Practical Sized Typing for Coq

ANONYMOUS AUTHOR(S)

Termination of recursive functions and productivity of corecursive functions are important for maintaining logical consistency in proof assistants. However, contemporary proof assistants, such as Coq, rely on fragile syntactic criteria that prevent users from easily writing obviously terminating or productive functions, such as quicksort. This is troublesome, since there exist theories for type-based termination and productivity checking.

In this paper, we present a design and implementation of sized type checking and inference for Coq. We extend past work on sized types for the Calculus of (Co)Inductive Constructions (CIC) to support definitions, and extend the sized type inference algorithm to support completely unannotated CIC terms. This allows our design to maintain complete backward compatibility with existing Coq developments. We provide an implementation that extends the Coq kernel with optional support for sized types.

1 INTRODUCTION

Proof assistants based on dependent type theory rely on the termination of recursive functions and the productivity of corecursive functions to ensure two important properties: logical consistency, so that it is not possible to prove false propositions; and decidability of type checking, so that checking that a program proves a given proposition is decidable.

In proof assistants such a Coq, termination and productivity are enforced by a *guard predicate* on fixpoints and cofixpoints respectively. For fixpoints, recursive calls must be *guarded by destructors*; that is, they must be performed on structurally smaller arguments. For cofixpoints, corecursive calls must be *guarded by constructors*; that is, they must be the structural arguments of a constructor. The following examples illustrate these structural conditions.

```
Fixpoint plus n m : nat :=
  match n with
  | O => m
  | S p => S (plus p m)
  end.
Variable A : Type.
CoFixpoint const a : Stream A := Cons a (const a).
```

In the recursive call to plus, the first argument p is structurally smaller than S p, which is the form of the original first argument n. Similarly, in const, the constructor Cons is applied to the corecursive call.

The actual implementation of the guard predicate extends beyond the guarded-by-destructors and guarded-by-constructors conditions to accept a larger set of terminating and productive functions. In particular, function calls will be unfolded (*i.e.*, inlined) in the bodies of (co)fixpoints as needed before checking the guard predicate. This has a few disadvantages: firstly, the bodies of these functions are required, which hinders modular design; and secondly, the (co)fixpoint bodies may become very large after unfolding, which can decrease the performance of type checking.

Furthermore, changes in the structural form of functions used in (co)fixpoints can cause the guard predicate to reject the program even if the functions still behave the same. The following simple example, while artificial, illustrates this structural fragility.

```
Fixpoint minus n m :=
  match n, m with
  | O, _ => n
  | _, O => n
  | S n', S m' => minus n' m'
  end.
```

```
Fixpoint div n m :=
  match n with
  | O => O
  | S n' => S (div (minus n' m) m)
  end.
```

If we replace `| O, _ => n` with `| O, _ => O` in `minus`, the behaviour does not change, but `O` is not a structurally smaller term of `n` in the recursive call to `div`, so `div` no longer satisfies the guard predicate. The acceptance of `div` then depends on a function external to it, which can lead to difficulty in debugging for larger programs. Furthermore, the guard predicate is unaware of the obvious fact that `minus` never returns a `nat` larger than its first argument, which the user would have to prove in order for `div` to be accepted with our alternate definition of `minus`.

An alternative to guard predicates for termination and productivity enforcement uses *sized types*. In essence, the (co)inductive type of an object is annotated with a size annotation, which provides some information about the size of the object. In this paper, we follow a simple size algebra: $s := \upsilon \mid \hat{s} \mid \infty$, where υ ranges over size variables. If the argument to a constructor has size s, then the fully-applied constructor would have a successor size $\hat{s}$. For instance, the `nat` constructors follow the below rules:

$$\frac{}{\Gamma \vdash \mathrm{O} : \mathrm{Nat}^{\hat{s}}} \qquad \frac{\Gamma \vdash n : \mathrm{Nat}^{s}}{\Gamma \vdash \mathrm{S}\ n : \mathrm{Nat}^{\hat{s}}}$$

Termination and productivity checking is then simply a type checking rule that uses size information. For termination, the recursive call must be done on an object with a smaller size, so when typing the body of the fixpoint, the reference to itself in the typing context must have a smaller size. For productivity, the returned object must have a larger size than that of the corecursive call, so the type of the body of the cofixpoint must be larger than the type of the reference to itself in the typing context. In short, they both follow the following (simplified) typing rule, where υ is an arbitrary fresh size variable annotated on the inductive types, and s is an arbitrary size expression as needed.

$$\frac{\Gamma(f : t^{\upsilon}) \vdash e : t^{\hat{\upsilon}}}{\Gamma \vdash (\mathrm{co})\mathrm{fix}\ f : t := e : t^{s}}$$

We can then assign `minus` the type $\mathrm{Nat}^{\iota} \to \mathrm{Nat} \to \mathrm{Nat}^{\iota}$. The fact that we can assign it a type indicates that it will terminate, and the ι annotations indicate that the function preserves the size of its first argument. Then `div` uses only the type of `minus` to successfully type check, not requiring its body. Furthermore, being type-based and not syntax-based, replacing `| O, _ => n` with `| O, _ => O` does not affect the type of `minus` or the typeability of `div`. Similarly, some other (co)fixpoints that preserve the size of arguments in ways that aren't syntactically obvious may be typed to be size preserving, expanding the set of terminating and productive functions that can be accepted.

Unfortunately, past work on sized types [Barthe et al. 2006; Sacchini 2011] in the Calculus of (Co)Inductive Constructions (CIC), Coq's underlying calculus, have some practical issues:

- They require nontrivial backwards-incompatible additions to the surface language. These include annotations that mark the positions of (co)recursive and size-preserved types, and polarity annotations on (co)inductive definitions that describe how subtyping works with respect to parameters.
- They require the (co)recursive arguments of (co)fixpoints to have literal (co)inductive types. That is, the types cannot be expressions that convert to (co)inductive types.

- Their languages do not support local and global definitions, which Coq includes.

In this paper, we present $\mathrm{CIC}\widehat{*}$ ("*CIC-star-hat*"), a calculus for representing the core of Coq with sized types, and an inference algorithm from CIC to $\mathrm{CIC}\widehat{*}$. It is an extension of $\widehat{\mathrm{CIC}}$ [Barthe et al. 2006] and $\mathrm{CIC}\widehat{\_}$ [Sacchini 2011] that resolves the issues above. We also have a fork of Coq available in the anonymous supplementary material that implements the size inference algorithm and size-based termination and productivity checking of $\mathrm{CIC}\widehat{*}$. To maximize backward compatibility, the surface language is completely unchanged, and the existing guard condition and sized types can be enabled or disabled independently, or used in conjunction. Sized typing can be turned on with a flag that is off by default for safe and gradual testing of the new kernel. We are currently working with the Coq development team to merge the work into Coq.

The remainder of this paper is organized as follows. We begin in Section 2 with a high-level overview of the design of $\mathrm{CIC}\widehat{*}$, its role in Coq, and the main lessons from our design. We formalize the calculus $\mathrm{CIC}\widehat{*}$ in Section 3. In Section 4, we present a size inference algorithm from CIC terms to sized $\mathrm{CIC}\widehat{*}$ terms that details how we annotate the types of (co)fixpoints, how we deal with the lack of polarities, and how definitions are supported, and termination and productivity checking. Section 5 states some of the metatheoretical properties of $\mathrm{CIC}\widehat{*}$ that have been proven or remain to be proven. Finally, we briefly compare with the past work on sized types for CIC and related languages in Section 6.

2 OVERVIEW

Our goal in the design of $\mathrm{CIC}\widehat{*}$ is to balance backward compatibility with performance. We could achieve backward compatibility easily by *just* using an extremely expressive size algebra, but we could never implement an efficient type checker. Similarly, we could easily add sized types to Coq with efficient checking if we *just* make the users and the Coq developers rewrite all their code in our cool new annotated language. Both of these are impractical. Instead, we try to thread the needle.

Our first design decision is complete backward compatibility: the Coq user must not be required to provide new annotations on existing code, and ideally should be able to get the advantages of sized typing in all new code. If we expect the user to reuse any standard library data types with sized types, this means that we need a *size inference* algorithm, which takes ordinary Coq programs, infers size annotations as needed, uses sizes during type checking, and returns ordinary Coq programs.

For performance, we want size inference to be local — size variables and constraints should be independent from one global declaration to another. In theory, we have an infinite set of unique size variables that can be summoned at will, but in practice, each variable needs to be generated and tracked, consuming time and space. Similarly, we could add an unrestricted number of constraints, but as we discuss in Section 4, the time complexity of size inference is proportional to the number of constraints. By keeping size inference local, the state of size information and size constraint sets can be kept smaller.

Below is a high-level view of our type checking process. The pipeline is local, *i.e.*, each top-level definition in a Coq program is run through this pipeline in turn.

$$\mathrm{CIC} \equiv \text{bare } \mathrm{CIC}\widehat{*} \xrightarrow{\text{inference}} \text{sized } \mathrm{CIC}\widehat{*} \xrightarrow{\text{erasure}} \text{limit } \mathrm{CIC}\widehat{*}$$

The first pass elaborates Coq code into a fully type-annotated CIC term. This is a standard pass for Coq which we will not discuss further. Naturally, the CIC term will have no size annotations; we also call this *bare* $\mathrm{CIC}\widehat{*}$.

Next, we perform size inference on bare CIC$\widehat{*}$ to obtain *sized* CIC$\widehat{*}$. There are three different tasks in size inference: (1) annotate all (co)inductive types with fresh size variables; (2) during type checking, whenever two terms are compared, follow subtyping rules to generate a set of constraints between their size annotations; and (3) when type checking a (co)fixpoint, check that the collected constraints are satisfiable. By representing the constraints as a weighted directed graph, this amounts to checking for negative cycles.

The final step in the pipeline is to erase unnecessary size annotations from sized CIC$\widehat{*}$ to obtain *limit* CIC$\widehat{*}$ by replacing size annotations with ∞. This is the same strategy adopted in a prototype implementation of CIC$\widehat{_}$ [Sacchini 2015a].

However, not all size annotations are erased. We preserve annotations for each *size-preserving* (co)recursive function, which is a key feature of CIC$\widehat{*}$ that enables additional expressiveness in termination and productivity checking. Consider, for example, an inductive list of type $\text{List}^r\ A$ and a coinductive stream of type $\text{Stream}^s\ A$. The intuitive notion of the size r is the length of the list. Since every list of As is an inhabitant of $\text{List}^\infty\ A$, we can imagine that the inhabitants of $\text{List}^r\ A$ are lists of length r or shorter. Dually, the inhabitants of $\text{Stream}^s\ A$ are streams of "length" s or "longer". From the perspective of productivity, these are streams that can produce at least s elements. A recursive function on lists that is size-preserving, then, is one that returns a list of equal or smaller size, while a size-preserving corecursive function on streams is one that returns a stream of equal or larger size. For instance, a `map` function over lists or streams is size-preserving, since it does not modify their lengths. Whether a (co)recursive function is size-preserving or not is determined during the size inference step, which annotates the types of (co)fixpoints with position annotations $*$ to mark the (co)recursive argument type as well as any size-preserving return types. These annotations are not erased.

This alone is not enough to express size-preservation of global declarations. A globally-defined (co)fixpoint is annotated with the *definition*'s type, which doesn't expose the size-preservedness expressed by the *(co)fixpoint*'s type. Therefore, we also annotate the types of such definitions with global annotations ι to mark what would have been position annotations.

The examples `filter` and `qsort` below demonstrate what limit CIC$\widehat{*}$ programs look like after erasure. During the inference step for `qsort`, the global annotations on `filter` are substituted by the *same* size variable, which tells `qsort` that `filter` preserves the size of the recursive argument, allowing us to use it in the recursive call. Global annotations then are essentially a limited form of size polymorphism with one universal quantifier, which is sufficient to express size preservation.

```
Def filter: (Nat^∞ → Bool^∞) →
  List^ι Nat^∞ → List^ι Nat^∞ :=
  fix filter': (Nat → Bool) →
    List^* Nat → List^* Nat :=
    λp: Nat → Bool. λl: List Nat.
    case l return List Nat of
    | Nil ⇒ Nil
    | Cons ⇒ λhd: Nat. λtl: List Nat.
      if p hd
      then Cons Nat hd (filter' p tl)
      else (filter' p tl)
    end.
```

```
(* [append], [gtb], [leb] omitted *)
Def qsort: List^ι Nat^∞ → List^∞ Nat^∞ :=
  fix qsort': List^* Nat →
    List Nat := λl : List Nat.
    case l return List Nat of
    | Nil ⇒ Nil
    | Cons ⇒ λhd: Nat. λtl: List Nat.
      append
      (qsort' (filter (gtb hd) tl))
      (Cons Nat hd
        (qsort' (filter (leb hd) tl)))
    end.
```

There is a second problem with erasure. Consider the following CIC$\widehat{*}$ program.

```
Def N: Type := Nat∞.
Def add: Nᶥ → N → N :=
  let id: N → N := λn: N. n in
  fix add': N* → N → N := λn: N. λm: N.
    case n return N of
    | O ⇒ m
    | S ⇒ λn': N. S (add' (id n') m)
    end.
```

N will reduce to Nat during type checking, but what size should Nat then have? It cannot have the limit size ∞ left after erasure; this would disallow us from using id in the recursive call to add', since termination checking requires that id preserve the size of n' and not return some *larger* Nat^{∞}. However, we cannot *not* erase, either, leaving Nat with some arbitrary fixed size annotation, since this makes add's type size-preserving when add is not. To handle this example, each instance of N should have its own fresh size annotation; during reduction, this becomes a size-annotated Nat.

We support this kind of program in $\text{CIC}\widehat{*}$ by treating global definitions essentially as implicitly abstracting over size expressions. Each instance of a variable bound to a definition needs to be instantiated with the correct number of size expressions, and so carries a vector of size expressions whose length is the number of ∞ annotations in the body after erasure. Like size annotations, these new vector annotations are only found in sized $\text{CIC}\widehat{*}$.

A final design decision remains to enable backward compatibility. Our sized typing enables many new programs to type check easily, such as qsort above, but our limited sized algebra means that there also exist programs that pass guard checking but not our sized typing pipeline. Guard checking unfolds definitions, which is bad for modularity and performance, but enables gcd as defined in the Coq standard library to type check using guard checking. On the other hand, gcd cannot be type checked using sized types with our size algebra[1]. We could enrich the size algebra, but as we discuss in Section 6, this greatly increases the time complexity of size inference. To take advantage of both schemas, our implementation enables each to be used simultaneously, so the type checker accepts a program if it passes either sized typing *or* guard checking:

```
Set Guard Checking. Set Sized Typing.
```

3 $\text{CIC}\widehat{*}$

In this section, we present $\text{CIC}\widehat{*}$, a core calculus for sized typing in Coq.

3.1 Syntax

Figure 1 presents the syntax of $\text{CIC}\widehat{*}$, whose terms are parameterized over a set of annotations α, which indicate the kind of annotations (if any) that appear on the term; details will be provided shortly. We draw variables from several distinct sets of variable names: $\mathcal{X}$ for term variable names, $\mathcal{V}$ for size variable names, $\mathcal{I}$ for (co)inductive type names, and $\mathcal{C}$ for (co)inductive constructor names. The brackets $\langle\cdot\rangle$ delimit a vector of some comma-separated constructions. For instance, if C, T are nonterminals, then $\langle C \Rightarrow T\rangle$ expands to $\langle C \Rightarrow T, \ldots, C \Rightarrow T\rangle$. We use m and n as well as i, j, k, ℓ as metavariables for positive naturals used in indexing; note that we use 1-based indexing.

$\text{CIC}\widehat{*}$ resembles the usual CIC, but there are some important differences compared to CIC and compared to past work $\widehat{\text{CIC}}$ and $\text{CIC}\widehat{\_}$:

- **Inductive types** carry annotations that represent their size, *e.g.*, Nat^{υ}. This is the defining feature of sized types. They can also have position annotations, *e.g.*, Nat^{*}, which mark the

[1]https://github.com/coq/coq/wiki/CoqTerminationDiscussion#sized

$$\begin{array}{rlr}
S & ::= \mathcal{V} \mid \mathcal{V}^* \mid \widehat{S} \mid \infty & \text{size expressions} \\
U & ::= \text{Prop} \mid \text{Set} \mid \text{Type}_n & \text{set of universes}
\end{array}$$

$$\begin{array}{lr}
T[\alpha] ::= & \\
\quad \mid U & \textit{universes} \\
\quad \mid \mathcal{X} \mid \mathcal{X}^{\langle \alpha \rangle} & \textit{variables} \\
\quad \mid \lambda \mathcal{X} : T^{\circ}. T[\alpha] & \textit{abstractions} \\
\quad \mid T[\alpha] T[\alpha] & \textit{applications} \\
\quad \mid \Pi \mathcal{X} : T[\alpha]. T[\alpha] & \textit{function types} \\
\quad \mid \text{let } \mathcal{X} : T^{\circ} := T[\alpha] \text{ in } T[\alpha] & \textit{let expressions} \\
\quad \mid \mathcal{I}^{\alpha} & \textit{(co)inductive types} \\
\quad \mid C & \textit{(co)ind. constructors} \\
\quad \mid \text{case}_{T^{\circ}}\ T[\alpha] \text{ of } \langle C \Rightarrow T[\alpha] \rangle & \textit{case expressions} \\
\quad \mid \text{fix}_{\langle n \rangle, m}\ \langle \mathcal{X} : T^* := T[\alpha] \rangle & \textit{fixpoints} \\
\quad \mid \text{cofix}_{m}\ \langle \mathcal{X} : T^* := T[\alpha] \rangle & \textit{cofixpoints}
\end{array}$$

Fig. 1. Syntax of $\text{CIC}\widehat{*}$ terms with annotations α

$$\begin{array}{rlrrlr}
T^{\circ} & ::= T[\{\epsilon\}] & \text{bare terms} & T^{\infty} & ::= T[\{\infty\}] & \text{limit terms} \\
T^{*} & ::= T[\{\epsilon, *\}] & \text{position terms} & T^{\iota} & ::= T[\{\infty, \iota\}] & \text{global terms} \\
T & ::= T[S] & \text{sized terms} & & &
\end{array}$$

Fig. 2. Kinds of annotated terms

type as that of the recursive argument of a fixpoint or the return type of a cofixpoint, as well as other size-preserving types. This is similar to `struct` annotations in Coq that specify the structurally-recursive argument.
- **Variables** may have a vector of annotations, *e.g.*, $x^{\langle v_1, v_2 \rangle}$. If the variable is bound to a term containing (co)inductive types, we assign the annotations to each (co)inductive type during reduction. For instance, if x is defined by $x : \text{Set} := \text{List Nat}$, then $x^{\langle v_1, v_2 \rangle}$ reduces to $\text{List}^{v_1}\ \text{Nat}^{v_2}$.
- **Definitions** are supported, in constrast to $\text{CIC}\widehat{\ }$ and $\text{CIC}\widehat{\_}$. This reflects the actual structure in Coq's kernel.
- **Mutual (co)fixpoints** are treated explicitly. In fixpoints, $\langle n_k \rangle$ is a vector of indices indicating the positions of the recursive arguments in each fixpoint type, and m picks out the mth (co)-fixpoint in the vector of mutual definitions.

Figure 2 lists shorthand for the kinds of annotated terms that we use, with ϵ indicating a lack of annotations. From $\text{CIC}\widehat{\_}$ and $\text{CIC}\widehat{\ }$, we have bare terms, which are necessary for subject reduction [Sacchini 2011]; position terms, which have asterisks to mark the types in (co)fixpoint types with at most (for fixpoints) or at least (for cofixpoints) the same size as that of the (co)recursive argument; and sized terms, used for termination and productivity checking. We also have limit terms, which occur after erasure, and global terms, which occur in the types of global declarations similarly to how position terms occur in (co)fixpoint types. These terms correspond to bare, sized, and limit $\text{CIC}\widehat{*}$: we begin with user-provided declarations as bare terms, produce size and position annotations during size inference while verifying termination and productivity, and finish by

$D[\alpha] ::=$	local declarations
$\mid \mathcal{X} : T[\alpha]$	*local assumption*
$\mid \mathcal{X} : T[\alpha] := T[\alpha]$	*local definition*
$D_G ::=$	global declarations
$\mid \text{Assum } \mathcal{X} : T^{\infty}.$	*global assumption*
$\mid \text{Def } \mathcal{X} : T^{\iota} := T^{\infty}.$	*global definition*
$\Gamma ::= \square \mid \Gamma(D[S])$	local environments
$\Gamma_G ::= \square \mid \Gamma_G(D_G)$	global environment
$\Delta[\alpha] ::= \square \mid \Delta[\alpha](\mathcal{X} : T[\alpha])$	assumption environments

Fig. 3. Declarations and environments

$e, a, b, p, q, \wp \in T[\alpha]$ (expressions)	$\tau, \upsilon \in \mathcal{V}$	$w \in U$
$t, u, v \in T[\alpha]$ (types)	$V \in \mathbb{P}(\mathcal{V})$	$I \in \mathcal{I}$
$f, g, h, x, y, z \in \mathcal{X}$	$r, s \in S$	$c \in C$

Fig. 4. Metavariables

$\text{(co)dom}(\Delta) \mapsto \overline{x}$	(co)domain of assumptions
$e\ \overline{a} \mapsto ((e\ a_1)\ldots)\ a_n$	multiple application
$t \to u \mapsto \Pi\_ : t.\, u$	nondependent function type
$(x : t) \to u \mapsto \Pi x : t.\, u$	dependent function type
$\Pi\Delta.\, t \mapsto \Pi x_1 : t_1.\ \ldots\ \Pi x_n : t_n.\, t$	product from assumptions
$\mathsf{SV}(e_1, e_2) \mapsto \mathsf{SV}(e_1) \cup \mathsf{SV}(e_2)$	size variables of terms
$\mathsf{SV}(\overline{a}) \mapsto \mathsf{SV}(a_1) \cup \cdots \cup \mathsf{SV}(a_n)$	size variables of terms
$\mathsf{SV}(\Delta) \mapsto \mathsf{SV}(\overline{t})$	size variables of assumptions
where $\overline{a} = a_1 \ldots a_n$	
$\Delta = (x_1 : t_1) \ldots (x_n : t_n)$	

Fig. 5. Syntactic sugar for terms and metafunctions

erasing sized terms to limit and global terms.

$$T^{\circ} \xrightarrow{\text{inference}} T, T^{*} \xrightarrow{\text{erasure}} T^{\infty}, T^{\iota}$$

Figure 3 illustrates the difference between *local* and *global* declarations and environments, a distinction also in the Coq kernel. Local assumptions and definitions occur in abstractions and let expressions, respectively, while global ones are declared at the top level. Local declarations and assumption environments are parameterized over a set of annotations α; we use the same shorthand for environments as for terms.

Figure 4 lists the metavariables we use in this work, which may be subscripted by n, m, i, j, k, ℓ, or natural number literals, or superscripted by $'$. We use the overline $\overline{\cdot}$ to denote a sequence of some construction; if it contains an index, the sequence spans the range of the index, usually given

$$\begin{aligned} \mathit{Ind} &::= \Delta \vdash \langle \mathcal{I}\ \overline{\mathcal{X}} : \Pi\Delta^{\infty}. U \rangle \coloneqq \langle \mathcal{C} : \Pi\Delta^{\infty}. \mathcal{I}\ \overline{\mathcal{X}}\ \overline{T^{\infty}} \rangle \\ \Sigma &::= \square \mid \Sigma(\mathit{Ind}) \end{aligned}$$

$$\Delta_p \vdash \langle I_i\ \mathsf{dom}(\Delta_p) : \Pi\Delta_i. w_i \rangle \coloneqq \langle c_j : \Pi\Delta_j. I_j\ \mathsf{dom}(\Delta_p)\ \bar{t}_j \rangle$$

Fig. 6. (Co)inductive definitions and signature

implicitly. For instance, given i inductive types, $\overline{I_k^{s_k}} = I_1^{s_1} \dots I_i^{s_i}$. Notice that this is *not* the same as an index outside of the overline, such as in $\bar{a}_k$, which represents the kth sequence of terms a. Indices also appear in syntactic vectors; for example, given a case expression with j branches, we write $\langle c_\ell \Rightarrow e_\ell \rangle$ for the vector $\langle c_1 \Rightarrow e_1, \dots, c_j \Rightarrow e_j \rangle$.

Figure 5 defines syntactic sugar on terms, most of which is standard.

We use $t[x \coloneqq e]$ to denote the term t with free variable x substituted by expression e, and $t[v \coloneqq s]$ to denote the term t with size variable v substituted by size expression s. Additionally, we use $t\overline{[\infty_i \coloneqq s_i]}$ to denote the substitutions of all ∞ annotations in t by the size expressions $\overline{s_i}$ in left-to-right order. The substitution is valid only if the number of ∞ annotations in t is same as the length of $\overline{s_i}$.

3.1.1 *Mutual (Co)Inductive Definitions.*

The definition of mutual (co)inductive types and their constructors are stored in a global signature Σ as defined in Figure 6. (Typing judgements are parameterized by all three of Σ, Γ_G, Γ.) A mutual (co)inductive definition contains:

- Δ_p, the parameters of the (co)inductive types and the constructors;
- I_i, the names of the (co)inductive types;
- Δ_i, the indices (or arguments) of I_i;
- w_i, the universes to which I_i belongs;
- c_j, the names of the constructors;
- Δ_j, the arguments of c_j;
- I_j, the (co)inductive types of the fully-applied constructors; and
- $\bar{t}_j$, the indices of I_j.

Given a constructor c_j, we will often refer to I_j as simply that constructor's inductive type. Note that I_j is *not* the jth inductive type in the definition, but rather the specific inductive type associated with the jth constructor. We would more precisely write I_{k_j}, to indicate that we pick out the k_jth inductive type, where the specific k depends on j, but we forgo this notation for clarity.

As an example, the usual Vector type would be defined in the language as (omitting $\square$ in nonempty environments and brackets in the syntax for singleton vectors):

$$\begin{aligned} (A : \mathsf{Type}) \vdash \mathsf{Vector}\ A : \mathsf{Nat} \to \mathsf{Type} \coloneqq \langle &\mathsf{VNil} : \mathsf{Vector}\ A\ \mathsf{O}, \\ &\mathsf{VCons} : (n : \mathsf{Nat}) \to A \to \mathsf{Vector}\ A\ n \to \mathsf{Vector}\ A\ (\mathsf{S}\ n) \rangle. \end{aligned}$$

As with mutual (co)fixpoints, we treat mutual (co)inductive definitions explicitly. Furthermore, in contrast to $\widehat{\text{CIC}}$ and $\underline{\widehat{\text{CIC}}}$, our definitions do not have a vector of polarities. In those works, each parameter has an associated polarity that tells us whether the parameter is covariant, contravariant, or invariant with respect to the (co)inductive type during subtyping. Since Coq's (co)inductive definitions do not have polarities, we forgo them so that our type checker can work with existing Coq code without modification. Consequently, we will see that the parameters of (co)inductive types are always invariant in the subtyping Rule ST-APP.

As usual, the well-formedness of (co)inductive definitions depends on certain syntactic conditions such as strict positivity. The conditions are defined in the supplementary material and reproduced in Appendix B. We refer the reader to clauses I1–I9 in Sacchini [2011], clauses 1–7 in

$$\frac{(x : t := e) \in \Gamma}{\Gamma_G, \Gamma \vdash x^{\langle s_i \rangle} \rhd_\delta |e|^\infty \overline{[\infty_i := s_i]}}\ \delta\text{-LOCAL} \qquad \frac{(\text{Def } x : t := e.) \in \Gamma_G}{\Gamma_G, \Gamma \vdash x^{\langle s_i \rangle} \rhd_\Delta e\overline{[\infty_i := s_i]}}\ \Delta\text{-GLOBAL}$$

Fig. 7. Reduction rules for local and global definitions

Barthe et al. [2006], and The Coq Development Team [2020] for further details. Generally, we can assume that non-nested (co)inductive definitions that are valid in Coq are valid in $\text{CIC}\widehat{*}$ as well.

Note that nested (co)inductive types are *not* supported in $\text{CIC}\widehat{*}$, as they break subject reduction (see Section 5 for details). This restriction manifests in the definition of strict positivity.

3.1.2 Metafunctions. We declare the following metafunctions:

- $\mathsf{FV} : T[\alpha] \to \mathbb{P}(\mathcal{X})$ returns the set of free term variables in the given term;
- $\mathsf{SV} : T \to \mathbb{P}(\mathcal{V})$ returns the set of size variables in the given sized term;
- $\lfloor \cdot \rfloor : S \setminus \{\infty\} \to \mathcal{V}$ returns the size variable in the given finite size expression;
- $\|\cdot\| : * \to \mathbb{N}^0$ returns the cardinality of the given argument (*e.g.*, vector length, set size, *etc.*);
- $[\![\cdot]\!] : T \to \mathbb{N}^0$ counts the number of size annotations in the given term;
- $|\cdot| : T \to T^\circ$ erases sized terms to bare terms;
- $|\cdot|^\infty : T \to T^\infty$ erases sized terms to limit terms;
- $|\cdot|^\upsilon : T \to T^*$ erases size annotations υ to $*$ and all others to bare; and
- $|\cdot|^s : T \to T^\iota$ erases size annotations s to ι and all others to ∞.

Their definitions are straightforward. Functions on T are inductive on the structure of terms, and they do not touch recursive bare and position terms.

We use the following additional expressions relating membership in contexts and signatures:

- $x \in \Gamma$, $(x : t) \in \Gamma$, or $(x : t := e) \in \Gamma$ indicate that there is some declaration with variable name x, some assumption with type t, or some definition with type t and body e in the local context, and similarly for Γ_G;
- $\Gamma(x)$ returns the type (and possibly body) bound to x in Γ, and similarly for Γ_G;
- $I \in \Sigma$ means the (co)inductive definition of type I is in the signature.

3.2 Reduction Rules

The reduction rules are the usual ones for CIC: β-reduction (function application), ζ-reduction (let expression evaluation), ι-reduction (case expressions), μ-reduction (fixpoint expressions), ν-reduction (cofixpoint expressions), δ-reduction (local definitions), and Δ-reduction (global definitions). We define convertibility ($\approx$) as the symmetric–reflexive–transitive compatible closure of reductions up to η-expansion. The complete reduction rules are reproduced in Appendix A; we refer the reader to previous work [Barthe et al. 2006; Sacchini 2011, 2013] and the Coq manual in particular [The Coq Development Team 2020] for precise details and definitions.

In the case of δ-/Δ-reduction, if the variable has a vector of annotations, we define additional rules, shown in Figure 7. These reduction rules are important supporting size inference with definitions. If the definition body contains (co)inductive types (or other defined variables), we can assign them fresh annotations for each distinct usage of the defined variable. This ensures that certain subsizing relations are not lost due to the erasure of definition bodies. Further details are discussed in later sections.

We also use the metafunction whnf to denote the reduction of a term to weak head normal form, which would have the form of a universe, a function type, an unapplied abstraction, a (co)inductive type (applied or unapplied), a constructor (applied or unapplied), or an unapplied (co)fixpoint, with arguments and inner terms unreduced.

$$\frac{}{s \sqsubseteq \infty}\ \textsc{ss-infty} \qquad \frac{}{s \sqsubseteq s}\ \textsc{ss-refl} \qquad \frac{}{s \sqsubseteq \hat{s}}\ \textsc{ss-succ} \qquad \frac{s_1 \sqsubseteq s_2 \qquad s_2 \sqsubseteq s_3}{s_1 \sqsubseteq s_3}\ \textsc{ss-trans}$$

Fig. 8. Subsizing rules

$$\frac{}{\text{Prop} \leq \text{Set} \leq \text{Type}_1} \qquad \frac{}{\text{Type}_i \leq \text{Type}_{i+1}}\ \textsc{st-cumul} \qquad \frac{t \approx u}{t \leq u}\ \textsc{st-conv}$$

$$\frac{t \leq u \qquad u \leq v}{t \leq v}\ \textsc{st-trans} \qquad \frac{t_1 \approx t_2 \qquad u_1 \leq u_2}{\Pi x : t_1 .\, u_1 \leq \Pi y : t_2 .\, u_2}\ \textsc{st-prod} \qquad \frac{t_1 \leq t_2 \qquad u_1 \approx u_2}{t_1\ u_1 \leq t_2\ u_2}\ \textsc{st-app}$$

$$\frac{I \text{ inductive} \qquad s \sqsubseteq s'}{I^s \leq I^{s'}}\ \textsc{st-ind} \qquad \frac{I \text{ coinductive} \qquad s' \sqsubseteq s}{I^s \leq I^{s'}}\ \textsc{st-coind}$$

Fig. 9. Subtyping rules

3.3 Subtyping Rules

First, we define the subsizing relation in Figure 8. Subsizing is straightforward since our size algebra is simple. Note that we define $\widehat{\infty}$ to be equal to ∞.

We extend the usual subtyping for CIC to sized types in Figure 9. The key features are:

- Universes are **cumulative**. (ST-CUMUL)
- Since convertibility is symmetric, if $t \approx u$, then we have both $t \leq u$ and $u \leq t$. (ST-CONV)
- Inductive types are **covariant** in their size annotations; coinductive types are **contravariant**. (ST-IND, ST-COIND)
- The argument types of function types are **invariant**. (ST-PROD)
- The arguments of applications (and therefore the parameters and arguments of (co)inductive types) are **invariant**. (ST-APP)

We can intuitively understand the covariance of inductive types by considering size annotations as a measure of the maximum number of constructors "deep" an object can be. If a list has type $\text{List}^s\ t$, then a list with one more element can be said to have type $\text{List}^{\hat{s}}\ t$. By the subsizing and subtyping rules, $\text{List}^s\ t \leq \text{List}^{\hat{s}}\ t$: if a list has at most s "many" elements, then it certainly also has at most $\hat{s}$ "many" elements.

Conversely, for coinductive types, we can consider size annotations as a measure of how many constructors an object must at least "produce". A coinductive stream $\text{Stream}^{\hat{s}}$ that produces at least $\hat{s}$ "many" elements can also produce at least s "many" elements, so we have the contravariant relation $\text{Stream}^{\hat{s}} \leq \text{Stream}^s$.

Rules ST-PROD and ST-APP differ from $\widehat{\text{CIC}}$ and $\underline{\text{CIC}}$ in their invariance, but correspond to CIC in Coq. As previously mentioned, inductive definitions do not have polarities, so there is no way to indicate whether parameters are covariant, contravariant, or invariant. As a compromise, we treat all parameters as invariant. Note that, algorithmically speaking, the subtyping relation would produce *both* subsizing constraints, and not *neither*. For instance, $\text{List}^{s_1}\ \text{Nat}^{s_3} \leq \text{List}^{s_2}\ \text{Nat}^{s_4}$ yields $\text{Nat}^{s_3} \approx \text{Nat}^{s_4}$, which yields both $s_3 \sqsubseteq s_4$ and $s_4 \sqsubseteq s_3$. Further details on the subtyping algorithm are presented in Section 4.

$$\frac{\Sigma \text{ is well-formed}}{\mathsf{WF}(\Sigma, \square, \square)}\ \textsc{wf-nil}$$

$$\frac{\Sigma, \Gamma_G, \Gamma \vdash t : w \qquad x \notin \Gamma}{\mathsf{WF}(\Sigma, \Gamma_G, \Gamma(x : t))}\ \textsc{wf-local-assum} \qquad \frac{\Sigma, \Gamma_G, \square \vdash t : w \qquad x \notin \Gamma_G}{\mathsf{WF}(\Sigma, \Gamma_G(\mathrm{Assum}\, x : |t|^{\infty}), \square)}\ \textsc{wf-global-assum}$$

$$\frac{\Sigma, \Gamma_G, \Gamma \vdash e : t \qquad x \notin \Gamma}{\mathsf{WF}(\Sigma, \Gamma_G, \Gamma(x : t := e))}\ \textsc{wf-local-def} \qquad \frac{\Sigma, \Gamma_G, \square \vdash e : t \qquad x \notin \Gamma_G}{\mathsf{WF}(\Sigma, \Gamma_G(\mathrm{Def}\, x : |t|^{s} := |e|^{\infty}), \square)}\ \textsc{wf-global-def}$$

Fig. 10. Well-formedness of environments

$$\begin{aligned}
\mathsf{indType}(\Sigma, I_k) &= \Pi\Delta_p.\, \Pi\Delta_k.\, w_k \\
\mathsf{constrType}(\Sigma, c_\ell, s) &= \Pi\Delta_p.\, \Pi\Delta_\ell[I_\ell^\infty := I_\ell^s].\, I_\ell^{\hat{s}}\ \mathsf{dom}(\Delta_p)\ \overline{t}_\ell \\
\mathsf{motiveType}(\Sigma, \overline{p}, w, I_k^s) &= \Pi\Delta_k[\mathsf{dom}(\Delta_p) := \overline{p}].\, \Pi\_ : I_k^s\ \overline{p}\ \mathsf{dom}(\Delta_k).\, w \\
\mathsf{branchType}(\Sigma, \overline{p}, c_\ell, s, \wp) &= \Pi\Delta_\ell[I_\ell^\infty := I_\ell^s][\mathsf{dom}(\Delta_p) := \overline{p}].\, \wp\ \overline{t}_\ell[\mathsf{dom}(\Delta_p) := \overline{p}]\ (c_\ell\ \overline{p}\ \mathsf{dom}(\Delta_\ell)) \\
&\textit{where}\ \ k \in \overline{\imath}, \ell \in \overline{\jmath}, \big(\Delta_p \vdash \langle I_i\ \_ : \Pi\Delta_i.\, w_i\rangle := \langle c_j : \Pi\Delta_j.\, I_j\ \_\ \overline{t}_j\rangle\big) \in \Sigma
\end{aligned}$$

Fig. 11. Metafunctions for typing rules

$$\frac{\upsilon \notin \mathsf{SV}(t)}{\upsilon \text{ pos } t}\ \textsc{pos-}\notin \qquad \frac{\upsilon \notin \mathsf{SV}(t)}{\upsilon \text{ neg } t}\ \textsc{neg-}\notin \qquad \frac{\upsilon \notin \mathsf{SV}(t) \qquad \upsilon \text{ pos } u}{\upsilon \text{ pos } \Pi x : t.\, u}\ \textsc{pos-}\Pi \qquad \frac{\upsilon \notin \mathsf{SV}(t) \qquad \upsilon \text{ neg } u}{\upsilon \text{ neg } \Pi x : t.\, u}\ \textsc{neg-}\Pi$$

$$\frac{\upsilon \notin \mathsf{SV}(\overline{a}) \qquad I \text{ inductive}}{\upsilon \text{ pos } I^s \overline{a}}\ \textsc{pos-ind} \qquad \frac{\upsilon \notin \mathsf{SV}(\overline{a}) \qquad I \text{ coinductive}}{\upsilon \text{ neg } I^s \overline{a}}\ \textsc{neg-coind}$$

Fig. 12. Positivity/negativity of size variables in terms

3.4 Typing Rules

We now present the typing rules of CIC$\widehat{*}$. Note that these are type checking rules for *sized* terms, whose annotations come from size inference in Section 4.

We begin with the rules for well-formedness of local and global environments, presented in Figure 10. As mentioned earlier, we elide the well-formedness of signatures. Recall from Section 2 that global declarations, in Rules wf-global-assum and wf-global-def, have size variables erased to implement a kind of size polymorphism for global definitions. If a global definition is size-preserving with respect to some size annotation s, we replace it with the global annotation ι. These annotations are instantiated with a concrete size expression as needed in the typing rules.

The typing rules for sized terms are given in Figure 13. As in CIC, we define the three sets Axioms, Rules, and Elims, which describe how universes are typed, how products are typed, and what eliminations are allowed in case expressions, respectively. These are listed in Figure 22 in Appendix C. Metafunctions that construct some important function types are listed in Figure 11; they are also used by the inference algorithm in Section 4. Finally, the typing rules use the notions of positivity and negativity, whose rules are given in Figure 12, describing where the position annotations of fixpoints are allowed to appear. Positivity and negativity are structured such that the properties $\upsilon \text{ pos } t \Leftrightarrow t \leq t[\upsilon := \hat{\upsilon}]$ and $\upsilon \text{ neg } t \Leftrightarrow t[\upsilon := \hat{\upsilon}] \leq t$ hold.

$\boxed{\Sigma, \Gamma_G, \Gamma \vdash T : T}$

$$\frac{\mathsf{WF}(\Sigma, \Gamma_G, \Gamma) \qquad (x : t) \in \Gamma}{\Sigma, \Gamma_G, \Gamma \vdash x : t}\ \textsc{var-assum} \qquad \frac{\mathsf{WF}(\Sigma, \Gamma_G, \Gamma) \qquad (\mathrm{Assum}\ x : t.) \in \Gamma_G}{\Sigma, \Gamma_G, \Gamma \vdash x : t}\ \textsc{const-assum}$$

$$\frac{\begin{array}{c}\mathsf{WF}(\Sigma, \Gamma_G, \Gamma) \qquad (x : t \coloneqq e) \in \Gamma \\ \Sigma, \Gamma_G, \Gamma \vdash |e|^{\infty}\overline{[\infty_i \coloneqq s_i]} : t\end{array}}{\Sigma, \Gamma_G, \Gamma \vdash x^{\langle s_i \rangle} : t}\ \textsc{var-def} \qquad \frac{\begin{array}{c}\mathsf{WF}(\Sigma, \Gamma_G, \Gamma) \qquad (\mathrm{Def}\ x : t \coloneqq e.) \in \Gamma_G \\ \Sigma, \Gamma_G, \Box \vdash |e|^{\infty}\overline{[\infty_i \coloneqq s_i]} : t[\iota \coloneqq s]\end{array}}{\Sigma, \Gamma_G, \Gamma \vdash x^{\langle s_i \rangle} : t[\iota \coloneqq s]}\ \textsc{const-def}$$

$$\frac{\mathsf{WF}(\Sigma, \Gamma_G, \Gamma) \qquad (w_1, w_2) \in \mathrm{Axioms}}{\Sigma, \Gamma_G, \Gamma \vdash w_1 : w_2}\ \textsc{univ} \qquad \frac{\Sigma, \Gamma_G, \Gamma \vdash e : t \qquad \Sigma, \Gamma_G, \Gamma \vdash u : w \qquad t \leq u}{\Sigma, \Gamma_G, \Gamma \vdash e : u}\ \textsc{conv}$$

$$\frac{\Sigma, \Gamma_G, \Gamma \vdash t : w_1 \qquad \Sigma, \Gamma_G, \Gamma(x : t) \vdash u : w_2 \qquad (w_1, w_2, w_3) \in \mathrm{Rules}}{\Sigma, \Gamma_G, \Gamma \vdash \Pi x : t.\, u : w_3}\ \textsc{prod}$$

$$\frac{\Sigma, \Gamma_G, \Gamma \vdash t : w \qquad \Sigma, \Gamma_G, \Gamma(x : t) \vdash e : u}{\Sigma, \Gamma_G, \Gamma \vdash \lambda x : |t|.\, e : \Pi x : t.\, u}\ \textsc{abs} \qquad \frac{\Sigma, \Gamma_G, \Gamma \vdash e_1 : \Pi x : t.\, u \qquad \Sigma, \Gamma_G, \Gamma \vdash e_2 : t}{\Sigma, \Gamma_G, \Gamma \vdash e_1\ e_2 : u[x \coloneqq e_2]}\ \textsc{app}$$

$$\frac{\Sigma, \Gamma_G, \Gamma \vdash e_1 : t \qquad \Sigma, \Gamma_G, \Gamma(x : t \coloneqq e_1) \vdash e_2 : u}{\Sigma, \Gamma_G, \Gamma \vdash \mathrm{let}\ x : |t| \coloneqq e_1\ \mathrm{in}\ e_2 : u[x \coloneqq e_1]}\ \textsc{let-in}$$

$$\frac{\mathsf{WF}(\Sigma, \Gamma_G, \Gamma)}{\Sigma, \Gamma_G, \Gamma \vdash I^s : \mathsf{indType}(\Sigma, I)}\ \textsc{ind} \qquad \frac{\mathsf{WF}(\Sigma, \Gamma_G, \Gamma)}{\Sigma, \Gamma_G, \Gamma \vdash c : \mathsf{constrType}(\Sigma, c, s)}\ \textsc{constr}$$

$$\frac{\begin{array}{c}\Sigma, \Gamma_G, \Gamma \vdash e : I_k^{\hat{s}}\ \overline{p}\ \overline{a} \qquad \mathsf{indType}(\Sigma, I_k) = \Pi\_.\, \Pi\_.\, w_k \qquad (w_k, w, I_k) \in \mathrm{Elims} \\ \Sigma, \Gamma_G, \Gamma \vdash \wp : \mathsf{motiveType}(\Sigma, \overline{p}, w, I_k^{\hat{s}}) \qquad \Sigma, \Gamma_G, \Gamma \vdash e_j : \mathsf{branchType}(\Sigma, \overline{p}, c_j, s, \wp)\end{array}}{\Sigma, \Gamma_G, \Gamma \vdash \mathsf{case}_{|\wp|}\ e\ \mathsf{of}\ \langle c_j \Rightarrow e_j \rangle : \wp\ \overline{a}\ e}\ \textsc{case}$$

$$\frac{\begin{array}{c}t_k \approx \Pi\Delta_{1k}.\, \Pi x_k : I_k^{\upsilon_k}\ \overline{a}_k.\, \Pi\Delta_{2k}.\, u_k \qquad \|\Delta_{1k}\| = n_m - 1 \qquad \upsilon_k\ \mathrm{pos}\ \Delta_{1k}, \Delta_{2k}, u_k \\ \upsilon_k \notin \mathsf{SV}(\Gamma, \overline{a}_k, e_k) \qquad \Sigma, \Gamma_G, \Gamma \vdash t_k : w_k \qquad \Sigma, \Gamma_G, \Gamma\overline{(f_k : t_k)} \vdash e_k : t_k[\upsilon_k \coloneqq \hat{\upsilon}_k]\end{array}}{\Sigma, \Gamma_G, \Gamma \vdash \mathsf{fix}_{\langle n_k \rangle, m}\ \langle f_k : |t_k|^{\upsilon_k} \coloneqq e_k \rangle : t_m[\upsilon_m \coloneqq s]}\ \textsc{fix}$$

$$\frac{\begin{array}{c}t_k \approx \Pi\Delta_k.\, I_k^{\upsilon_k}\ \overline{a}_k \qquad \upsilon_k\ \mathrm{neg}\ \Delta_k \\ \upsilon_k \notin \mathsf{SV}(\Gamma, \overline{a}_k, e_k) \qquad \Sigma, \Gamma_G, \Gamma \vdash t_k : w_k \qquad \Sigma, \Gamma_G, \Gamma\overline{(f_k : t_k)} \vdash e_k : t_k[\upsilon_k \coloneqq \hat{\upsilon}_k]\end{array}}{\Sigma, \Gamma_G, \Gamma \vdash \mathsf{cofix}_m\ \langle f_k : |t_k|^{\upsilon_k} \coloneqq e_k \rangle : t_m[\upsilon_m \coloneqq s]}\ \textsc{cofix}$$

Fig. 13. Typing rules

Rules VAR-ASSUM, CONST-ASSUM, UNIV, CONV PROD, and APP are essentially unchanged from CIC. Rules ABS and LET-IN differ only in that type annotations are erased to bare. This is to preserve subject reduction without requiring size substitution during reduction, and is discussed further by Sacchini [2011].

The first significant usage of size annotations are in Rules VAR-DEF and CONST-DEF. If a variable or a constant is bound to a body in the local or global environment, it is annotated with a vector of size expressions such that the body is well-typed after substituting in those size expressions, allowing for proper δ-/Δ-reduction of variables and constants. Note that each usage of a variable or a constant does not have to have the same size annotations. Furthermore, every global annotation in a constant's type is instantiated to the same size expression s, which enforces size-preservedness.

Before discussing typing (co)inductive types, there are some indexing conventions to note. In Rules IND, CONSTR, and CASE, we use i to range over the number of (co)inductive types in a single mutual (co)inductive definition, j to range over the number of constructors of a given (co)inductive type, k for a specific index in the range $\overline{\imath}$, and ℓ for a specific index in the range $\overline{\jmath}$. In Rules FIX and COFIX, we use k to range over the number of mutually-defined (co)fixpoints and m for a specific index in the range $\overline{k}$. When a judgement contains a ranging index not contained within $\langle\cdot\rangle$, it means that the judgement or side condition should hold for *all* indices in its range. For instance, the branch judgement in Rule CASE should hold for all branches, and the fixpoint type judgement in Rule FIX should hold for all mutually-defined fixpoints. Finally, we use _ (underscore) to omit irrelevant constructions for readability.

In Rule IND, the type of a (co)inductive type is a function type from its parameters Δ_p and its indices Δ_k to its universe w_k. The (co)inductive type itself holds a single size annotation.

In Rule CONSTR, the type of a constructor is a function type from its parameters Δ_p and its arguments Δ_ℓ to its (co)inductive type I_ℓ applied to the parameters and its indices $\overline{t}_\ell$. Size annotations appear in two places:

- In the argument types of the constructor. We annotate each occurrence of I_ℓ in Δ_ℓ with a size expression s.
- On the (co)inductive type of the fully-applied constructor. If the constructor belongs to the inductive type I_ℓ, then it is annotated with the size expression $\hat{s}$. Using the successor guarantees that the constructor always constructs an object that is *larger* than any of its arguments of the same type.

As an example, consider a possible typing of VCons:

$$\text{VCons} : (A : \text{Type}) \to (n : \text{Nat}^\infty) \to A \to \text{Vector}^s\ A\ n \to \text{Vector}^{\hat{s}}\ A\ (\text{S}\ n)$$

It has a single parameter A and S n corresponds to the index $\overline{t}_j$ of the constructor's inductive type. The input Vector has size s, while the output Vector has size $\hat{s}$.

In Rule CASE, a case expression has three important parts:

- The **target** e. It must have a (co)inductive type I_k with a successor size annotation $\hat{s}_k$ so that any constructor arguments of the same type can have the predecessor size annotation.
- The **motive** $\wp$. It is an abstraction over the indices Δ_k of the target type I_k and the target itself, and produces the return type of the case expression. Note that in the motive's type in Figure 11, the parameter variables $\text{dom}(\Delta_p)$ in the indices are bound to the parameters of the target type.
  (This presentation follows CIC, but differs from that by Sacchini [2011, 2013, 2014], where the case expression contains a return type in which the index and target variables are free and explicitly stated, in the syntactic form $\overline{y}.x.\wp$.)
- The **branches** e_j. Each branch is associated with a constructor c_j and is an abstraction over the arguments Δ_j of the constructor, producing some term. The type of each branch, listed in Figure 11, is the motive $\wp$ applied to the indices $\overline{t}_j$ of that constructor's type and the constructor applied to the parameters and its arguments.
  Note that, like in the type of constructors, we annotate each occurence of c_j's (co)inductive type I_k in Δ_j with the size expression s. The parameter variables in Δ_j and $\overline{t}_j$ are similarly bound to the parameters $\overline{p}$ of the target.

The type of the entire case expression is then the motive applied to the target type's indices and the target itself. Notice that we also restrict the universe of this type based on the universe of the target type using Elims.

Finally, we have the typing of mutual (co)fixpoints in rules fix and cofix. We take the annotated type t_k of the kth (co)fixpoint definition to be convertible to a function type containing a (co)inductive type, as usual. However, instead of the guard condition, we ensure termination/productivity using size expressions.

The main difficulty in these rules is supporting size preserving (co)fixpoints. We must restrict how the size variable v_k appears in the type of the (co)fixpoints, using the pos and neg judgments. For fixpoints, the type of the n_kth argument, the recursive argument, is an inductive type annotated with a size variable v_k. For cofixpoints, the return type is a coinductive type annotated with v_k. The positivity or negativity of v_k in the rest of t_k indicate where v_k may occur other than in the (co)recursive position. For instance, List^{v} Nat $\rightarrow$ List^{v} Nat $\rightarrow$ List^{v} Nat is a valid fixpoint type with respect to v, while Stream^{v} Nat $\rightarrow$ List^{v} Nat $\rightarrow$ List Nat^{v} is not, since v appears negatively in Stream and must not appear at all in the parameter of the List return type. This is because v_k indicates the types that are size-preserved. For fixpoints, it indicates not only the recursive argument but also which argument or return types have size *at most* that of the recursive argument. For cofixpoints, it indicates the arguments that have size *at least* that of the return type. Therefore, it cannot appear on types of the incorrect recursivity, or on types not being (co)recurred upon.

As in Rule abs, we cannot keep the size annotations. Instead, we mark (co)fixpoint type annotations, which recall are position terms, as size-preserving using the erasure $|t_k|^{v_k}$ to replace size annotations in t_k whose size variable is v_k with $*$.

Checking termination and productivity is relatively straightforward. If t_k are well typed, then the (co)fixpoint bodies should have type t_k with a successor size in the local context where (co)fixpoint names f_k are bound to their types t_k. Intuitively, this tells us that the recursive call to f_k in fixpoint bodies are on smaller-sized arguments, and that corecursive bodies produce objects larger than those from the corecursive call to f_k. The type of the whole (co)fixpoint is then the mth type t_m with its size variable v_m bound to some size expression s.

In Coq, the indices of the recursive elements are rarely given, and there are no user-provided position annotations at all. In Section 4, we present how we compute the indices and the position annotations during size inference.

4 SIZE INFERENCE

In this section, we present a size inference algorithm, whose goal is to take unannotated programs in T° (corresponding to terms in CIC), simultaneously assign annotations to them while collecting a set of subsizing constraints based on the typing rules, check the constraints to ensure well-typedness, and produce annotated programs in T^{ι} that are stored in the global environment and can be used in the inference of future programs. Constraints are generated when comparing two types t, u to ensure that the subtyping relation $t \leq u$ holds. Therefore, this algorithm is also a type checking algorithm, since it could be that t fails to subtype u, in which case the algorithm fails.

Our algorithm is an extension to the size inference algorithm of $\widehat{\text{CIC}}$, and Barthe et al. [2006] presents soundness and completeness of their algorithm with respect to $\widehat{\text{CIC}}$'s typing rules. We discuss soundness and completeness theorems of our algorithm with respect to $\text{CIC}\widehat{*}$'s typing rules in Section 5.

4.1 Notation

We define three judgements to represent *checking*, *inference*, and *well-formedness*. They all use the symbol $\rightsquigarrow$, with inputs on the left and outputs on the right. We use $C : \mathbb{P}(S \times S)$ to represent subsizing constraints: if $(s_1, s_2) \in C$, then we must enforce $s_1 \sqsubseteq s_2$.

$\boxed{C, \Gamma_G, \Gamma \vdash T^\circ \Leftarrow T \rightsquigarrow C, T}$

$$\frac{C, \Gamma_G, \Gamma \vdash e^\circ \rightsquigarrow C_1, e \Rightarrow t}{C, \Gamma_G, \Gamma \vdash e^\circ \Leftarrow u \rightsquigarrow C_1 \cup t \preceq u, e} \text{ A-CHECK}$$

Fig. 14. Size inference algorithm: Checking

- Checking, $C, \Gamma_G, \Gamma \vdash e^\circ \Leftarrow t \rightsquigarrow C', e$, takes a set of constraints C, environments Γ_G, Γ, a bare term e°, and an annotated type t, and produces the annotated term e with a new set of constraints C' that ensures that the type of e subtypes t.
- Inference, $C, \Gamma_G, \Gamma \vdash e^\circ \rightsquigarrow C', e \Rightarrow t$, takes a set of constraints C, environments Γ_G, Γ, and a bare term e°, and produces the annotated term e, its annotated type t, and a new set of constraints C'.
- Well-formedness, $\Gamma_G^\circ \rightsquigarrow \Gamma_G$, takes a global environment with bare declarations and produces a global environment where each declaration has been properly annotated via size inference.

The algorithm is implicitly parameterized over a fixed signature Σ, as well as two mutable sets of size variables $\mathcal{V}, \mathcal{V}^*$, such that $\mathcal{V}^* \subseteq \mathcal{V}$. Their assignment is denoted with $:=$ and they are initialized as empty. The set $\mathcal{V}^*$ contains *position* size variables, which mark size-preserving types, and we use τ for these position size variables. We define two additional metafunctions: PV returns all position size variables in a given term, while $|\cdot|^*$ erases position size variables to position annotations and all other annotations to bare. Finally, on the right-hand size of inference judgements, we use $e \Rightarrow^* t$ to mean $e \Rightarrow t' \wedge t = \text{whnf}(t')$.

We define a number of metafunctions to translate the side conditions from the typing rules into procedural form. They are introduced as needed, but are also summarized in Figure 23 in Appendix C.

4.2 Inference Algorithm

Size inference begins with a bare term; even type annotations of (co)fixpoints are bare, *i.e.*,

$$T^\circ ::= \cdots \mid \text{fix}_{\langle n_k \rangle, m} \ \langle \mathcal{X} : T^\circ := T^\circ \rangle \mid \text{cofix}_m \ \langle \mathcal{X} : T^\circ := T^\circ \rangle$$

Notice that fixpoints still have a vector of indices, with n_k being the index of the recursive argument of the kth mutual fixpoint, whereas Coq programs have no indices. To produce these indices, we do what Coq currently does: brute-force search. We attempt type checking on every combination of indices from left to right (even if the type of the argument at that index is not inductive). This continues until one combination works, or fails if none do.

Figure 14, Figure 15, and Figure 17 present the size inference algorithm, which uses the same indexing conventions as the typing rules. We go over parts of the algorithm in detail shortly.

Rule A-CHECK is the checking component of the algorithm. To ensure that the inferred type subtypes the given type, we use the metafunction $\preceq$ that takes two sized terms and attempts to produce a set of subsizing constraints based on the subtyping rules of Figure 9. $\preceq$ may reduce terms to check convertibility and will fail if two terms are incompatible.

Rules A-VAR-ASSUM, A-CONST-ASSUM, A-UNIV, A-PROD, A-ABS, A-APP, and A-LET-IN are all fairly straightforward. Note that after type annotations pass through inference to become sized types, they must be erased to bare types again. These rules use the metafunctions axiom, rule, and elim, which correspond to the sets Axioms, Rules, and Elims, defined in Figure 22. The metafunction axiom produces the type of a universe; rule produces the type of a function type given the universes of its argument and return types; and elim directly checks membership in Elims and can fail.

In Rules A-VAR-DEF and A-CONST-DEF, we annotate variables and constants using a vector of annotations from fresh, which generates the given number of fresh size variables and adds them

$$\boxed{C, \Gamma_G, \Gamma \vdash T^\circ \rightsquigarrow C, T \Rightarrow T}$$

$$\frac{}{C, \Gamma_G, \Gamma \vdash x \rightsquigarrow C, x \Rightarrow \Gamma(x)}\ \text{A-VAR-ASSUM} \qquad \frac{}{C, \Gamma_G, \Gamma \vdash x \rightsquigarrow C, x \Rightarrow \Gamma_G(x)}\ \text{A-CONST-ASSUM}$$

$$\frac{(e : t) = \Gamma(x) \qquad \overline{v_i} = \mathsf{fresh}(\llbracket e \rrbracket)}{C, \Gamma_G, \Gamma \vdash x \rightsquigarrow C, x^{\langle v_i \rangle} \Rightarrow t}\ \text{A-VAR-DEF} \qquad \frac{\begin{array}{c}(e : t) = \Gamma_G(x)\\ \overline{v_i} = \mathsf{fresh}(\llbracket e \rrbracket) \qquad v = \mathsf{fresh}(1)\end{array}}{C, \Gamma_G, \Gamma \vdash x \rightsquigarrow C, x^{\langle v_i \rangle} \Rightarrow t[\iota := v]}\ \text{A-CONST-DEF}$$

$$\frac{}{C, \Gamma_G, \Gamma \vdash w \rightsquigarrow C, w \Rightarrow \mathsf{axiom}(w)}\ \text{A-UNIV}$$

$$\frac{C, \Gamma_G, \Gamma \vdash t^\circ \rightsquigarrow C_1, t \Rightarrow^* w_1 \qquad C_1, \Gamma_G, \Gamma(x : t) \vdash u^\circ \rightsquigarrow C_2, u \Rightarrow^* w_2}{C, \Gamma_G, \Gamma \vdash \Pi x : t^\circ. u^\circ \rightsquigarrow C_2, \Pi x : t. u \Rightarrow \mathsf{rule}(w_1, w_2)}\ \text{A-PROD}$$

$$\frac{C, \Gamma_G, \Gamma \vdash t^\circ \rightsquigarrow C_1, t \Rightarrow^* w \qquad C_1, \Gamma_G, \Gamma(x : t) \vdash e^\circ \rightsquigarrow C_2, e \Rightarrow u}{C, \Gamma_G, \Gamma \vdash \lambda x : t^\circ. e^\circ \rightsquigarrow C_2, \lambda x : |t|. e \Rightarrow \Pi x : t. u}\ \text{A-ABS}$$

$$\frac{C, \Gamma_G, \Gamma \vdash e_1^\circ \rightsquigarrow C_1, e_1 \Rightarrow^* \Pi x : t. u \qquad C_1, \Gamma_G, \Gamma \vdash e_2^\circ \Leftarrow t \rightsquigarrow C_2, e_2}{C, \Gamma_G, \Gamma \vdash e_1^\circ\ e_2^\circ \rightsquigarrow C_2, e_1\ e_2 \Rightarrow u[x := e_2]}\ \text{A-APP}$$

$$\frac{\begin{array}{c}C, \Gamma_G, \Gamma \vdash t^\circ \rightsquigarrow C_1, t \Rightarrow^* w \qquad C_1, \Gamma_G, \Gamma \vdash e_1^\circ \Leftarrow t \rightsquigarrow C_2, e_1\\ C_2, \Gamma_G, \Gamma(x : t := e_1) \vdash e_2^\circ \rightsquigarrow C_3, e_2 \Rightarrow u\end{array}}{C, \Gamma_G, \Gamma \vdash \mathsf{let}\ x : t^\circ := e_1^\circ\ \mathsf{in}\ e_2^\circ \rightsquigarrow C_3, \mathsf{let}\ x : |t| := e_1\ \mathsf{in}\ e_2 \Rightarrow u[x := e_1]}\ \text{A-LET-IN}$$

$$\frac{v = \mathsf{fresh}(1)}{C, \Gamma_G, \Gamma \vdash I \rightsquigarrow C, I^v \Rightarrow \mathsf{indType}(\Sigma, I)}\ \text{A-IND} \qquad \frac{\tau = \mathsf{fresh}(1) \qquad \mathcal{V} := \mathcal{V} \cup \{\tau\}}{C, \Gamma_G, \Gamma \vdash I^* \rightsquigarrow C, I^\tau \Rightarrow \mathsf{indType}(\Sigma, I)}\ \text{A-IND-STAR}$$

$$\frac{v = \mathsf{fresh}(1)}{C, \Gamma_G, \Gamma \vdash c \rightsquigarrow C, c \Rightarrow \mathsf{constrType}(\Sigma, c, v)}\ \text{A-CONSTR}$$

$$\frac{\begin{array}{c}C, \Gamma_G, \Gamma \vdash e^\circ \rightsquigarrow C_1, e \Rightarrow^* I_k^s\ \overline{p}\ \overline{a} \qquad C_1, \Gamma_G, \Gamma \vdash \wp^\circ \rightsquigarrow C_2, \wp \Rightarrow t_p\\ \Pi\_. \Pi\Delta_k. w_k = \mathsf{indType}(\Sigma, I_k) \qquad w = \mathsf{decompose}(t_p, \|\Delta_k\| + 1) \qquad \mathsf{elim}(w_k, w, I_k)\\ v = \mathsf{fresh}(1) \qquad C_3 = \mathsf{caseSize}(I_k^s, \hat{v}) \qquad C_4 = t_p \preceq \mathsf{motiveType}(\Sigma, \overline{p}, w, I_k^{\hat{v}})\\ C_5 = C_2 \cup C_3 \cup C_4 \qquad C_5, \Gamma_G, \Gamma \vdash e_j^\circ \Leftarrow \mathsf{branchType}(\Sigma, \overline{p}, c_j, v, \wp) \rightsquigarrow C_{6j}, e_j \qquad C_6 = \bigcup_j C_{6j}\end{array}}{C, \Gamma_G, \Gamma \vdash \mathsf{case}_{\wp^\circ}\ e^\circ\ \mathsf{of}\ \langle c_j \Rightarrow e_j^\circ \rangle \rightsquigarrow C_6, \mathsf{case}_{|\wp|}\ e\ \mathsf{of}\ \langle c_j \Rightarrow e_j \rangle \Rightarrow \wp\ \overline{a}\ e}\ \text{A-CASE}$$

$$\frac{\begin{array}{c}C, \Gamma_G, \Gamma \vdash t_k^\circ \rightsquigarrow \_, \_ \Rightarrow \_ \qquad C, \Gamma_G, \Gamma \vdash \mathsf{setRecStars}(t_k^\circ, n_k) \rightsquigarrow C_{1k}, t_k \Rightarrow^* w\\ \bigcup_k C_{1k}, \Gamma_G, \Gamma(\overline{f_k : t_k}) \vdash e_k^\circ \Leftarrow \mathsf{shift}(t_k) \rightsquigarrow \overline{C_{2k}, e_k} \qquad C_2 = \bigcup_k C_{2k} \cup t_k \preceq \mathsf{shift}(t_k)\\ C_3 = \mathsf{RecCheckLoop}(C_2, \mathsf{getRecVar}(t_k, n_k), \overline{t_k}, \overline{e_k})\end{array}}{C, \Gamma_G, \Gamma \vdash \mathsf{fix}_{\langle n_k \rangle, m}\ \langle f_k : t_k^\circ := e_k^\circ \rangle \rightsquigarrow C_3, \mathsf{fix}_{\langle n_k \rangle, m}\ \langle f_k : |t_k|^* := e_k \rangle \Rightarrow t_m}\ \text{A-FIX}$$

$$\frac{\begin{array}{c}C, \Gamma_G, \Gamma \vdash t_k^\circ \rightsquigarrow \_, \_ \Rightarrow \_ \qquad C, \Gamma_G, \Gamma \vdash \mathsf{setCorecStars}(t_k^\circ) \rightsquigarrow C_{1k}, t_k \Rightarrow^* w\\ \bigcup_k C_{1k}, \Gamma_G, \Gamma(\overline{f_k : t_k}) \vdash e_k^\circ \Leftarrow \mathsf{shift}(t_k) \rightsquigarrow \overline{C_{2k}, e_k} \qquad C_2 = \bigcup_k C_{2k} \cup \mathsf{shift}(t_k) \preceq t_k\\ C_3 = \mathsf{RecCheckLoop}(C_2, \mathsf{getCorecVar}(t_k), \overline{t_k}, \overline{e_k})\end{array}}{C, \Gamma_G, \Gamma \vdash \mathsf{cofix}_m\ \langle f_k : t_k^\circ := e_k^\circ \rangle \rightsquigarrow C_3, \mathsf{cofix}_m\ \langle f_k : |t_k|^* := e_k \rangle \Rightarrow t_m}\ \text{A-COFIX}$$

Fig. 15. Size inference algorithm: Inference

to $\mathcal{V}$. The length of the vector corresponds to the number of size annotations found in the body of the definitions. For instance, if $(x : \text{Type} := \text{List}^{s_1}\ \text{Nat}^{s_2}) \in \Gamma$, then a use of x would be annotated as $x^{\langle v_1, v_2 \rangle}$. If x is δ-reduced during inference, such as in a fixpoint type, then it is replaced by $\text{List}^{v_1}\ \text{Nat}^{v_2}$. Furthermore, since the types of global definitions can have global annotations marking sized-preserved types, we replace the global annotations with a fresh size variable.

A position-annotated type from a (co)fixpoint can be passed into the algorithm, so we deal with the possibilities separately in Rules A-IND and A-IND-STAR. In both rules, a bare (co)inductive type is annotated with a size variable; in Rule A-IND-STAR, it is also added to the set of position size variables $\mathcal{V}^*$.

In Rule A-CONSTR, we generate a single fresh size variable, which gets annotated on the constructor's (co)inductive type in the argument types of the constructor type, as well as the return type, which has the successor of that size variable. All other (co)inductive types which are not the constructor's (co)inductive type continue to have ∞ annotations.

The key constraint in Rule A-CASE is generated by caseSize. Similar to Rule A-CONSTR, we generate a single fresh size variable v to annotate on I_k in the branches' argument types, which correspond to the constructor arguments of the target. Then, given the unapplied target type I_k^s, caseSize returns $\{s \sqsubseteq \hat{v}\}$ if I_k is inductive and $\{\hat{v} \sqsubseteq s\}$ if I_k is coinductive. This ensures that the target type satisfies $I_k^s\ \overline{p}\ \overline{a} \leq I_k^{\hat{v}_k}\ \overline{p}\ \overline{a}$, so that Rule CASE is satisfied.

The rest of the rule proceeds as we would expect: we infer the sized type of the target and the motive, we check that the motive and the branches have the types we expect given the target type, and we infer that the sized type of the case expression is the annotated motive applied to the target type's annotated indices and the annotated target itself. We also ensure that the elimination universes are valid using elim on the motive type's return universe and the target type's universe. To obtain the motive type's return universe, we use decompose. Given a type t and a natural n, this metafunction reduces t to a function type $\Pi\Delta.\, u$ where $\|\Delta\| = n$, reduces u to a universe w, and returns w. It can fail if t cannot be reduced to a function type, if $\|\Delta\| < n$, or if u cannot be reduced to a universe.

Finally, we come to size inference and termination/productivity checking for (co)fixpoints. It uses the following metafunctions:

- setRecStars, given a function type t and an index n, decomposes t into arguments and a return type, reduces the nth argument type to an inductive type, annotates that inductive type with position annotation $*$, annotates all other argument and return types with the same inductive type with $*$, and rebuilds the function type. This is how fixpoint types obtain their position annotations without being user-provided; the algorithm will remove other position annotations if size-preservation fails.
  Similarly, setCorecStars annotates the coinductive return type first, then the argument types with the same coinductive type. Both of these can fail if the nth argument type or the return type respectively are not (co)inductive types. Note that the decomposition of t may perform reductions using whnf.
- getRecVar, given a function type t and an index n, returns the position size variable of the annotation on the nth inductive argument type, while getCorecVar returns the position size variable of the annotation on the coinductive return type. Essentially, they retrieve the position size variable of the annotation on the primary (co)recursive type of a (co)fixpoint type.
- shift replaces all position size annotations s (*i.e.*, $\lfloor s \rfloor \in \mathcal{V}^*$) by its successor $\hat{s}$.

Although the desired (co)fixpoint is the mth one in the block of mutually-defined (co)fixpoints, we must still size-infer and type-check the entire mutual definition. Rules A-FIX and A-COFIX first run the size inference algorithm on each of the (co)fixpoint *types*, ignoring the results, to ensure

```
let rec RecCheckLoop $C_2$ $\overline{\tau_k}$ $\overline{t_k}$ $\overline{e_k}$ =
  try let $C_3$ = {} in
      let for i = 1 to k do
          let $\mathsf{pv}_i$ = PV $t_i$ in
          let $\mathsf{sv}_i$ = (SV $t_i$ ∪ SV $e_i$) \ $\mathsf{pv}_i$ in
          $C_3$ := $C_3$ ∪ RecCheck $C_2$ $\tau_i$ $\mathsf{pv}_i$ $\mathsf{sv}_i$
      done in $C_3$
  with RecCheckFail $V$ ->
      if (empty? $V$)
      then raise RecCheckLoopFail
      else $\mathcal{V}^*$ := $\mathcal{V}^* \setminus V$; RecCheckLoop $C_2$ $\overline{\tau_k}$ $\overline{t_k}$ $\overline{e_k}$
```

Fig. 16. Pseudocode implementation of RecCheckLoop

that any reduction on those types will terminate. Then we annotate the bare types with position annotations (using setRecStars/setCorecStars) and pass these position types through the algorithm to get sized types $\overline{t_k}$. Next, we check that the (co)fixpoint bodies have the successor-sized types of $\overline{t_k}$ when the (co)fixpoints have types $\overline{t_k}$ in the local environment. Lastly, we call RecCheckLoop, and return the constraints it gives us, along with the mth (co)fixpoint type.

Notice that setRecStars and setCorecStars optimistically annotates *all* possible (co)inductive types in the (co)fixpoint type with position annotations, but not all (co)fixpoints are size-preserving. RecCheckLoop filters these annotations to generate the final constraint set. This is a recursive function that calls RecCheck, which checks satisfiability of a given constraint set. If the set is unsatisfiable due to a bad position annotation, then RecCheckLoop removes it and tries again.

More specifically, RecCheck can fail raising RecCheckFail, which contains a set V of position size variables that must be set to infinity; since position size variables always appear on size-preserved types, they cannot be infinite. RecCheckLoop then removes V from the set of position size variables, allowing them to be set to infinity, and recursively calls itself. The number of position size variables from the (co)fixpoint type shrinks on every iteration until no more can be removed. If no satisfiable set is found even when no positions are considered size preserving, termination/productivity checking and thus type inference has failed. An OCaml-like pseudocode implementation of RecCheckLoop is provided by Figure 16.

4.3 RecCheck

As in previous work on CC$\widehat{\omega}$ with coinductive streams [Sacchini 2013] and in CIC$\widehat{\ }$, we use the same RecCheck algorithm from F$\widehat{\ }$ [Barthe et al. 2005]. This algorithm attempts to ensure that the subsizing rules in Figure 8 can be satisfied within a given set of constraints. It does so by checking the set of constraints for invalid circular subsizing relations, setting the size variables involved in the cycles to ∞, and producing a new set of constraints without these problems or fails, which indicates nontermination or nonproductivity. It takes four arguments:

- A set of subsizing constraints C.
- The size variable τ of the annotation on the type of the recursive argument (for fixpoints) or on the return type (for cofixpoints). While other arguments (and the return type, for fixpoints) may optionally be marked as size-preserving, each (co)fixpoint type requires at *least* τ for the primary (co)recursive type.
- A set of size variables V^* that must be set to some non-infinite size. These are the size annotations in the (co)fixpoint type that have position size variables. Note that $\tau \in V^*$.

$\boxed{\Gamma_G^\circ \rightsquigarrow \Gamma_G}$

$$\frac{}{\square \rightsquigarrow \square}\ \textsc{a-global-nil} \qquad \frac{\Gamma_G^\circ \rightsquigarrow \Gamma_G \qquad \emptyset, \Gamma_G, \square \vdash t^\circ \rightsquigarrow \_, t \Rightarrow w}{\Gamma_G^\circ(\text{Assum } x : t^\circ) \rightsquigarrow \Gamma_G(\text{Assum } x : |t|^\infty)}\ \textsc{a-global-assum}$$

$$\frac{\begin{array}{c}\Gamma_G^\circ \rightsquigarrow \Gamma_G \qquad \emptyset, \Gamma_G, \square \vdash t^\circ \rightsquigarrow \_, t \Rightarrow w \\ C_1, \Gamma_G, \square \vdash e^\circ \rightsquigarrow \_, e \Rightarrow u \qquad \_ = u \preceq t \qquad t' = \text{eraseToGlobal}(u, t)\end{array}}{\Gamma_G^\circ(\text{Def } x : t^\circ := e^\circ) \rightsquigarrow \Gamma_G(\text{Def } x : t' := |e|^\infty)}\ \textsc{a-global-def}$$

Fig. 17. Size inference algorithm: Well-formedness

- A set of size variables $V^{\neq}$ that must be set to ∞. These are all other non-position size annotations, found in the (co)fixpoint types and bodies.

Here, we begin to treat C as a weighted, directed graph. Each size variable corresponds to a node, and each subsizing relation is an edge from the lower to the upper variable. A size expression consists of a size variable with an arbitrary finite nonnegative number of successor "hats"; instead of using a perniculous tower of carets, we can write the number as a superscript, as in $\hat{v}^n$. Then given a subsizing relation $\hat{v}_1^{n_1} \sqsubseteq \hat{v}_2^{n_2}$, the weight of the edge from v_1 to v_2 is $n_2 - n_1$. Subsizings to ∞ do not need to be added to C since they are given by Rule $\textsc{ss-infty}$; subsizings from ∞ are given an edge weight of 0.

Given a set of size variables V, its *upward closure* $\bigsqcup V$ in C is the set of size variables that can be reached from V by travelling along the edges of C; that is, $v_1 \in V \land \hat{v}_1^{n_1} \sqsubseteq \hat{v}_2^{n_2} \implies v_2 \in V$. Similarly, the *downward closure* $\bigsqcap V$ in C is the set of size variables that can reach V by travelling along the edges of C, or $v_2 \in V \land \hat{v}_1^{n_1} \sqsubseteq \hat{v}_2^{n_2} \implies v_1 \in V$.

We use the notation $v \sqsubseteq V$ to denote the set of constraints from v to each size variable in V and similarly for $V \sqsubseteq v$.

The algorithm proceeds as follows:

(1) Add $V^* \sqsubseteq \tau$ to C. This ensures that all position size variables are size-preserving.
(2) Let $V^\iota = \bigsqcap V^*$, and add $\tau \sqsubseteq V^\iota$ to C. This ensures that τ is the smallest size variable among all the noninfinite size variables.
(3) Find all negative cycles in C, and let V^- be the set of all size variables in some negative cycle.
(4) Remove all edges with size variables in V^- from C, and add $\infty \sqsubseteq V^-$. Since $\widehat{\infty} \sqsubseteq \infty$, this is the only way to resolve negative cycles.
(5) Add $\infty \sqsubseteq \left(\bigsqcup V^{\neq} \cap \bigsqcup V^\iota\right)$ to C.
(6) Let $V^\perp = (\bigsqcup\{\infty\}) \cap V^\iota$. This is the set of size variables that we have determined to both be infinite and noninfinite. If $V^\perp$ is empty, then return C.
(7) Otherwise, let $V = V^\perp \cap (V^* \setminus \{\tau\})$, and fail with RecCheckFail(V). This is the set of contradictory position size variables excluding τ, which we can remove from $\mathcal{V}^*$ in RecCheckLoop. If V is empty, there are no position size variables left to remove, so the check and therefore the size inference algorithm fails.

Disregarding closure operations and set operations like intersection and difference, the time complexity of a single pass is $O(\|V\|\|C\|)$, where V is the set of size variables appearing in C. This comes from the negative-cycle finding in (3).

4.4 Well-Formedness

A self-contained chunk of code, be it a file or a module, consists of a sequence of (co)inductive definitions (signatures) and programs (global declarations). For our purposes, we assume that there is a singular well-formed signature defined independently. Then we need to perform size inference on each declaration of Γ_G in order. This is given by Rules A-GLOBAL-NIL, A-GLOBAL-ASSUM, and A-GLOBAL-DEF in Figure 17. The first two are straightforward.

In Rule A-GLOBAL-DEF, we obtain two types: u, the inferred sized type of the definition body, and t, its sized declared type. Evidently, u must subtype t. Furthermore, only u has position size variables due to the body e, so we use eraseToGlobal to replace the size variables of t in the same locations as the position size variables of u with global annotations. For instance, if $\mathcal{V}^* = \{\tau_1, \tau_2\}$

$$\text{eraseToGlobal}(\text{Nat}^{\tau_1} \to \text{Nat}^{\upsilon_1} \to \text{Nat}^{\tau_2}, \text{Nat}^{\upsilon_2} \to \text{Nat}^{\upsilon_3} \to \text{Nat}^{\upsilon_4}) = \text{Nat}^{\iota} \to \text{Nat}^{\infty} \to \text{Nat}^{\iota}$$

Note that we cannot simply globally erase u and use that in the global definition type, since t may be a more general type than u.

5 METATHEORY OF $\text{CIC}\widehat{*}$ AND FUTURE WORK

In this section, we describe the metatheory of $\text{CIC}\widehat{*}$. Some of the metatheory is inherited or essentially similar to past work [Barthe et al. 2006; Sacchini 2011, 2013], although we must adapt key proofs to account for differences in subtyping and definitions. Complete proofs for a language like $\text{CIC}\widehat{*}$ are too involved to present in full, so we provide key lemmas and proof sketches; full proofs can be found in the supplementary material.

In short, $\text{CIC}\widehat{*}$ satisfies confluence and subject reduction (with the same caveats as in CIC for cofixpoints). Proofs of strong normalization and logical consistency for $\text{CIC}\widehat{*}$, and soundness and completeness of the size inference algorithm with respect to the typing rules, remain future work. We conjecture how the proofs of strong normalization and consistency should proceed based on past work [Barthe et al. 2006; Sacchini 2011, 2013].

The metatheoretical investigations provide many interesting questions, so we discuss future work in context as these questions arise.

5.1 Confluence

We define $\rhd$ as the least compatible closure of $\beta\zeta\delta\Delta\iota\mu\nu$-reduction and $\rhd^*$ as the reflexive–transitive closure of $\rhd$. Precise definitions are also provided in Appendix A.

Theorem 5.1 (Confluence). *Let $e, e_{1,2}$ be terms. If $e_1 \, {}^*\!\lhd \, e \rhd^* e_2$ then $e_1 \rhd^* e' \, {}^*\!\lhd \, e_2$ for a term e'.*

Proof Sketch. We use the Takahashi translation technique due to Komori et al. [2014], which is a simplification of the standard parallel reduction technique. The proof is straightforward. □

5.2 Subject Reduction

Subject reduction does not hold in CIC, or in Coq, due to cofixpoints[2]. In essence, the problem is that ν-reduction can either be guarded to reduce under a case expression (dual to μ-reduction which only reduces a fixpoint when applied to a constructor), enabling strong normalization but breaking subject reduction, or unrestricted, enabling subject reduction to hold but breaking strong normalization.

$\text{CIC}\widehat{*}$, following Coq, implements the guarded ν-reduction, so cofixpoints do not satisfy subject reduction. Ongoing work in Coq seeks to change the semantics of cofixpoints to enable both subject reduction and strong normalization.

With unrestricted ν-reduction, subject reduction and confluence hold for cofixpoints, but we conjecture strong normalization must fail. Sacchini [2013] provides an nice discussion of this in a similar context.

[2]Discussion and counterexample can be found here: https://github.com/coq/coq/issues/5288/

THEOREM 5.2 (SUBJECT REDUCTION). *Let Σ be a well-formed signature. Further, define ν-reduction to allow unrestricted unfolding of cofixpoints. Then, $\Sigma, \Gamma_G, \Gamma \vdash e : t$ and $e \rhd e'$ implies $\Sigma, \Gamma_G, \Gamma \vdash e' : t$.*

PROOF SKETCH. By induction on $\Sigma, \Gamma_G, \Gamma \vdash e : t$. Most cases are straightforward.

The case for Rule CASE where $e \rhd e'$ by ι-reduction relies on the fact if x is the name of a (co)-inductive types and appears strictly positively in t then x appears covariantly in t. (This is only true without nested (co)inductive type, which recall, $\text{CIC}\widehat{*}$ disallows in well-formed signatures.)

The case for Rule CASE and e (guarded) ν-reduces to e' requires an unrestricted ν-reduction. After guarded ν-reduction, the target (a cofixpoint) appears in the motive unguarded by a case expression, but must be unfolded to re-establish typing the type t. □

Recall from Section 3 that we disallow nested (co)inductive types in our definition of strict positivity. Unfortunately, as suggested in the above proof, subject reduction breaks in the presence of nested (co)inductive types. We present a counterexample shortly. The root cause of this issue is the removal of polarity annotations from $\text{CIC}\widehat{\ }$ to make $\text{CIC}\widehat{*}$ backward compatible with Coq. In $\text{CIC}\widehat{\ }$, these polarity annotations must be provided by the user in the surface syntax and affect subtyping for (co)inductive types.

Interestingly, counterexamples seem to be impossible to express in our implementation. The counterexamples rely on crafting a set of bad annotations, but the user cannot provide annotations directly and is forced to rely on size inference. The size inference algorithm seems unable to produce these bad annotations. We conjecture that $\text{CIC}\widehat{*}$ programs whose size annotations are generated by the algorithm do enjoy subject reduction. Future study of the inference algorithm should provide insight on how to add nested (co)inductive types to $\text{CIC}\widehat{*}$.

To see how subject reduction for $\text{CIC}\widehat{*}$ fails in the presence of nested (co)inductive types, consider the following example. Again, we omit □ in nonempty environments and brackets in singleton vectors, and we use a Coq-like syntax for case branches.

$$\begin{aligned}\Sigma = ((A : \text{Type})(x : A) &\vdash \text{Eq} : A \to \text{Type} \coloneqq \text{eq\_refl} : \text{Eq}\ A\ x\ x)\\ (\square \vdash \text{N} : \text{Type} &\coloneqq \langle \text{O} : \text{N}, \text{S} : (n : \text{N}) \to (n{=}n : \text{Eq N}\ n\ n) \to \text{N}\rangle)\end{aligned}$$

Note that although N behaves extensionally like $\mathbb{N}$, it cannot be encoded without the use of nested inductive types due to the $(n{=}n : \text{Eq N}\ n\ n)$ argument of S. It is possible to see that

$$\Sigma, \square, \square \vdash \left(\begin{array}{l}\text{case}_{\lambda\_:\text{N}.\text{N}}\ (\text{S O}\ (\text{eq\_refl N}^{\upsilon+1}\ \text{O}))\ \text{of}\\ \mid \text{O} \Rightarrow \text{O}\\ \mid \text{S} \Rightarrow \lambda n : \text{N}.\, \lambda n{=}n : \text{Eq N}\ n\ n.\\ \qquad\qquad \text{seq}\ (\text{eq\_refl}\ (\text{Eq}^{\infty}\ \text{N}^{\upsilon+2}\ n\ n)\ n{=}n)\ \text{O}\end{array}\right) : \text{N}^{\infty} \qquad \text{(case-ex)}$$

where υ is a size variable, $\text{seq} = \lambda\_ : \_.\, \lambda n : \text{N}.\, n$, and we use the shorthand $\upsilon + k$ to mean the k-th successor stage of υ. However, this case expression reduces to the ill-typed expression in $\Sigma, \square, \square$:

$$\left(\begin{array}{l}\lambda n : \text{N}.\, \lambda n{=}n : \text{Eq N}\ n\ n.\\ \quad \text{seq}\ (\text{eq\_refl}\ (\text{Eq}^{\infty}\ \text{N}^{\upsilon+2}\ n\ n)\ n{=}n)\ \text{O}\end{array}\right)\ \text{O}\ (\text{eq\_refl N}^{\upsilon+1}\ \text{O}) \qquad \text{(case-red-ex)}$$

In detail, first observe that the original case expression is well-typed as follows. By Rule CONSTR and Rule CONV, $\Sigma, \square, \square \vdash \text{S O}\ (\text{eq\_refl N}^{\upsilon+1}\ \text{O}) : \text{N}^{\upsilon+2} \leq \text{N}^{\upsilon+3}$. Now, let $\wp = \lambda\_ : \text{N}^{\upsilon+3}.\text{N}^{\infty}$, and let $e_{\text{O}}, e_{\text{S}}$ respectively be the O and S branches in the case expression above. Put $s = \upsilon + 2$ so that $\Sigma, \square, \square \vdash \text{S O}\ (\text{eq\_refl N}^{\upsilon+1}\ \text{O}) : \text{N}^{s+1}$. We can show $\Sigma, \square, \square \vdash e_{\text{O}} = \text{O} : \text{N}^{\infty} \approx \text{branchType}(\Sigma, \cdot, \text{O}, s, \wp)$.

$$\Sigma, \square, (n : \text{N}^{s})(n{=}n : \text{Eq}^{\infty}\ \text{N}^{s}\ n\ n) \vdash (\text{eq\_refl}\ (\text{Eq}^{\infty}\ \text{N}^{\upsilon+2}\ n\ n)\ n{=}n) : \text{Eq}^{\infty}\ \text{N}^{s}\ n\ n$$

by Rule CASE and Rule ABS and because $s = \upsilon + 2$ by choice. Since seq returns its second argument, which is O in e_{S}, it follows that $\Sigma, \square, \square \vdash e_{\text{S}} : \text{N}^{\infty} \approx \text{branchType}(\Sigma, \cdot, \text{S}, s, \wp)$. This verifies (case-ex).

To see that (case-red-ex) is not well-typed, note that eq_refl $(\mathrm{Eq}^{\infty}\ \mathrm{N}^{v+2}\ n\ n)$ $n{=}n$ can only be well-typed in a Γ such that $\Sigma, \Box, \Gamma \vdash n{=}n : \mathrm{Eq}^{\infty}\ \mathrm{N}^{v+2}\ n\ n$, which means that $(n{=}n : \mathrm{Eq}^{-}\ \mathrm{N}^{v+2}\ n\ n) \in \Gamma$ because Rule CONV and Rule ST-APP states that arguments must be convertible (here Eq^{-} means the annotation for Eq is irrelevant to our discussion). In other words, all attempts to type e_S in $\Sigma, \Box, \Box$ will result in $\Sigma, \Box, \Box \vdash e_S : \Pi n : \mathrm{N}^{-}. \Pi n{=}n : \mathrm{Eq}^{-}\ \mathrm{N}^{v+2}\ n\ n. \mathrm{N}^{-}$. On the other hand, since Rule CONV and Rule ST-APP states that arguments must be convertible in order to have subtyping, all attempts to type (eq_refl N^{v+1} O O) in $\Sigma, \Box, \Box$ will result in $\Sigma, \Box, \Box \vdash$ (eq_refl N^{v+1} O O) $: \mathrm{Eq}^{-}\ \mathrm{N}^{v+1}\ \_\ \_$. Since $\mathrm{N}^{v+1} \not\approx \mathrm{N}^{v+2}$, all attempts to type (case-red-ex) will fail, thus breaking subject reduction.

As observed, the key to this counterexample breaking subject reduction is the eq_refl $(\mathrm{Eq}^{\infty}$ $\mathrm{N}^{v+2}\ n\ n)$ $n{=}n$ subterm of e_S: it exploits the fact that subtyping between inductive types requires the parameters to be convertible, as specified in Rule ST-APP. A straightforward attempt to fix this issue would be to modify Rule ST-APP so that $t_1\ t_2 \leq t_1'\ t_2'$ provided that $t_i \leq t_i'$ for $i = 1, 2$. However, since $\mathrm{CIC}\widehat{*}$ allows both universe subtyping and size subtyping, and the universe subtyping rules in the Coq manual [The Coq Development Team 2020] require arguments and parameters to be convertible, this straightforward solution may cause unforeseen problems with universe subtyping.

Alternatively, we may take inspiration from $\mathrm{CIC}\widehat{\_}$, which satisfies subject reduction. This issue is not present in $\mathrm{CIC}\widehat{\_}$ because it syntactically requires parameters supplied to constructors to be bare, so the above example (specifically the term eq_refl $(\mathrm{Eq}^{\infty}\ \mathrm{N}^{v+2}\ n\ n)$ $n{=}n$) would not conform to $\mathrm{CIC}\widehat{\_}$ grammar. However, this requires constructors to be treated separately from normal functions in applications and thus more changes to the Coq kernel with unforeseen consequences.

Another possible solution is to consider the principle of "size irrelevance" in subtyping and type equality checking. This idea, investigated by Abel et al. [2017], essentially allows the type system to ignore sizes where they act as *type arguments*. That is, size information will be irrelevant in constructor applications and term-level function applications. For instance, the size information $\infty, v + 2$ in eq_refl $(\mathrm{Eq}^{\infty}\ \mathrm{N}^{v+2}\ n\ n)$ $n{=}n$ will be treated as irrelevant, thus allowing us to type this term in an environment where $n{=}n$ has type $\mathrm{Eq}^{-}\ \mathrm{N}^{v+1}\ \_\ \_$. However, sizes still remain relevant in places where they act as *regular arguments*, such as v in N^{v}. At this point, how such a solution would be expressed in $\mathrm{CIC}\widehat{*}$ and its metatheoretical implications remain unclear.

5.3 Soundness and Completeness of Size Inference

The size inference algorithm assigns fresh size variables to each (co)inductive type and produces a set of size constraints; however, the typing rules of $\mathrm{CIC}\widehat{*}$ state a relationship between a particular sized term and its sized type. To prove soundness and completeness, we need the notion of a size substitution and statisfaction of a constraint system.

Definition 5.3 (Size substitutions and constraint satisfaction).

- A **size substitution** $\rho : S \to S$ is a map from size expressions to size expressions. The size substitution $[v := s]$ maps the size variable v to the size expression s and maps every other size expresssion to itself. We write $\rho(s)$ to denote the size expression that s is mapped to by ρ. We define $(\rho_1 \circ \rho_2)(s) = \rho_1(\rho_2(s))$ and $\rho(V) = \{s : \forall s' \in V, \rho(s') = s\}$.
  Additionally, we write ρe to denote the sized term e with every size expression in e replaced by its mapping and $\rho\Gamma$ to denote the environment Γ with every size expression in the sized terms of its codomain replaced by its mapping.
- A size substitution ρ **satisfies the constraint system** C, denoted $\rho \vDash C$, if for every constraint $s_1 \sqsubseteq s_2 \in C$, $\rho(s_1) \sqsubseteq \rho(s_2)$ holds.

Soundness and completeness of the size inference algorithm rely on the soundness and completeness of RecCheck, stated below. We refer the reader to Barthe et al. [2005] for the full proofs of SRC and CRC.

Theorem 5.4 (Soundness of RecCheck (SRC)). *If RecCheck*$(C', \tau, V^*, V^{\neq}) = C$*, for every* ρ *such that* $\rho \vDash C$*, given a fresh stage variable* υ*, there exists a* ρ' *such that the following all hold:*

- $\rho' \vDash C'$ *and* $\rho'(\tau) = \upsilon$
- *For all* $\upsilon' \in V^{\neq}$, $([\upsilon := \rho(\tau)] \circ \rho')(\upsilon') = \rho(\upsilon')$
- *For all* $\tau' \in V^*$, $([\upsilon := \rho(\tau)] \circ \rho')(\tau') \sqsubseteq \rho(\tau')$
- $\lfloor \rho'(V^*) \rfloor = \upsilon$ *and* $\lfloor \rho'(V^{\neq}) \rfloor \neq \upsilon$

Intuitively, if RecCheck succeeds with C*, then given any substitution* ρ *that satisfies* C*, there is another substitution* ρ' *that satisfies the original constraint system* C' *while also not violating any of the properties that RecCheck enforces to produce* C*.*

Theorem 5.5 (Completeness of RecCheck (CRC)). *Suppose the following all hold:*

- $\rho \vDash C'$ *and* $\rho(\tau) = \upsilon$
- $\lfloor \rho(V^*) \rfloor = \upsilon$ *and* $\lfloor \rho(V^{\neq}) \rfloor \neq \upsilon$

Then RecCheck$(C', \tau, V^*, V^{\neq}) = C$ *and* $\rho \vDash C$*.*

We can now state soundness and completeness of the checking, inference, and well-formedness portions of the algorithm.

Conjecture 5.6 (Soundness). *Let* Σ *be some fixed signature.*

(1) If $C, \Gamma_G, \Gamma \vdash e^\circ \Leftarrow t \rightsquigarrow C', e$*, then for every* ρ *such that* $\rho \vDash C'$*, we have* $\Sigma, \Gamma_G, \rho\Gamma \vdash \rho e : \rho t$*.*
(2) If $C, \Gamma_G, \Gamma \vdash e^\circ \rightsquigarrow C', e \Rightarrow t$*, then for every* ρ *such that* $\rho \vDash C'$*, we have* $\Sigma, \Gamma_G, \rho\Gamma \vdash \rho e : \rho t$*.*
(3) If $\Gamma_G^\circ \rightsquigarrow \Gamma_G$*, then* $WF(\Sigma, \Gamma_G, \square)$ *holds.*

Proof Sketch. By simultaneous induction on the checking and inference judgements of the algorithm. We refer the reader to Barthe et al. [2005]; Sacchini [2013] for the bulk of the proof details. SRC is used in the cases of Rule a-fix and Rule a-cofix: if RecCheckLoop succeeds, then there is some position annotation of the bare (co)fixpoint type such that RecCheck succeeds, and we can apply SRC. We have added two new typing rules, Rules var-def and const-def, and we conjecture that the algorithm is sound with respect to these two rules as well. (3) requires the additional property that if $u \preceq t$ succeeds then $u \leq t$. □

Soundness remains a conjecture. Key properties that remain to be proven include well-foundedness of environments in the leaf rules of inference, and showing that the premises of Rules var-def and const-def hold in Rules a-var-def and a-const-def, respectively. These are both new additions to the typing rules in contrast to those of $\hat{F}$ [Barthe et al. 2005], so the soundness proof techniques must be adapted as well.

Conjecture 5.7 (Completeness). *Let* Σ *be some fixed signature.*

(1) If $\Sigma, \Gamma_G, \rho\Gamma \vdash e : \rho t$ *and* $\rho \vDash C$*, then there exist* C', ρ', e' *such that* $\forall \upsilon \in SV(\Gamma, t), \rho(\upsilon) = \rho'(\upsilon)$ *and* $\rho' \vDash C'$ *and* $\rho' e' = e$ *and* $C, \Gamma_G, \Gamma \vdash |e| \Leftarrow t \rightsquigarrow C', e'$*.*
(2) If $\Sigma, \Gamma_G, \rho\Gamma \vdash e : t$ *and* $\rho \vDash C$*, then there exist* C', ρ', e', t' *such that* $\forall \upsilon \in SV(\Gamma), \rho(\upsilon) = \rho'(\upsilon)$ *and* $\rho' \vDash C'$ *and* $\rho' e' = e$ *and* $\rho' t' \leq t$ *and* $C, \Gamma_G, \Gamma \vdash |e| \rightsquigarrow C', e' \Rightarrow t'$*.*
(3) If $WF(\Sigma, \Gamma_G, \square)$*, then* $|\Gamma_G| \rightsquigarrow \Gamma_G$*.*

Proof Sketch. By induction on the typing judgement. We refer the reader to Barthe et al. [2005]; Sacchini [2013] for the bulk of the proof details. We conjecture that if a (co)fixpoint with position-annotated types $\overline{t_k}$ is well-typed, then RecCheckLoop can find position annotations for

$\overline{|t_k|}$ that have at least the same position annotations as $\overline{t_k}$ such that RecCheck succeeds, so that we can apply CRC. (3) requires the additional property that if $t' \leq t$ then $t' \preceq t$ will succeed. □

Completeness also remains a conjecture. The core of the proof relies on a property of RecCheck-Loop that we have yet been able to figure out how to prove. Further investigation into the completeness proof of RecCheck from F$\widehat{\ }$ may yield a proof technique.

5.4 Strong Normalization and Logical Consistency

Our ultimate goal and primary future work is to prove strong normalization and logical consistency of CIC$\widehat{*}$.

Conjecture 5.8 (Strong Normalization). *If $\Sigma, \Gamma_G, \Gamma \vdash e : t$ then e contains no infinite reduction sequences.*

Conjecture 5.9 (Logical Consistency). *The type $\Pi p : Prop.\, p$ is uninhabited in CIC$\widehat{*}$.*

Taking inspiration from Barthe et al. [2006]; Sacchini [2011, 2013], we conjecture that these statements can be proven using the Λ-set technique from Altenkrich [1993]; Melliès and Werner [1998]. Our proof attempt is still ongoing.

However, with coinduction and cofixpoints, results from Sacchini [2013] suggest that subject reduction and strong normalization may not be true at the same time, because subject reduction appears to require unrestricted unfolding of cofixpoints, which breaks normalization.

Recall that Coq itself is not strongly normalizing, only weakly normalizing.[3]

```
Definition cbv_omega := fix f (n : nat) := let x := f n in 0.
```

This is a side-effect of relaxing the guard condition to enable more sound programs to type check. Ideally, sufficiently expressive sized typing will allow us to replace the guard condition entirely and regain strong normalization in Coq.

However, an intermediate goal may be to achieve only weak normalization. This would also maintain backward compatibility with Coq, although it is unclear if counterexamples like the above are ever desired in user programs.

6 RELATED WORK

This work is based on CIC$\widehat{\ }$ [Barthe et al. 2006], which describes CIC with sized types and a size inference algorithm. It assumes that position annotations are given by the user, requires each parameter of (co)inductive types to be assigned polarities, and deals only with terms. We have added on top of it global declarations, local definitions, constants and variables annotated by a vector of size expressions, their δ-/Δ-reductions, an explicit treatment of mutually-defined (co)-inductive types and (co)fixpoints, and an intermediate procedure RecCheckLoop to handle missing position annotations, while removing parameter polarities and subtyping rules based on these polarities.

The language CIC$\underline{\widehat{\ }}$ [Sacchini 2011] is similar to CIC$\widehat{\ }$, described in greater detail, but with one major difference: CIC$\underline{\widehat{\ }}$ disallows size variables in the bodies of abstractions, in the arguments of applications, and in case expression branches, making CIC$\underline{\widehat{\ }}$ a strict subset of CIC$\widehat{\ }$. Any size expressions found in these locations must be set to ∞. This "solves" the problem discussed in Section 2 of how to handle defined variables used in (co)fixpoints by disallowing it entirely. In practice, such as in Coq's standard library, aliases are often defined for (co)inductive types, so we designed CIC$\widehat{*}$ to accommodate (co)fixpoints defined over aliases and other type-level computations.

[3]At least, the underlying calculus is not. The counterexample "normalizes" to a stack overflow on machine with finite memory.

$$\frac{}{v : \text{Size}}\ \textsc{size-var} \qquad \frac{}{\infty : \text{Size}}\ \textsc{size-infty} \qquad \frac{s : \text{Size}}{\uparrow s : \text{Size}}\ \textsc{size-succ} \qquad \frac{r : \text{Size} \qquad s : \text{Size}}{r \sqcup^s s : \text{Size}}\ \textsc{size-max}$$

$$\frac{r : \text{Size}{<}s}{r : \text{Size}}\ \textsc{size-lt} \qquad \frac{}{\text{Size} : \text{SizeUniv}}\ \textsc{sizeuniv-size} \qquad \frac{s : \text{Size}}{\text{Size}{<}s : \text{SizeUniv}}\ \textsc{sizeuniv-size-lt}$$

Fig. 18. Typing rules for sizes in Agda

The implementation of RecCheck comes from F$\widehat{~}$ [Barthe et al. 2005], an extension of System F with type-based termination using sized types. Rules relating to coinductive constructions and cofixpoints come from the natural extension of CC$\widehat{\omega}$ [Sacchini 2013], which describes only infinite streams. The size inference algorithm is based on those of CIC$\widehat{~}$, CC$\widehat{\omega}$, and CIC$\widehat{~}_l$ [Sacchini 2014].

Whereas our size algebra supports only a successor operation, *linear* sized types in CIC$\widehat{~}_l$ extends the algebra by including size expressions of the form $n \cdot S$, so that all annotations are of the form $n \cdot v + m$, where m is the number of "hats". Unfortunately, this causes the time complexity of its RecCheck procedure to be worst-case doubly exponential in the number of size variables. However, the set of typeable (and therefore terminating or productive) functions is expanded compared to CIC$\widehat{*}$; functions such as list-doubling could be typed as size-preserving in addition to being terminating. If successor sized types prove practical, augmenting the type system to linear sized types would be worth investigating, depending on whether common programs would cause worst-case behaviour. The most significant change required would be in RecCheck, which must then solve a set of constraints in Presburger arithmetic.

Well-founded sized types in CIC$\widehat{~}_{\sqsubseteq}$ [Sacchini 2015b] are yet another extension of successor sized types. The unpublished manuscript contains a type system, some metatheoretical results, and a size inference algorithm. In essence, it preserves subject reduction for coinductive constructions, and also expands the set of typeable functions.

The proof assistant Agda implements sized types as user-provided size parameters, similar to type parameters. Correspondingly, sizes have the type Size, while Size itself has the type SizeUniv, which is its own type. Figure 18 presents the typing rules for Size; the operator $\uparrow \cdot$ corresponds to our $\hat{\cdot}$, while $\cdot \sqcup^s \cdot$ takes the maximum of two sizes. Additionally, Agda defines the size constructor Size<, which allows the user to specify a size constraint $r \sqsubseteq s$ with the annotation r : Size<s. Whereas CIC$\widehat{~}$'s philosophy is to hide all size annotations from the user with a focus on size inference, Agda opts for allowing users to explicitly write size annotations and treat them almost like terms, yielding greater flexibility in deciding how things should be typed. However, this approach is a non-starter if we wish to maintain backward compatibility with Coq.

7 CONCLUSION

We have presented a design and implementation of sized types for Coq. Our work extends the core language and type checking algorithm of prior theoretical work on sized types for CIC with pragmatic features found in Coq, such as global definitions, and extends the inference algorithm to infer sizes over completely unannotated CIC terms to enable backward compatibility. We implement the design presented in this paper as an extension to Coq's kernel available in the anonymous supplementary material. The design and implementation can be used alone or in conjunction with syntactic guard checking to maximize typeability and compatibility.

REFERENCES

Andreas Abel, Andrea Vezzosi, and Theo Winterhalter. 2017. Normalization by Evaluation for Sized Dependent Types. *Proc. ACM Program. Lang.* 1, ICFP, Article 33 (Aug. 2017), 30 pages. https://doi.org/10.1145/3110277

Thorsten Altenkrich. 1993. *Constructions, Inductive Types and Strong Normalization.* Theses. University of Edinbrugh. https://www.cs.nott.ac.uk/~psztxa/publ/phd93.pdf

Gilles Barthe, Benjamin Grégoire, and Fernando Pastawski. 2005. Practical inference for type-based termination in a polymorphic setting. In *Typed Lambda Calculi and Applications (Lecture Notes in Computer Science, Vol. 3461)*, Urzyczyn, P (Ed.). Springer-Verlag Berlin, Heidelberger Platz 3, D-14197 Berlin, Germany, 71–85. https://doi.org/10.1007/11417170_7

Gilles Barthe, Benjamin Grégoire, and Fernando Pastawski. 2006. $\widehat{\text{CIC}}$: Type-Based Termination of Recursive Definitions in the Calculus of Inductive Constructions. In *Logic for Programming, Artificial Intelligence, and Reasoning, Proceedings (Lecture Notes in Artificial Intelligence, Vol. 4246)*, Hermann, M and Voronkov, A (Ed.). Springer-Verlag Berlin, Heidelberger Platz 3, D-14197 Berlin, Germany, 257–271. https://doi.org/10.1007/11916277_18

Yuichi Komori, Naosuke Matsuda, and Fumika Yamakawa. 2014. A Simplified Proof of the Church—Rosser Theorem. *Stud. Log.* 102, 1 (Feb. 2014), 175–183. https://doi.org/10.1007/s11225-013-9470-y

Paul-André Melliès and Benjamin Werner. 1998. *A Generic Normalisation Proof for Pure Type Systems.* Research Report RR-3548. INRIA. https://hal.inria.fr/inria-00073135 Projet COQ.

Jorge Luis Sacchini. 2011. *On type-based termination and dependent pattern matching in the calculus of inductive constructions.* Theses. École Nationale Supérieure des Mines de Paris. https://pastel.archives-ouvertes.fr/pastel-00622429

Jorge Luis Sacchini. 2013. Type-Based Productivity of Stream Definitions in the Calculus of Constructions. In *2013 28TH Annual IEEE/ACM Symposium on Logic in Computer Science (LICS) (IEEE Symposium on Logic in Computer Science).* IEEE, 345 E 47th St., New York, NY 10017 USA, 233–242. https://doi.org/10.1109/LICS.2013.29

Jorge Luis Sacchini. 2014. Linear Sized Types in the Calculus of Constructions. In *Functional and Logic Programming, FLOPS 2014 (Lecture Notes in Computer Science, Vol. 8475)*, Codish, M and Sumii, E (Ed.). Springer-Verlag Berlin, Heidelberger Platz 3, D-14197 Berlin, Germany, 169–185. https://doi.org/10.1007/978-3-319-07151-0_11

Jorge Luis Sacchini. 2015a. *jsacchini/cicminus.* https://doi.org/10.5281/zenodo.3928999

Jorge Luis Sacchini. 2015b. Well-Founded Sized Types in the Calculus of (Co)Inductive Constructions. (2015). https://web.archive.org/web/20160606143713/http://www.qatar.cmu.edu/~sacchini/well-founded/well-founded.pdf Unpublished paper.

The Coq Development Team. 2020. *The Coq Proof Assistant, version 8.11.0.* https://doi.org/10.5281/zenodo.3744225

$\boxed{\Gamma_G, \Gamma \vdash T \triangleright_{\beta\zeta\delta\Delta\iota\mu\nu} T}$

$$\frac{}{\Gamma_G, \Gamma \vdash (\lambda x : t.\, e_1)\ e_2 \triangleright_\beta e_1[x := e_2]}\ \beta \qquad \frac{}{\Gamma_G, \Gamma \vdash \text{let } x : t := e_1 \text{ in } e_2 \triangleright_\zeta e_2[x := e_1]}\ \zeta$$

$$\frac{(x : t := e) \in \Gamma}{\Gamma_G, \Gamma \vdash x^{\langle s_i \rangle} \triangleright_\delta |e|^\infty \overline{[\infty_i := s_i]}}\ \delta \qquad \frac{(\text{Def } x : t := e.) \in \Gamma_G}{\Gamma_G, \Gamma \vdash x^{\langle s_i \rangle} \triangleright_\Delta e\overline{[\infty_i := s_i]}}\ \Delta$$

$$\frac{}{\Gamma_G, \Gamma \vdash \text{case}_\wp\ (c_\ell\ \overline{p}\ \overline{a}) \text{ of } \langle c_j \Rightarrow e_j \rangle \triangleright_\iota e_\ell\ \overline{a}}\ \iota \qquad \frac{\|\overline{b}\| = n_m - 1 \qquad q_i = \text{fix}_{\langle n_k \rangle, i}\ \langle f_k : t_k := e_k \rangle}{\Gamma_G, \Gamma \vdash q_m\ \overline{b}\ (c\ \overline{p}\ \overline{a}) \triangleright_\mu e_m\overline{[f_k := q_k]}\ \overline{b}\ (c\ \overline{p}\ \overline{a})}\ \mu$$

$$\frac{q_i = \text{cofix}_i\ \langle f_k : t_k := e_k \rangle}{\Gamma_G, \Gamma \vdash \text{case}_\wp\ (q_m\ \overline{b}) \text{ of } \langle c_j \Rightarrow a_j \rangle \triangleright_\nu \text{case}_\wp\ (e_m\overline{[f_k := q_k]}\ \overline{b}) \text{ of } \langle c_j \Rightarrow a_j \rangle}\ \nu$$

Fig. 19. Reduction rules

$\boxed{\Gamma_G, \Gamma \vdash T \triangleright_{\beta\zeta\delta\Delta\iota\mu\nu'} T}$

$$\vdots$$

$$\frac{q_i = \text{cofix}_i\ \langle f_k : t_k := e_k \rangle}{\Gamma_G, \Gamma \vdash q_m \triangleright_{\nu'} e_m\overline{[f_k := q_k]}}\ \nu'$$

Fig. 20. Reduction rules (with unrestricted cofixpoint reduction)

$\boxed{\Gamma_G, \Gamma \vdash T \approx T}$

$$\frac{\begin{array}{c}\Gamma_G, \Gamma \vdash e_1 \triangleright^* e \\ \Gamma_G, \Gamma \vdash e_2 \triangleright^* e\end{array}}{\Gamma_G, \Gamma \vdash e_1 \approx e_2}\ \approx\text{-}\triangleright^* \qquad \frac{\begin{array}{c}\Gamma_G, \Gamma \vdash e_1 \triangleright^* \lambda x : t.\, e \\ \Gamma_G, \Gamma \vdash e_2 \triangleright^* e_2' \\ \Gamma_G, \Gamma(x : t) \vdash e \approx e_2'\ x\end{array}}{\Gamma_G, \Gamma \vdash e_1 \approx e_2}\ \approx\text{-}\eta\text{-L} \qquad \frac{\begin{array}{c}\Gamma_G, \Gamma \vdash e_1 \triangleright^* e_1' \\ \Gamma_G, \Gamma \vdash e_2 \triangleright^* \lambda x : t.\, e \\ \Gamma_G, \Gamma(x : t) \vdash e_1'\ x \approx e\end{array}}{\Gamma_G, \Gamma \vdash e_1 \approx e_2}\ \approx\text{-}\eta\text{-R}$$

Fig. 21. Convertibility rules

A REDUCTION, CONVERTIBILITY, TAKAHASHI TRANSLATION

Figure 19 lists the complete definitions for all reduction rules, including our new rules for δ- and Δ-reduction. Notice that in fixpoints, the n_mth recursive argument needs to be an applied constructor, while cofixpoints can only be reduced as a case expression target. These conditions are required for strong normalization. However, this restricted cofixpoint unfolding causes issues with subject reduction [Sacchini 2013]. Figure 20 presents an alternate unrestricted nonterminating ν'-reduction of cofixpoints outside of case expressions so that subject reduction holds at the cost of normalization.

We define $\triangleright$ as the least compatible closure of $\triangleright_{\beta\zeta\delta\Delta\iota\mu\nu}$, $\triangleright_n$ as the n-step reduction of $\triangleright$, and $\triangleright^*$ as the reflexive–transitive closure of $\triangleright$. Convertibility ($\approx$) is the symmetric closure of $\triangleright^*$ up to η-expansion, and is formally defined in Figure 21.

The Takahashi translation $e^\dagger$ of a term e [Komori et al. 2014] is used in our proof of confluence. Informally, we define it as the simultaneous single-step reduction of all $\beta\zeta\delta\Delta\iota\mu\nu$-redexes of e in left-most inner-most order. We further define $e^{n\dagger}$ as the n-step Takahashi translation of e.

B WELL-FORMEDNESS OF (CO)INDUCTIVE DEFINITIONS

In this section we define what it means for a (co)inductive definition to be *well-formed*, including some required auxilliary definitions. A signature is then well-formed if each of its (co)inductive definitions are well-formed. Note that although we prove subject reduction for CIC$\widehat{*}$ without nested inductive types, we include their definitions for completeness.

Definition B.1 (Strict Positivity). Given some existing sigature Σ, the variable x occurs *strictly positively* in the term t, written $x \oplus t$, if any of the following holds:

- $x \notin \mathsf{FV}(t)$
- $t \approx x\, \overline{e}$ and $x \notin \mathsf{FV}(\overline{t})$
- $t \approx \Pi x : u.\, v$ and $x \notin \mathsf{FV}(u)$ and $x \oplus v$

If nested inductive types are permitted, then $x \oplus t$ may hold if the following also holds:

- $t \approx I_k^{\infty}\, \overline{p}\, \overline{a}$ where $\langle \Delta_p \vdash I_i\ \_ : \_ \rangle \coloneqq \langle c_j : \Pi \Delta_j.\, I_j\ \mathsf{dom}(\Delta_p)\ \overline{t}_j \rangle \in \Sigma$ for some $k \in \overline{\imath}$ and all of the following hold:
  - $\|\overline{p}\| = \|\Delta_p\|$
  - $x \notin \mathsf{FV}(\overline{a})$
  - For every j, if $I_j = I_k$, then $x \circledplus_{I_k} (\Pi \Delta_j.\, I_j\, \overline{p}\, \overline{t}_j)[\mathsf{dom}(\Delta_p) \coloneqq \overline{p}]$

Definition B.2 (Nested Positivity). Given some existing signature Σ, the variable x is *nested positive* in t of I_k, written $x \circledplus_{I_k} t$, if $\langle \Delta_p \vdash I_i\ \_ : \_ \rangle \coloneqq \_ \in \Sigma$ for some $k \in \overline{\imath}$ and any of the following holds:

- $t \approx I_k^{\infty}\, \overline{p}\, \overline{a}$ and $\|\overline{p}\| = \|\Delta_p\|$ and $x \notin \mathsf{FV}(\overline{a})$
- $t \approx \Pi x : u.\, v$ and $x \oplus u$ and $x \circledplus_{I_k} v$

In short, $x \circledplus_I t$ if $t \approx \Pi \Delta.\, I\, \overline{p}\, \overline{a}$ and $x \oplus \Delta$ and $x \notin \mathsf{FV}(\overline{a})$.

Definition B.3 (Constructor Type). The term t is a constructor type for I when:

- $t = I\, \overline{e}$; or
- $t = \Pi x : u.\, v$ where $x \notin \mathsf{FV}(u)$ and v is a constructor type for I; or
- $t = u \to v$ where $x \oplus u$ and v is a constructor type for I.

Note that in particular, this means that $t = \Pi \Delta.\, I\, \overline{e}$ such that $I \oplus u$ for every $u \in \mathsf{codom}(\Delta)$, and the recursive arguments of t are not dependent.

Definition B.4 (Well-formedness of (Co)Inductive Definitions). Suppose we have some signature Σ and some global environment Γ_G. Consider the following (co)inductive definition, where $\overline{p} = \mathsf{dom}(\Delta_p)$.

$$\Delta_p \vdash \langle I_i\, \overline{p} : \Pi \Delta_i.\, w_i \rangle \coloneqq \langle c_j : \Pi \Delta_j.\, I_j\, \overline{p}\, \overline{t}_j \rangle$$

This (co)inductive definition is *well-formed* if the following all hold:

(I1). For every i, there is some w_i' such that $\Sigma, \Gamma_G, \Delta_p \vdash \Pi \Delta_i.\, w_i : w_i'$ holds.
(I2). For every j, there is some w_j such that $\Sigma, \Gamma_G, \Delta_p (I_j^{\infty} : \Pi \Delta_p.\, \Pi \Delta_i.\, w_i) \vdash \Pi \Delta_j.\, I_j^{\infty}\, \overline{p}\, \overline{t}_j : w_j$ holds.
(I3). For every j, $\Pi \Delta_j.\, I_j\, \overline{p}\, \overline{t}_j$ is a constructor type for I_j. Note that this implies $I_j \oplus \mathsf{codom}(\Delta_j)$.
(I4). For every i, j, all (co)inductive types in the terms $\mathsf{codom}(\Delta_p), \mathsf{codom}(\Delta_i), \mathsf{codom}(\Delta_j)$ are annotated with ∞.

C SUPPLEMENTARY FIGURES

Figure 22 lists the sets Axioms, Rules, and Elims, which are the same as for CIC. Figure 23 catalogues the various metafunctions introduced in Section 4.

$$\begin{aligned}
\text{Axioms} &= \{(\text{Prop}, \text{Type}_1), (\text{Set}, \text{Type}_1), (\text{Type}_i, \text{Type}_{i+1})\} \\
\text{Rules} &= \{(w, \text{Prop}, \text{Prop}) : w \in U\} \cup \{(w, \text{Set}, \text{Set}) : w \in \{\text{Prop}, \text{Set}\}\} \\
&\quad \cup \{(\text{Type}_i, \text{Type}_j, \text{Type}_k) : k = \max(i, j)\} \\
\text{Elims} &= \{(w_i, w, I_i) : w_i \in \{\text{Set}, \text{Type}\}, w \in U, I_i \in \Sigma\} \cup \{(\text{Prop}, \text{Prop}, I_i) : I_i \in \Sigma\} \\
&\quad \cup \{(\text{Prop}, w, I_i) : w \in U, I_i \in \Sigma, I_i \text{ empty or singleton}\}
\end{aligned}$$

Fig. 22. Universe relations: Axioms, Rules, and Eliminations

axiom : $U \rightarrow U$	Produces type of universe
rule : $U \times U \rightarrow U$	Produces universe of product type given universe of argument and return types
elim : $U \times U \times \mathcal{I} \rightarrow ()$	Checks that given universe w_k of (co)inductive type I_k of case expression target can be eliminated to a type with given universe w; can fail
$\cdot \preceq \cdot : T \times T \rightarrow \mathbb{P}(S \times S)$	Checks subtypeability and produces a size constraint set; can fail
fresh : $\mathbb{N}^+ \rightarrow \overline{\mathcal{V}}$	Produces given number of fresh size variables, putting them into $\mathcal{V}$
decompose : $T \times \mathbb{N}^0 \rightarrow \Delta \times T$	Splits function type into given number of arguments and return type; can fail
caseSize : $\mathcal{I} \times S \times \mathcal{V} \rightarrow \mathbb{P}(S \times S)$	Given (co)inductive type I_k, size expression s, and size variable υ_k, returns $\{s \sqsubseteq \hat{\upsilon}_k\}$ if I_k is inductive and $\{\hat{\upsilon}_k \sqsubseteq s\}$ if I_k is coinductive
shift : $T \rightarrow T$	Replaces each position size annotation by successor
setRecStars : $T^\circ \times \mathbb{N}^+ \rightarrow T^*$	Given index n, annotates nth argument type I and all other argument and return types with same type I with position annotations; can fail
setCorecStars : $T^\circ \rightarrow T^*$	Annotates return argument type I and all other argument types with same type I with position annotations; can fail
getRecVar : $T \times \mathbb{N}^+ \rightarrow \mathcal{V}^*$	Given index n, retrieves position size variable of nth argument type; can fail
getCorecVar : $T \rightarrow \mathcal{V}^*$	Retrieve position size variable of return type; can fail
eraseToGlobal : $T \times T \rightarrow T^\iota$	Given a terms u and t, erase t to a global term such that global annotations appear in t where position size variables appear in u
RecCheckLoop : $C \times \overline{\mathcal{V}^*} \times \overline{T} \times \overline{T} \rightarrow C$	Calls RecCheck recursively, shrinking $\mathcal{V}^*$ each time; can fail via RecCheck
RecCheck : $C \times \mathcal{V}^* \times \mathbb{P}(\mathcal{V}^*) \times \mathbb{P}(\mathcal{V}) \rightarrow C$	Checks termination and productivity using size constraints, returning a new set of constraints; can fail

Fig. 23. Summary of metafunctions used in the size inference algorithm