without our hypothesis, `Nil` : `list(nat)` → `unit` could be replaced by `Nil` : `list(nat)` → `x` in G/`unit` so that G/`unit` doesn't really contain a parity game for `list(nat)`.

2.3. **Strategies from Terms.** Strategies (Definition 2.2) are defined as order-theoretic trees. Because of the way types parity games are defined, they are equivalent to values in the corresponding type.

Lemma 2.10. *For any type T, and associated parity game G, the set of strategies for G starting from node T is isomorphic to the set of maximal elements in $[\![T]\!]$.*

Proof. Let t be a maximal element in $[\![T]\!]$, that is, an element of $[\![T]\!]$ which doesn't contain $\bot$. By definition, each finite branch of a maximal element of $[\![T]\!]$ is a finite path from T in the graph of T.

- If T is a datatype/odd position, t chooses precisely one constructor. The set of finite branches of t is thus *deterministic* on odd positions.
- If T is a codatatype/even position, t contains one field for each destructor. The set of finite branches of t is thus *complete* on even positions.

Conversely, we can construct a maximal element $\widehat{s}$ of $[\![T]\!]$ from any strategy s from T in G coinductively.

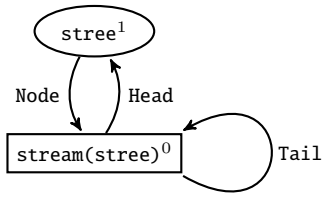
- If T is a data, by determinism, all non-empty paths of s start with the same constructor $\mathtt{C}$: this is the head constructor of $\widehat{s}$ and we continue by considering the strategy $s/\mathtt{C}$ obtained from s by removing the head constructor of each of its paths: $\widehat{s} := \mathtt{C}\ \widehat{s/\mathtt{C}}$.
- If T is a codata, by completeness, there are paths starting with each destructor $.\mathtt{D}_k$ of T. In that case, we can put $\widehat{s} = \{\ldots;\mathtt{D}_k = \widehat{s/\mathtt{D}_k};\ldots\}$.

This construction works because by construction, $s/\mathtt{C}$ are strategies for the appropriate types. □

Putting together Proposition 2.9, Proposition 2.3 and Lemma 2.10 we finally get

Corollary 2.11. *If T is a type and G a parity game for T, we have $\mathcal{W}(G)_T \cong |T|$. In particular, $v \in [\![T]\!]$ is total iff every branch of v has even principal priority.*

The only thing to note is that priorities are not part of the value $v \in [\![T]\!]$ but are read in G. As a final example in this section, let's consider the problematic example from Section 1.5. The parity game for `stree` is given by



Clearly, any strategies has to go through the `stree` node infinitely often, so that it contains an infinite branch with principal priority 1: it cannot be total.

2.4. **Forgetting Types.** If a term t in `chariot` (not necessarily a value) is of type T, it will generate a strategy in G, a parity game for T. Thanks to Corollary 2.11, the definition of t is total if and only if the corresponding strategy is winning.[13]

[13]Usually, t will be a function, resulting in some additional complexity.

In order to talk about priorities in `chariot`, we annotate each occurrence of constructor / destructor in a definition with its priority taken from one of the type's parity game. This can be done during Hindley-Milner type checking:

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

Once type checking is done, the type of constructors / destructors can be erased and only their priorities are kept. We end up with definitions like

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

Note that priorities are inferred and only used while checking totality. They are never shown to the end user.

We thus refine the notion of value from the previous section by adding priorities on constructors and destructors.

Definition 2.12. The set of *values with leaves in* $X_1, \ldots, X_n$, written $\mathcal{V}(X_1, \ldots, X_n)$ is defined coinductively by the grammar

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

where

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

Corollary 2.11 gives an intrinsic notion of totality on $\mathcal{V}$.

Definition 2.13. Totality for $\mathcal{V}$ is defined as $v \in |\mathcal{V}|$ iff and only if every branch of v has even principal priority.

Because of Corollary 2.11, checking totality of a recursive definition can thus take the following form:

(1) annotate the definition with priorities during type checking,
(2) check that, in the infinite unfolding of the recursive definition, either
  (a) we inspect a non-total infinite branch of the argument,
  (b) or we only construct total infinite branches of the result.

The patterns on the left side of clauses are the parts that "inspect the argument" and the values on the right side of clauses are the parts that "construct the result". A recursive definition satisfying this property is total. When applied to a total value (which has no non-total branch), the result is necessarily total (it contains only total branches).

Because we cannot really inspect the infinite unfolding of the definition, the size-change principle will be used to give a computable approximation of the above. Making this precise is the aim of an upcoming paper [Hyv].

Concluding Remarks

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

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

In a similar vein, the language could have coinductive constructors to deal with coinductive types like finite or infinite lists.[14] At the moment, the only way to introduce this type is with

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

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

Needless to say, using this quickly gets tiring.

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

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

or (inductive)

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

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

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

References

[AA02] Andreas Abel and Thorsten Altenkirch. A predicative analysis of structural recursion. *Journal of Functional Programming*, 12:1–41, January 2002.

[Abe10] Andreas Abel. Miniagda: Integrating sized and dependent types. In *In Partiality and Recursion (PAR 2010)*, 2010. arXiv:1012.4896.

[Abe12] Andreas Abel. Type-based termination, inflationary fixed-points, and mixed inductive-coinductive types. In *Proceedings 8th Workshop on Fixed Points in Computer Science, FICS 2012, Tallinn, Estonia, 24th March 2012.*, pages 1–11, 2012.

[AD12] Thorsten Altenkirch and Nils Anders Danielsson. Termination checking in the presence of nested inductive and coinductive types. In Ekaterina Komendantskaya, Ana Bove, and Milad Niqui, editors, *PAR-10. Partiality and Recursion in Interactive Theorem Provers*, volume 5 of *EasyChair Proceedings in Computing*, pages 101–106. EasyChair, 2012.

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

[Bar92] Michael Barr. Algebraically compact functors. *J. Pure Appl. Algebra*, 82(3):211–231, 1992.

[Ber93] Ulrich Berger. Total sets and objects in domain theory. *Annals of Pure and Applied Logic*, 60(2):91–117, 1993.

[CBGB16] Ranald Clouston, Ales Bizjak, Hans Bugge Grathwohl, and Lars Birkedal. The guarded lambda-calculus: Programming and reasoning with guarded recursion for coinductive types. *Logical Methods in Computer Science*, 12(3), 2016.

[CF92] Robin Cockett and Tom Fukushima. About Charity. Yellow Series Report No. 92/480/18, Department of Computer Science, The University of Calgary, June 1992.

[Cla13] Pierre Clairambault. Strong functors and interleaving fixpoints in game semantics. *RAIRO - Theor. Inf. and Applic.*, 47(1):25–68, 2013.

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

[Coq93] Thierry Coquand. Infinite objects in type theory. In H. Barendregt and T. Nipkow, editors, *Types for Proofs and Programs*, pages 62–78. Springer, Berlin, Heidelberg, 1993.

[Dou17a] Amina Doumane. Constructive completeness for the linear-time μ-calculus. In *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017*, pages 1–12, 2017.

[Dou17b] Amina Doumane. *On the infinitary proof theory of logics with fixed points. (Théorie de la démonstration infinitaire pour les logiques à points fixes)*. PhD thesis, Paris Diderot University, France, 2017.

[For14] Jérôme Fortier. *Expressive Power of Circular Proofs*. PhD, Aix Marseille Université ; Université du Québec à Montréal, December 2014.

[FS14] Jéôme Fortier and Luigi Santocanale. Cuts for circular proofs. In Nikolaos Galatos, Alexander Kurz, and Constantine Tsinakis, editors, *TACL 2013. Sixth International Conference on Topology, Algebra and Categories in Logic*, volume 25 of *EasyChair Proceedings in Computing*, pages 72–75. EasyChair, 2014.

[Gua18] Adrien Guatto. A generalized modality for recursion. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, pages 482–491, 2018.

[Hyv] Pierre Hyvernat. The size-change principle for mixed inductive and coinductive definitions. in preparation for submission to Logical Methods in Computer Science.

[Hyv14] Pierre Hyvernat. The size-change termination principle for constructor based languages. *Logical Methods in Computer Science*, 10(1), 2014.

[LJBA01] Chin Soon Lee, Neil D. Jones, and Amir Ben-Amram. The size-change principle for program termination. In *Symposium on Principles of Programming Languages*, volume 28, pages 81–92. ACM press, january 2001.

[LR18] Rodolphe Lepigre and Christophe Raffalli. Practical subtyping for curry-style languages, 2018. accepted for publication in ACM Transactions on Programming Languages and Systems (TOPLAS).

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[Nak00] Hiroshi Nakano. A modality for recursion. In *15th Annual IEEE Symposium on Logic in Computer Science, Santa Barbara, California, USA, June 26-29, 2000*, pages 255–266, 2000.

[Nor08] Ulf Norell. Dependently typed programming in agda. In *In Lecture Notes from the Summer School in Advanced Functional Programming*, 2008.

[PJ87] Simon Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, January 1987.

[San02a] Luigi Santocanale. A calculus of circular proofs and its categorical semantics. In Mogens Nielsen and Uffe Engberg, editors, *FoSSaCS*, volume 2303 of *Lecture Notes in Computer Science*, pages 357–371. Springer, 2002.

[San02b] Luigi Santocanale. From parity games to circular proofs. *Electr. Notes Theor. Comput. Sci.*, 65(1):305–316, 2002.

[San02c] Luigi Santocanale. μ-bicomplete categories and parity games. *Theoretical Informatics and Applications*, 36:195–227, 2002.

[The04] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004.