CIC$\widehat{\ }$: type-based termination of recursive definitions in the Calculus of Inductive Constructions*

Gilles Barthe[1], Benjamin Grégoire[1], and Fernando Pastawski[1,2]

[1] INRIA Sophia-Antipolis, France
{Gilles.Barthe,Benjamin.Gregoire,Fernando.Pastawski}@sophia.inria.fr
[2] FaMAF, Univ. Nacional de Córdoba, Argentina

Abstract. Sized types provides a type-based mechanism to enforce termination of recursive definitions in typed λ-calculi. Previous work has provided strong indications that type-based termination provides an appropriate foundation for proof assistants based on type theory; however, most work to date has been confined to non-dependent type systems. In this article, we introduce a variant of the Calculus of Inductive Constructions with sized types and study its meta theoretical properties: subject reduction, normalization, and thus consistency and decidability of type-checking and of size-inference. A prototype implementation has been developed alongside case studies.

1 Introduction

Proof assistants based on dependent type theory rely on termination of typable programs to guarantee decidability of convertibility and hence decidability of typing. In order to enforce termination of programs, proof assistants typically require that recursive calls in a function definition are always performed on structurally smaller arguments; in the Coq proof assistant, which forms the focus of this article, the requirement is captured by a guard predicate $\mathcal{G}$ on expressions, that is applied to the body of recursive definitions for deciding whether or not the function should be accepted. Providing a liberal yet intuitive and correct syntactic guard criterion to guarantee termination is problematic.

Type-based termination is an alternative approach to enforce termination of recursive definitions through an extended type system that manipulates *sized types*, i.e. types that convey information about the size of their inhabitants. In a nutshell, the key ingredients of type-based termination are the explicit representation of the successive approximations of datatypes in the type system, a subtyping relation to reflect inclusion of the successive approximations and the adoption of appropriate rules for constructors, case expressions, and fixpoints.

Previous work by the authors [6, 7] and by others (see Section 2) has shown that type-based termination is an intuitive and robust mechanism, and a good candidate for enforcing termination in proof assistants based on dependent type theories. However, these works were concerned with non-dependent type systems. The contribution of the paper is an extension of these results to dependent type theories, and more precisely

* More details on difficulties with Coq, case studies, remaining issues and proofs and an implementation are available from the second author's web page.

to the Calculus of Inductive Constructions; concretely, we introduce CIC$\hat{}$, a variant of the Calculus of Inductive Constructions that enforces termination of recursive definitions through sized types. Besides, we show that the system CIC$\hat{}$ enjoys essential properties required for proof assistants, in particular logical consistency and decidability of type-checking, and decidability of size-inference. We have developed a prototype implementation of CIC$\hat{}$ and used it to prove the correctness of quicksort.

2 Related work

The idea of ensuring termination and productivity of fixpoint definitions by typing can be traced back to early work by Mendler [21] on recursion schemes, and to work by Hughes, Pareto and Sabry [18] on the use of sized types to ensure productivity of programs manipulating infinite objects. We refer the reader to [2, 6] for a review of related work, and focus on work that deals with dependent types or with size inference.

Inference Chin and Khoo [14] were among the first to study size inference in a non-dependent setting; they provided an algorithm that generates formulae of Presburger arithmetic to witness termination for a class of strongly normalizing terms typable in a (standard) simply typed λ-calculus with recursive definitions.

Type-checking algorithms for systems that enforce type-based termination were developed by Xi [25] for a system with restricted dependent types and by Abel [1, 2] for a higher order polymorphic λ-calculus. More recently, Blanqui and Riba [12] have shown (termination and) decidability of type checking for a simply typed λ-calculus extended with higher-order rewriting and based on constraint-based termination, a generalization of type-based termination inspired from [14].

Dependent types Giménez [17] was the first to consider a dependent type theory that uses type-based termination: concretely, he defined a variant of the Calculus of Inductive Constructions with type-based termination, and stated strong normalization for his system. The paper does not contain proofs and does not deal with size inference; besides, Giménez does not use an explicit representation of stages, which makes the system impractical for mutually recursive definitions. This work was pursued by Barras [4], who considered a variant of Giménez system with an explicit representation of stages, and proved in Coq decidability of type-checking assuming strong normalization.

Blanqui [10, 11] has defined a type-based variant of the Calculus of Algebraic Constructions (CAC) [9], an extension of the Calculus of Constructions with higher-order rewriting *à la* Jouannaud-Okada [19], and showed termination and decidability of type-checking. It is likely that strong normalization for CIC$\hat{}$ (which we conjecture) can be derived from [10], in the same way that strong normalization of CIC can be derived from strong normalization of CAC [9]. On the other hand, our inference result is more powerful than [11]. Indeed, our system only requires terms to carry a minimal amount of size annotations, and uses a size inference algorithm to compute these annotations, whereas size annotations are pervasive in Blanqui's system, and merely checked. We believe that size inference has a significant impact on the usability of the system, and is a requirement for a practical use of type-based termination in a proof-assistant.

3 A primer on type-based termination

The object of this section is to provide a (necessarily) brief introduction to type-based termination. For more information (including a justification of some choices for the syntax of non-dependent systems and inherited here), we refer the reader to [6, 7].

Consider for example the datatype of lists; in our system, the user declares the datatype with a declaration

$$\mathrm{List}\ [A : \mathbf{Set}] : \mathbf{Set} := \mathrm{nil} : \mathrm{List} \mid \mathrm{cons} : A \to \mathrm{List} \to \mathrm{List}$$

Lists are then represented by an infinite set of approximations $\mathrm{List}^s\ A$, where s is a size (or stage) expression taken from the following grammar:

$$s ::= \imath \mid \widehat{s} \mid \infty$$

where $\widehat{\cdot}$ denotes the successor function on stage expressions, and where we adopt the convention that $\widehat{\infty} = \infty$. Intuitively, $\mathrm{List}^s\ A$ denotes the type of A-lists of size at most s, and in particular, $\mathrm{List}^\infty\ A$ denotes the usual type of A-lists. In order to reflect inclusion between successive approximations of lists, we introduce a subtyping relation with the rules

$$\mathrm{List}^s\ A \leq \mathrm{List}^{\widehat{s}}\ A \qquad \mathrm{List}^s\ A \leq \mathrm{List}^\infty\ A$$

together with the usual rules for reflexivity, transitivity, and function space.

The typing rules for constructors confirm the intuitive meaning that the size of a constructor term is one plus the maximum size of its subterms:

$$\frac{\Gamma \vdash A : \omega}{\Gamma \vdash \mathrm{nil}\ |A| : \mathrm{List}^{\widehat{\imath}}\ A} \qquad \frac{\Gamma \vdash A : \omega \qquad \Gamma \vdash a : A \qquad \Gamma \vdash l : \mathrm{List}^s\ A}{\Gamma \vdash \mathrm{cons}\ |A|\ a\ l : \mathrm{List}^{\widehat{s}}\ A}$$

Note that the empty list cannot be of type $\mathrm{List}^\imath$ because it would break normalization. Furthermore, note that parameters in constructors do not carry any size annotations (they are removed by the erasure function $|.|$), both to ensure subject reduction and to guarantee that we do not have multiple canonical inhabitants for parametrized types: e.g. in this way we guarantee that $\mathrm{nil}\ \mathrm{Nat}$ is the only empty list in $\mathrm{List}\ \mathrm{Nat}^\infty$; otherwise we would have for each stage s an empty list $\mathrm{nil}\ \mathrm{Nat}^s$ of type $\mathrm{List}\ \mathrm{Nat}^\infty$.

Then, the typing rule for fixpoints ensures that recursive function calls are always performed on arguments smaller than the input:

$$\frac{\Gamma, f : \mathrm{List}^\imath\ A \to B \vdash e : \mathrm{List}^{\widehat{\imath}}\ A \to B[\imath := \widehat{\imath}]}{\Gamma \vdash (\mathsf{fix}\ f : |\mathrm{List}^\imath\ A \to B|^\imath := e) : \mathrm{List}^s\ A \to B[\imath := s]}$$

where $\imath$ occurs positively in B and does not occur in $\Gamma,\ A$. Note that the tag of f in the recursive definition does not carry size annotations (we use the erasure function $|.|^\imath$2); instead, it simply carries position annotations in some places, to indicate which recursive arguments have a size related to the decreasing argument. The purpose of position annotations is to guarantee the existence of compact most general typings (without position annotations we would need union types), see [7].

In the conclusion, the stage s is arbitrary, so the system features some *implicit stage polymorphism*. Further, the substitution in the conclusion is useful to compute so-called *precise typings*. For example, CIC$\hat{}$ allows the precise typings for map and filter:

$$\mathsf{map} : \Pi A : \mathbf{Set}.\ \Pi B : \mathbf{Set}.\ (A \to B) \to (\mathsf{list}^{\imath}\ A) \to (\mathsf{list}^{\imath}\ B)$$
$$\mathsf{filter} : \Pi A : \mathbf{Set}.\ (A \to \mathrm{bool}) \to (\mathsf{list}^{\imath}\ A) \to (\mathsf{list}^{\imath}\ A)$$

to reflect that the map and filter function outputs a list whose length is smaller or equal to the length of the list that they take as arguments. In turn, the precise typing of filter is used to type functions that are rejected by many syntactic criteria of termination, such as the quicksort function. Wahlstedt[23] presents a size-change principle that accepts many recursion schemes that escape our type system but is unable to gain expressiveness from size preserving functions.

4 System CIC$\hat{}$

The system CIC$\hat{}$ is a type-based termination version of the Calculus of Inductive Constructions (CIC) [24]. The latter is an extension of the Calculus of Constructions with (co-)inductive types, and the foundation of the Coq proof assistant. In this paper, we omit co-inductive types. On the other hand, we present CIC$\hat{}$ as an instance of a Sized Inductive Type System, which is an extension of Pure Type Systems [3] with inductive types using size-based termination.

Specifications The type system is implicitly parametrized by a specification that is a quadruple of sorts $\mathbb{S}$, axioms Axioms, product rules Rules, and elimination rules Elim. Sorts are the universes of the type system; typically, there is one sort **Prop** of propositions and a sort **Set** of types. Axioms establish typing relations between sorts, product rules determine which dependent products may be formed and in which sort they live, and elimination rules determine which case analysis may be performed. The specification for CIC$\hat{}$ is that of CIC [22].

Terms Following $F\hat{}$ [7], CIC$\hat{}$ features three families of expressions to ensure subject reduction and efficient type inference. The first family is made of bare expressions that do not carry any size information: bare expressions are used in the tags of λ-abstractions and case expressions and as parameters in inductive definitions/constructors. The second family is made of positions expressions that are used in the tags of recursive definitions and rely on a mark $\star$ to denote (in a recursive definition) which positions have a size related to that of the recursive argument. Finally, the third family is made of sized expressions that carry annotations (except in their tags).

Definition 1 (Stages and expressions).

1. *The set $\mathbb{S}$ of* stage expressions *is given by the abstract syntax: $s, r ::= \imath \mid \infty \mid \widehat{s}$. Stage substitution is defined in the obvious way, and we write $s[\imath := s']$ to denote the stage obtained by replacing $\imath$ by s' in s. Furthermore, the base stage of a stage expression is defined by the clauses $\lfloor \imath \rfloor = \imath$ and $\lfloor \widehat{s} \rfloor = \lfloor s \rfloor$ (the function is not defined on stages that contain ∞).*

$$\frac{}{\mathsf{WF}([])}\ \text{empty} \qquad \frac{\mathsf{WF}(\Gamma) \quad \Gamma \vdash T : \omega}{\mathsf{WF}(\Gamma(x:T))}\ \text{cons} \qquad \frac{\mathsf{WF}(\Gamma) \quad (\omega, \omega') \in \mathsf{Axioms}}{\Gamma \vdash \omega : \omega'}\ \text{sort}$$

$$\frac{\mathsf{WF}(\Gamma) \quad \Gamma(x) = T}{\Gamma \vdash x : T}\ \text{var} \qquad \frac{\Gamma \vdash T : \omega_1 \quad \Gamma(x:T) \vdash U : \omega_2 \quad (\omega_1, \omega_2, \omega_3) \in \mathsf{Rules}}{\Gamma \vdash \Pi x{:}T.\, U : \omega_3}\ \text{prod}$$

$$\frac{\Gamma \vdash \Pi x{:}T.\, U : \omega \quad \Gamma(x:T) \vdash u : U}{\Gamma \vdash \lambda x{:}|T|.\, u : \Pi x{:}T.\, U}\ \text{abs} \qquad \frac{\Gamma \vdash u : \Pi x{:}T.\, U \quad \Gamma \vdash t : T}{\Gamma \vdash u\, t : U[x := t]}\ \text{app}$$

$$\frac{\Gamma \vdash t : T \quad \Gamma \vdash U : \omega \quad T \preceq U}{\Gamma \vdash t : U}\ \text{conv} \qquad \frac{\mathsf{WF}(\Gamma) \quad I \in \Sigma}{\Gamma \vdash I^s : \mathsf{TypeInd}(I)}\ \text{ind}$$

$$\frac{\begin{array}{c} I \in \Sigma \quad \Gamma(x : \mathsf{TypeConstr}(c, s)) \vdash x\, \boldsymbol{p}\, \boldsymbol{a} : U \\ \mathsf{params}(c) = \#\boldsymbol{p} \quad \mathsf{args}(c) = \#\boldsymbol{a} \quad x \text{ fresh in } \Gamma, \boldsymbol{p}, \boldsymbol{a} \end{array}}{\Gamma \vdash c(|\boldsymbol{p}|, \boldsymbol{a}) : U}\ \text{constr}$$

$$\frac{\begin{array}{c} \Gamma \vdash t : I^{\widehat{s}}\, \boldsymbol{p}\, \boldsymbol{a} \quad I \in \Sigma \quad \Gamma \vdash P : \mathsf{TypePred}(I, s, \boldsymbol{p}, \omega') \\ (I.\omega, \omega') \in \mathsf{Elim} \qquad \mathsf{params}(I) = \#\boldsymbol{p} \qquad \Gamma \vdash b_i : \mathsf{TypeBranch}(c_i, s, P, \boldsymbol{p}) \end{array}}{\Gamma \vdash \mathsf{case}_{|P|}\ t\ \mathsf{of}\ \{c_i \Rightarrow b_i\} : P\, \boldsymbol{a}\, t}\ \text{case}$$

$$\frac{\begin{array}{c} T = \Pi \Delta.\, \Pi x{:}I^{\imath}\, \boldsymbol{u}.\, U \qquad \imath\ \mathsf{pos}\ U \qquad \#\Delta = n - 1 \\ \imath \text{ does not occur in } \Delta, \boldsymbol{u}, \Gamma, t \qquad \Gamma \vdash T : \omega \qquad \Gamma(f : T) \vdash t : T[\imath := \widehat{\imath}] \end{array}}{\Gamma \vdash (\mathsf{fix}_n\ f{:}|T|^{\imath} := t) : T[\imath := s]}\ \text{fix}$$

Fig. 1. Typing rules

2. *The set of $\mathcal{P}$ size positions is defined as $\{\star, \epsilon\}$.*
3. *The generic set of terms over the set a is defined by the abstract syntax:*

$$\begin{array}{c} \mathcal{T}[a] ::= \Omega \mid \mathcal{X} \mid \lambda \mathcal{X}{:}\mathcal{T}^{\circ}.\, \mathcal{T}[a] \mid \mathcal{T}[a]\, \mathcal{T}[a] \mid \Pi \mathcal{X}{:}\mathcal{T}[a].\, \mathcal{T}[a] \mid \mathcal{C}(\boldsymbol{\mathcal{T}^{\circ}}, \boldsymbol{\mathcal{T}[a]}) \mid \mathcal{I}^{a} \\ \mid \mathsf{case}_{\mathcal{T}^{\circ}}\ \mathcal{T}[a]\ \mathsf{of}\ \{\mathcal{C} \Rightarrow \mathcal{T}[a]\} \mid \mathsf{fix}_n\ \mathcal{X}{:}\mathcal{T}^{\star} := \mathcal{T}[a] \end{array}$$

   where Ω, $\mathcal{X}$, $\mathcal{I}$ and $\mathcal{C}$ range over sorts, variables, datatypes and constructors.
4. *The set of bare expressions, position expressions, and sized expressions are defined by the clauses $\mathcal{T}^{\circ} ::= \mathcal{T}[\epsilon]$ and $\mathcal{T}^{\star} ::= \mathcal{T}[\mathcal{P}]$ and $\mathcal{T} ::= \mathcal{T}[\mathcal{S}]$.*

Note that we require that constructors are fully applied; as mentioned above, we also separate arguments of constructors into parameters that do not carry any size information and arguments that may carry size information. Besides, the fixpoint definition carries an index n that determines the recursive argument of the function. Finally, observe that case expressions are tagged with a function that gives the type of each branch, as required in a dependently typed setting.

Reduction and conversion The computational behavior of expressions is given by the usual rules for β-reduction (function application), $\imath$-reduction (pattern matching) and μ-reduction (unfolding of recursive definitions). The definition of these rules relies on substitution, whose formalization must be adapted to deal with the different categories of expressions.

Definition 2 (Erasure and substitution).

1. *The function* $|.| : \mathcal{T}^\star \cup \mathcal{T} \to \mathcal{T}^\circ$ *is defined as the obvious erasure function from sized terms (resp. position terms) to bare terms.*
2. *The function* $|.|^\imath : \mathcal{T} \to \mathcal{T}^\star$ *is defined as the function that replaces stage annotations* s *with* $\star$ *if the base stage of* s *is* $\imath$ *(*$\lfloor s \rfloor = \imath$*) and by* ϵ *otherwise.*
3. *The substitution of* x *by* N *into* M *is written as* $M[x := N]$*. (In fact we need three substitution operators, one for each category of terms; all are defined in the obvious way, and use the erasure functions when required.)*
4. *The substitution of stage variable* $\imath$ *by stage expression* s *is defined as* $M[\imath := s]$*.*

We are now in position to define the reduction rules.

Definition 3 (Reduction rules and conversion).

– *The reduction relation* $\to$ *is defined as the compatible closure of the rules:*

$$\begin{aligned}
(\lambda x{:}T^\circ.\ M)\ N &\ \to\ M[x := N] && (\beta)\\
\mathsf{case}_{T^\circ}\ c_j(\boldsymbol{p}^\circ, \boldsymbol{a})\ \mathsf{of}\ \{c_i \Rightarrow t_i\} &\ \to\ t_j\ \boldsymbol{a} && (\imath)\\
(\mathsf{fix}_n\ f{:}T^\star := M)\ \boldsymbol{b}\ c(\boldsymbol{p}^\circ, \boldsymbol{a}) &\ \to\ M[f := (\mathsf{fix}_n\ f{:}T^\star := M)]\ \boldsymbol{b}\ c(\boldsymbol{p}^\circ, \boldsymbol{a}) && (\mu)
\end{aligned}$$

where $\boldsymbol{b}$ *is of length* $n-1$*.*

– *We write* $\overset{*}{\to}$ *and* $\approx$ *respectively for the reflexive-transitive and reflexive-symmetric-transitive closure of* $\to$*.*

Both reduction and conversion are closed under substitution. Moreover, reduction is Church-Rosser.

Lemma 1 (Church-Rosser). *For every expressions* u *and* v *such that* $u \approx v$ *there exists* t *such that* $u \overset{*}{\to} t$ *and* $v \overset{*}{\to} t$*.*

In particular normal forms are unique, hence we write $\mathsf{NF}(A)$ for the normal form of A (if it exists) w.r.t. $\to$.

Subtyping In order to increase its expressiveness, the type system features a subtyping relation that is derived from a partial order on stages. The partial order reflects two intuitions: first, that an approximation $\mathcal{I}^s$ is contained in its successor approximation $\mathcal{I}^{\widehat{s}}$; second, that $\mathcal{I}^\infty$ is closed under constructors, and the fixpoint of the monotonic operator attached to inductive types.

Definition 4 (Substage). *The relation* s *is a substage of* s'*, written* $s \sqsubseteq s'$*, is defined by the rules:*

$$\frac{}{s \sqsubseteq s} \qquad \frac{s \sqsubseteq r \quad r \sqsubseteq p}{s \sqsubseteq p} \qquad \frac{}{s \sqsubseteq \widehat{s}} \qquad \frac{}{s \sqsubseteq \infty}$$

The substage relation defines a subtyping relation between types, using for each inductive type a declaration that indicate the polarity of its parameters.

Definition 5 (Polarity declaration). *We assume that each inductive type I comes with a vector $I.\boldsymbol{\nu}$ of polarity declarations, where each element of a polarity declaration can be positive, negative or invariant:*

$$\nu \ ::= \ + \mid - \mid \circ$$

Subtyping is then defined in the expected way, using an auxiliary relation that defines subtyping between vectors of expressions relative to a vector of positivity declarations.

Definition 6 (Subtyping).

- *Let R be an equivalence relation stable under substitution of terms and stages. The subtyping relations $\preceq_R$ and $\preceq_R^{\boldsymbol{\nu}}$ are simultaneously defined by the rules:*

$$\frac{t_1\ R\ t_2}{t_1 \preceq_R t_2} \qquad \frac{T_2 \preceq_R T_1 \quad U_1 \preceq_R U_2}{\Pi x{:}T_1.\ U_1 \preceq_R \Pi x{:}T_2.\ U_2} \qquad \frac{s \sqsubseteq s' \quad \boldsymbol{t}_1 \preceq_R^{I.\boldsymbol{\nu}} \boldsymbol{t}_2}{I^s\ \boldsymbol{t}_1 \preceq_R I^{s'}\ \boldsymbol{t}_2}$$

$$\frac{t_1\ R\ u_1 \quad \boldsymbol{t} \preceq_R^{\boldsymbol{\nu}} \boldsymbol{u}}{t_1.\boldsymbol{t} \preceq_R^{\circ.\boldsymbol{\nu}} u_1.\boldsymbol{u}} \qquad \frac{t_1 \preceq_R u_1 \quad \boldsymbol{t} \preceq_R^{\boldsymbol{\nu}} \boldsymbol{u}}{t_1.\boldsymbol{t} \preceq_R^{+.\boldsymbol{\nu}} u_1.\boldsymbol{u}} \qquad \frac{u_1 \preceq_R t_1 \quad \boldsymbol{t} \preceq_R^{\boldsymbol{\nu}} \boldsymbol{u}}{t_1.\boldsymbol{t} \preceq_R^{-.\boldsymbol{\nu}} u_1.\boldsymbol{u}} \qquad \frac{\boldsymbol{t}\ R\ \boldsymbol{u}}{\boldsymbol{t} \preceq_R^{\emptyset} \boldsymbol{u}}$$

- *We define $\preceq$ as the transitive closure of $\preceq_{\approx}$. (Note that $\preceq$ is reflexive and allows redex elimination through $\approx$.)*
- *We define $\leq$ as $\preceq_{=}$. (Note that $\leq$ is reflexive and transitive.)*

The subtyping relation $\preceq$ shall be used to define the type system of CIC$\widehat{\ }$, whereas the subtyping relation $\leq$ shall be used by the inference algorithm. The two subtyping relations are related by the following lemma.

Lemma 2. *If A and A' are normalizing, then $A \preceq A'$ iff $\mathsf{NF}(A) \leq \mathsf{NF}(A')$.*

Positivity In order to formulate the type system and to specify which inductive definitions are correct and supported by CIC$\widehat{\ }$, we need several notions of positivity and strict positivity. Strict positivity is used to guarantee termination of recursive functions, whereas positivity is used to verify that polarity declarations are correct and in the rule for fixpoints. We begin by defining positivity of stage variables. In contrast to simple type systems, positivity cannot be defined syntactically, and we are forced to use a semantic definition.

Definition 7 (Positivity of stage variables). *$\imath$ is positive in T, written $\imath\ \mathsf{pos}\ T$, iff $T[\imath := s_1] \preceq T[\imath := s_2]$ for all $s_1,\ s_2$ such that $s_1 \sqsubseteq s_2$.*

The above definition involves a universal quantification and thus is not practical for algorithmic purposes. We provide an equivalent definition that can be used for type checking.

Lemma 3 (Redefinition of positivity). *If T is normalizing then*

$$\imath\ \mathsf{pos}\ T \Leftrightarrow T \preceq T[\imath := \widehat{\imath}] \Leftrightarrow \mathsf{NF}(T) \leq \mathsf{NF}(T[\imath := \widehat{\imath}])$$

We can generalize the notion of positivity to term variables.

Definition 8 (Positivity of term variables).

- x *is positive in* T*, written* x pos T*, iff* $T[x := t_1] \preceq T[x := t_2]$ *for all* $t_1,\ t_2$ *such that* $t_1 \preceq t_2$.
- x *is negative in* T*, written* x neg T*, iff* $T[x := t_2] \preceq T[x := t_1]$ *for all* $t_1,\ t_2$ *such that* $t_1 \preceq t_2$.

We conclude this section with a definition of strict positivity. Indeed, contrary to earlier work with non-dependent type systems, we cannot allow positive inductive types in our system, because it would lead to an inconsistency [15].

Definition 9 (Strictly positive). *A variable* x *is strictly positive in* T*, written* x POS T*, if* x *does not appear in* T *or if* $T \approx \Pi\Delta.\ x\ \boldsymbol{t}$ *and* x *does not appear in* Δ *and* $\boldsymbol{t}$.

Inductive types Inductive definitions are declared in a signature Σ; each inductive definition is introduced with a declaration of the form

$$\mathrm{Ind}(I[\Delta_p]^{\boldsymbol{\nu}} : \Pi\Delta_a.\ \omega := \overrightarrow{c_i : \Pi\Delta_i.\ \delta\ \boldsymbol{t}_i})$$

where I is the name of the inductive type, Δ_p is a context defining its parameters and their type, Δ_a is a context defining its arguments and their type, ω is a sort and $\boldsymbol{\nu}$ is its polarity declaration. To the right of the $:=$ symbol, we find a list of constructors with their types: c_i represents the name of the i-th constructor, and Δ_i is the context for its arguments, and $\delta\ \boldsymbol{t_i}$ represents the type of the resulting constructor term—for technical reasons, we use a special variable δ representing the current inductive type. In the sequel, we shall use some notations to deal with inductive definitions. First, we write $I \in \Sigma$ for $\mathrm{Ind}(I[\Delta_p]^{\boldsymbol{\nu}} : \Pi\Delta_a.\ \omega := \overrightarrow{c_i : \Pi\Delta_i.\ \delta\ \boldsymbol{t}_i}) \in \Sigma$. Figure 2 introduces further notations referring to inductive definitions : $I.\omega$ and $\mathsf{TypeInd}(I)$ are respectively the sort and the type of I; $\mathsf{params}(I) = \mathsf{params}(c)$ indicates the number of parameters of the inductive type and of its constructors. Then, we define the type of a constructor ($\mathsf{TypeConstr}(c, s)$), of the case predicate ($\mathsf{TypePred}(I, s, \boldsymbol{p}, \omega')$) and the type of case branches $\mathsf{TypeBranch}(c_i, s, P, \boldsymbol{p})$.

As usual, we separate between parameters and arguments of inductive types—they are handled differently in the syntax and shall be handled differently in the conversion rule—and assume that inductive types do not share constructors. Furthermore, contexts of inductive definitions are subject to well-formedness constraints; some constraints rely on the type system defined in the next paragraph. A context of inductive definitions is well-formed if it is empty [] or if it is of the form $\Sigma; \mathrm{Ind}(...)$ where Σ is well formed and all of the following hold:

1. the inductive definition is well-typed, i.e. $\vdash \Pi\Delta_p.\ \Pi\Delta_a.\ \omega : \omega'$ for some ω' is a valid typing judgment with signature Σ;
2. the constructors are well-typed, i.e. $\Delta_p\ (\delta : \Pi\Delta_a.\ \omega) \vdash \Pi\Delta_i.\ \delta\ \boldsymbol{t}_i : \omega_i$ for some ω_i is a valid typing judgment with signature Σ;
3. variable δ is strictly positive in the type of every constructor argument (δ pos Δ_i).
4. each occurrence of inductive types in $\Delta_p, \Delta_a, \Delta_i$ is annotated with ∞;
5. each variable in Δ_p satisfies the polarity condition in the type of each constructor. This means $\mathsf{dom}(\Delta_p)$ $\mathsf{pos}^{\ I.\boldsymbol{\nu}}\ \Delta_p$ and for every constructor c_i, $\mathsf{dom}(\Delta_p)$ $\mathsf{pos}^{\ I.\boldsymbol{\nu}}\ \Delta_i$.

$$
\begin{array}{ll}
I.\omega & := \omega \\
\mathsf{TypeInd}(I) & := \Pi\Delta_p.\ \Pi\Delta_a.\ \omega \\
\mathsf{params}(I) & := \#\Delta_p \\
\mathsf{params}(c) & := \#\Delta_p \\
\mathsf{args}(c) & := \#\Delta_i \\
\mathsf{TypeConstr}(c_i, s) & := \Pi\Delta_p.\ \Pi(\Delta_i[\delta := I^s\ \mathsf{dom}(\Delta_p)]).\ I^{\widehat{s}}\ \mathsf{dom}(\Delta_p)\ \boldsymbol{t_i} \\
\mathsf{TypePred}(I, s, \boldsymbol{p}, \omega') & := \Pi\Delta_a[\mathsf{dom}(\Delta_p) := \boldsymbol{p}].\ \Pi x{:}I^{\widehat{s}}\ \boldsymbol{p}\ \mathsf{dom}(\Delta_a).\ \omega' \\
\mathsf{TypeBranch}(c_i, s, P, \boldsymbol{p}) & := (\Pi\Delta_i.\ P\ \boldsymbol{t}_i\ c_i(|\boldsymbol{p}|, \mathsf{dom}(\Delta_i)))\ [\mathsf{dom}(\Delta_p) := \boldsymbol{p}][\delta := I^s\ \boldsymbol{p}]
\end{array}
$$

Fig. 2. Definitions over inductive constructions

6. positive and negative variables in Δ_p do not appear in arguments $\boldsymbol{t}_i$ that appear in the types of constructors.
7. from subtyping rules, we have that $\boldsymbol{p}_1 \preceq^{I.\boldsymbol{\nu}} \boldsymbol{p}_2$ implies $I\ \boldsymbol{p}_1\boldsymbol{a} \preceq I\ \boldsymbol{p}_2\boldsymbol{a}$. We require $\mathsf{dom}(\Delta_p)\ \mathsf{pos}^{\,I.\boldsymbol{\nu}}\Delta_a$ to guarantee that if $I\ \boldsymbol{p}_1\boldsymbol{a}$ and all the components of p_2 are well typed, then $I\ \boldsymbol{p}_2\boldsymbol{a}$ will be well typed.

Clause 3 ensures termination, whereas Clause 4 ensures that constructors use previously defined datatypes, but not approximations of previously defined datatypes—it is not clear whether lifting such a restriction would make the system more useful and how much the theory would be impacted. Clauses 5 and 6 reflect the subtyping rules for inductive types, and are used in the proof of subject reduction. Lastly, clause 7 is required to guarantee the completeness of type inference.

Typing Typing judgments are defined in the usual way. They are implicitly parameterized by a signature of inductive declarations, and by a specification that consists of a set of axioms, product rules, and elimination rules. Axioms establish typing relations between sorts, product rules determine which dependent products may be formed and in which sort they live, and elimination rules determine which case analysis may be performed. For example, Coq does not allow $\mathsf{Elim}(\mathbf{Prop}, \mathbf{Set})$.

Definition 10 (Contexts and judgments).

- *A context is a finite list of declarations* $\Gamma := (x_1 : T_1) \ldots (x_n : T_n)$ *where* $x_1, \ldots, x_n$ *are pairwise disjoint variables and* $T_1, \ldots, T_n$ *are expressions.*
- *A typing judgment is a tuple of the form* $\Gamma \vdash t : T$*, where* Γ *is a context,* t *and* T *are expressions.*
- *A judgment is derivable iff it can be obtained using the rules of Figure 1.*

5 Meta-theory

This section states the main properties that CIC$\widehat{\ }$ inherits from its non-dependent ancestors, and that justifies it as a foundation for proof assistants. Once the distinction between terms and types is reestablished for CIC$\widehat{\ }$, the algorithm and most proofs may be adapted with only minor modifications inherent to the complexity of CIC. All properties are proved for arbitrary specifications, and rely on the assumption of normalization.

Subject reduction and consistency In order to prove subject reduction, we must first establish substitution lemmas, generation lemmas, correctness of types and inversion of products.

Lemma 4.

- ***Correctness of types:*** *If* $\Gamma \vdash t : T$ *then there exists* $\omega \in \mathcal{S}$ *such that* $T = \omega$ *or* $\Gamma \vdash T : \omega$
- ***Inversion of product*** *If* $\Pi x{:}A.\ B \preceq \Pi x{:}C.\ D$ *then* $C \preceq A$ *and also* $B \preceq D$
- ***Subject reduction*** *If* $\Gamma \vdash M : T$ *and* $M \to M'$ *then* $\Gamma \vdash M' : T$

The proof of subject reduction is in most parts analogous to the one for CIC. The difficulty posed by fixpoint reduction is dealt with thanks to a lemma stating preservation of typing under stage substitution[6].

As usual, subject reduction and confluence allow to deduce consistency from normalization.

Size inference Proof assistants based on dependent type theory rely on the Curry-Howard isomorphism to reduce proof-checking to type-checking. In this context, it is important to be able to decide whether a term is typable or not. Furthermore, it is important for usability that size annotations should not be provided by users, for whom sized types should be as transparent as possible. Thus, we want to device a procedure that takes as input a context and a bare expression and returns a decoration of the bare expression and a most general typing if it exists, or an error if no decoration of the expression is typable.

There are two fundamental steps in designing a type-checking algorithm for dependent types—without subtyping and inductive types. The first step is to give a syntax-directed formulation of the typing rules, with conversion used only in specific places; the second step is to give a procedure to decide the convertibility of two terms. The syntax-directed algorithm always calls convertibility checking on typable terms, which are thus known to be normalizing, and convertibility is decidable in this case— thanks to confluence and normalization, one can compute both normal forms and check the equality.

In our setting, convertibility is replaced by subtyping $T \preceq U$, but we can adopt the strategy for testing convertibility for well typed terms (that are strongly normalizing): compute both normal forms and check whether they are related by subtyping $\leq$, see Lemma 2. However, termination is enforced with size information, which must be inferred during type-checking. Although it would be tempting to perform type-checking using erased types and perform termination checking afterwards, this is not possible with dependent types because it would entail not knowing termination at type-checking, which itself results in undecidability. Thus, we must check termination during type-checking, and more concretely when checking recursive definitions. Informally, we achieve the desired effect by creating and propagating constraints between stages while checking expressions, and resolving the constraints while checking a recursive definition.

Formally, our algorithm returns for every context Γ and unannotated expression e° either an error if no annotation e of e° is typable in Γ or else a most general annotation e

of e° and typing of the form $C \Rightarrow T$ where C is a set of constraints (stage inequalities), and T is an annotated type subject to the following properties:

Soundness: for every stage substitution ρ satisfying C, we have $\rho\Gamma \vdash \rho e : \rho T$.
Completeness: for every stage substitution ρ' and annotation e' of e° such that $\rho'\Gamma \vdash e' : T'$, there exists ρ, a stage substitution such that ρ satisfies C and $\rho\Gamma = \rho'\Gamma$ and $\rho e = e'$ and $\rho T \preceq T'$.

The notion of constraint system and satisfaction are defined formally as follows.

Definition 11 (Constraint and constraint systems).

1. *A* stage constraint *is a pair of stages, written* $s_1 \sqsubseteq s_2$.
2. *A* constraint system *is a finite set of stage constraints.*
3. *A stage substitution* ρ *satisfies a constraint system* C*, written* $\rho \models C$*, if for every constraint* $s_1 \sqsubseteq s_2$ *in* C*, we have* $\rho(s_1) \sqsubseteq \rho(s_2)$.

Note that the stage substitution that maps all stage variables to ∞ is a solution of all constraint systems.

We now turn to the formal description of the algorithm, which is adapted from [7]. The inference algorithm $\mathsf{Infer}(V, \Gamma, e^\circ)$ takes as input a context Γ, an unannotated expression e° and an auxiliary parameter V that represents the stage variables that have been previously used during inference (we need the latter to guarantee that we only introduce fresh variables). It returns a tuple (V', C, e, T) where e is an annotated version of e°, T is a sized type, C is a constraint system, and V' is an extended set of stage variables that has been used by the algorithm. The invariants are $\mathsf{FV}(\Gamma) \subseteq V$ and $V \subseteq V'$ and $\mathsf{FV}(C, e, T) \subseteq V'$. For practical reasons, we also use a second algorithm $\mathsf{Check}(V, \Gamma, e^\circ, T)$ which returns a tuple (V', C, e), where e is an annotated version of e° such that e has type T in environment Γ (and fails if no such e exists). The invariants are $\mathsf{FV}(\Gamma, T) \subseteq V$ and $V \subseteq V'$ and $\mathsf{FV}(C, e) \subseteq V'$.

Definition 12. *The algorithms* Infer *and* Check *are defined in Figure 3.*

The algorithms rely on several auxiliary functions. First, there are functions axioms , rules , elim that verify compatibility with the specification—here we assume Axioms and Rules to be functional. Then, there is an auxiliary function whnf that computes the weak head normal form of an expression—here we assume that the type system is normalizing, and use the fact that the function will only be called on typable expressions. As mentioned above, we also need an auxiliary function that generates constraints from subtyping judgments—the function is used in Check and the rule for fixpoints. Besides, there are auxiliary functions for fixpoints.

The algorithm is close to the usual type checking algorithm of CIC. The most difficult part is the case of fixpoints. First, the algorithm type checks the type annotation $T^\star = \Pi\Delta^\circ.\ \Pi x : I^\star\ \boldsymbol{u}^\circ.\ U^\star$ and gets, as part of the result, an annotated term T that corresponds to the final type of the recursive definition, as well as it's sort, W. Here, we identify the stage variable α annotating the decreasing inductive argument. Next, we compute from U and $U^\star$, the expected return type, $\widehat{U}$, for the body of the recursive definition using the shift function, which replaces all stage annotations s in

recursive positions by $\widehat{s}$; in addition, shift returns the set $V^\star$ of replaced variables. Once done, we check that the body e° can be decorated into an expression e of type $\widehat{T} = \Pi\Delta.\ \Pi x\!:\!I^{\widehat{\alpha}}\ \boldsymbol{u}.\ \widehat{U}$. Finally, we call the auxiliary function RecCheck to guarantee termination. The function RecCheck takes as input:

- the stage variable α which corresponds to the recursive argument, and which must be mapped to a fresh base stage $\imath$;
- a set of stage variables $V^\star$ that must be mapped to a stage expression with the same base stage as α. The set $V^\star$ is determined by the position types in the tag of the recursive definition. In particular, we have $\alpha \in V^\star$;
- a set of stage variables $V^{\neq}$ that must be mapped to a stage expression with a base stage different from $\imath$;
- a set of constraints C';

and returns an error or a set of constraints subject to some conditions. In [7], we provide an implementation of RecCheck and a proof of some soundness and completeness conditions. We use these results in the proof of the proposition below.

Proposition 1. *Assume that typable terms and normalizing and that the specification is functional.*

- Check *and* Infer *are sound:*

$$\begin{array}{rcl} \mathsf{Check}(V,\Gamma,e^\circ,T) = (V',C,e) & \Rightarrow & \forall \rho \models C.\ \rho\Gamma \vdash \rho e : \rho T \\ \mathsf{Infer}(V,\Gamma,e^\circ) = (V',C,e,T) & \Rightarrow & \forall \rho \models C.\ \rho\Gamma \vdash \rho e : \rho T \end{array}$$

- Check *and* Infer *terminate and are complete:*
  1. *If* $\rho\Gamma \vdash e : \rho T$ *and* $\mathsf{FV}(\Gamma,T) \subseteq V$ *then there exist* V', C, e', ρ' *such that* $\rho' \models C$ *and* $\rho =_V \rho'$ *and* $\rho' e' = e$ *and* $\mathsf{Check}(V,\Gamma,|e|,T) = (V',C,e')$.
  2. *If* $\rho\Gamma \vdash e : T$ *and* $\mathsf{FV}(\Gamma) \subseteq V$ *there exist* V', C, e', T', ρ' *such that* $\rho' \models C$ *and* $\rho' T' \preceq T$ *and* $\rho' =_V \rho$ *and* $\rho' e' = e$ *and* $\mathsf{Infer}(V,\Gamma,|e|) = (V',C,e',T')$.

Proof. By simultaneous induction on the structure of e° for soundness and on the typing derivation for completeness.

Normalization Both consistency and decidability of type checking and size inference rely on normalization.

Conjecture 1 *If* $\Gamma \vdash M : A$ *taking as specification that of CIC [22] then* M *is strongly normalizing.*

Our earlier work on non-dependent systems demonstrates that it is rather direct to adapt existing model constructions to type-based termination, and that the resulting model is in fact easier to justify than for systems that use a syntactic guard predicate to enforce termination. Thus we strongly believe—but have not checked details—that existing model constructions for CIC, e.g. [16, 24], can be adapted immediately to CIC$\widehat{\ }$, using the construction of [6] for inductive definitions. As discussed in Section 2, it is likely that the conjecture can also be deduced from [10].

6 Implementation and case studies

We have developed a prototype implementation of the type checker and size inference algorithm for a fragment of CIC$\hat{}$, and used it to program quicksort and prove its correctness.

We have also used CIC$\hat{}$ to define general recursive functions, following the approach developed by Bove and Capretta [13] for Martin-Löf's type theory—we have not carried this work with the prototype because it currently does not support inductive families. In a nutshell, the approach consists in defining an inductive predicate that characterizes the domain of the function to be defined, and to define the function by induction on the proof that the argument is in the domain. One difficulty with this approach is that it requires to prove some inversion lemmas in a very contrived way, not using Coq standard tactics for inversion [5, 8]. In a type-based setting, the problem disappears, i.e. there is no restriction on the way the lemmas are proved, because the statements make it clear that the recursive call will be performed on a smaller proof. The example illustrates that type-based termination makes it easier to define general recursive definitions, and suggests that CIC$\hat{}$ is a more appropriate setting than CIC to pursue the program of [5] to support general recursive definitions via tools that generate termination proofs for functions that are shown terminating with e.g. the size-change principle [20].

7 Concluding remarks

We have defined CIC$\hat{}$, a variant of the Calculus of Inductive Constructions that enforces termination of recursive definitions via sized types, and shown that it enjoys the required meta-theoretical properties to serve as a basis for proof assistants. A prototype implementation has been developed and applied on medium size case studies.

The immediate objective for further work is to resolve outstanding issues that CIC$\hat{}$ inherited from $F\hat{}$, and that must be solved prior to integrating type-based termination in Coq, namely mutually recursive types and global definitions. Our longer term goal is to integrate type-based termination in Coq. We believe that it shall result in a more robust and flexible system that is easier for users to understand and for developers to evolve.

References

1. A. Abel. Termination checking with types. *RAIRO– Theoretical Informatics and Applications*, 38:277–320, October 2004.
2. A. Abel. *A Polymorphic Lambda-Calculus with Sized Higher-Order Types*. PhD thesis, Ludwig-Maximilians-Universität München, 2006.
3. H. Barendregt. Lambda calculi with types. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, pages 117–309. Oxford Science Publications, 1992. Volume 2.
4. B. Barras. *Auto-validation d'un système de preuves avec familles inductives*. PhD thesis, Université Paris 7, 1999.

5. G. Barthe, J. Forest, D. Pichardie, and V. Rusu. Defining and reasoning about recursive functions: a practical tool for the Coq proof assistant. In M. Hagiya and P. Wadler, editors, *Proceedings of FLOPS'06*, volume 3945 of *Lecture Notes in Computer Science*, pages 114–129. Springer-Verlag, 2006.
6. G. Barthe, M. J. Frade, E. Giménez, L. Pinto, and T. Uustalu. Type-based termination of recursive definitions. *Mathematical Structures in Computer Science*, 14:97–141, February 2004.
7. G. Barthe, B. Grégoire, and F. Pastawski. Practical inference for typed-based termination in a polymorphic setting. In P. Urzyczyn, editor, *Proceedings of TLCA'05*, volume 3641 of *Lecture Notes in Computer Science*, pages 71–85. Springer-Verlag, 2005.
8. Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development—Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer-Verlag, 2004.
9. F. Blanqui. *Théorie des Types et Récriture*. PhD thesis, Université Paris XI, Orsay, France, 2001. Available in english as "Type theory and Rewriting".
10. F. Blanqui. A type-based termination criterion for dependently-typed higher-order rewrite systems. In V. van Oostrom, editor, *Proceedings of RTA'04*, volume 3091 of *Lecture Notes in Computer Science*, pages 24–39, 2004.
11. F. Blanqui. Decidability of type-checking in the calculus of algebraic constructions with size annotations. In C.-H.L. Ong, editor, *Proceedings of CSL'05*, volume 3634 of *Lecture Notes in Computer Science*, pages 135–150. Springer-Verlag, 2005.
12. F. Blanqui and C. Riba. Constraint based termination. Manuscript, 2006.
13. A. Bove and V. Capretta. Modelling general recursion in type theory. *Mathematical Structures in Computer Science*, 15:671–708, February 2005.
14. W.-N. Chin and S.-C. Khoo. Calculating sized types. *Higher-Order and Symbolic Computation*, 14(2–3):261–300, September 2001.
15. T. Coquand and C. Paulin. Inductively defined types. In P. Martin-Löf and G. Mints, editors, *Proceedings of COLOG'88*, volume 417 of *Lecture Notes in Computer Science*, pages 50–66. Springer-Verlag, 1988.
16. H. Geuvers. A short and flexible proof of strong normalisation for the Calculus of Constructions. In P. Dybjer, B. Nordström, and J. Smith, editors, *Proceedings of TYPES'94*, volume 996 of *Lecture Notes in Computer Science*, pages 14–38. Springer-Verlag, 1995.
17. E. Giménez. Structural recursive definitions in Type Theory. In K.G. Larsen, S. Skyum, and G. Winskel, editors, *Proceedings of ICALP'98*, volume 1443 of *Lecture Notes in Computer Science*, pages 397–408. Springer-Verlag, 1998.
18. J. Hughes, L. Pareto, and A. Sabry. Proving the correctness of reactive systems using sized types. In *Proceedings of POPL'96*, pages 410–423. ACM Press, 1996.
19. J.-P. Jouannaud and M. Okada. Executable higher-order algebraic specification languages. In *Proceedings of LICS'91*, pages 350–361. IEEE Computer Society Press, 1991.
20. C.-S. Lee, N. D. Jones, and A. M. Ben-Amram. The size-change principle for program termination. In *Proceedings of POPL'01*, pages 81–92. ACM Press, 2001.
21. N. P. Mendler. Inductive types and type constraints in the second-order lambda calculus. *Annals of Pure and Applied Logic*, 51(1-2):159–172, March 1991.
22. C. Paulin-Mohring. *Définitions Inductives en Theorie des Types d'Ordre Superieur*. Habilitation à diriger les recherches, Université Claude Bernard Lyon I, 1996.
23. David Wahlstedt. Type theory with first-order data types and size-change termination. Technical report, Chalmers University of Technology, 2004. Licentiate thesis 2004, No. 36L.
24. B. Werner. *Méta-théorie du Calcul des Constructions Inductives*. PhD thesis, Université Paris 7, 1994.
25. H. Xi. Dependent Types for Program Termination Verification. In *Proceedings of LICS'01*, pages 231–242. IEEE Computer Society Press, 2001.

$$
\begin{array}{lcl}
\mathsf{Check}(V, \Gamma, e^\circ, T) & = & V_e, C_e \cup T_e \preceq T, e \\
& & \text{where } (V_e, C_e, e, T_e) := \mathsf{Infer}(V, \Gamma, e^\circ) \\
\mathsf{Infer}(V, \Gamma, \omega) & = & V, \emptyset, \omega, \mathsf{axioms}(\omega) \\
\mathsf{Infer}(V, \Gamma, x) & = & V, \emptyset, x, \Gamma(x) \\
\mathsf{Infer}(V, \Gamma, \lambda x{:}T_1^\circ.\, e^\circ) & = & V_e, C_1 \cup C_e, \lambda x{:}T_1^\circ.\, e, \Pi x{:}T_1.\, T_2 \\
& & \text{where } (V_1, C_1, T_1, W_1) := \mathsf{Infer}(V, \Gamma, T_1^\circ) \text{ and } \mathsf{whnf}(W_1) = \omega_1 \\
& & \quad (V_e, C_e, e, T_2) := \mathsf{Infer}(V_1, \Gamma; x{:}T_1, e^\circ) \\
\mathsf{Infer}(V, \Gamma, \Pi x{:}T_1^\circ.\, T_2^\circ) & = & \\
& & \quad V_2, C_1 \cup C_2, \Pi x{:}T_1.\, T_2, \mathsf{rules}(\omega_1, \omega_2) \\
& & \text{where } (V_1, C_1, T_1, W_1) := \mathsf{Infer}(V, \Gamma, T_1^\circ) \text{ and } \mathsf{whnf}(W_1) = \omega_1 \\
& & \quad (V_2, C_2, T_2, W_2) := \mathsf{Infer}(V_1, \Gamma; x{:}T_1, T_2^\circ) \\
& & \text{and } \mathsf{whnf}(W_2) = \omega_2 \\
\mathsf{Infer}(V, \Gamma, e_1^\circ\ e_2^\circ) & = & V_2, C_1 \cup C_2, e_1\ e_2, T[x := e_2] \\
& & \text{where } (V_1, C_1, e_1, T_1) := \mathsf{Infer}(V, \Gamma, e_1^\circ) \\
& & \quad \mathsf{whnf}(T_1) = \Pi x{:}T_2.\, T \\
& & \quad (V_2, C_2, e_2) := \mathsf{Check}(V_1, \Gamma, e_2^\circ, T_2) \\
\mathsf{Infer}(V, \Gamma, I) & = & V \cup \{\alpha\}, \emptyset, I^\alpha, \mathsf{TypeInd}(I) \quad \text{with } \alpha \notin V \\
\mathsf{Infer}(V, \Gamma, c(\boldsymbol{p}^\circ, \boldsymbol{a}^\circ)) & = & V_c, C, c(\boldsymbol{p}^\circ, \boldsymbol{a}), T \\
& & \text{where } T_c := \mathsf{TypeConstr}(c, \alpha) \quad \text{with } \alpha \notin V \\
& & \quad \mathsf{params}(c) = \#\boldsymbol{p}^\circ \text{ and } \mathsf{args}(c) = \#\boldsymbol{a}^\circ \text{ and } x \text{ free in } \Gamma, \boldsymbol{p}, \boldsymbol{a} \\
& & \quad (V_c, C, x\ \boldsymbol{p}\ \boldsymbol{a}, T) = \mathsf{Infer}(V \cup \{\alpha\}, \Gamma(x : T_c), x\ \boldsymbol{p}^\circ\ \boldsymbol{a}^\circ) \\
\mathsf{Infer}(V, \Gamma, \mathsf{case}_{P^\circ}\ e_c^\circ\ \mathsf{of}\ \{c_i \Rightarrow e_i^\circ\}) & = & \\
& & \quad V_n, C_c \cup C_p \cup \bigcup_{i=0}^n C_i, \mathsf{case}_{P^\circ}\ e_c\ \mathsf{of}\ \{c_i \Rightarrow e_i\}, P\ \boldsymbol{a}\ e_c \\
& & \text{where } (V_c, C_c, e_c, T_c) := \mathsf{Infer}(V, \Gamma, e_c^\circ) \\
& & \quad \mathsf{whnf}(T_c) = I^r\ \boldsymbol{p}\ \boldsymbol{a} \text{ and } \mathsf{params}(I) = \#\boldsymbol{p} \text{ and } \alpha \notin V_c \\
& & \quad (V_P, C_P, P, T_P) := \mathsf{Infer}(V_c \cup \{\alpha\}, \Gamma, P^\circ) \text{ and } T_{P_0} := T_P \\
& & \quad \forall i = 1 \ldots \mathsf{args}(I) + 1, \Pi x_i{:}T_i.\, T_{P_i} := \mathsf{whnf}(T_{P_{i-1}}) \\
& & \quad \omega' := \mathsf{whnf}(T_{P_{\mathsf{args}(I)+1}}) \text{ and } \mathsf{elim}(I.\omega, \omega') \\
& & \quad C_0 := r \sqsubseteq \widehat{\alpha} \cup T_P \preceq \mathsf{TypePred}(I, \alpha, \boldsymbol{p}, \omega')) \text{ and } V_0 := V_P \\
& & \quad \forall i = 1 \ldots n, (V_i, C_i, e_i) := \\
& & \qquad \mathsf{Check}(V_{i-1}, \Gamma, e_i^\circ, \mathsf{TypeBranch}(c_i, \alpha, P, \boldsymbol{p})) \\
& & \text{given } T^\star \equiv \Pi\Delta^\circ.\, \Pi x{:}I^\star\ \boldsymbol{u}^\circ.\, U^\star \quad \text{with } \#\Delta^\circ = n - 1 \\
\mathsf{Infer}(V, \Gamma, \mathsf{fix}_n\ f{:}T^\star := e_B^\circ) & = & \\
& & \quad V_B, C_f, \mathsf{fix}_n\ f{:}T^\star := e_B, \Pi\Delta.\, \Pi x{:}I^\alpha\ \boldsymbol{u}.\, U \\
& & \text{where } (V_T, C_T, \Pi\Delta.\, \Pi x{:}I^\alpha\ \boldsymbol{u}.\, U, W) := \mathsf{Infer}(V, \Gamma, |T^\star|) \\
& & \text{and } \mathsf{whnf}(W) = \omega \\
& & \quad (V^\star, \widehat{U}) := \mathsf{shift}(U, U^\star) \text{ and } T' := \Pi\Delta.\, \Pi x{:}I^\alpha\ \boldsymbol{u}.\, U \\
& & \quad (V_B, C_B, e_B) := \\
& & \qquad \mathsf{Check}(V_T, \Gamma(f : T'), e_B^\circ, \Pi\Delta.\, \Pi x{:}I^{\widehat{\alpha}}\ \boldsymbol{u}.\, \widehat{U}) \\
& & \quad C_f := \mathsf{RecCheck}(\alpha, V^\star, V_B \backslash V^\star, C_T \cup C_B \cup U \preceq \widehat{U})
\end{array}
$$

Fig. 3. Inference Algorithm