Flexible Coinduction in Agda

Luca Ciccone
University of Torino, Italy

Francesco Dagnino
DIBRIS, University of Genova, Italy

Elena Zucca
DIBRIS, University of Genova, Italy

Abstract

We provide an Agda library for *inference systems*, also supporting their recent generalization allowing *flexible coinduction*, that is, interpretations which are neither inductive, nor purely coinductive. A specific inference system can be obtained as an instance by writing a set of meta-rules, in an Agda format which closely resembles the usual one. In this way, the user gets for free the related properties, notably the inductive and coinductive intepretation and the corresponding proof principles. Moreover, a significant modularity is achieved. Indeed, rather than being defined from scratch and with a built-in interpretation, an inference system can also be obtained by composition operators, such as union and restriction to a smaller universe, and its semantics can be modularly chosen as well. In particular, flexible coinduction is obtained by composing in a certain way the interpretations of two inference systems. We illustrate the use of the library by several examples. The most significant one is a big-step semantics for the λ-calculus, where flexible coinduction allows to obtain a special result (∞) for all and only the diverging computations, and the proof of equivalence with small-step semantics is carried out by relying on the proof principles offered by the library.

2012 ACM Subject Classification Theory of computation → Semantics and reasoning

Keywords and phrases inference systems, induction, coinduction

Digital Object Identifier 10.4230/LIPIcs.ITP.2021.13

Supplementary Material The complete code of the library with all the examples discussed in the paper is available at
Software (Source Code): `https://github.com/LcicC/inference-systems-agda`
archived at `swh:1:dir:955d11416c46a1ee1b0b913f0bc5f30bf3de9e97`

Acknowledgements We warmly thank the anonymous ITP reviewers and Andreas Abel for the suggestions which greatly improved the paper, in particular for pointing out the relation with indexed containers.

1 Introduction

An *inference system* [5, 19, 21], that is, a set of (meta-)rules stating that a consequence can be derived from a set of premises, is a simple, general and widely-used way to express and reason about a recursive definition. In most cases such recursive definition is seen as inductive, that is, the denoted set consists of the elements with a finite derivation. This enables *inductive reasoning*, that is, to prove that the elements an inductively defined set satisfy a property, it is enough to show that, for each (meta-)rule, the property holds for the consequence assuming that it holds for the premises. In other cases, the recursive definition is seen as coinductive, that is, the denoted set consists of the elements with a possibly infinite derivation. This enables *coinductive reasoning*, that is, to prove that all the elements satisfying a property belong to the coinductively defined set, it is enough to show that, when the property holds for an element, it can be derived from premises for which the property holds as well. Recently, a generalization of inference systems has been proposed [8, 13, 15] which handles cases where neither the inductive, nor the purely coinductive intepretation provides the desired meaning.

This approach is called *flexible coinduction*, and, correspondingly, coinductive reasoning is generalized as well by a principle which is called *bounded coinduction*.

The Agda proof assistant [23] offers language constructs to inductively/coinductively define predicates, and correspondingly built-in proof principles. However, in this way the recursive definition is monolithic, and hard-wired with its chosen interpretation. Our aim, instead, is to provide an Agda library allowing the user to express a recursive definition as an *instance of a parametric type* of inference systems. In this way, the user is not committed from the beginning to a given interpretation but, rather, gets for free a bunch of properties which have been proved once and for all, including the inductive and coinductive intepretation and the corresponding proof principles. Moreover, it is possible to define composition operators on inference systems, for instance union and restriction. Finally, flexible coinduction is modularly obtained as well, by composing in a certain way the interpretations of two inference systems.

Indexed containers [6] provide a way to specify possibly recursive definitions of predicates independently from their interpretation and are supported in the Agda standard library. An Agda implementation of inference systems can be provided by seeing them as indexed containers. However, this approach requires to structure definitions in an unusual way. Indeed, inference systems are usually presented through a (finite) set of *meta-rules*, denoting all the rules which can be obtained by instantiating meta-variables with values satisfying the side condition. Hence, we provide a different implementation following this schema, to allow users to write their own inference system in an Agda format which closely resembles that "on paper". We then prove that the two implementations are equivalent, showing that every indexed container can be encoded in terms of meta-rules and viceversa.

In Sect. 2 we recall basic notions on inference systems, and in Sect. 3 the generalization supporting flexible coinduction. In Sect. 4 we describe how to implement (generalized) inference systems in Agda. Notably, in Sect. 4.1 we present the approach mimicking meta-rules, showing step-by-step the correspondence with the previous definitions. In Sect. 4.2, instead, we explain the view of inference systems as indexed containers, and prove the equivalence. Then, we illustrate the use of the library by several examples. In Sect. 5 we consider three different predicates on possibily infinite lists (*colists* in Agda terminology) defined by induction, coinduction, and flexible coinduction, respectively. In Sect. 6 we provide a more significant and elaborated example: a big-step semantics for the λ-calculus where flexible coinduction allows to obtain a special result (∞) for all and only the diverging computations, and the proof of equivalence with small-step semantics is carried out by relying on the proof principles offered by the library. Finally, we summarize the contribution and outline further work in Sect. 7.

2 Inference systems

We recall basic definitions on inference systems [5, 19, 21, 15]. Throughout this section and the following we assume a set $\mathcal{U}$, named *universe*, whose elements j are called *judgments*. An *inference system* $\mathcal{I}$ is a set of *rules*, which are pairs $\langle pr, j\rangle$, with $pr \subseteq \mathcal{U}$ the set of *premises*, and $j \in \mathcal{U}$ the *conclusion* (a.k.a. *consequence*). A rule with an empty set of premises is an *axiom*. A rule $\langle pr, j\rangle$ is often written as a fraction $\dfrac{pr}{j}$.

In practice, inference systems are described by a (finite) set of *meta-rules*, written in some meta-language. For instance, taking as universe the set $\mathbb{N}^\infty$ of finite and infinite lists of natural numbers, and denoting Λ the empty list, and $x{:}u$ the list with head x and tail u, the following two sets of meta-rules describe inference systems for the predicates holding when an element belongs to the list, and all elements are positive, respectively.

$$\text{(mem-h)}\ \frac{}{\mathsf{member}(x, x{:}xs)} \qquad \text{(mem-t)}\ \frac{\mathsf{member}(x, xs)}{\mathsf{member}(x, y{:}xs)}$$

$$\text{(allP-}\Lambda\text{)}\ \frac{}{\mathsf{allPos}(\Lambda)} \qquad \text{(allP-t)}\ \frac{\mathsf{allPos}(xs)}{\mathsf{allPos}(x{:}xs)}\quad x > 0$$

The aim of an inference system is to define, in a way which provides canonical techniques to prove properties, a subset of the universe. There are several ways to choose this set, depending on the *interpretation* given to the inference system.

To define an interpretation in a model-theoretic way, the basis is the *inference operator* associated with $\mathcal{I}$, which is the function $F_{\mathcal{I}} : \wp(\mathcal{U}) \to \wp(\mathcal{U})$ defined by

$$F_{\mathcal{I}}(X) = \{j \in \mathcal{U} \mid pr \subseteq X, \langle pr, j\rangle \in \mathcal{I}, \text{ for some } pr \in \wp(\mathcal{U})\}.$$

A subset X of the universe is *$\mathcal{I}$-closed* if, for all rules $\langle pr, j\rangle \in \mathcal{I}$, if $pr \subseteq X$ then $j \in X$, it is *$\mathcal{I}$-consistent* if, for all $j \in X$, there is a rule $\langle pr, j\rangle \in \mathcal{I}$ and $pr \subseteq X$.

The *inductive interpretation* $Ind[\![\mathcal{I}]\!]$ is the least fixed point of $F_{\mathcal{I}}$, which, by the Knaster-Tarski theorem, coincides with the least pre-fixed point of $F_{\mathcal{I}}$ and so with the least $\mathcal{I}$-closed set. As an immediate consequence, when we define a set inductively, that is, as $Ind[\![\mathcal{I}]\!]$ for some $\mathcal{I}$, we can prove that such definition is *sound* with respect to a given specification, namely, a subset $\mathcal{S} \subseteq \mathcal{U}$, by the *induction principle*:

(ind) If a set $\mathcal{S} \subseteq \mathcal{U}$ is $\mathcal{I}$-closed, then $Ind[\![\mathcal{I}]\!] \subseteq \mathcal{S}$.

Proving that $\mathcal{S}$ is $\mathcal{I}$-closed amounts to show that, for each (meta-)rule, if the premises satisfy $\mathcal{S}$ then the consequence satisfies $\mathcal{S}$ as well.

The *coinductive interpretation* $CoInd[\![\mathcal{I}]\!]$ is the greatest fixed point of $F_{\mathcal{I}}$, which, by the Knaster-Tarski theorem, coincides with the largest post-fixed point of $F_{\mathcal{I}}$ and so with the largest $\mathcal{I}$-consistent set. As an immediate consequence, when we define a set coinductively, that is, as $CoInd[\![\mathcal{I}]\!]$ for some $\mathcal{I}$, we can prove that such definition is *complete* with respect to a given specification $\mathcal{S} \subseteq \mathcal{U}$ by the *coinduction principle*:

(coind) If a subset $\mathcal{S} \subseteq \mathcal{U}$ is $\mathcal{I}$-consistent, then $\mathcal{S} \subseteq CoInd[\![\mathcal{I}]\!]$.

Proving that $\mathcal{S}$ is $\mathcal{I}$-consistent amounts to show that, for each j satisfying $\mathcal{S}$, there is a rule with consequence j and premises satisfying $\mathcal{S}$ as well.

To prove completeness of the inductive interpretation, and soundness of the coinductive interpretation, instead, there is no canonical technique, so some ad-hoc proof is needed.

Alternatively, the interpretation can also be specified proof-theoretically, that is, through the notion of *proof tree*. For the aim of this paper a semi-formal definition is enough, we refer to [13, 15] for a rigorous treatment. Set $\mathcal{T}$ the set of trees with nodes (labeled by) judgments. Given $\tau \in \mathcal{T}$, $\mathsf{r}(\tau)$ is the (label of the) root, $\mathsf{dst}(\tau)$ the set of direct subtrees, and $\mathsf{chl}(\tau)$ the set of (the labels of) their roots. The inference operator can be naturally extended to a function $T_{\mathcal{I}} : \wp(\mathcal{T}) \to \wp(\mathcal{T})$ as follows:

$$T_{\mathcal{I}}(Y) = \{\tau \in \mathcal{T} \mid \mathsf{dst}(\tau) \subseteq Y, \langle \mathsf{chl}(\tau), \mathsf{r}(\tau)\rangle \in \mathcal{I}\}$$

Then, a *proof tree* (a.k.a. *derivation*) is a tree such that, for each subtree τ, $\langle \mathsf{chl}(\tau), \mathsf{r}(\tau)\rangle \in \mathcal{I}$, that is, there is a node (labelled by) j with set of children (labelled by) pr only if the rule $\langle pr, j\rangle$ belongs to $\mathcal{I}$. A *proof tree for* j is a proof tree τ such that $\mathsf{r}(\tau) = j$.

Then, $\mathit{Ind}[\![\mathcal{I}]\!]$ and $\mathit{CoInd}[\![\mathcal{I}]\!]$ can be equivalently defined as the sets of judgments with respectively a finite and a possibly infinite proof tree. Moreover, the sets of finite and possibly infinite proof trees turn out to be the least fixed point and the greatest fixed point, respectively, of the inference operator extended to trees. See [13, 14] for detailed proofs carried out with a rigorous definition of trees.

Coming back to the two examples above, it is easy to see that, in order to obtain the desired meaning, the inference system for member should be interpreted inductively, while that for allPos coinductively. Indeed, the fact that an element belongs to the list can be shown by a finite proof tree, even for an infinite list, whereas, for such a list, to show that all elements are positive an infinite proof tree is needed.

3 Corules

We recall the notion of inference systems with corules, which mixes induction and coinduction in a flexible way [8, 13, 15]. For $X \subseteq \mathcal{U}$, we write $\mathcal{I}_{|X}$ for the *restriction* of $\mathcal{I}$ to X, that is, the inference system $\{\langle pr, j\rangle \in \mathcal{I} \mid j \in X\}$.

▶ **Definition 1.** *A* generalized inference system*, or* inference system with corules*, is a pair $\langle \mathcal{I}, \mathcal{I}_{\mathsf{co}}\rangle$ where $\mathcal{I}$ and $\mathcal{I}_{\mathsf{co}}$ are inference systems. Elements in $\mathcal{I}$ and $\mathcal{I}_{\mathsf{co}}$ are called rules and corules, respectively. The interpretation of $\langle \mathcal{I}, \mathcal{I}_{\mathsf{co}}\rangle$ is defined by $\mathit{FCoInd}[\![\mathcal{I}, \mathcal{I}_{\mathsf{co}}]\!] = \mathit{CoInd}[\![\mathcal{I}_{|\mathit{Ind}[\![\mathcal{I}\cup\mathcal{I}_{\mathsf{co}}]\!]}]\!]$.*

Thus, the interpretation $\mathit{FCoInd}[\![\mathcal{I}, \mathcal{I}_{\mathsf{co}}]\!]$ is basically *coinductive*, but restricted to a universe of judgements which is *inductively defined* by the (potentially) larger system $\mathcal{I} \cup \mathcal{I}_{\mathsf{co}}$.

In [8, 13, 15] the following results are proved:

- $\mathit{FCoInd}[\![\mathcal{I}, \mathcal{I}_{\mathsf{co}}]\!]$ is the largest post-fixed point of $F_{\mathcal{I}}$ included in $\mathit{Ind}[\![\mathcal{I} \cup \mathcal{I}_{\mathsf{co}}]\!]$
- in proof-theoretic terms, $\mathit{FCoInd}[\![\mathcal{I}, \mathcal{I}_{\mathsf{co}}]\!]$ is the set of judgments which have a possibly infinite proof tree in $\mathcal{I}$ whose nodes all have a finite proof tree in $\mathcal{I} \cup \mathcal{I}_{\mathsf{co}}$, that is, the (standard) inference system consisting of rules and corules.

As an immediate consequence, when we define a set by flexible coinduction, that is, as $\mathit{FCoInd}[\![\mathcal{I}, \mathcal{I}_{\mathsf{co}}]\!]$ for some $\langle \mathcal{I}, \mathcal{I}_{\mathsf{co}}\rangle$, we can prove that such definition is *complete* with respect to a given specification $\mathcal{S} \subseteq \mathcal{U}$ by the *bounded coinduction principle*, which generalizes the coinduction principle:

(b-coind) If a subset $\mathcal{S} \subseteq \mathcal{U}$ is bounded, that is, $\mathcal{S} \subseteq \mathit{Ind}[\![\mathcal{I} \cup \mathcal{I}_{\mathsf{co}}]\!]$, and $\mathcal{I}$-consistent, then $\mathcal{S} \subseteq \mathit{FCoInd}[\![\mathcal{I}, \mathcal{I}_{\mathsf{co}}]\!]$.

Proving that $\mathcal{S}$ is bounded means proving completeness of the inference system extended by corules, interpreted inductively, with respect to $\mathcal{S}$. Hence, there is no canonical technique, and for each concrete case we must find an ad-hoc proof. Proving that $\mathcal{S}$ is $\mathcal{I}$-consistent, as for the standard coinduction principle, amounts to show that, for each j satisfying $\mathcal{S}$, there is a rule with consequence j and premises satisfying $\mathcal{S}$ as well. As for the purely coinductive interpretation, an ad-hoc proof is also needed for soundness. However, as shown in the examples in Sect. 5 and Sect. 6, in many cases we can take advantage of the fact that $\mathit{FCoInd}[\![\mathcal{I}, \mathcal{I}_{\mathsf{co}}]\!]$ is a subset of $\mathit{Ind}[\![\mathcal{I} \cup \mathcal{I}_{\mathsf{co}}]\!]$, and reason by induction on the latter.

We illustrate the role of corules by a simple example: defining the maximal element of a list. A (meta-)corule is written as a fraction with a thicker line.

$$\text{(max-}\Lambda\text{)}\ \frac{}{\mathsf{maxElem}(x{:}\Lambda, x)} \qquad \text{(max-t)}\ \frac{\mathsf{maxElem}(u, y)}{\mathsf{maxElem}(x{:}u, z)}\ z = \max(x, y) \qquad \text{(max-co)}\ \frac{}{\mathsf{maxElem}(x{:}u, x)}$$

Considering the standard inference system consisting of the two rules, its inductive interpretation only provides the desired meaning on finite lists, since for infinite lists an infinite proof is needed. However, the coinductive interpretation allows also wrong judgements. For instance, let $L = 1{:}2{:}1{:}2{:}1{:}2{:}\ldots$. Then any judgment $\mathsf{maxElem}(L, x)$ with $x \geq 2$ can be derived, as illustrated by the following examples.

$$\dfrac{\dfrac{\dfrac{\ldots}{\mathsf{maxElem}(L,2)}}{\mathsf{maxElem}(2{:}L,2)}}{\mathsf{maxElem}(1{:}2{:}L,2)} \qquad \dfrac{\dfrac{\dfrac{\ldots}{\mathsf{maxElem}(L,5)}}{\mathsf{maxElem}(2{:}L,5)}}{\mathsf{maxElem}(1{:}2{:}L,5)}$$

By adding the coaxiom, we force the element to belong to the list, so that wrong results are "filtered out". Indeed, the judgment $\mathsf{maxElem}(1{:}2{:}L, 2)$ has the infinite proof tree shown above, and each node has a finite proof tree in the inference system extended by the corule:

$$\dfrac{\dfrac{\dfrac{\ldots}{\mathsf{maxElem}(L,2)}}{\mathsf{maxElem}(2{:}L,2)}}{\mathsf{maxElem}(1{:}2{:}L,2)} \qquad \dfrac{\genfrac{}{}{2pt}{}{}{\mathsf{maxElem}(2{:}L,2)}}{\mathsf{maxElem}(1{:}2{:}L,2)}$$

On the other hand, the judgment $\mathsf{maxElem}(1{:}2{:}L, 5)$ has the infinite proof tree shown above, but has *no finite proof tree* in the inference system extended by the corule. Indeed, since 5 does not belong to the list, the corule can never be applied. Hence, this judgment cannot be derived in the inference system with corules. We refer to [8, 13, 15] for other examples.

Note that the inductive and coinductive interpretation of $\mathcal{I}$ are special cases, notably:

- the inductive interpretation of $\mathcal{I}$ is the interpretation of $\langle \mathcal{I}, \emptyset \rangle$
- the coinductive interpretation of $\mathcal{I}$ is the interpretation of $\langle \mathcal{I}, \{\langle \emptyset, j\rangle \mid j \in \mathcal{U}\}\rangle$.

4 Generalized inference systems in Agda

We describe how to implement (generalized) inference systems in Agda. Notably, in Section 4.1 we present an approach mimicking meta-rules, showing step-by-step the correspondence with the definitions of the previous sections. In Sect. 4.2, instead, we explain the view of inference systems as indexed containers, and prove the equivalence.

4.1 An Agda library for writing meta-rules

In this section and the following we report the most interesting parts of the Agda code.

As anticipated, the aim of the Agda library is to allow a user to write meta-rules as "on paper". To illustrate this format, let us consider, e.g., the previous example:

$$\text{(allP-t)}\ \dfrac{\mathsf{allPos}(xs)}{\mathsf{allPos}(x{:}xs)} \quad x > 0$$

In a meta-rule, we have *meta-variables*, which range over certain sets, in a way possibly restricted by a *side condition*. We call *context* the set of the instantiations of meta-variables which satisfy the side-condition, hence produce a rule of the inference system. In the example, there are two meta-variables, x and xs, which range over $\mathbb{N}$ and $\mathbb{N}^\infty$, respectively, with the restriction that x should be positive. Hence the context is $\{\langle x, l\rangle \in \mathbb{N} \times \mathbb{N}^\infty \mid x > 0\}$, see Sect. 5 for the Agda version of this meta-rule.

Correspondingly, the following Agda declaration defines a meta-rule as a record, parametric on the universe `U`. The first two components are the context and a set of positions for premises. For each element of the context (instantiation of meta-variables satisfying the side condition), the last two components produce the premises, one for each position, and the conclusion of the rule obtained by this instantiation.

```
record MetaRule {ℓc ℓp : Level} (U : Set ℓu) : Set _ where
  field
    Ctx : Set ℓc
    Pos : Set ℓp
    prems : Ctx → Pos → U
    conclu : Ctx → U

  RF[_] : ∀{ℓ} → (U → Set ℓ) → (U → Set _)
  RF[_] P u = Σ[ c ∈ Ctx ] (u ≡ conclu c × (∀ p → P (prems c p)))

  RClosed : ∀{ℓ} → (U → Set ℓ) → Set _
  RClosed P = ∀ c → (∀ p → P (prems c p)) → P (conclu c)
```

Recall that in Agda the declaration `U : Set` introduces the type (set) `U`, and `P : U → Set` the dependent type (predicate on `U`) `P`. For each element `u` of `U`, `P u` is the type of the proofs that `u` satifies `P`, hence `P u` inhabited means that `u` satisfies `P`. To avoid paradoxes, not every Agda type is in `Set`; there is an infinite sequence `Set 0`, `Set 1`, . . . , `Set ℓ`, . . . such that `Set ℓ : Set (suc ℓ)`, where ℓ is a *level*, and `Set` is an abbreviation for `Set 0`. The programmer can write a wildcard for a level which can be inferred; to make the Agda code reported in the paper lighter, we sometimes use a wildcard even for a level which is explicit in the real code.

In the Agda code in this section, predicates `P : U → Set` encode subsets of the universe as in Sect. 2 and Sect. 3, so we speak of subsets and membership, rather than of predicates and satisfaction, to closely follow the previous formulation.

The function `RF[_]` encodes the inference operator associated with the meta-rule. Given a subset `P` of the universe, `u` belongs to the resulting subset if we can find an instantiation `c` of meta-variables satisfying the side condition, producing `u` as conclusion, and, for each position, a premise in `P`. Note the use of existential quantification `Σ[ x ∈ A ] B` where `B` depends on `x`.

The predicate `RClosed` encodes the property of being closed with respect to the meta-rule. A subset `P` of the universe is closed if, for each instantiation `c` of the meta-variables satisfying the side-condition, if all the premises are in `P` then the conclusion is in `P` as well. Note the use of universal quantification `∀ (x : A) → B`, where `B` depends on `x`.

Since in practical cases meta-rules are very often *finitary*, that is, premises are a finite set, the library also offers an interface to write a (finitary) meta-rule, by providing, besides the context, two components which are the *vector* of premises, with fixed length `n`, and the conclusion. The injection `from` transforms this more concrete format in the generic one for meta-rules, by specifying that the set of positions is `Fin n` (the set of indexes from 0 to $n-1$).

```
record FinMetaRule {ℓc n} (U : Set ℓu) : Set _ where
  field
    Ctx : Set ℓc
    comp : C → Vec U n × U

  from : MetaRule {ℓc} {zero} U
  from .MetaRule.Ctx = Ctx
  from .MetaRule.Pos = Fin n
  from .MetaRule.prems c i = get (proj₁ (comp c)) i
  from .MetaRule.conclu c = proj₂ (comp c)
```

An inference system is defined as a record, parametric on the universe U, consisting of a set of meta-rule names and a family of meta-rules. The function ISF [_] and the predicate ISClosed are defined composing those given for a single meta-rule.

```
record IS {ℓc ℓp ℓn : Level} (U : Set ℓu) : Set _ where
  field
    Names : Set ℓn
    rules : Names → MetaRule {ℓc} {ℓp} U

  ISF[_] : ∀{ℓ} → (U → Set ℓ) → (U → Set _)
  ISF[_] P u = Σ[ rn ∈ Names ] RF[ rules rn ] P u

  ISClosed : ∀{ℓ} → (U → Set ℓ) → Set _
  ISClosed P = ∀ rn → RClosed (rules rn) P
```

Recall that the inductive interpretation $Ind[\![\mathcal{I}]\!]$ of an inference system $\mathcal{I}$ is the set of elements of the universe which have a finite proof tree, and finite proof trees are, in turn, inductively defined, that is, by a least fixed point operator. In Agda, inductive structures are encoded as *datatypes*, which specify their constructors.

```
data Ind⟦_⟧ {ℓc ℓp ℓn : Level}
(is : IS {ℓc} {ℓp} {ℓn} U) : U → Set _ where
  fold : ∀ {u} → ISF[ is ] Ind⟦ is ⟧ u → Ind⟦ is ⟧ u
```

For each u, Ind ⟦ is ⟧ u is the type of the proofs that u satisfies Ind ⟦ is ⟧, which are essentially the finite proof trees[1] for u. Indeed, the fold constructor, given a proof that u can be derived by applying a rule from premises belonging to Ind ⟦ is ⟧, which essentially consists of a rule with conclusion u and finite proof trees for its premises, builds a finite proof tree for u.

The coinductive interpretation $CoInd[\![\mathcal{I}]\!]$, instead, is the set of elements of the universe which have a possibly infinite proof tree, and possibly infinite proof trees are, in turn, coinductively defined, that is, by a greatest fixed point operator. In Agda, coinductive structures can be encoded in two different ways: either as *coinductive records* [3], or as datatypes by using the mechanism of *thunks* (suspended computations) together with *sized types* [1, 2, 4] to ensure termination. To allow compatibility with existing code implemented in either way, both versions are supported by the library.

```
record CoInd⟦_⟧ {ℓc ℓp ℓn : Level}
 (is : IS {ℓc} {ℓp} {ℓn} U) (u : U) : Set _ where
  coinductive
  constructor cofold_
  field
    unfold : ISF[ is ] CoInd⟦ is ⟧ u

  data SCoInd⟦_⟧ {ℓc ℓp ℓn : Level}
   (is : IS {ℓc} {ℓp} {ℓn} U) : U → Size → Set _ where
    sfold : ∀ {u i} → ISF[ is ] (λ u → Thunk (SCoInd⟦ is ⟧ u) i) u
      → SCoInd⟦ is ⟧ u i
```

For each u, CoInd ⟦ u ⟧ is the type of the proofs that u satisfies CoInd ⟦ is ⟧, which are essentially the possibly infinite proof trees for u, and analogously for SCoInd ⟦ is ⟧.

In the first version, a possibly infinite proof tree for u is a record with only one field unfold containing an element of ISF [is] CoInd ⟦ is ⟧ u, that is, a proof that u can be derived by applying a rule from premises belonging to CoInd ⟦ is ⟧, which essentially consists of a rule with conclusion u and possibly infinite proof trees for its premises.

[1] With some more structure, since the Agda proofs keep trace of the applied meta-rules.

In the second version, a possibly infinite proof tree is obtained by a **data** constructor, analogously to a finite one in the inductive interpretation; however, since proof trees are encoded as thunks, hence evaluated lazily, this encoding represents infinite trees as well. In other words, coinduction is "hidden" in the library type `Thunk`, which is a coinductive record with only one field `force`, intuitively representing the suspended computation.

The interpretation of a generalized inference system can then be encoded following exactly the definition in Sect. 3: it is the coinductive interpretation of `I`, restricted to rules whose conclusion is in the inductive interpretation of the (standard) inference system consisting of both rules `I` and corules `C`.

```
FCoInd⟦_,_⟧ : ∀{ℓc ℓp ℓn ℓn'} → (I : IS {ℓc} {ℓp} {ℓn} U)
  → (C : IS {ℓc} {ℓp} {ℓn'} U) → U → Set _
FCoInd⟦ I , C ⟧ = CoInd⟦ I ⊓ Ind⟦ I ∪ C ⟧ ⟧

SFCoInd⟦_,_⟧ : ∀{ℓc ℓp ℓn ℓn'} → (I : IS {ℓc} {ℓp} {ℓn} U)
  → (C : IS {ℓc} {ℓp} {ℓn'} U) → U → Size → Set _
SFCoInd⟦ I , C ⟧ = SCoInd⟦ I ⊓ Ind⟦ I ∪ C ⟧ ⟧
```

The definition is provided in two flavours where the coinductive interpretation is encoded by coinductive records and thunks, respectively, and uses two operators on inference systems, restriction ⊓ and union ∪. We report the former, which adds to each rule the side condition that the conclusion should satisfy `P`, as specified by the function `addSideCond`, omitted.

```
_⊓_ : ∀ {ℓc ℓp ℓn ℓ}{U : Set ℓu} → IS {ℓc} {ℓp} {ℓn} U
  → (U → Set ℓ) → IS {ℓc ⊔ ℓ} {_} {_} U
(is ⊓ P) .Names = is .Names
(is ⊓ P) .rules rn = addSideCond (is .rules rn) P
```

The library also provides the proofs of relevant properties, e.g., that closed sets coincide with pre-fixed points, and consistent sets coincide with post-fixed points. Moreover, it is shown that the two versions of encoding of the coinductive interpretation (by coinductive records and thunks) are equivalent. Finally, the library provides the induction, coinduction, and bounded coinduction principles. We only report here the statements.

```
ind[_] : ∀{ℓc ℓp ℓn ℓ}
     → (is : IS {ℓc} {ℓp} {ℓn} U)          −− IS
     → (S : U → Set ℓ)                     −− specification
     → ISClosed is S                       −− S is closed
     → Ind⟦ is ⟧ ⊆ S
```

If `S` is closed, then each element of the inductively defined set `Ind ⟦ is ⟧` satisfies `S`.

```
coind[_] : ∀{ℓc ℓp ℓn ℓ}
     → (is : IS {ℓc} {ℓp} {ℓn} U)
     → (S : U → Set ℓ)
     → (S ⊆ ISF[ is ] S)                   −− S is consistent
     → S ⊆ CoInd⟦ is ⟧
```

If `S` is consistent, then each element satisfying `S` is in the coinductively defined set `CoInd ⟦ is ⟧`.

```
bounded−coind[_,_] : ∀{ℓc ℓp ℓn ℓn' ℓ}
     → (I : IS {ℓc} {ℓp} {ℓn} U)
     → (C : IS {ℓc} {ℓp} {ℓn'} U)
     → (S : U → Set ℓ)
     → S ⊆ Ind⟦ I ∪ C ⟧                     −− S is bounded w.r.t. I ∪ C
     → S ⊆ ISF[ I ] S                      −− S is consistent w.r.t. I
     → S ⊆ FCoInd⟦ I , C ⟧
```

If S is bounded, and consistent with respect to I, then each element which satisfies S belongs to the set FCoInd ⟦ I , C ⟧ defined by flexible coinduction.

Another easy theorem useful in proofs is that $\mathit{FCoInd}[\![\mathcal{I},\mathcal{I}_{\mathsf{co}}]\!] \subseteq \mathit{Ind}[\![\mathcal{I}\cup\mathcal{I}_{\mathsf{co}}]\!]$:

```
fcoind-to-ind : ∀{ℓc ℓp ℓn ℓn'}
    {is : IS {ℓc} {ℓp} {ℓn} U}{cois : IS {ℓc} {ℓp} {ℓn'} U}
    → FCoInd⟦ is , cois ⟧ ⊆ Ind⟦ is ∪ cois ⟧
```

4.2 Inference systems as indexed containers

Indexed containers [6] are a rather general notion, meant to capture families of datatypes with some form of indexing. They are part of the Agda standard library. We report below the definition, simplified and adapted a little for presentation purpose. Notably, we use ad-hoc field names, chosen to reflect the explanation provided below.

```
record Container {ℓi ℓo}
 (I : Set ℓi) (O : Set ℓo) (ℓc ℓp : Level) : Set _ where
  constructor _ ◁ _/_
  field
    Cons  : (o : O) → Set ℓc
    Pos : ∀ {o} → Cons o → Set ℓp
    input : ∀ {o} (c : Cons o) → Pos c → I

⟦_⟧: ∀ {ℓi ℓo ℓc ℓp ℓ} {I : Set ℓi} {O : Set ℓo} → Container I O ℓc ℓp →
       (I → Set ℓ) → (O → Set _)
⟦ C ◁ P / inp ⟧ X o = Σ[ c ∈ C o ] ((p : P c) → X (inp c p))
```

To explain the view of an inference system as an indexed container, we can think of the latter as describing a family of datatype constructors where I and O are input and output sorts, respectively. Then, Cons specifies, for each output sort o, the set of its constructors; for each constructor for o, Pos specifies a set of positions to store inputs to the constructor; finally, input specifies the input sort for each position in a constructor.

The function ⟦_⟧ models the "semantics" of an indexed container, that is, given a family of inputs X indexed by I, it returns the family of outputs indexed by O which can be constructed by providing to some constructor inputs from P of correct sorts.

Then, inference systems can be defined as indexed containers where input and output sorts coincide, and are the elements of the universe, as follows.

```
  ISCont : {ℓc ℓp : Level} → (U : Set ℓu) → Set _
  ISCont {ℓc} {ℓp} U = Container U U ℓc ℓp
```

In this way, for each u : U:

- Cons u is the set of (indexes for) all the rules which have consequence u
- Pos c is the set of (indexes for) the premises of the c-th rule
- input c p is the p-th premise of the c-th rule.

This view comes out quite naturally observing that an inference system is an element of $\wp(\wp(\mathcal{U}) \times \mathcal{U})$; equivalently, a function which, for each $j \in \mathcal{U}$, returns the set of the sets of premises of all the rules with consequence j. In a constructive setting such as Agda, the powerset construction is not available, hence we have to use functions. So, for each element u, we need a type to index all rules with consequence u, and, for each rule, a type to index its premises, which are exactly the data of an indexed container. In other words, this view of inference systems as indexed containers explicitly interprets rules as constructors for proofs.

Moreover, definitions in Sect. 2 can be easily obtained as instances of analogous definitions for indexed containers, building on the fact that the inference operator associated with an inference system turns out to be the semantics ⟦_⟧ of the corresponding container.

Whereas this encoding allows reuse of notions and code, a drawback is that information is structured in a rather different way from that "on paper"; notably, we group together rules with the same consequence, rather than those obtained as instances of the same "schema", that is, meta-rule. For instance, the inference system for allPos would be as follows:

```
allPosCont : ISCont (Colist ℕ ∞)
allPosCont .Cons [] = ⊤
allPosCont .Cons (x :: xs) = x > 0
allPosCont .Pos {[]} c = ⊥
allPosCont .Pos {x :: xs} c = Fin 1
allPosCont .input {x :: xs} c zero = xs .force
```

For this reason we developed the Agda library mimicking meta-rules described in Sect. 4.1, and we use this library for the examples in the following sections.

However, we can prove that the two notions are equivalent, as shown below. To this end, we define a translation C[_] from inference systems as in Sect. 4.1 to indexed containers, and a converse translation IS[_]. Note that in the translation C[_] each meta-rule is transformed in all its instantiations; more precisely, for each u : C, Cons u gives all the instantiations of meta-rules having u as consequence. Conversely, in the translation IS[_], each rule is transformed in a meta-rule with trivial context.

```
C[_] : ∀{ℓc ℓp ℓn} → IS {ℓc} {ℓp} {ℓn} U → Container U U _ ℓp
C[ is ] .Cons u = Σ[ rn ∈ is .Names ] Σ[ c ∈ is .rules rn .Ctx ]
    u ≡ is .rules rn .conclu c
C[ is ] .Pos (rn , _ , refl) = is .rules rn .Pos
C[ is ] .input (rn , c , refl) p = is .rules rn .prems c p

IS[_] : ∀{ℓc ℓp} → Container U U ℓc ℓp → IS {zero} {ℓp} {l ⊔ ℓc} U
IS[ C ] .Names = Σ[ u ∈ U ] C .Cons u
IS[ C ] .rules (u , c) =
  record {
    Ctx = ⊤ ;
    Pos = C .Pos c ;
    prems = λ _ r → C .input c r ;
    conclu = λ _ → u }

isf−to−c : ∀{ℓc ℓp ℓn ℓp} {is : IS {ℓc} {ℓp} {ℓn} U}{P : U → Set ℓp}
    → ISF[ is ] P ⊆ ⟦ C[ is ] ⟧ P
  isf−to−c (rn , c , refl , pr) = (rn , c , refl) , pr

c−to−isf : ∀{l' ℓp ℓp} {C : Container U U l' ℓp}{P : U → Set ℓp}
    → ⟦ C ⟧ P ⊆ ISF[ IS[ C ] ] P
c−to−isf (c , pr) = (_ , c) , tt , refl , pr
```

5 Using the library

We show how to use the library to define specific inference systems and prove their properties. Consider the examples in Sect. 2 and Sect. 3. For member, the universe are pairs of elements and possibly infinite lists, implemented by the Agda library Colist which uses thunks:

```
U = A × Colist A ∞
data memberRN : Set where mem-h mem-t : memberRN
```

```
mem-h-r : FinMetaRule U
mem-h-r .Ctx = A × Thunk (Colist A) ∞
mem-h-r .comp (x , xs) =
  [] ,
  ──────────────
  (x , x :: xs)

mem-t-r : FinMetaRule U
mem-t-r .Ctx = A × A × Thunk (Colist A) ∞
mem-t-r .comp (x , y , xs) =
  ((x , xs .force) :: []) ,
  ──────────────
  (x , y :: xs)

memberIS : IS U
memberIS .Names = memberRN
memberIS .rules mem-h = from mem-h-r
memberIS .rules mem-t = from mem-t-r
```

Here memberRN are the rule names, and each rule name has an associated element of FinMetaRule U, which exactly encodes the meta-rule in Sect. 2. Note, in mem-t-r, the use of the force field of Thunk to actually obtain the tail colist.

This inference system is expected to define exactly the pairs (x , xs) such that x belongs to xs, that is, those satisfying the following specification

```
memSpec : U → Set
memSpec (x , xs) = Σ[ i ∈ ℕ ] (Colist.lookup i xs ≡ just x)
```

where the library function lookup : ℕ → Colist A ∞ → Maybe A returns the i-th element of xs, if any.

As said in Sect. 2, to obtain the desired meaning this inference system has to be interpreted inductively, and soundness can be proved by the induction principle, that is, by providing a proof that the specification is closed with respect to the two meta-rules, as shown below.

```
_member_ : A → Colist A ∞ → Set
x member xs = Ind⟦ memberIS ⟧ (x , xs)

memSpecClosed : ISClosed memberIS memSpec
memSpecClosed mem-h _ _ = zero , refl
memSpecClosed mem-t _ pr =
  let (i , proof) = pr Fin.zero in (suc i) , proof

memberSound : ∀ {x xs} → x member xs → memSpec (x , xs)
memberSound = ind[memberIS] memSpec memSpecClosed
```

For completeness there is no canonical technique; in this example, it can be proved by induction on the position (the index i in the specification).

For allPos , the universe are possibly infinite lists.

```
U : Set
U = Colist ℕ ∞
data allPosRN : Set where allP-Λ allP-t : allPosRN

allP-Λ-r : FinMetaRule U
allP-Λ-r .Ctx = ⊤
allP-Λ-r .comp c =
  [] ,
  ──────────────
  []
```

```
allP-t-r : FinMetaRule U
allP-t-r .Ctx = Σ[ (x , _) ∈ ℕ × Thunk (Colist ℕ) ∞ ] x > 0
allP-t-r .comp ((x , xs) , _) =
  ((xs .force) :: []) ,
  ──────────────────
  x :: xs)

allPosIS : IS U
allPosIS .Names = allPosRN
allPosIS .rules allP-Λ = from allP-Λ-r
allPosIS .rules allP-t = from allP-t-r
```

This inference system is expected to define exactly the lists such that all elements are positive, that is, those satisfying the following specification (where for simplicity, we use the predicate ∈, omitted, directly defined inductively).

```
allPosSpec : U → Set
allPosSpec xs = ∀ {x} → x ∈ xs → x > 0
```

As said in Sect. 2, to obtain the desired meaning this inference system has to be interpreted coinductively, and completeness can be proved by the coinduction principle, that is, by providing a proof that the specification is consistent with respect to the inference system, as shown below.

```
allPos : U → Set
allPos = CoInd⟦ allPosIS ⟧

allPosSpecCons : ∀ {xs} → allPosSpec xs → ISF[ allPosIS ] allPosSpec xs
allPosSpecCons {[]} _ = allP-Λ , (tt , (refl , tt , λ ()))
allPosSpecCons {(x :: xs)} Sxs =
  allP-t ,
  ((x , xs) , (refl , (Sxs here , λ {Fin.zero → λ mem → Sxs (there mem)})))

allPosComplete : allPosSpec ⊆ allPos
allPosComplete = coind[ allPosIS ] allPosSpec allPosSpecCons
```

For soundness there is no canonical technique; in this example, when the colist is empty the proof that the specification holds is trivial. If the colist is not empty, then the proof proceeds by induction on the position of the element to be proved to be positive.

Finally, for `maxElem`, the universe are pairs of natural numbers and possibly infinite lists.

```
U : Set
U = ℕ × Colist ℕ ∞
data maxElemRN : Set where max-h max-t : maxElemRN
data maxElemCoRN : Set where co-max-h : maxElemCoRN

max-h-r : FinMetaRule U
max-h-r .Ctx = Σ[ (_ , xs) ∈ ℕ × Thunk (Colist ℕ) ∞ ] xs .force ≡ []
max-h-r .comp ((x , xs) , _) =
  [] ,
  ──────────────
  x , x :: xs

max-t-r : FinMetaRule U
max-t-r .Ctx =
  Σ[ (x , y , z , _) ∈ ℕ × ℕ × ℕ × Thunk (Colist ℕ) ∞ ] z ≡ max x y
max-t-r .comp ((x , y , z , xs) , _) =
  (x , xs .force) :: [] ,
  ──────────────
  z , y :: xs

co-max-h-r : FinMetaRule U
co-max-h-r .Ctx = ℕ × Thunk (Colist ℕ) ∞
co-max-h-r .comp (x , xs) =
  [] ,
  ──────────────
  (x , x :: xs)

maxElemIS : IS U
maxElemIS .Names = maxElemRN
maxElemIS .rules max-h = from max-h-r
maxElemIS .rules max-t = from max-t-r

maxElemCoIS : IS U
maxElemCoIS .Names = maxElemCoRN
maxElemCoIS .rules co-max-h = from co-max-h-r
```

Note that in this example we have defined two inference systems, the rules and the corules. This generalized inference system is expected to define exactly the pairs `(x , xs)` such that `x` is the maximal element of `xs`, that is, those satisfying the following specification, where to be the maximal element `x` should belong to `xs`, and be greater or equal than any `n` in `xs`.

```
maxSpec inSpec geqSpec : U → Set
inSpec (x , xs) = x ∈ xs
geqSpec (x , xs) = ∀{n} → n ∈ xs → x ≡ max x n
maxSpec u = inSpec u × geqSpec u
```

As said in Sect. 3, the desired meaning is provided by the interpretation of the generalized inference system.

```
_maxElem_ : ℕ → Colist ℕ ∞ → Set
x maxElem xs = FCoInd⟦ maxElemIS , maxElemCoIS ⟧ (x , xs)
```

and completeness can be proved by the bounded coinduction principle, see (bcoind) at page 4.

```
maxElemComplete : ∀{x xs} → maxSpec (x , xs) → x maxElem xs
maxElemComplete =
  bounded-coind[ maxElemIS , maxElemCoIS ] maxSpec
    (λ{(x , xs) → maxSpecBounded x xs}) λ{(x , xs) → maxSpecCons x xs}
```

Notably, we have to prove that the specification is:

- bounded, that is, contained in `_maxElem`$_i$`_`, the inductive interpretation of the standard inference system consisting of both rules and corules, as shown below:

  ```
  _maxElemᵢ_ : ℕ → Colist ℕ ∞ → Set
  x maxElemᵢ xs = Ind⟦ maxElemIS ∪ maxElemCoIS ⟧ (x , xs)

  maxSpecBounded : ∀{x xs} → inSpec (x , xs)
    → geqSpec (x , xs) → x maxElemᵢ xs
  ```

- consistent with respect to the inference system consisting of only rules, as shown below:

  ```
  maxSpecCons : ∀{x xs} → inSpec (x , xs) →
    geqSpec (x , xs) → ISF[ maxElemIS ] maxSpec (x , xs)
  ```

These proofs are omitted.

For soundness there is no canonical technique. The proof can be split for the two components of the specification. It is worth noting that, for soundness with respect to `inSpec`, we first use `fcoind-to-ind` at page 9, and then define `maxElemSound-in-ind`, omitted, by induction on the inference system consisting of rules and corules. The use of `fcoind-to-ind` in the proof corresponds to the fact that without corules unsound judgments could be derived, see Sect. 3.

```
maxElemSound-in : ∀ {x xs} → x maxElem xs → inSpec (x , xs)
maxElem-sound-in max = maxElemSound-in-ind (fcoind-to-ind max)
```

Soundness with respect to `geqSpec` is proved by induction on the position, that is, the proof of membership, of the element that must be proved to be less or equal. In this case, soundness would hold even in the purely coinductive case.

6 A worked example

We describe a more significant example of instantiation: an inference system with corules providing a big-step semantics of lambda-calculus including divergence among the possible results [9], reported in Fig. 1. In this example, corules play a key role: indeed , considering, e.g., the divergent term $\Omega = (\lambda x.x)\,(\lambda x.x)$, in the standard inductive big-step semantics no result can be derived (an infinite proof tree is needed), as for a stuck term; in the purely coinductive interpretation, any judgment $\Omega \Downarrow v^\infty$ would be obtained [19]. Since each node of the infinite proof tree for a judgment should also have a finite proof tree using the corules, the coaxiom (coa) forces to obtain only ∞ as result, see [9] for a more detailed explanation.[2]

$$\begin{array}{rcll} t & ::= & v \mid x \mid t_1\, t_2 \mid \ldots & \text{term} \\ v & ::= & \lambda x.t \mid \ldots & \text{value} \\ v^\infty & ::= & v \mid \infty & \text{result} \end{array}$$

$$\text{(coa)}\ \frac{}{e \Downarrow \infty} \qquad \text{(val)}\ \frac{}{v \Downarrow v}$$

$$\text{(app)}\ \frac{t_1 \Downarrow \lambda x.t \quad t_2 \Downarrow v \quad t[x/v] \Downarrow v^\infty}{t_1\ t_2 \Downarrow v^\infty}$$

$$\text{(l-div)}\ \frac{t_1 \Downarrow \infty}{t_1\ t_2 \Downarrow \infty} \qquad \text{(r-div)}\ \frac{t_1 \Downarrow v \quad t_2 \Downarrow \infty}{t_1\ t_2 \Downarrow \infty}$$

Figure 1 λ-calculus: syntax and big-step semantics.

In rule (app), v^∞ is used for the result, so the rule also covers the case when the evaluation of the body of the lambda abstraction diverges. As usual, $t[x/v]$ denotes capture-avoiding substitution. Rules (l-div) and (r-div) cover the cases when either t_1 or t_2 diverges, assuming a left-to-right evaluation strategy.

Terms, values, and results are inductively defined, hence encoded by Agda datatypes. As customary in implementations of lambda-calculus, we use the De Bruijn notation: notably, `Term n` is the set of terms with `n` free variables.

```
data Term (n : ℕ) : Set where
  var : Fin n → Term n
  lambda : Term (suc n) → Term n
  app : Term n → Term n → Term n

data Value : Set where lambda : Term 1 → Value

term : Value → Term 0
term (lambda x) = lambda x

data Value∞ : Set where
```

[2] Other examples of big-step semantic definitions with more sophisticated corules are given in [10, 7].

```
  res : Value → Value∞
  ∞ : Value∞
```

The universe consists of big-step judgments (pairs consisting of a term and a result).

```
U : Set
U = Term 0 × Value∞
```

The two inference systems of rules and corules are encoded below:

```
data BigStepRN : Set where val app l−div r−div : BigStepRN
data BigStepCoRN : Set where COA : BigStepCoRN

BigStepIS : IS U
BigStepIS .Names = BigStepRN
BigStepIS .rules val = from val-r
BigStepIS .rules app = from app-r
BigStepIS .rules L-DIV = from l-div-r
BigStepIS .rules R-DIV = from r-div-r

BigStepCoIS : IS U
BigStepCoIS .Names  = BigStepCoRN
BigStepCoIS .rules COA = from coa-r
```

where BigStepRN are the rule names, and each rule name has an associated element of FinMetaRule U. For instance, app-r is given below. The auxiliary function subst , omitted, implements capture-avoiding substitution.

```
app-r : FinMetaRule U
app-r .Ctx = Term 0 × Term 1 × Term 0 × Value × Value∞
app-r .comp (t1 , t , t2 , v , v∞) =
 (t1 , res (lambda t)) :: (t2 , res v) :: (subst t (term v) , v∞) :: [] ,
 ─────────────────────────
  (app t1 t2 , v∞)
```

The big-step semantics can be obtained as the interpretation of the generalized inference system, as shown below. We use the flavour with thunks.

```
_⇓_ : Term 0 → Value∞ → Size → Set
(t ⇓ v∞) i = SFCoInd⟦ BigStepIS , BigStepCoIS ⟧ (t , v∞) i

_⇓ᵢ_ : Term 0 → Value∞ → Set
t ⇓ᵢ v∞ = Ind⟦ BigStepIS ∪ BigStepCoIS ⟧ (t , v∞)
```

The second predicate (i stands for "inductive") models that a judgment has a finite proof tree in the inference system consisting of rules and coaxiom, and will be used in proofs.

Small-step semantics, reported in Fig. 2, can also be obtained appropriately instantiating

$$(\beta)\ \frac{}{(\lambda x.t)\ v \Rightarrow t[x/v]} \qquad (\text{l-app})\ \frac{t_1 \Rightarrow t_1'}{t_1\ t_2 \Rightarrow t_1'\ t_2} \qquad (\text{r-app})\ \frac{t_2 \Rightarrow t_2'}{v\ t_2 \Rightarrow v\ t_2'}$$

Figure 2 λ-calculus: small-step semantics.

the library. In this case, the universe consists of small-step judgments, which are pairs of terms. There is only one inference system, where SmallStepRN are the rule names, and each rule name has an associated element of FinMetaRule U.

```
U : Set
U = Term 0 × Term 0
```

```
data SmallStepRN : Set where β L-app R-app : SmallStepRN

SmallStepIS : IS U
SmallStepIS .Names = SmallStepRN
SmallStepIS .rules β = from β-r
SmallStepIS .rules L-app = from l-app-r
SmallStepIS .rules R-app = from r-app-r
```

For instance, β-r is given below.

```
β-r : FinMetaRule U
β-r .Ctx = Term 1 × Value
β-r .comp (t , v) =
  [] ,
  ──────────────────────────────────
  (app (lambda t) (term v) , subst t (term v))
```

The one-step relation $\Rightarrow$ is obtained as the inductive interpretation of the (standard) inference system. Then, finite computations are modeled by its reflexive and transitive closure $\Rightarrow^\star$, defined using Star in the Agda library, as shown below.

```
_⇒_ : Term 0 → Term 0 → Set
t ⇒ t' = Ind⟦ SmallStepIS ⟧ (t , t')

_⇒⋆_ : Term 0 → Term 0 → Set
_⇒⋆_ = Star _⇒_
```

Infinite computations, instead, are modeled by the relation $\Rightarrow^\infty$, coinductively defined by the meta-rule $\dfrac{t' \Rightarrow^\infty}{t \Rightarrow^\infty}\ t \Rightarrow t'$, encoded in Agda by thunks.

```
  data _⇒∞ : Term 0 → Size → Set where
    step : ∀ {t t' i} → t ⇒ t' → Thunk (t' ⇒∞) i → t ⇒∞ i
```

The proof of equivalence between big-step and small-step semantics is structured as follows, where $\mathcal{S} = \{\langle t, v\rangle \mid t \Rightarrow^\star v\} \cup \{\langle t, \infty\rangle \mid t \Rightarrow^\infty\}$.

Soundness

- **$t \Downarrow v$ implies $t \Rightarrow^\star v$** We use fcoind -to-ind at page 9, and then reason by induction on the judgment $t \Downarrow_{\mathrm{i}} v$. That is, we show that $t \Rightarrow^\star v$ is closed w.r.t. the inference system consisting of rules and corules. As already pointed out for the maxElem example, the use of fcoind -to-ind in the proof corresponds to the fact that, without the coaxiom (coa), unsound judgments would be derived, e.g., $\Omega \Downarrow v$ for $v \in$ Val.
- **$t \Downarrow \infty$ implies $t \Rightarrow^\infty$** This implication, instead, would hold even in the purely coinductive case. It can be proved from *progress* and *subject reduction* properties:
  - **Progress** $t \Downarrow \infty$ implies that there exists t' such that $t \Rightarrow t'$.
  - **Subject reduction** $t \Downarrow \infty$ and $t \Rightarrow t'$ implies $t' \Downarrow \infty$.

Completeness By bounded coinduction, see (bcoind) at page 4.

- **Boundedness**
  - **$t \Rightarrow^\star v$ implies $t \Downarrow_{\mathrm{i}} v$** By induction on the number of steps.
  - **$t \Rightarrow^\infty$ implies $t \Downarrow_{\mathrm{i}} \infty$** Trivial, since the coaxiom coa can be applied.
- **Consistency** We have to show that, for each $\langle t, v^\infty\rangle \in \mathcal{S}$, $\langle t, v^\infty\rangle$ is the consequence of a big-step rule where the premises are in $\mathcal{S}$ as well. We distinguish two cases.
  - **$t \Rightarrow^\star v$** By induction on the number of steps. If it is 0, then t is a value, hence we can use rule (val). Otherwise, t is an application, and we can use rule (app).
  - **$t \Rightarrow^\infty$** The term t is an application $t_1\, t_2$. We distinguish the following cases:
    - t_1 diverges, hence we can use rule (l-div)

- t_1 converges and t_2 diverges, hence we can use rule (r-div)
- both t_1 and t_2 converge, hence we can use rule (app).

Note that in this proof by cases we need to use the *excluded middle* principle, which is defined in the standard library, and postulated in our proof.

7 Conclusion

We have presented an Agda implementation of inference systems which, besides the standard inductive and coinductive interpretations, supports also flexible coinduction and the associated proof principle. The key feature is that the library allows the separation of the definitions from their semantics, thus enabling modular composition and reasoning. This is particularly useful for flexible coinduction, because the interpretation of a generalized inference system is just defined by mixing the inductive and the coinductive interpretations of two inference systems built from rules and corules.

Of course, as Agda supports both inductive and coinductive dependent types, one could directly write Agda code for inductive, coinductive and even flexible coinductive definitions of concrete examples. We have explored this possibility in [12]. However, in this way, the definition is hard-wired with its semantics, and, for flexible coinduction, one has to manually construct the interpretation by combining in the correct way an inductive and a coinductive type and to prove the bounded coinduction principle for each example. For instance, the definition of maxElem will look as follows:

```
data _maxElem_ : ℕ → CoList ℕ ∞ → Size → Set where
  max-h : ∀ {x xs i} →force xs ≡ [] → x maxElem (x :: xs) i
  max-t : ∀ {x y xs i} → Thunk (x maxElem (force xs)) i
                          → z ≡ max x y
                          → z maxElem_i (y :: xs)
                          → z maxElem (y :: xs) i

data _maxElem_i_ : ℕ → CoList ℕ ∞ → Set where
  imax-h : ∀ {x xs} →force xs ≡ [] → x maxElem_i (x :: xs)
  imax-t : ∀ {x y xs} → x maxElem_i (force xs)) → z ≡ max x y
                         → z maxElem_i (y :: xs)
  co-max-h : ∀ {x xs} → x maxElem_i (x :: xs)
```

Clearly, this approach causes duplication of rules and code, as rules of the coinductive type have to be duplicated in the inductive one, making things rather complex. Our library instead hides all these details, exposing interfaces for interpretations and proof principles, so that the user only has to write code describing rules.

For future work we plan to extend the library in several directions. The first one is to support other interpretations of inference systems, such as the *regular* one [14], which is basically coinductive but allows only proof trees with finitely many distinct subtrees.To this end, useful starting points are works on regular terms and streams [22, 24] and on finite sets [17] in dependent type theories. The challenging part is the finiteness constraint, which is not trivial in a type-theoretic setting. A second direction is to implement other proof techniques for (flexible) coinduction, as parametrized coinduction [18] and up-to techniques [20, 16]. Finally, another direction could be the development of a full framework for composition of inference systems, along the lines of seminal work on module systems [11]. On the more practical side, a further development is to transform the methodology in an automatic translation. That is, a user should be allowed to write an inference system (with corules) in a natural syntax, and the corresponding Agda types should be generated automatically, either by an external tool, or, more interestingly, using *reflection*, recently added in Agda.

References

1 Andreas Abel. Type-based termination, inflationary fixed-points, and mixed inductive-coinductive types. In Dale Miller and Zoltán Ésik, editors, *Proceedings 8th Workshop on Fixed Points in Computer Science, FICS 2012*, volume 77 of *EPTCS*, pages 1–11, 2012. doi:10.4204/EPTCS.77.1.

2 Andreas Abel and Brigitte Pientka. Well-founded recursion with copatterns and sized types. *Journal of Functional Programming*, 26:e2, 2016. doi:10.1017/S0956796816000022.

3 Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. Copatterns: programming infinite structures by observations. In Roberto Giacobazzi and Radhia Cousot, editors, *The 40th Annual ACM Symposium on Principles of Programming Languages, POPL'13*, pages 27–38. ACM Press, 2013. doi:10.1145/2429069.2429075.

4 Andreas Abel, Andrea Vezzosi, and Théo Winterhalter. Normalization by evaluation for sized dependent types. *Proceedings of ACM on Programming Languages*, 1(ICFP):33:1–33:30, 2017. doi:10.1145/3110277.

5 Peter Aczel. An introduction to inductive definitions. In Jon Barwise, editor, *Handbook of Mathematical Logic*, volume 90 of *Studies in Logic and the Foundations of Mathematics*, pages 739–782. Elsevier, 1977.

6 Thorsten Altenkirch, Neil Ghani, Peter G. Hancock, Conor McBride, and Peter Morris. Indexed containers. *Journal of Functional Programming*, 25, 2015. doi:10.1017/S095679681500009X.

7 Davide Ancona, Francesco Dagnino, Jurriaan Rot, and Elena Zucca. A big step from finite to infinite computations. *Science of Computer Programming*, 197:102492, 2020. doi:10.1016/j.scico.2020.102492.

8 Davide Ancona, Francesco Dagnino, and Elena Zucca. Generalizing inference systems by coaxioms. In Hongseok Yang, editor, *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017*, volume 10201 of *Lecture Notes in Computer Science*, pages 29–55, Berlin, 2017. Springer. doi:10.1007/978-3-662-54434-1_2.

9 Davide Ancona, Francesco Dagnino, and Elena Zucca. Reasoning on divergent computations with coaxioms. *Proceedings of ACM on Programming Languages*, 1(OOPSLA):81:1–81:26, 2017. doi:10.1145/3133905.

10 Davide Ancona, Francesco Dagnino, and Elena Zucca. Modeling infinite behaviour by corules. In Todd D. Millstein, editor, *32nd European Conference on Object-Oriented Programming, ECOOP 2018*, volume 109 of *LIPIcs*, pages 21:1–21:31, Dagstuhl, 2018. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.ECOOP.2018.21.

11 G. Bracha. *The Programming Language JIGSAW: Mixins, Modularity and Multiple Inheritance*. PhD thesis, Department of Comp. Sci., Univ. of Utah, 1992.

12 Luca Ciccone. Flexible coinduction in Agda. Master's thesis, University of Genova, 2019. URL: https://arxiv.org/abs/2002.06047.

13 Francesco Dagnino. Coaxioms: flexible coinductive definitions by inference systems. *Logical Methods in Computer Science*, 15(1), 2019. doi:10.23638/LMCS-15(1:26)2019.

14 Francesco Dagnino. Foundations of regular coinduction. Technical report, DIBRIS, University of Genova, June 2020. Submitted for journal publication. URL: https://arxiv.org/abs/2006.02887.

15 Francesco Dagnino. *Flexible Coinduction*. PhD thesis, DIBRIS, University of Genova, 2021.

16 Nils Anders Danielsson. Up-to techniques using sized types. *Proceedings of ACM on Programming Languages*, 2(POPL):43:1–43:28, 2018. doi:10.1145/3158131.

17 Denis Firsov and Tarmo Uustalu. Dependently typed programming with finite sets. In Patrick Bahr and Sebastian Erdweg, editors, *Proceedings of the 11th ACM Workshop on Generic Programming, WGP@ICFP 2015*, pages 33–44. ACM, 2015. doi:10.1145/2808098.2808102.

18 Chung-Kil Hur, Georg Neis, Derek Dreyer, and Viktor Vafeiadis. The power of parameterization in coinductive proof. In Roberto Giacobazzi and Radhia Cousot, editors, *The 40th Annual ACM Symposium on Principles of Programming Languages, POPL'13*, pages 193–206. ACM Press, 2013. doi:10.1145/2429069.2429093.

19 Xavier Leroy and Hervé Grall. Coinductive big-step operational semantics. *Information and Computation*, 207(2):284–304, 2009. doi:10.1016/j.ic.2007.12.004.

20 Damien Pous. Coinduction all the way up. In Martin Grohe, Eric Koskinen, and Natarajan Shankar, editors, *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS'16*, pages 307–316. ACM Press, 2016. doi:10.1145/2933575.2934564.

21 Davide Sangiorgi. *Introduction to Bisimulation and Coinduction*. Cambridge University Press, USA, 2011.

22 Régis Spadotti. *A mechanized theory of regular trees in dependent type theory. (Une théorie mécanisée des arbres réguliers en théorie des types dépendants)*. PhD thesis, Paul Sabatier University, Toulouse, France, 2016. URL: https://tel.archives-ouvertes.fr/tel-01589656.

23 The Agda Team. *The Agda Reference Manual*. URL: http://agda.readthedocs.io/en/latest/index.html.

24 Tarmo Uustalu and Niccolò Veltri. Finiteness and rational sequences, constructively. *Journal of Functional Programming*, 27:e13, 2017. doi:10.1017/S0956796817000041.