Practical Sized Typing for Coq

Jonathan Chan
University of British Columbia
jon@alumni.ubc.ca

William J. Bowman
University of British Columbia
wjb@williamjbowman.com

Abstract

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 syntactic criteria that prevent users from easily writing obviously terminating or productive programs, 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) with support for global definitions found in Gallina, and extend the sized-type inference algorithm to support completely unannotated Gallina terms. This allows our design to maintain complete backwards 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 the proof assistant 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 add n m : nat :=
  match n with
  | O => m
  | S p => S (add p m)
  end.
Variable A : Type.
CoFixpoint const a : Stream A := Cons a (const a).
```

In the recursive call to add, 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, _ | _, 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, it does not change its behaviour, but since it can return O which is not a structurally-smaller term of n in the recursive call to div, the guard predicate is no longer satisfied. Then acceptance of div depends 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 write a proof for 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, (co)-inductive types are annotated with a size annotation, which follow a simple size algebra: $s := \upsilon \mid \hat{s} \mid \infty$. If some object has size s, then the object wrapped in a 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 type of the function of the recursive call must have a smaller size than that of the outer fixpoint; for productivity, the outer cofixpoint must have a larger size than that of the function of the corecursive call. In short, they both follow the following (simplified) typing rule.

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

We can then assign `minus` the type $\texttt{nat}^{\iota} \rightarrow \texttt{nat} \rightarrow \texttt{nat}^{\iota}$, indicating that it 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 `| 0, _ => n` with `| 0, _ => 0` 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 sized-preserving, expanding the set of terminating and productive functions that can be accepted.

However, past work on sized types in the Calculus of (Co)-Inductive Constructions (CIC) [2, 4] have some practical issues:

- They require nontrivial additions to the language, making existing Coq code incompatible without adjustments that must be made manually. 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 their parameters.
- They require the (co)recursive arguments of (co)fixpoints to have literal (co)inductive types. That is, the types cannot be any other expressions that might otherwise reduce to (co)inductive types.
- They do not specify how global definitions should be handled. Ideally, size inference should be done locally, i.e. confined to within a single global definition.

In this paper, we present CIC$\widehat{*}$, an extension of CIC$\widehat{~}$ [2] that resolves these issues without requiring any changes to the surface syntax of Coq. We have also implemented a size inference algorithm based on CIC$\widehat{*}$ within Coq's kernel[1]. In Section 2, we define the syntax of the language, as well as typing rules that handle both terms and global definitions. We then present in Section 3 a size inference algorithm from CIC terms to sized CIC$\widehat{*}$ terms that details how we annotate the types of (co)fixpoints, how we deal with the lack of polarities, and how global definitions are typed, along with the usual termination and productivity checking. Finally, we review and compare with the past work done on sized typing in CIC and related languages in Section 5. Additionally, we provide some illustrating examples in Section 4.

[1]Link removed for double-blinding; see anonymous supplementary material.

$$\begin{array}{rcll}
\overline{\cdot} & ::= & \cdot \mid \cdot\ \overline{\cdot} & \text{sequences} \\
S & ::= & \mathcal{V} \mid \mathcal{P} \mid \widehat{S} \mid \infty & \text{stage annotations} \\
U & ::= & \text{Prop} \mid \text{Set} \mid \text{Type}_n & \text{set of universes}
\end{array}$$

$$\begin{array}{rll}
T[\alpha] ::= & (T[\alpha]) & \\
& \mid U & \textit{universes} \\
& \mid \mathcal{X} \mid \mathcal{X}^{\langle \alpha \rangle} & \textit{variables} \\
& \mid \lambda \mathcal{X} : T^{\circ}.T[\alpha] & \textit{abstraction} \\
& \mid T[\alpha]T[\alpha] & \textit{application} \\
& \mid \Pi \mathcal{X} : T[\alpha].T[\alpha] & \textit{function types} \\
& \mid \text{let } \mathcal{X} : T^{\circ} := T[\alpha] \text{ in } T[\alpha] & \textit{let-in (definitions)} \\
& \mid \mathcal{I}^{\alpha} & \textit{(co)inductive types} \\
& \mid C & \textit{(co)ind. constructors} \\
& \mid \text{case}_{T^{\circ}}\ T[\alpha] \text{ of } \langle C \Rightarrow T[\alpha] \rangle & \textit{case analysis} \\
& \mid \text{fix}_{\langle n \rangle, m}\ \langle \mathcal{X} : T^{*} := T[\alpha] \rangle & \textit{fixpoint} \\
& \mid \text{cofix}_{n}\ \langle \mathcal{X} : T^{*} := T[\alpha] \rangle & \textit{cofixpoint}
\end{array}$$

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

2 CIC$\widehat{*}$

In this section, we present CIC$\widehat{*}$, a superset of CIC, the underlying formal language of Coq, and adds to it sized types in the style of CIC$\widehat{~}$. Beginning with user-provided code in CIC, we produce sized CIC$\widehat{*}$ terms with sized types, check for termination and productivity, and finish by erasing the sizes to produce full CIC$\widehat{*}$ terms.

$$\text{CIC} \xrightarrow{\text{inference}} \text{sized CIC}\widehat{*} \xrightarrow{\text{erasure}} \text{full CIC}\widehat{*}$$

Before we delve into the details of what sized and full terms are, or how inference and erasure are done, we first introduce our notation.

2.1 Notation

Figure 1 presents the syntax of CIC$\widehat{*}$, whose terms are parametrized 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 use $\mathcal{X}$ for term variable names, $\mathcal{V}$ for stage variable names, $\mathcal{P}$ for position stage variable names, $\mathcal{I}$ for (co)inductive type names, and C for (co)inductive constructor names. (The distinction between $\mathcal{V}$ and $\mathcal{P}$ will be important when typing (co)fixpoints and global definitions). We use the overline $\overline{\cdot}$ to denote a sequence of some construction: for instance, $\overline{\mathcal{V}}$ is a sequence of stage variables $\mathcal{V} \ldots \mathcal{V}$.

In the syntax, the brackets $\langle \cdot \rangle$ delimits a vector of comma-separated constructions. In the grammar of Figure 1, the

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

Figure 2. Kinds of annotated terms

construction inside the brackets denote the pattern of the elements in the vector. For instance, the branches of a case analysis are $\langle C \Rightarrow T, \ldots, C \Rightarrow T \rangle$. Finally, we use i, j, k, ℓ, m, n to represent strictly positive integers.

$\text{CIC}\widehat{*}$ resembles the usual CIC, but there are some important differences:

- **Inductive types** can carry annotations that represent their size, e.g. $\texttt{Nat}^{\upsilon}$. This is the defining feature of sized types. They can also have position annotations, e.g. $\texttt{Nat}^{*}$, which marks the type as that of the recursive argument or return value of a (co)fixpoint. This is similar to `struct` annotations in Coq that specify the structurally-recursive argument.
- **Variables** may have a vector of annotations, e.g. $x^{\langle \upsilon_1, \upsilon_2 \rangle}$. If the variable is bound to a type containing (co)inductive types, we can assign the annotations to each (co)inductive type during reduction. For instance, if x were defined by $x : \text{Set} := \texttt{List Nat}$, then the example would reduce to $\texttt{List}^{\upsilon_1}\ \texttt{Nat}^{\upsilon_2}$. This is important in the typing algorithm in Section 3.
- **Definitions** are explicitly part of the syntax, in constrast to $\widehat{\text{CIC}}$ and $\underline{\widehat{\text{CIC}}}$ [4]. This reflects the actual structure in Coq's kernel.
- We also treat **mutual (co)fixpoints** 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.

We also refer to definitions [3] as *let-ins* to avoid confusion with local and global definitions in environments.

Figure 2 lists shorthand for the kinds of annotated terms that we will use. Bare terms as used in the grammar are necessary for subject reduction [4]. Position terms 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. Global terms appear in the types of global definitions, with ι marking types with preserved sizes. Sized terms are used for termination- and productivity-checking, and full terms appear in the types and terms of global declarations.

In terms of type checking and size inference, we proceed as follows:

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

$$
\begin{array}{rcll}
D[\alpha] & ::= & & \text{local declarations} \\
 & | & \mathcal{X} : T[\alpha] & \textit{local assumption} \\
 & | & \mathcal{X} : T[\alpha] := T[\alpha] & \textit{local definition} \\
D_G & ::= & & \text{global declarations} \\
 & | & \text{Assum } \mathcal{X} : T^{\infty}. & \textit{global assumption} \\
 & | & \text{Def } \mathcal{X} : T^{\iota} := T^{\infty}. & \textit{global definition}
\end{array}
$$

$$
\begin{array}{rcll}
\Gamma & ::= & \square \mid \Gamma(D) & \text{local environments} \\
\Gamma_G & ::= & \square \mid \Gamma_G(D_G) & \text{global environment} \\
\Delta[\alpha] & ::= & \square \mid \Delta[\alpha](\mathcal{X} : T[\alpha]) & \text{assumption environments}
\end{array}
$$

Figure 3. Declarations and environments

$$
\begin{array}{lll}
e, a, p, \wp \in T[\alpha] \text{ (expressions)} & \upsilon, \rho \in \mathcal{V} \cup \mathcal{P} & \in U \\
t, u, v \in T[\alpha] \text{ (types)} & V \in \mathbb{P}(\mathcal{V}) & I \in \mathcal{I} \\
f, g, h, x, y, z \in \mathcal{X} & s \in S & c \in C
\end{array}
$$

Figure 4. Metavariables

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-ins, respectively, while global ones are entire programs. Notice that global declarations have no sized terms: by discarding size information, we can infer sizes locally rather than globally. Local declarations and assumption environments are parametrized 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 indexed by n, m, i, j, k, ℓ, or integer literals. If an index appears under an overline, the sequence it represents spans the range of the index, usually given implicitly; for instance, given i inductive types, $\overline{I_k^{s_k}} = I_1^{s_1} \ldots I_i^{s_i}$. Notice that this is *not* the same as an index outside of the underline, such as in $\overline{a}_k$, which represents the kth sequence of terms a. Indices also appear in syntactic vectors; for example, given a case analysis with j branches, we write $\langle c_\ell \Rightarrow e_\ell \rangle$ for the vector $\langle c_1 \Rightarrow e_1, \ldots, c_j \Rightarrow e_j \rangle$.

Finally, we use $t[x := e]$ to denote the term t with free variable x substituted by expression e, and $t[\upsilon := s]$ to denote the term t with stage variable υ substituted by stage annotation s. Occasionally we use $t\overline{[\infty_i := s_i]}$ to denote the substitutions of all full annotations in t by the stage annotations in $\overline{s_i}$ in an arbitrary order.

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

$$\Delta_p \vdash \langle I_i \ \mathrm{dom}(\Delta_p) : \Pi\Delta_{a_i}.{}_i \rangle \coloneqq \langle c_j : \Pi\Delta_j.I_{k_j}\ \mathrm{dom}(\Delta_p)\ \bar{t}_j \rangle$$

Figure 5. Inductive definitions and signature

2.1.1 Mutual (Co)Inductive Definitions

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

- Δ_p, the parameters of the (co)inductive types;
- I_i, their names;
- Δ_{a_i}, the indices (or arguments) of these (co)inductive types;
- ${}_i$, their universes;
- c_j, the names of their constructors;
- Δ_j, the arguments of these constructors;
- I_{k_j}, the (co)inductive types of the fully-applied constructors; and
- $\bar{t}_j$, the indices of those (co)inductive types.

As an example, the usual `Vector` type would be defined in the language as:

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

As with mutual (co)fixpoints, we treat mutual (co)inductive definitions explicitly. Furthermore, in contrast to CIC$\widehat{\ }$ and CIC$\widehat{_}$, 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 bivariant in the subtyping Rule (st-app).

The well-formedness of (co)inductive definitions depends on certain syntactic conditions such as strict positivity. Since we assume definitions in Coq to be valid here, we do not list these conditions, and instead refer the reader to clauses I1–I9 in [4], clauses 1–7 in [2], and [8].

2.1.2 Metafunctions

We declare the following metafunctions:

- $\text{SV} : T \to \mathbb{P}(\mathcal{V} \cup \mathcal{P})$ returns the set of stage variables in the given sized term;
- $\text{PV} : T \to \mathbb{P}(\mathcal{P})$ returns the set of position stage variables in the given sized term;
- $\lfloor . \rfloor : S \setminus \{\infty\} \to \mathcal{V} \cup \mathcal{P}$ returns the stage variable in the given finite stage annotation;
- $\|\cdot\| : * \to \mathbb{N}^0$ returns the cardinality of the given argument (e.g. vector length, set size, etc.);
- $[\![.]\!] : T \to \mathbb{N}^0$ counts the number of stage 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 full terms;
- $|\cdot|^{*} : T \to T^{*}$ erases stage annotations with variables in $\mathcal{P}$ to $*$ and all others to bare; and
- $|\cdot|^{\iota} : T \to T^{\iota}$ erases stage annotations with variables in $\mathcal{P}$ to ι and all others to ∞.

They are defined in the obvious way. 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 to denote membership in contexts and signatures:

- $x \in \Gamma$ means there is some assumption or definition with variable name x in the local context, and similarly for Γ_G;
- $I \in \Sigma$ means the (co)inductive definition of type I is in the signature.

$$\frac{\mathrm{WF}(\Sigma, \Gamma_G, \Gamma) \qquad (x : t \coloneqq e) \in \Gamma}{\Sigma, \Gamma_G, \Gamma \vdash x^{\langle s_i \rangle} \rhd_{\delta} |e|^{\infty}\overline{[\infty_i \coloneqq s_i]}}\ (\delta\text{-local})$$

$$\frac{\mathrm{WF}(\Sigma, \Gamma_G, \Gamma) \qquad (\text{Def } x : t \coloneqq e.) \in \Gamma_G}{\Sigma, \Gamma_G, \Gamma \vdash x^{\langle s_i \rangle} \rhd_{\Delta} e\overline{[\infty_i \coloneqq s_i]}}\ (\Delta\text{-global})$$

Figure 6. Reduction rules for local and global definitions

2.2 Reduction Rules

The reduction rules are the usual ones for β-reduction (function application), ζ-reduction (let-in evaluation), ι-reduction (case expressions), μ-reduction (fixpoint expressions), ν-reduction (cofixpoint expressions), δ-reduction (local definitions), Δ-reduction (global definitions), and η-equivalence. We define convertibility ($\approx$) as the reflexive–symmetric–transitive closure of reductions up to η-equivalence. We refer the reader to [2, 4, 5, 8] for precise details and definitions.

In the case of δ-/Δ-reduction, if the variable has annotations, we define additional rules, as shown in Figure 6. These reduction rules are particularly important for the size inference algorithm. 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 allows for correct substaging relations derived from subtyping relations. 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, an (un)applied (co)inductive type, an (un)applied constructor, or an unapplied (co)fixpoint, with inner terms unreduced.

$$\frac{}{s \sqsubseteq \infty}\ \text{(ss-infty)} \qquad \frac{}{s \sqsubseteq s}\ \text{(ss-refl)} \qquad \frac{}{s \sqsubseteq \hat{s}}\ \text{(ss-succ)}$$

$$\frac{s_1 \sqsubseteq s_2 \qquad s_2 \sqsubseteq s_3}{s_1 \sqsubseteq s_3}\ \text{(ss-trans)}$$

Figure 7. Substaging rules

$$\frac{}{\text{Prop} \leq \text{Set} \leq \text{Type}_1} \quad \frac{}{\text{Type}_i \leq \text{Type}_{i+1}}\ \text{(st-cumul)}$$

$$\frac{t \approx u}{t \leq u}\ \text{(st-conv)} \qquad \frac{t \leq u \qquad u \leq v}{t \leq v}\ \text{(st-trans)}$$

$$\frac{t_2 \approx t_1 \qquad u_1 \leq u_2}{\Pi x : t_1 . u_1 \leq \Pi y : t_2 . u_2}\ \text{(st-prod)}$$

$$\frac{t_1 \leq t_2 \qquad u_1 \approx u_2}{t_1 u_1 \leq t_2 u_2}\ \text{(st-app)}$$

$$\frac{I \text{ inductive} \qquad s_1 \sqsubseteq s_2}{I^{s_1} \leq I^{s_2}}\ \text{(st-ind)}$$

$$\frac{I \text{ coinductive} \qquad s_2 \sqsubseteq s_1}{I^{s_1} \leq I^{s_2}}\ \text{(st-coind)}$$

Figure 8. Subtyping rules

$$\frac{}{\text{WF}(\square, \square, \square)}\ \text{(wf-nil)}$$

$$\frac{\Sigma, \Gamma_G, \Gamma \vdash t : \qquad x \notin \Gamma}{\text{WF}(\Sigma, \Gamma_G, \Gamma(x : t))}\ \text{(wf-local-assum)}$$

$$\frac{\Sigma, \Gamma_G, \Gamma \vdash e : t \qquad x \notin \Gamma}{\text{WF}(\Sigma, \Gamma_G, \Gamma(x : t := e))}\ \text{(wf-local-def)}$$

$$\frac{\Sigma, \Gamma_G, \Gamma \vdash t : \qquad x \notin \Gamma_G}{\text{WF}(\Sigma, \Gamma_G(\text{Assum}\ x : |t|^{\infty}.), \square)}\ \text{(wf-global-assum)}$$

$$\frac{\Sigma, \Gamma_G, \Gamma \vdash e : t \qquad x \notin \Gamma_G}{\text{WF}(\Sigma, \Gamma_G(\text{Def}\ x : |t|^{\iota} := |e|^{\infty}.), \square)}\ \text{(wf-global-def)}$$

Figure 9. Well-formedness of environments

$$\textsc{indType}(\Sigma, I_k) = \Pi\Delta_p . \Pi\Delta_{a_k} . k$$

$$\textsc{constrType}(\Sigma, c_\ell, \overline{s_i}) =$$

$$\Pi\Delta_p . \Pi\Delta_\ell \overline{[I_i^{\infty} := I_i^{s_i}]} . I_{k_\ell}^{\hat{s}_{k_\ell}}\ \text{dom}(\Delta_p)\ \overline{t}_\ell$$

$$\textsc{motiveType}(\Sigma, \overline{p}, , I_k^s) =$$

$$\Pi\Delta_{a_k}[\text{dom}(\Delta_p) := \overline{p}] . \Pi\_ : I_k^s\ \overline{p}\ \text{dom}(\Delta_{a_k}).$$

$$\textsc{branchType}(\Sigma, \overline{p}, c_\ell, \overline{s_i}, \wp) =$$

$$\Pi\Delta_\ell \overline{[I_i^{\infty} := I_i^{s_i}]}[\text{dom}(\Delta_p) := \overline{p}] . \wp\ \overline{t}_\ell\ (c_\ell\ \overline{p}\ \text{dom}(\Delta_\ell))$$

$$\textit{where} \quad k \in \overline{\imath}, \ell \in \overline{\jmath},$$

$$\left(\Delta_p \vdash \langle I_i\ \_ : \Pi\Delta a_i . i\rangle := \langle c_j : \Pi\Delta_j . I_{k_j}\ \_\ \overline{t}_j\rangle\right) \in \Sigma$$

Figure 10. Metafunctions for typing rules

2.3 Subtyping Rules

First, we define the substaging relation for our stage annotations in Figure 7. Additionally, we define $\widehat{\infty}$ to be equivalent to ∞.

We define the subtyping rules for sized types in Figure 8. There are some key features to note:

- 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 stage annotations; coinductive types are **contravariant**. (st-ind) (st-coind)
- By the type application rule, the parameters of polymorphic types are **bivariant**. (st-app)

We can intuitively understand the covariance of inductive types by considering stage annotations as a measure of how many constructors "deep" an object can at most be. If a list has type $\texttt{List}^s t$, then a list with one more element can be said to have type $\texttt{List}^{\hat{s}} t$. Furthermore, by the substaging and subtyping rules, $\texttt{List}^s t \leq \texttt{List}^{\hat{s}} t$: 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 stage annotations as a measure of how many constructors an object must at least "produce". A coinductive stream $\texttt{Stream}^{\hat{s}}$ that produces at least $\hat{s}$ "many" elements can also produce at least s "many" elements, so we have the contravariant relation $\texttt{Stream}^{\hat{s}} \leq \texttt{Stream}^s$, in accordance with the rules.

As previously mentioned, inductive definitions do not have polarities, so there is no way to indicate whether parameters are are covariant, contravariant, or invariant. As a compromise, we treat all parameters as invariant, which we instead call *bivariant*. This is because, algorithmically speaking, the subtyping relation would produce *both* substaging constraints (and not *neither*, as *invariant* suggests). For instance, $\texttt{List}^{s_1}\ \texttt{Nat}^{s_3} \leq \texttt{List}^{s_2}\ \texttt{Nat}^{s_4}$ yields $\texttt{Nat}^{s_3} \approx \texttt{Nat}^{s_4}$, which yields both $s_3 \sqsubseteq s_4$ and $s_4 \sqsubseteq s_3$. A formal description of the subtyping algorithm is presented in Section 3.

2.4 Typing Rules

We now present the typing rules of CIC$\widehat{*}$. Note that these are type-checking rules for *sized* terms, whose annotations will come from size inference in Section 3.

We begin with the rules for well-formedness of local and global environments, presented in Figure 9. As mentioned earlier, we do not cover the well-formedness of signatures. Because well-typed terms are sized, we erase annotations when putting declarations in the global environment in Rules (wf-global-assum) and (wf-global-def) as an explicit indicator that we only use stage variables within individual global declarations. The declared type of global definitions are annotated with global annotations in Rule (wf-global-def); these annotations are used by the typing rules.

$$\frac{v \notin \mathrm{SV}(t)}{v \text{ pos } t} \qquad \frac{v \notin \mathrm{SV}(t)}{v \text{ neg } t}$$

$$\frac{v \text{ neg } t \qquad v \text{ pos } u}{v \text{ pos } \Pi x : t.u} \qquad \frac{v \text{ pos } t \qquad v \text{ neg } u}{v \text{ neg } \Pi x : t.u}$$

$$\frac{v \notin \mathrm{SV}(\overline{a}) \qquad I \text{ inductive}}{v \text{ pos } I^s \overline{a}}$$

$$\frac{v \notin \mathrm{SV}(\overline{a}) \qquad I \text{ coinductive}}{v \text{ neg } I^s \overline{a}}$$

$$\frac{v \notin \mathrm{SV}(\overline{a}) \qquad I \text{ inductive} \qquad v \neq \lfloor s \rfloor}{v \text{ neg } I^s \overline{a}}$$

$$\frac{v \notin \mathrm{SV}(\overline{a}) \qquad I \text{ coinductive} \qquad v \neq \lfloor s \rfloor}{v \text{ pos } I^s \overline{a}}$$

Figure 11. Positivity/negativity of stage variables in terms

The typing rules for sized terms are given in Figure 12. In the style of a Pure Type System, 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 analyses, respectively. These are the same as in CIC and are listed in Figure 17 in Appendix A for reference. Metafunctions that construct some important function types are listed in Figure 10; they are also used by the inference algorithm in Section 3. Finally, the typing rules use the notions of positivity and negativity, whose rules are given in Figure 11, describing where the position annotations of fixpoints are allowed to appear. We go over the typing rules in detail shortly.

Before we proceed, 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 an unbound ranging index, i.e. 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 fixpoint type judgement in Rule (fix) for all mutually-defined fixpoints. Finally, we use _ to omit irrelevant constructions for readability.

The typing rule for assumptions, universes, products, applications, and convertibility are unchanged from CIC and are provided for reference in Figure 15 in Appendix A. Rules (abs) and (let-in) differ from CIC 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 in [4].

The first significant usage of stage 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 stages with the same length as the number of stage annotations in the body, 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 stage annotations.

The type of a (co)inductive type is a function type from its parameters Δ_p and its indices Δ_{a_k} to its universe $_k$. The (co)inductive type itself holds a single stage annotation.

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_k applied to the parameters and its indices $\overline{t}_\ell$. Stage annotations appear in two places:

- In the argument types of the constructor. For each (co)-inductive type I_i, we annotate their occurrences in Δ_ℓ with its own stage annotation s_i.
- On the (co)inductive type of the fully-applied constructor. If the constructor belongs to the inductive type I_k, then it is annotated with the successor of the kth stage annotation, $\hat{s}_k$. 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`:

$$\begin{aligned}&\mathrm{VCons} : (A : \mathrm{Type}) \to (n : \mathrm{Nat}^\infty) \to A \to \mathrm{Vector}^s\ A\ n\\ &\qquad \to \mathrm{Vector}^{\hat{s}}\ A\ (\mathrm{S}\ n).\end{aligned}$$

It has a single parameter A and $\mathrm{S}\ n$ 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 $\hat{s}$.

A case analysis has three important parts:

- The **target** e. It must have a (co)inductive type I_k and a successor stage annotation $\hat{s}_k$ so that any constructor arguments can have the predecessor stage annotation.
- The **motive** $\wp$. It is an abstraction over the indices Δ_a of the target type and the target itself, and produces the return type of the case analysis.
  This presentation of the return type differs from those of [4–6], where the case analysis 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.
  Note that, like in the type of constructors, for each (co)-inductive type I_i, we annotate their occurrences in Δ_j with its own stage annotation s_i, with the kth stage annotation being the predecessor of the target's stage annotation, s_k.

The type of the entire case analysis 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.

$$\frac{\mathrm{WF}(\Sigma, \Gamma_G, \Gamma) \qquad (x : t := e) \in \Gamma \qquad \|\overline{s_i}\| = [\![e]\!]}{\Sigma, \Gamma_G, \Gamma \vdash x^{\langle s_i \rangle} : t}\ \text{(var-def)} \qquad \frac{\mathrm{WF}(\Sigma, \Gamma_G, \Gamma) \qquad (\mathrm{Def}\ x : t := e.) \in \Gamma_G \qquad \|\overline{s_i}\| = [\![e]\!]}{\Sigma, \Gamma_G, \Gamma \vdash x^{\langle s_i \rangle} : t[\iota := s]}\ \text{(const-def)}$$

$$\frac{\Sigma, \Gamma_G, \Gamma(x : t) \vdash e : u}{\Sigma, \Gamma_G, \Gamma \vdash \lambda x : |t|.e : \Pi x : t.u}\ \text{(abs)} \qquad \frac{\Sigma, \Gamma_G, \Gamma \vdash e_1 : t \qquad \Sigma, \Gamma_G, \Gamma(x : t := e_1) \vdash e_2 : u}{\Sigma, \Gamma_G, \Gamma \vdash \text{let}\ x : |t| := e_1\ \text{in}\ e_2 : u[x := e_1]}\ \text{(let-in)}$$

$$\frac{\mathrm{WF}(\Sigma, \Gamma_G, \Gamma)}{\Sigma, \Gamma_G, \Gamma \vdash I_k^s : \textsc{indType}(\Sigma, I_k)}\ \text{(ind)} \qquad \frac{\mathrm{WF}(\Sigma, \Gamma_G, \Gamma)}{\Sigma, \Gamma_G, \Gamma \vdash c_\ell : \textsc{constrType}(\Sigma, c_\ell, \overline{s_i})}\ \text{(constr)}$$

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

$$\frac{\begin{array}{c} t_k \approx \Pi\Delta_{k_1}.\Pi x_k : I_k^{\upsilon_k}\ \overline{a}_k.\Pi\Delta_{k_2}.u_k \qquad \|\Delta_k\| = n_m - 1 \\ \upsilon_k\ \text{pos}\ \Delta_{k_1}, \Delta_{k_2}, u_k \qquad \upsilon_k \notin \mathrm{SV}(\Gamma, \Delta_k, \overline{a}_k, e_k) \qquad \upsilon_k, \lfloor s \rfloor \in \mathcal{P} \\ \Sigma, \Gamma_G, \Gamma \vdash t_k : _k \qquad \Sigma, \Gamma_G, \Gamma\overline{(f_k : t_k)} \vdash e_k : t_k[\upsilon_k := \hat{\upsilon}_k] \end{array}}{\Sigma, \Gamma_G, \Gamma \vdash \text{fix}_{\langle n_k \rangle, m}\ \langle f_k : |t_k|^* := e_k \rangle : t_m[\upsilon_m := s]}\ \text{(fix)} \qquad \frac{\begin{array}{c} t_k \approx \Pi\Delta_k.I_k^{\upsilon_k}\ \overline{a}_k \\ \upsilon_k\ \text{neg}\ \Delta_k \qquad \upsilon_k \notin \mathrm{SV}(\Gamma, \overline{a}_k, e_k) \qquad \upsilon_k, \lfloor s \rfloor \in \mathcal{P} \\ \Sigma, \Gamma_G, \Gamma \vdash t_k : _k \qquad \Sigma, \Gamma_G, \Gamma\overline{(f_k : t_k)} \vdash e_k : t_k[\upsilon_k := \hat{\upsilon}_k] \end{array}}{\Sigma, \Gamma_G, \Gamma \vdash \text{cofix}_m\ \langle f_k : |t_k|^* := e_k \rangle : t_m[\upsilon_m := s]}\ \text{(cofix)}$$

Figure 12. Typing rules (excerpt)

Finally, we have the types of fixpoints and cofixpoints, whose typing rules are very similar. 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. For fixpoints, the type of the n_kth argument, the recursive argument, is an inductive type annotated with a stage variable υ_k. For cofixpoints, the return type is a coinductive type annotated with υ_k. The positivity or negativity of υ_k in the rest of t_k indicate where υ_k may occur other than in the (co)recursive position. For instance,

$$\text{List}^{\upsilon}\ \text{Nat} \longrightarrow \text{List}^{\upsilon}\ \text{Nat} \longrightarrow \text{List}^{\upsilon}\ \text{Nat}$$

is a valid fixpoint type with respect to υ, while

$$\text{Stream}^{\upsilon}\ \text{Nat} \longrightarrow \text{List}^{\upsilon}\ \text{Nat} \longrightarrow \text{List}\ \text{Nat}^{\upsilon}$$

is not, since υ appears negatively in `Stream` and must not appear at all in the parameter of the `List` return type.

In general, υ_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 that are not being (co)recurred upon.

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 stage variable υ_m bound to some annotation s.

Additionally, all (co)fixpoint types are annotated with position annotations: $|t_k|^{\upsilon_k}$ replaces all occurrences of υ_k with $*$. We cannot keep the stage annotations for the same reason as in Rule (abs), but we use $*$ to remember which types are size-preserving.

In actual Coq code, the indices of the recursive elements are rarely given, and there are no user-provided position annotations at all. In Section 3, we present how we compute the indices and the position annotations during size inference.

3 Size Inference

The goal of the size inference algorithm is to take unannotated programs in T° (corresponding to terms in CIC), simultaneously assign annotations to them while collecting a set of substaging constraints based on the typing rules, check the constraints to ensure termination and productivity, 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 two sized types are deemed to satisfy the subtyping relation $t \leq u$, from which we deduce the subtyping relations that must hold for their annotations from the subtyping rules. 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.

3.1 Notation

We use three kinds of judgements to represent *checking*, *inference*, and *well-formedness*. For convenience, 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 substaging constraints: if $(s_1, s_2) \in C$, then we must enforce $s_1 \sqsubseteq s_2$.

- $C, \Gamma_G, \Gamma \vdash e^\circ \Leftarrow t \rightsquigarrow C', e$ 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 that ensures that the type of e subtypes t.
- $C, \Gamma_G, \Gamma \vdash e^\circ \rightsquigarrow C', e \Rightarrow t$ 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'.
- $\Gamma^\circ \vdash \Gamma$ takes a global environment with bare declarations and produces global environment where each declaration has been properly annotated via inference.

The algorithm is implicitly parametrized over a set of stage variables $\mathcal{V}$, a set of position stage variables $\mathcal{P}$, and a signature Σ. The sets $\mathcal{V}, \mathcal{P}$ are treated as mutable for brevity, their assignment denoted with $:=$, and initialized as empty. The variable assignment $V = \mathcal{V}$ is a copy-by-value and not a reference. We will have $\mathcal{P} \subseteq \mathcal{V}$ throughout. Finally, we use $e \Rightarrow^* t$ to mean $e \Rightarrow t' \land t = \textsc{whnf}(t')$.

We define a number of metafunctions to translate the side conditions from the typing rules into procedural form, which are introduced as needed.

3.2 Inference Algorithm

Size inference begins with a bare term. In this case, even type annotations of (co)fixpoints are bare; that is,

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

Notice that fixpoints still have their vector of recursive argument indices, whereas real Coq code can have no indices given. To produce these indices, we do what Coq's kernel currently does: attempt type checking on every combination of indices from left to right until one combination works, or fail if none do.

Figure 13 presents the size inference algorithm, which uses the same indexing conventions as the typing rules. We will 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 sized given type, it uses the metafunction $\preceq$ that takes two sized terms and attempts to produce a set of stage constraints based on the subtyping rules of Figure 8. It performs reductions as necessary and fails 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. Again, we erase type annotations to bare. They use the metafunctions $\textsc{axiom}$, $\textsc{rule}$, and $\textsc{elim}$, which are functional counterparts to the sets Axioms, Rules, and Elims in Figure 17.

In Rules (a-var-def) and (a-const-def), we annotate variables and constants using $\textsc{fresh}$, which generates the given number of fresh stage annotations, adds them to $\mathcal{V}$, and returns them as a vector. Its length corresponds to the number of stage 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 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 stage variable.

A position-annotated type (i.e. an annotated (co)recursive 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 the former, a bare (co)inductive type is annotated with a stage variable; in the latter, a (co)inductive type with a position annotation has its annotation replaced by a position stage variable. The metafunction $\textsc{fresh}^*$ does the same thing as $\textsc{fresh}$ except that it also adds the freshly-generated stage variables to $\mathcal{P}$.

In Rule (a-constr), we generate a fresh stage variable for each (co)inductive type in the mutual definition that defines the given constructor. The number of types is given by $\textsc{inds}$. These are used to annotate the types of its (co)inductive arguments, as well as the return type, which of course has a successor stage annotation.

The key constraint generated by Rule (a-case) is $\hat{v}_k \sqsubseteq s$, where s is the annotation on the target type I_k. Similar to Rule (a-constr), we generate fresh stage variables $\overline{v_i}$ for each (co)-inductive type in the mutual definition that defines the type of the target. They are assigned to the branches' arguments of types $\overline{I_i}$, which correspond to the constructor arguments of the target. Then this constraint ensures that the constructor argument types have a smaller size than that of the target, since by Rules (ss-succ) and (ss-trans) we have $v_k \sqsubseteq s$.

The rest of the rule proceeds as we would expect: we get the 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 give the type of the case analysis as the motive applied to the target type's indices and the target itself. We also ensure that the elimination universes are valid using $\textsc{elim}$ on the motive type's return universe and the target type's universe. To obtain the motive type's return universe, we decompose the motive's type using $\textsc{decompose}$, which splits a function type into the given number of arguments and a return type, which in this case is the return universe.

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

- $\textsc{setRecStars}$, given a function type t and an index n, decomposes t into arguments and return type, reduces the nth argument type to an inductive type, annotates that

$$\frac{}{C, \Gamma_G, \Gamma \vdash x \rightsquigarrow C, x \Rightarrow \Gamma(x)}\ \text{(a-var-assum)} \qquad \frac{e : t = \Gamma(x) \qquad \overline{v_i} = \textsc{fresh}([\![e]\!])}{C, \Gamma_G, \Gamma \vdash x \rightsquigarrow C, x^{\langle v_i \rangle} \Rightarrow t}\ \text{(a-var-def)}$$

$$\frac{}{C, \Gamma_G, \Gamma \vdash x \rightsquigarrow C, x \Rightarrow \Gamma_G(x)}\ \text{(a-const-assum)} \qquad \frac{e : t = \Gamma_G(x) \qquad \overline{v_i} = \textsc{fresh}([\![e]\!]) \qquad v = \textsc{fresh}(1)}{C, \Gamma_G, \Gamma \vdash x \rightsquigarrow C, x^{\langle v_i \rangle} \Rightarrow t[\iota := v]}\ \text{(a-const-def)}$$

$$\frac{}{C, \Gamma_G, \Gamma \vdash \ \rightsquigarrow C, \ \Rightarrow \textsc{axiom}()}\ \text{(a-univ)} \qquad \frac{C, \Gamma_G, \Gamma \vdash e^\circ \rightsquigarrow C_1, e \Rightarrow t}{C, \Gamma_G, \Gamma \vdash e^\circ \Leftarrow u \rightsquigarrow C_1 \cup t \preceq u, e}\ \text{(a-check)}$$

$$\frac{C, \Gamma_G, \Gamma \vdash t^\circ \rightsquigarrow C_1, t \Rightarrow^* \ _1 \qquad C_1, \Gamma_G, \Gamma(x : t) \vdash u^\circ \rightsquigarrow C_2, u \Rightarrow^* \ _2}{C, \Gamma_G, \Gamma \vdash \Pi x : t^\circ . u^\circ \rightsquigarrow C_2, \Pi x : t.u \Rightarrow \textsc{rule}(_1, _2)}\ \text{(a-prod)}$$

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

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

$$\frac{C, \Gamma_G, \Gamma \vdash t^\circ \rightsquigarrow C_1, t \Rightarrow^* \qquad C_1, \Gamma_G, \Gamma \vdash e_1^\circ \Leftarrow t \rightsquigarrow C_2, e_1 \qquad C_2, \Gamma_G, \Gamma(x : t := e_1) \vdash e_2^\circ \rightsquigarrow C_3, e_2 \Rightarrow u}{C, \Gamma_G, \Gamma \vdash \text{let } x : t^\circ := e_1^\circ \text{ in } e_2^\circ \rightsquigarrow C_3, \text{let } x : |t| := e_1 \text{ in } e_2 \Rightarrow u[x := e_1]}\ \text{(a-let-in)}$$

$$\frac{v = \textsc{fresh}(1)}{C, \Gamma_G, \Gamma \vdash I_k \rightsquigarrow C, I_k^v \Rightarrow \textsc{indType}(\Sigma, I_k)}\ \text{(a-ind)} \qquad \frac{\rho = \textsc{fresh}^*(1)}{C, \Gamma_G, \Gamma \vdash I_k^* \rightsquigarrow C, I_k^\rho \Rightarrow \textsc{indType}(\Sigma, I_k)}\ \text{(a-ind-star)}$$

$$\frac{\overline{v} = \textsc{fresh}(\textsc{inds}(c_\ell))}{C, \Gamma_G, \Gamma \vdash c_\ell \rightsquigarrow C, c_\ell \Rightarrow \textsc{constrType}(\Sigma, c_\ell, \overline{v})}\ \text{(a-constr)}$$

$$\frac{\begin{array}{c} C, \Gamma_G, \Gamma \vdash e^\circ \rightsquigarrow C_1, e \Rightarrow^* I_k^s\, \overline{p}\, \overline{a} \qquad C_1, \Gamma_G, \Gamma \vdash \wp^\circ \rightsquigarrow C_2, \wp \Rightarrow t_p \\ \Pi\_ :\ _k = \textsc{indType}(\Sigma, I_k) \qquad (\_,) = \textsc{decompose}(t_p, \|\Delta_{a_k}\| + 1) \qquad \textsc{elim}(_k, , I_k) \qquad \overline{v_i} = \textsc{fresh}(\textsc{inds}(I_k)) \\ C_3 = C_2 \cup \{\hat{v}_k \sqsubseteq s\} \cup (t_p \preceq \textsc{motiveType}(\Sigma, \overline{p}, , I_k^s)) \qquad C_3, \Gamma_G, \Gamma \vdash e_j^\circ \Leftarrow \textsc{branchType}(\Sigma, \overline{p}, c_j, \overline{v_i}, \wp) \rightsquigarrow C_{4j}, e_j \end{array}}{C, \Gamma_G, \Gamma \vdash \text{case}_{\wp^\circ}\ e^\circ \text{ of } \langle c_j \Rightarrow e_j^\circ \rangle \rightsquigarrow \bigcup_j C_{4j}, \text{case}_{|\wp|}\ e \text{ of } \langle c_j \Rightarrow e_j \rangle \Rightarrow \wp \overline{a} e}\ \text{(a-case)}$$

$$\frac{\begin{array}{c} C, \Gamma_G, \Gamma \vdash t_k^\circ \rightsquigarrow \_, \_ \Rightarrow \_ \qquad V_{\text{outer}} = \mathcal{V} \\ C, \Gamma_G, \Gamma \vdash \textsc{setRecStars}(t_k^\circ, n_k) \rightsquigarrow C_{1k}, t_k \Rightarrow^* \\ \bigcup_k C_{1k}, \Gamma_G, \Gamma\overline{(f_k : t_k)} \vdash e_k^\circ \Leftarrow \textsc{shift}(t_k) \rightsquigarrow C_{2k}, e_k \\ C_4 = \textsc{RecCheckLoop}(\bigcup_k C_{2k}, V_{\text{outer}}, \overline{\textsc{getRecVar}(t_k, n_k)}, \overline{t_k}, \overline{e_k}) \end{array}}{C, \Gamma_G, \Gamma \vdash \text{fix}_{\langle n_k \rangle, m}\ \langle f_k : t_k^\circ := e_k \rangle \rightsquigarrow C_4, \text{fix}_{\langle n_k \rangle, m}\ \langle f_k : |t_k|^* := e_k \rangle \Rightarrow t_m}\ \text{(a-fix)}$$

$$\frac{\begin{array}{c} C, \Gamma_G, \Gamma \vdash t_k^\circ \rightsquigarrow \_, \_ \Rightarrow \_ \qquad V_{\text{outer}} = \mathcal{V} \\ C, \Gamma_G, \Gamma \vdash \textsc{setCorecStars}(t_k^\circ) \rightsquigarrow C_{1k}, t_k \Rightarrow^* \\ \bigcup_k C_{1k}, \Gamma_G, \Gamma\overline{(f_k : t_k)} \vdash e_k^\circ \Leftarrow \textsc{shift}(t_k) \rightsquigarrow C_{2k}, e_k \\ C_4 = \textsc{RecCheckLoop}(\bigcup_k C_{2k}, V_{\text{outer}}, \overline{\textsc{getCorecVar}(t_k)}, \overline{t_k}, \overline{e_k}) \end{array}}{C, \Gamma_G, \Gamma \vdash \text{cofix}_m\ \langle f_k : t_k^\circ := e_k \rangle \rightsquigarrow C_4, \text{cofix}_m\ \langle f_k : |t_k|^* := e_k \rangle \Rightarrow t_m}\ \text{(a-cofix)}$$

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

$$\frac{\begin{array}{c} \Gamma_G^\circ \rightsquigarrow \Gamma_G \qquad \emptyset, \Gamma_G, \square \vdash t^\circ \rightsquigarrow C_1, t \Rightarrow \\ C_1, \Gamma_G, \square \vdash e^\circ \rightsquigarrow \_, e \Rightarrow u \qquad \_ = u \preceq t \qquad \mathcal{P} := \mathcal{P} \cup \textsc{getPosVars}(t, u) \end{array}}{\Gamma_G^\circ(\text{Def } x : t^\circ := e^\circ.) \rightsquigarrow \Gamma_G(\text{Def } x : |t|^\iota := |e|^\infty.)}\ \text{(a-global-def)}$$

Figure 13. Size inference algorithm

```
let rec RecCheckLoop $C_2$ $V_{\text{outer}}$ $\overline{\rho_k}$ $\overline{t_k}$ $\overline{e_k}$ =
  try let pv$_k$ = PV $t_k$ in
      let sv$_k$ = ($V_{\text{outer}}$ ∪ SV $t_k$ ∪ SV $e_k$) \ pv$_k$ in
      let $C_{3k}$ = RecCheck $C_2$ $\rho_k$ pv$_k$ sv$_k$
      in $\bigcup_k C_{3k}$
  with RecCheckFail $V$ ->
      $\mathcal{P}$ := $\mathcal{P} \setminus V$;
      RecCheckLoop $C_2$ $\overline{\rho_k}$ $\overline{t_k}$ $\overline{e_k}$
```

Figure 14. Pseudocode implementation of RecCheckLoop

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 stage variable of the annotation on the nth inductive argument type, while getCorecVar returns the position stage variable of the annotation on the coinductive return type. Essentially, they retrieve the position stage variable of the annotation on the primary (co)recursive type of a (co)fixpoint type, which is used to check termination and productivity.
- shift replaces all stage annotations s with a position stage variable (i.e. $\lfloor s \rfloor \in \mathcal{P}$) 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 that any reduction we perform on it will terminate (otherwise the algorithm would have failed). Then we annotate the bare types with position annotations 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 environment. Lastly, we call RecCheckLoop, and return the constraints it gives us, along with the mth (co)fixpoint type.

Notice that in setRecStars and setCorecStars, we annotate *all* possible (co)inductive types in the (co)fixpoint type with position annotations. Evidently not all (co)fixpoints are size-preserving; some of those position annotations (excluding the one on the recursive argument type or the corecursive return type) will need to be removed. RecCheckLoop is a recursive function that calls RecCheck, which checks that a given set of stage constraints can be satisfied; if it cannot, then RecCheckLoop removes the position annotations that RecCheckLoop has found to be problematic, then retries.

More specifically, RecCheck can fail with RecCheckFail, which contains a set V of position stage variables that must be set to infinity; since position stage variables always appear on size-preserved types, they cannot be infinite. RecCheckLoop then removes V from the set of position stage variables, allowing them to be set to infinity, and recursively calls itself. The number of position stage variables from the (co)fixpoint type shrinks on every iteration until no more can be removed, at which point RecCheckLoop fails the algorithm. An OCaml-like pseudocode implementation of RecCheckLoop is provided by Figure 14.

3.3 RecCheck

As in previous work on CC$\widehat{\omega}$ with coinductive streams [5] and in CIC$\widehat{\ }$, we use the same RecCheck algorithm from F$\widehat{\ }$ [1]. Its goal is to check a set of constraints for circular substaging relations, set the stage variables involved in the cycles to ∞, and to produce a new set of constraints without these problems or fail, indicating nontermination or nonproductivity. It takes four arguments:

- A set of substaging constraints C.
- The stage variable ρ 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 sized-preserving, each (co)fixpoint type requires at *least* ρ for the primary (co)recursive type.
- A set of stage variables V^* that must be set to some non-infinite stage. These are the stage annotations with position stage variables found in the (co)fixpoint type. Note that $\rho \in V^*$.
- A set of stage variables $V^{\neq}$ that must be set to ∞. These are all other non-position stage annotations, found in the (co)fixpoint type, the (co)fixpoint body, and outside the (co)fixpoint.

Here, we begin to treat C as a weighted, directed graph. Each stage variable corresponds to a node, and each substaging relation is an edge from the lower to the upper variable. A stage annotation consists of a stage variable with an arbitrary finite nonnegative number of successor "hats"; we can write the number as a superscript, as in $\hat{v}^n$. Then given a substaging relation $\hat{v}_1^{n_1} \sqsubseteq \hat{v}_2^{n_2}$, the weight of the edge from v_1 to v_2 is $n_2 - n_1$. Substagings from ∞ are given an edge weight of 0.

Given a set of stage variables V, its *upward closure* $\bigsqcup V$ in C is the set of stage 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 stage variables that can reach V by travelling along the edges of C, or $\upsilon_2 \in V \land \hat{\upsilon}_1^{n_1} \sqsubseteq \hat{\upsilon}_2^{n_2} \implies \upsilon_1 \in V$.

We use the notation $\upsilon \sqsubseteq V$ to denote the set of constraints from υ to each stage variable in V.

The algorithm proceeds as follows:

1. Let $V^\iota = \bigsqcap V^*$, and add $\rho \sqsubseteq V^\iota$ to C. This ensures that ρ is the smallest stage variable among all the noninfinite stage variables.
2. Find all negative cycles in C, and let V^- be the set of all stage variables present in some negative cycle.
3. Remove all edges with stage variables in V^- from C, and add $\infty \sqsubseteq V^-$. Since $\widehat{\infty} \sqsubseteq \infty$, this is the only way to resolve negative cycles.
4. Add $\infty \sqsubseteq \left(\bigsqcup V^{\neq} \cap \bigsqcup V^\iota\right)$ to C.
5. Let $V^\perp = (\bigsqcup\{\infty\}) \cap V^\iota$. This is the set of stage variables that we have determined to both be infinite and noninfinite. If $V^\perp$ is empty, then return C.
6. Otherwise, let $V = V^\perp \cap (V^* \setminus \{\rho\})$. This is the set of contradictory position stage variables excluding ρ, which we can remove from $\mathcal{P}$ in RecCheckLoop. If V is empty, there are no position stage variables left to remove, so the check and therefore the size inference algorithm fails. If V is not empty, fail with RecCheckFail(V), which is handled by RecCheckLoop.

3.4 Well-Formedness

A self-contained chunk of code, be it a file or a module, consists of a sequence of (co)inductive definitions, or signatures, and programs, or global declarations. For our purposes, we assume that there is a singular well-formed signature defined independently. Assuring that the chunk of code is properly typed is then performing size inference on each declaration of Γ_G. These are given by Rules (a-global-empty), (a-global-assum), and (a-global-def). 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 stage variables due to the body e, so we use getPosVars to find the stage variables of t in the same locations as the position stage variables of u. For instance, if $\mathcal{P} = \{\rho\}$,

$$\text{getPosVars}(\text{Nat}^{\upsilon} \to \text{Nat}^{\upsilon'}, \text{Nat}^{\rho} \to \text{Nat}^{\upsilon''}) = \{\upsilon\}.$$

These then get added to $\mathcal{P}$ so that $|\cdot|^\iota$ properly erases the right stage annotations to global annotations. We cannot simply replace t with u, since t may have a more general type, e.g. $u = \text{Nat} \to \text{Set}$ vs. $t = \text{Nat} \to \text{Type}$.

4 Examples

Returning to our example programs in Section 1, in CIC$\widehat{*}$ they would be written as:

```
Def minus: Nat^ι → Nat^ι → Nat^ι := ....
Def div: Nat^ι → Nat → Nat^ι := ....
```

The body of `div` only needs to know that `minus` has type $\text{Nat}^\iota \to \text{Nat}^\iota \to \text{Nat}^\iota$ and nothing else. Furthermore, we have no problems using variables in our fixpoint types (note that we use 1-based indexing):

```
Def aNat: Set := Nat.
Def add: aNat^⟨ι⟩ → aNat → aNat :=
  fix_⟨1⟩,1 add': aNat^⟨*⟩ → Nat → Nat := ....
```

For the following examples we use a more succinct, Coq-like syntax for brevity, adding in global annotations where necessary. Assuming the usual definition for `List`s and `Bool`s, and the usual `if-then-else` syntax, we can construct a `filter` function with size-preserving types, since the output list is never longer than the input list.

```
Definition filter:
  (A: Set) -> (A -> Bool) -> List^ι A -> List^ι A :=
  fix filter' A pred (l: List^* A): List^* A :=
    match l with
      | Nil => Nil
      | Cons _ hd tl =>
        if pred hd
        then Cons A hd (filter' A pred tl)
        else (filter' tl)
    end.
Definition append:
  (A: Set) -> List^ι A -> List A -> List A := ....
```

We also have an append function that is *not* size-preserving. Now we are all set to implement `quicksort` on `Nat`s:

```
Definition quicksort:
  (A: Set) -> List^ι Nat -> List Nat :=
  fix quicksort' A (l: List^* Nat): List Nat :=
    match l with
    | Nil => Nil
    | Cons _ hd tl => append A
      (quicksort' (filter Nat (gtb hd) tl))
      (Cons Nat hd
        (quicksort' (filter Nat (leb hd) tl)))
    end.
```

Even though the output list has the same length as the input list, there is no way to add sizes in our current size algebra, so the return type of `append` is not annotated with the same size as the input type of `quicksort`. While asserting that `quicksort` does not change the length of the list requires additional proof, the fact that it *terminates* is given to us by virtue of being typeable.

On the other hand, it is because we cannot express any size relations more complicated than size-preservation that `gcd`, while terminating, is not typeable.

```
Definition modulo: Nat -> Nat^ι -> Nat^ι := ...
Fail Definition gcd: Nat -> Nat -> Nat :=
  fix gcd' a b :=
    match a with
    | O => b
    | S a' => gcd' (modulo b a) a
    end.
```

Because modulo can only determine that the return type is at most as large as its second argument, the first argument to the recursive call in gcd' has a type with the same size as a, and is not deemed to decrease on its first argument.

The above examples are annotated CIC$\widehat{*}$ implementations. In our Coq implementation, we write the Gallina equivalents of each function, and size inference infers the annotations for the above examples.

In our implementation, we can separately enable or disable syntactic guard checking and sized type checking.

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

This way, we can type check either: (1) programs that type check only with sized types, or (2) programs that type check only with syntactic guard checking.

5 Related Work

This work is based on CIC$\widehat{\ }$ [2], 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, constants and variables annotated by a vector of stage annotations, their δ-/Δ-reductions, a let-in construction, 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$\widehat{_}$ [4] is similar to CIC$\widehat{\ }$, described in greater detail, but with one major difference: CIC$\widehat{_}$ disallows stage variables in the bodies of abstractions, in the arguments of applications, and in case analysis branches, making CIC$\widehat{_}$ a strict subset of CIC$\widehat{\ }$. Any stage annotations found in these locations must be set to ∞. This solves the problem of knowing which stage annotations to use when using a variable defined as, for instance, an inductive type, simply by disallowing stage annotations in these definitions. However, this prevents us from using a variable as the (co)recursive type of a (co)fixpoint, and forces these types to be literal (co)inductive types. In practice, such as in Coq's default theorems and libraries, aliases are often defined for (co)inductive types, so we have worked around it with annotated variables and constants.

The implementation of RecCheck comes from F$\widehat{\ }$ [1], an extension of System F with type-based termination used sized types. Rules relating to coinductive constructions and cofixpoints comes from the natural extension of CC$\widehat{\omega}$ [5], which describes only infinite streams. Additionally, the judgement syntax for describing the size inference algorithm comes from CC$\widehat{\omega}$ and CIC$\widehat{l}$ [6].

Whereas our successor sized types uses a size algebra that only has a successor operation, *linear* sized types in CIC$\widehat{l}$ extends the algebra by including stage annotations 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". Although this causes the time complexity of their RecCheck procedure to be exponential in the number of stage variables, the (co)fixpoints written in practice may not so complicated as to be meaningfully detrimental compared to the benefits that linear sized types would bring. The set of typeable (and therefore terminating or productive) functions would be expanded even further; functions such as append and quicksort could be typed as size-preserving in addition to being terminating. If successor sized types prove to be practically useable in Coq, augmenting the type system to linear sized types would be a valuable consideration.

Well-founded sized types in CIC$\widehat{\sqsubseteq}$ [7] are yet another extension of successor sized types. This 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.

6 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 Gallina, such as global definitions, and extends the inference algorithm to infer sizes from completely unannotated Gallina terms to enable backwards compatibility. We implement the design presented in this paper as an extension to Coq's kernel, which can be found on GitHub https://github.com/ionathanch/coq/tree/dev. The design and implementation can be used alone or in conjunction with syntactic guard checking to maximize typeability and compatibility.

References

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

[2] Gilles Barthe, Benjamin Gregoire, and Fernando Pastawski. 2006. CIC$\widehat{\ }$: 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)*, Hermann, M and Voronkov, A (Ed.), Vol. 4246. Springer-Verlag Berlin, Heidelberger Platz 3, D-14197 Berlin, Germany, 257–271. https://doi.org/10.1007/11916277_18

[3] Fairouz Kamareddine, Twan Laan, and Rob Nederpelt. 2005. *Pure Type Systems with definitions.* Springer Netherlands, Dordrecht, 233–241. https://doi.org/10.1007/1-4020-2335-9_9

[4] 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

[5] 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

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

[7] Jorge Luis Sacchini. 2015. 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.

[8] The Coq Development Team. 2019. The Coq Proof Assistant, version 8.9.0. https://doi.org/10.5281/zenodo.2554024

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

$$\frac{\mathrm{WF}(\Sigma, \Gamma_G, \Gamma) \qquad (_1, _2) \in \mathrm{Axioms}}{\Sigma, \Gamma_G, \Gamma \vdash _1 : _2} \text{(univ)} \qquad \frac{\Sigma, \Gamma_G, \Gamma \vdash e : t \qquad u : \qquad t \preceq u}{\Sigma, \Gamma_G, \Gamma \vdash e : u} \text{(conv)}$$

$$\frac{\Sigma, \Gamma_G, \Gamma \vdash t : _1 \qquad \Sigma, \Gamma_G, \Gamma(x : t) \vdash u : _2 \qquad (_1, _2, _3) \in \mathrm{Rules}}{\Sigma, \Gamma_G, \Gamma \vdash \Pi x : t.u : _3} \text{(prod)}$$

$$\frac{\Sigma, \Gamma_G, \Gamma \vdash e_1 : \Pi x : t.u \qquad \Sigma, \Gamma_G, \Gamma \vdash e_2 : t}{\Sigma, \Gamma_G, \Gamma \vdash e_1 e_2 : u[x := e_2]} \text{(app)}$$

Figure 15. Typing rules common to CIC and $\widehat{\mathrm{CIC*}}$

Appendix A Supplementary Figures

Figure 16 lists the syntactic sugar we use in this work for writing terms and metafunctions on terms. Figure 17 lists the sets Axioms, Rules, and Elims, which are relations on universes. They desribe how universes are typed, how products are typed, and what eliminations are allowed in case analyses, respectively. Figure 15 gives the typing rules for assumptions, universes, products, applications, and convertibility, which are all common to CIC.

$\mathrm{dom}(\Delta)$	$\mapsto \overline{x}$	domain of assum. env.
$e\overline{a}$	$\mapsto (((ea_1)\dots)a_n)$	multiple application
$t \to u$	$\mapsto \Pi\_ : t.u$	nondependent product
$(x : t) \to u$	$\mapsto \Pi x : t.u$	dependent product
$\Pi\Delta.t$	$\mapsto \Pi x_1 : t_1.\dots\Pi x_n : t_n.t$	product from assums.
$\mathrm{SV}(e_1, e_2)$	$\mapsto \mathrm{SV}(e_1) \cup \mathrm{SV}(e_2)$	stage vars. of terms
$\mathrm{SV}(\overline{a})$	$\mapsto \mathrm{SV}(a_1) \cup \dots \cup \mathrm{SV}(a_n)$	stage vars. of terms

where $\overline{a} = a_1 \dots a_n$
$\Delta = (x_1 : t_1)\dots(x_n : t_n)$

Figure 16. Syntactic sugar for terms and metafunctions

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

Figure 17. Universe relations: Axioms, Rules, and Eliminations