On type-based termination and dependent pattern matching in the calculus of inductive constructions

Jorge Luis Sacchini

ParisTech
INSTITUT DES SCIENCES ET TECHNOLOGIES
PARIS INSTITUTE OF TECHNOLOGY

MINES ParisTech

École doctorale n°84 :
Sciences et technologies de l'information et de la communication

Doctorat européen ParisTech

T H È S E

pour obtenir le grade de docteur délivré par

l'École nationale supérieure des mines de Paris

Spécialité « Informatique temps-réel, robotique et automatique »

présentée et soutenue publiquement par

Jorge Luis SACCHINI

le 29 juin 2011

Terminaison basée sur les types et filtrage dépendant pour le calcul des constructions inductives

~ ~ ~

On Type-Based Termination and Dependent Pattern Matching in the Calculus of Inductive Constructions

Directeur de thèse : **Gilles BARTHE**
Co-encadrement de la thèse : **Benjamin GRÉGOIRE**

Jury

Benjamin WERNER, Directeur de recherche, LIX, École Polytechnique	Président
Herman GEUVERS, Professeur, ICIS, Radboud University Nijmegen	Rapporteur
Alexandre MIQUEL, Maitre de conférences, LIP, ENS de Lyon	Rapporteur
Andreas ABEL, Professeur assistant, Ludwig-Maximilians-Universität	Examinateur
Gilles BARTHE, Directeur de recherche, IMDEA	Examinateur
Benjamin GRÉGOIRE, Chargé de recherche, Marelle, INRIA Sophia Antipolis	Examinateur

T H È S E

MINES ParisTech
Centre de Mathématiques Appliquées
Rue Claude Daunesse B.P. 207, 06904 Sophia Antipolis Cedex, France

Abstract

Proof assistants based on dependent type theory are progressively used as a tool to develop certified programs. A successful example is the Coq proof assistant, an implementation of a dependent type theory called the Calculus of Inductive Constructions (CIC). Coq is a functional programming language with an expressive type system that allows to specify and prove properties of programs in a higher-order predicate logic.

Motivated by the success of Coq and the desire of improving its usability, in this thesis we study some limitations of current implementations of Coq and its underlying theory, CIC. We propose two extension of CIC that partially overcome these limitations and serve as a theoretical basis for future implementations of Coq.

First, we study the problem of termination of recursive functions. In Coq, all recursive functions must be terminating, in order to ensure the consistency of the underlying logic. Current techniques for checking termination are based on syntactical criteria and their limitations appear often in practice. We propose an extension of CIC using a type-based mechanism for ensuring termination of recursive functions. Our main contribution is a proof of Strong Normalization and Logical Consistency for this extension.

Second, we study pattern-matching definitions in CIC. With dependent types it is possible to write more precise and safer definitions by pattern matching than with traditional functional programming languages such as Haskell and ML. Based on the success of dependently-typed programming languages such as Epigram and Agda, we develop an extension of CIC with similar features.

Résumé

Les assistants de preuve basés sur des théories des types dépendants sont de plus en plus utilisé comme un outil pour développer programmes certifiés. Un exemple réussi est l'assistant de preuves Coq, fondé sur le Calcul des Constructions Inductives (CCI). Coq est un langage de programmation fonctionnel dont un expressif système de type qui permet de préciser et de démontrer des propriétés des programmes dans une logique d'ordre supérieur.

Motivé par le succès de Coq et le désir d'améliorer sa facilité d'utilisation, dans cette thèse nous étudions certaines limitations des implémentations actuelles de Coq et sa théorie sous-jacente, CCI. Nous proposons deux extension de CCI que partiellement resourdre ces limitations et que on peut utiliser pour des futures implémentations de Coq.

Nous étudions le problème de la terminaison des fonctions récursives. En Coq, la terminaison des fonctions récursives assure la cohérence de la logique sous-jacente. Les techniques actuelles assurant la terminaison de fonctions récursives sont fondées sur des critères syntaxiques et leurs limitations apparaissent souvent dans la pratique. Nous proposons une extension de CCI en utilisant un mécanisme basé sur les type pour assurer la terminaison des fonctions récursives. Notre principale contribution est une preuve de la normalisation forte et la cohérence logique de cette extension.

Nous étudions les définitions par filtrage dans le CCI. Avec des types dépendants, il est possible d'écrire des définitions par filtrage plus précises, par rapport à des langages de programmation fonctionnels Haskell et ML. Basé sur le succès des langages de programmation avec types dépendants, comme Epigram et Agda, nous développons une extension du CCI avec des fonctions similaires.

Acknowledgments

I would like to express my deepest gratitude to my supervisor, Benjamin Grégoire, for his encouragement and trust during the past four years, and for always being available as a colleague and as a friend. Also to my co-supervisor, Gilles Barthe, for continually providing advice and expertise.

I am extremely grateful to the other members of the dissertation committee. To Herman Geuvers and Alexandre Miquel for accepting to review this work, for their kind comments and insightful questions. To Benjamin Werner for presiding the jury and to Andreas Abel for his detailed comments and questions that helped improved this work. It is truly an honor to have such an excellent committee.

I would like to thank Hugo Herbelin, Pierre Corbineau, Bruno Barras, and Benjamin Werner, for many fruitful discussions and collaborations. I would like to specially thank Bruno and Benjamin for hosting me in the Typical team during the summer of 2009.

I would also like to thank the members of the Marelle and Everest teams for providing a stimulating work environment, and the Latin community at INRIA Sophia-Antipolis for all the barbecues and memorable moments we shared together. I am specially indebted to César Kunz, Gustavo Petri, Santiago Zanella, Tamara Rezk, Guido Pusiol, and Daniel Zullo for all the help they gave me when I was moving to and from France.

I also offer my gratitude to my high-school friends in Argentina and the OTCB members for always being there in spite of the distance. I specially thank Dante Zanarini from whom I have learned a lot while we were students at University of Rosario, and Erica Hinrichsen for teaching me mathematical thinking over 15 years ago. I would also like to thank my friends at Casa Argentina for an unforgettable summer in Paris.

Many thanks to Iliano Cervesato for his moral and financial support for my defense and to our new friends in Doha for helping us adapt to a new country.

I wish to thank my family for their support and encouragement when I decided to study abroad.

Finally, I thank my wife Tomoki for her love, for her patience, and for standing next to me during the most difficult times of this process.

Contents

List of Figures

Chapter 1

Introduction

The reliance on software systems in our daily life has been increasing for many years now. In critical applications such as medicine, aviation, or finance, software bugs (software errors) can cause enormous human and economic losses. Therefore, there is an increasing interest in developing tools and techniques that provide cost-effective assurance of the safety properties of software products.

Proof assistants based on dependent type theory are gaining attention in recent years, not only as a tool for developing formal mathematical proofs, but also as a tool for developing certified programs. By certified programs we mean a program or algorithm whose behavior is specified in a formal language together with a (formal) proof that the specification is met.

Dependent type theory was introduced by Per Martin-Löf [58], under the name Type Theory, as a formalism for developing constructive mathematics. It can be seen as a typed functional programming language, similar to Haskell or ML, where predicate logic can be interpreted through the so-called Curry-Howard isomorphism. Also known as formulas-as-types, proof-as-programs, the Curry-Howard isomorphism establishes a correspondence between logical systems and functional programming languages. Formulas are represented by types, while programs of a given type can be seen as proofs of the formula the type represents. Type Theory is thus a unified formalism where programs and specifications can be developed. It is the foundation for several proof assistants such as Lego [34], Alf [57], NuPrl [28], and Coq [23,80]. Also several programming languages based on dependent types have recently been proposed such as Cayenne [10], Epigram [60], Agda [69], ATS [86], Ωmega [74], and Guru [79], to name a few.

One important property of proof assistants as functional programming languages is that the computation (evaluation) of well-typed programs always terminate (this property is called strong normalization). This is a distinguishing feature with respect to traditional functional programming languages such as Haskell or ML (or any general-purpose programming language for that matter), where non-terminating programs can be written. Termination of programs ensures two important properties for proof assistants: logical consistency and decidability of type checking. Logical consistency means that the proof assistant, when viewed as a logic, is consistent, i.e., it is not possible to prove false propositions. An inconsistent logic is useless to prove program specifications, as any proposition is valid. Decidability of type checking means that the problem of checking that a formula is proved by a given proof is decidable. Through the Curry-Howard isomorphism, proof checking reduces to type checking.

We are interested in Coq, arguably one of the most successful proof assistants. Developed

continually for over 20 years, it is a mature system where a large body of formal proofs and certified programs have been developed. Notorious examples include the formal verification of the Four Color Theorem, developed by Gonthier and Werner [43], and the CompCert project [52] that resulted in a formally verified compiler for a large subset of C.

Coq is based on a dependent type theory called the Calculus of (Co-)Inductive Constructions (CIC) [72]. The main feature of CIC is the possibility of defining *inductive types*. Inductive types can be seen as a generalization of the datatype constructions commonly found in Haskell or ML. They are an essential tool in the development of certified programs. Coupled with dependent types, inductive definitions can be used to define data structures as well as logical predicates, in an intuitive and efficient way.

Programs (and proofs) on inductive types are defined using pattern-matching and recursive functions, similarly to traditional functional programming languages. It is important that their definition is clear and precise, and their semantics intuitive. Current implementations of these mechanisms in Coq have some limitations that often appear in practice, hindering the usability of the system. In this thesis, we study some of these limitations and propose extensions of CIC that (partially) overcome them. Our objective is that these extensions will serve as the theoretical basis for future implementations of Coq that would improve its usability and efficiency.

Concretely, we propose an extension of CIC with a type-based termination mechanism for ensuring termination of recursive functions. Type-based termination is a known technique to ensure termination that improves on current syntactical mechanisms used by Coq. Our contribution is a proof of strong normalization for the proposed extension. As a consequence, we obtain logical consistency, a necessary property of the core theory of a proof assistant.

A second contribution of this thesis is a proposal for a new pattern-matching construction for CIC. The new construction takes advantage of dependent types to allow the user to write precise and safe definitions by pattern matching by allowing, for example, the automatic elimination of impossible cases.

In the rest of the chapter we detail the motivations for this work and our contributions. In the next section, we present a short introduction to CIC. In Sect. 1.2 and Sect. 1.3 we describe the limitations of CIC and Coq that are the focus of this work. Our contributions are summarized in Sect. 1.4. Finally, in Sect. 1.5 we give a brief overview of the rest of this work.

1.1 The Calculus of (Co-)Inductive Constructions

CIC is a functional programming language with a powerful typing system featuring dependent types. It is based on Luo's Extended Calculus of Constructions [56] (ECC), which is itself based on Coquand's Calculus of Constructions [32] (CC). The latter is a basic impredicative dependent type theory, inspired by Martin-Löf's Type Theory. With respect to CC, ECC adds a predicative hierarchy of universes and the so-called Σ-types. With respect to ECC, CIC adds the possibility of defining new types using inductive and co-inductive definitions.

In this section we present a short introduction to CIC focusing on inductive definitions. Readers familiar with CIC can skip this section and continue with Sections 1.2 and 1.3 where we describe the motivation for this work.

CIC as a programming language. CIC is a typed functional programming language, and as such, it allows to write programs and types. Similarly to ML or Haskell, programs in CIC are constructed by defining and combining functions. New types can be defined by the user. Typical examples include data structures such as lists, trees, or arrays. Functions operating on these data structures are defined using recursion and pattern matching.

Each program in CIC can be given a *type*. A *typing judgment* is a relation that associates a program with its type. A typing judgment stating that program M has type T is denoted by

$$\vdash M : T \ .$$

Typing judgments are described by a set of *typing rules* that define, for each program construction, how to assign a type to a program given the type of its parts.

Programs can be computed (or evaluated). Computation is defined by a set of *reduction rules* that repeatedly transform (or reduce) a program until no more reductions can be applied. Programs that can not be further reduced are said to be in *normal form*.

CIC as a logic. It is well-known that there is a correspondence between intuitionistic propositional logic, natural deduction and the simply-typed λ-calculus. Logical formulas can be represented by types (in particular, logical implication is represented by function-space arrow), while proofs are represented by programs. For example, a proof of $P \to Q$ (P implies Q) is a function (program) that transforms a proof of P into a proof of Q. This isomorphism is known as the *Curry-Howard isomorphism* (or formulas-as-types, proofs-as-programs).

The Curry-Howard isomorphism implies that typed λ-calculi can be used as a formalism to express formulas and proofs. The typing judgment $\vdash M : T$ can also be read as formula T is proved by M. Checking that a formula is proved by a given proof reduces then to a type-checking problem. Formulas that can be represented by types include, for example, logical connectives ($P \wedge Q$, $P \to Q$, ...), or properties of computational object such as a predicate stating that a list is sorted, or that a number is prime.

Under the Curry-Howard isomorphism, CIC corresponds to higher-order predicate logic. I.e., proofs and formulas of higher-order predicate logic can be represented (respectively) as programs and types of CIC. As we show below, this correspondence allows us to write specifications and prove that programs meet their specifications.

Dependent types. The tool that allows to write formulas of predicate logic in a typed λ-calculus is *dependent types*. Basically, a dependent type is a type that can contain a term. In other words, the type *depends* on terms, hence the name. This feature allows us to express properties of terms (i.e., properties of programs). For example, we can represent a formula stating that n is a prime number, by a type of the form

$$\mathsf{prime}(n)$$

where n has type int. As types and contain terms (and viceversa), the distinction between types and terms is blurred in the case of dependent types. This is contrary to Haskell and ML, as well as non-dependent typed λ-calculi (e.g., simply-typed λ-calculus, or system F_ω) where terms (programs) and types are defined separately.

Going back to the example of prime numbers given above, the predicate prime is actually a *type family*, since it defines a type for each term of type int. In particular, the type $\mathsf{prime}(n)$

only makes sense if n has type int. This means that we need to check that types are well-typed, using the typing rules. While terms are classified by types, types are classified by a special class of types called *universes* or *sorts*. The class of sorts in CIC is defined by:

$$\mathsf{Sorts} ::= \mathsf{Prop} \mid \mathsf{Type}_i \quad \text{for } i \geq 0$$

The universe Prop is the *propositional universe*; types representing logical formulas (such as prime) are usually defined in Prop. The universes in $\{\mathsf{Type}_i\}_i$ are the *computational universes*; types representing data (such as int) are usually defined in Type_i, for some i.

For example, to check that the type of prime numbers given above, prime, is well-typed, we need to check that the following typing judgment are valid:

$$(x : \mathsf{int}) \vdash \mathsf{prime}(x) : \mathsf{Prop}$$

Let us mention that universes have also a type, as term in CIC. The typing rules for universes are the following:

$$\vdash \mathsf{Prop} : \mathsf{Type}_0 \qquad \vdash \mathsf{Type}_i : \mathsf{Type}_{i+1} \quad \text{for } i \geq 0$$

The main feature of dependent type system is dependent product, a generalization of the function arrow found in non-dependent type systems. A dependent product is a type of the form

$$\Pi(x : T).U(x)$$

where T and U are types such that U may contain x. It represents a type of functions f such that given a term $a : T$, $f(a)$ has type $U(a)$. For example, $\Pi(n : \mathsf{int}).\mathsf{even}(n + n)$, where $\mathsf{even}(n)$ is a predicate stating that n is even, is a formula stating that for any number n, $n+n$ is an even number. Usual function space $T \to U$ is a special case of dependent product where U does not depend on x, i.e., x does not appear in U.

As we mentioned, the motivation for introducing dependent types comes from the necessity of expressing formulas in predicate logic. Dependent types representing logical predicates are defined in Prop. However, it also makes sense to define dependent types in the computational universes. A typical use is to represent more precise data structures. For example, we can define a type $\mathsf{array}(A, n)$ representing the type of arrays whose elements have type A and have exactly n elements. It satisfies a typing judgment of the form:

$$(A : \mathsf{Type})(n : \mathsf{int}) \vdash \mathsf{array}(A, n) : \mathsf{Type}$$

Using this dependent type we can write a *safe* access function to an array. More precisely, we can define a function get to access the k-th element of an array with type:

$$\mathsf{get} : \Pi(A : \mathsf{Type})(k : \mathsf{int}).\mathsf{inBound}(n, k) \to \mathsf{array}(A, n) \to A$$

where $\mathsf{inBound}(n, k)$ is a logical predicate stating that k is in the bounds of the array. To call function get we need to supply a proof of this predicate, thus ensuring that the access to the array is safe. No runtime error can occur.

To sum up, in CIC we can define two kind of types: *computational types* and *propositional types*. The former kind consists of types representing entities intended to be used in computation such as lists or arrays. While the latter kind consists of types representing logical formulas. Program specification are represented by propositional types.

Programs of computational types are intended to be used for "real" computations, where the interest lies in the result of the evaluation. On the other hand, programs of propositional types (i.e., proofs) do not have any real computational value. Only their existence is important, since it implies that the formula they prove is true. Two proofs of the same formula can be seen as equivalent (this property is called proof-irrelevance).

Conversion. The type of an expression is usually not unique. For example, in Haskell, we can define type synonyms:

```
type IntList = [Int]
```

Then `IntList` and `[Int]` represent the same type, namely, the type of lists of integers. Any term of type `IntList` can be used in a place where a term of type `[Int]` is needed. Thus, when type-checking a term, it is necessary to check if two types are different representations of a same expression, like `IntList` and `[Int]` above. In that case, the checking can be done by first unfolding all type synonyms.

In the case of dependent types, the situation is more complicated since types can contain arbitrary terms. For example, consider the types $\mathsf{prime}(3+4)$ and $\mathsf{prime}(2+5)$. Both types represent, intuitively, the same proposition: namely, that 7 is a prime number. Both types can be transformed, by computation, into the same type $\mathsf{prime}(7)$. Then, any proof M of $\mathsf{prime}(3+4)$ should also be a proof of $\mathsf{even}(2+5)$ or $\mathsf{prime}(7)$. This intuition is represented in the conversion rule

$$\frac{M : T \qquad T \approx U}{M : U}$$

where $T \approx U$ means that T and U can be computed into the same type.

In large developments in Coq, the majority of the time required for type-checking can be devoted to tests of conversion. It is thus important that computation (i.e., reduction) be implemented efficiently. Some of the design choices of the system we present in this thesis are influenced by this requirement.

Strong normalization and logical consistency. Two properties of CIC that have important theoretical and practical consequences are strong normalization (SN) and logical consistency (LC). SN states that, for well-typed terms, all reductions sequences reach a normal form. In other words, there is no infinite reduction sequence starting from a well-typed term. LC states that there are formulas that can not be proved in CIC. If all formulas can be proved in CIC, the logic becomes inconsistent, since we would be able to prove also false statements such as $0 = 1$.

SN is a notoriously difficult property to establish, but it has important consequences. LC can be proved from SN (together with a property called Canonicity that characterizes the shape of expressions in normal form).

On the practical side, a consequence of SN is decidability of type checking. The problem with deciding of type-checking in dependent type theories is how to decide the conversion rule. That is, how to decide if two types are convertible. Decidability of conversion follows from SN and a property called Confluence stating that all reduction sequences starting from a term lead to the same normal form. A test to decide conversion is the following: compute the normal form of both types, which is guaranteed to exist, and then check that both results are equal.

The objective of this work is to propose sound extensions of CIC that overcome some of its limitations. It is important that SN and LC are still valid in any extension. Without LC, the use of CIC to reason about programs is lost. Without SN, we would not have decidability of type-checking, which could introduce some difficulties in practice and complicate the implementation.

Inductive definitions in CIC. One of the main features of CIC is the possibility of defining new types using inductive definitions. Inductive definitions can be seen as a generalization of datatypes construction of Haskell and ML. While this comparison is sufficient to give an intuition, there are some important differences. First, the dependent type system of CIC allows to define inductive definitions that have no correspondence in ML. Second, both concepts are semantically different.

Let us illustrate with an example. Consider Peano natural numbers, defined by zero and the successor operator. We can define it in CIC using the following inductive definition:

$$\mathsf{Ind}(\mathsf{nat} : \mathsf{Type} := \mathsf{O} : \mathsf{nat}, \mathsf{S} : \mathsf{nat} \to \mathsf{nat}) \ .$$

This statement introduces the type nat and constructors O and S with their respective type. Besides some syntactic differences, the same definition can be written in ML. In CIC, this definition has the following properties:

- nat is the smallest set that can be constructed using O and S,
- O is different from $\mathsf{S}(x)$ (the "no confusion" property),
- S is injective,
- we can define functions on nat using primitive recursion.

Intuitively, the semantics of type nat is the set of terms $\{\mathsf{O}, \mathsf{S}\,\mathsf{O}, \mathsf{S}(\mathsf{S}\,\mathsf{O}), \ldots\}$. In this intuitive semantics, the properties given above are sound. In particular, the last property that allows to define functions on nat using primitive recursion. Using the formula-as-types interpretation, it amounts to proving properties on nat using the principle of mathematical induction.

Constructors define the *introduction rules* of an inductive definition. They state how to build elements of the type. To reason about the elements of the type, we use *elimination rules*. There are several ways to define *elimination rules*.

In functional programming languages such as Haskell or ML, the elimination rules are pattern matching and recursive functions. In CIC, elimination rules have to be carefully considered to ensure that logical consistency is preserved. For example, in Haskell, the following function can have any type:

```
let f x = f (x + 1) in f 0
```

Allowing this kind of unrestricted recursion in CIC leads immediately to inconsistencies, since we can give the above function any type. In particular, a type representing a false statement, like $0 = 1$. Under the proposition-as-types view, the above function is a proof that $0 = 1$. The theory is inconsistent, since anything can be proved.

To avoid this kind of issues, all function in CIC are required to be total and terminating. This contrast with general-purpose programming languages like Haskell, where termination and totality are not issues that are dealt with inside the language. If these properties are a concern for the programmer, she would have to use external mechanisms to ensure them.

Early proposals of dependent type theories (including Martin-Löf Type Theory and CIC) used primitive recursion as a elimination rule. Basically, a primitive recursor for an inductive

definition is an induction principle derived from its structure. In the case of nat above, the primitive recursor derived is the principle of mathematical induction:

$$\mathsf{Elim}_{\mathsf{nat}} : \Pi(P : \mathsf{nat} \to \mathsf{Prop}).P\,\mathsf{O} \to (\Pi(n : \mathsf{nat}).P\,n \to P(\mathsf{S}\,n)) \to \Pi(n : \mathsf{nat}).P\,n;$$

basically, a property P is satisfied by any natural number n if it is satisfied by O, and for the successor of any n satisfying it. There is another variant of the primitive recursor where P has type $\mathsf{nat} \to \mathsf{Type}$ which allows to define computational objects.

Functions defined using primitive recursion are immediately total and terminating. Therefore, establishing logical consistency of primitive recursion is relatively easy compared to the other approaches we describe below. The main disadvantage of primitive recursors is that it is cumbersome to write function using them. For example, let us consider a function half that divides a natural number by 2. In Haskell, function half can be easily written by the following equations:

```
half O = O
half (S O) = O
half (S (S n)) = S (half n)
```

On the other hand, writing half using primitive recursion is more complicated. The reason is that, using primitive recursion, we have to define $\mathsf{half}\,(\mathsf{S}\,n)$ from the result of the recursive call $\mathsf{half}\,n$. But $\mathsf{half}\,(\mathsf{S}\,n)$ cannot be determined solely from $\mathsf{half}\,n$. It also depends on the parity of n: $\mathsf{half}\,(\mathsf{S}\,n) = \mathsf{S}\,(\mathsf{half}\,n)$ when n is odd, and $\mathsf{half}\,(\mathsf{S}\,n) = \mathsf{half}\,n$ when n is even. One way to proceed would be to define an auxiliary function half' such that $\mathsf{half}'\,n$ computes both $\mathsf{half}\,n$ and the parity of n (as a boolean). In Haskell, we could define it as

```
half' O = (O, true)
half' (S n) = let (r,p) = half' n in
                if p then (r, not p) else (S r, not p)
```

Writing half' using the primitive recursion scheme is direct since $\mathsf{half}'\,(\mathsf{S}\,n)$ can be determined from $\mathsf{half}'\,n$. However, defining the function this way is more complicated and less clear than the direct approach above.

The above example shows that using primitive recursion is more cumbersome than using pattern matching and recursion as in Haskell. An alternative approach, similar to that of Haskell, is to divide the elimination rule in two constructions: case analysis and fixpoints construction (i.e., recursive function definition).

Case analysis. The case analysis construction of CIC is inspired from the pattern matching mechanism of functional programming languages. In the case of natural numbers, the case analysis constructions has the form

$$\begin{array}{l}\mathsf{case}_P\ x\ \mathsf{of}\\ |\ \mathsf{O} \Rightarrow t_1\\ |\ \mathsf{S}\,y \Rightarrow t_2\end{array}$$

where P is the return type, which depends on the argument x. Each branch can have a different type. In the case above, t_1 has type $P\,\mathsf{O}$, while t_2 has type $P\,(\mathsf{S}\,y)$.

The computational rule associates is the obvious one: the case analysis construction reduces to the corresponding branch if the argument is headed by a constructor. In the case of natural numbers, we have the two reductions:

$$(\mathsf{case}\ \mathsf{O}\ \mathsf{of}\ |\ \mathsf{O} \Rightarrow t_1\ |\ \mathsf{S}\, y \Rightarrow t_2) \to t_1$$
$$(\mathsf{case}\ \mathsf{S}\, t\ \mathsf{of}\ |\ \mathsf{O} \Rightarrow t_1\ |\ \mathsf{S}\, y \Rightarrow t_2) \to t_2\,[y := t]$$

Case analysis, in the presence of inductive families, can be a powerful tool to write safer programs. However, it is rather cumbersome to use for this purpose. We describe the problem in Sect. 1.3. We propose a more expressive case analysis construction that solves some of the difficulties found in the usual case analysis construction (Sect. 1.4).

Fixpoints. Recursive functions can be defined in CIC using the fixpoint construction. The intuitive way of defining recursive definitions is using a construction of the form

$$\mathsf{fix}\ f : T := M$$

which defines a function of type T, where recursive calls to f can be performed in M. We say that M is the *body* of the fixpoint and f is a function defined by fixpoint, or simply f is a fixpoint. Intuitively, the function satisfies the equation $(\mathsf{fix}\ f : T := M) = M\,[f := \mathsf{fix}\ f : T := M]$. However, adding this equation as a computation rule for fixpoint would immediately break SN.

To avoid this problem, reduction should be restricted. The actual fixpoint construction has the form

$$\mathsf{fix}_n\ f : T := M$$

where n denotes the *recursive argument* of f. Reduction is allowed only when the fixpoint is applied to at least n arguments, and the recursive argument is in constructor form:

$$(\mathsf{fix}_n\ f : T := M)\,\vec{N}\,C \to (M\,[f := \mathsf{fix}_n\ f : T := M])\,\vec{N}\,C$$

where $\vec{N}$ is a sequence of $n-1$ arguments and C is in constructor form (i.e., it is a constructor applied to some arguments).

As we mentioned before, non-terminating recursive definitions might lead to inconsistencies. Hence, in CICand its implementation in Coq[1], several restrictions are imposed to ensure termination. In the latter, a syntactic criterion called *guard predicate* is used to check that fixpoint definitions terminate. This method has several limitations that we describe in Sect. 1.2.1. We show an alternative approach to ensure termination called *type-based termination* that possesses several advantages over guard predicates (Sect. 1.2.2). We propose an extension of CIC that uses the latter approach to ensure termination (Sect. 1.4).

Coinductive definitions. Infinite data structures arise naturally in some applications. In system specification, infinite data structures are useful to specify network protocols or systems that are meant to run indefinitely, like a webserver or an operating system.

Haskell and some implementations of ML (e.g., OCAML) allow the possibility to define and operate on infinite data structures. A typical example is infinite lists. The following program in Haskell defines an infinite list of alternating ones and zeros:

1. When we refer to Coq, we refer to any version from 8.0 to 8.3 inclusive.

```
alt = 1 : 0 : alt
```

Of course, being an infinite sequence, it is not possible to compute `alt` entirely. Instead, infinite data structures are computed *lazily*, on demand. For example, the program

```
take 10 alt
```

computes the first 10 elements of `alt`, while the rest of the list is not computed.

Infinite data structures can be defined in CIC using *coinductive definitions*. From a category-theory point of view, coinductive definitions are the *dual* of inductive definitions. The latter are initial algebras of some category, while the former are final co-algebras.

Coinductive definitions are expressed in the same way as inductive definitions although they differ semantically. Let us illustrate with the example. Natural numbers can also be defined as a coinductive type:

$$\mathsf{CoInd}(\mathsf{conat} : \mathsf{Type} := \mathsf{coO} : \mathsf{conat}, \mathsf{coS} : \mathsf{conat} \to \mathsf{conat})$$

Semantically, this type defines the *greatest* set that is closed under constructor application. This is the dual meaning of inductive types, where an inductive type defines the *smallest* set that is closed under constructor application. The above definition includes the term $\mathsf{coS}(\mathsf{coS}(\mathsf{coS}\ldots$ which is not in the inductive definition of natural numbers.

The introduction rules are the constructors, similar to inductive types. Two elimination rules are defined for coinductive types: case analysis and *cofixpoint definitions*. Case analysis on coinductive types is the same as case analysis on inductive types.

For inductive definitions, fixpoints *consume* data structures. Dually, for coinductive definitions, cofixpoint definitions *produce* data structures. In other words, fixpoints take an inductive type as argument, while cofixpoints return a coinductive type. Note that it does not make sense to define a fixpoint on an infinite data structure, since it would never terminate.

The dual of the termination requirement for fixpoint definitions is *productivity*. A term of a coinductive type can be seen as a sequence of constructor applications, possibly infinite. I.e., it has the form $C_{i1}(C_{i2}(\ldots$. A cofixpoint definition is productive if it is possible to compute any element of the sequence in finite time. For example, the definition of `alt` given above is productive, since any element of the list can be computed in finite time. The following program defining the list of natural numbers is also productive:

```
nats = 1 : map (+1) nats
```

Let us see why. The first element of `nats` can be easily computed. Then, the first element of `map (+1) nats` can also be computed, which allows to compute the second element of `nats`. In general, computing the n-th element of `nats` allows to compute the n-th element of `map (+1) nats` which allows to compute the $n+1$-th element of `nats`. Hence, the definition is productive.

A simple example of a non-productive definitions is the following:

```
nonp = 1 : tail nonp
```

Note that only the first element of the sequence can be computed.

Non-productivity of corecursive definitions might lead to inconsistencies. In CIC, several conditions are imposed on cofixpoint definition to ensure productivity. Basically, a definition

is accepted if all corecursive calls are performed under a constructor. For example, `alt` defined above is accepted, while `nats` and `nonp` are not, since the recursive call is not directly performed under a constructor, but as an argument of a function.

This criterion is rather restrictive in practice. As we explained above, `nats` is a productive definition, but it is rejected. In Sect. 5.3, we discuss how to extend the type-based termination approach into a *type-based productivity* criterion for corecursive definitions. However, we do not treat the general case of coinductive type. Instead, we only consider the case of *streams*, i.e. infinite sequences, defined by the following coinductive type:

$$\mathsf{CoInd}(\mathsf{stream}[A : \mathsf{Type}] : \mathsf{Type} := \mathsf{scons} : A \to \mathsf{stream}\, A \to \mathsf{stream}\, A)$$

1.2 Termination of Recursive Functions

Functions defined by recursion need to be terminating in CIC. In this section we describe two approaches to termination: guard predicates and type-based termination. The former is the mechanism used in Coq. It has some important limitations that we describe in detail. The latter is an alternative approach that is more expressive than guard predicates (i.e., it accepts more functions as terminating) and more intuitive. We describe the basic ideas of this approach.

1.2.1 Guard predicates

Guard predicates are a mechanism to ensure termination of recursive functions. The basic idea is to ensure that recursive calls are performed on *structurally smaller arguments*. For example, consider the following function to compute the sum of the first n natural numbers, using an accumulator:

$$\begin{aligned} F\, x\, r \stackrel{\text{def}}{=} \mathsf{fix}\ F := \lambda\, x\, r.\mathsf{case}\ x\ \mathsf{of} \\ \mid \mathsf{O} \Rightarrow r \\ \mid \mathsf{S}\, x' \Rightarrow F\, x'\, (x + r) \end{aligned}$$

Function F has two parameters and is defined by recursion on the first. This means that the first argument must be smaller in each recursive call. There is only one recursive call in this example. The first argument is x' which is *smaller* than $S\, x'$ which is itself equal to x, the original argument. We denote this with $x' \prec S\, x' \equiv x$. The function terminates, since recursion will eventually reach the base case O.

Note that this is valid in CIC because of the interpretation of inductive types, where constructors can only be applied a finite number of times. The same reasoning is not true in Haskell, where terms using infinite application of constructors is allowed.

The guard predicate for a function f in the body M defined by recursion on an argument x, denoted $\mathcal{G}_x^f(M)$, checks that recursive calls to f in the term M are performed on arguments that are structurally smaller than x. The structurally smaller relation, denoted with $\prec$, is a transitive relation defined by the rule $x \prec C(\ldots, x, \ldots)$, where C is a constructor of an inductive type. Recall that elements of inductive types are composed of finite applications of the constructors. Hence, the structurally-smaller relation is well-founded and recursion is terminating. The typing rule for fixpoint thus looks like:

$$\frac{\Gamma(f : T) \vdash M : T}{\Gamma \vdash \mathsf{fix}_n\ f : T := M : T}\ \mathcal{G}_x^f(M)$$

where x is the name of the n-th argument of f.

The guard predicates used in Coq are based on the work of Giménez [38,39,40]. The original definition was very restrictive; it has been extended in Coq to account for more cases. We do not give the formal definition here but, instead, we illustrate its features and limitations through examples.

A classical example is division of natural numbers. We define a recursive function, div, that computes $\lceil \frac{x}{y+1} \rceil$. It is defined by repeated subtraction using the function minus that computes $x - y$. They can be defined as follows:

$$\begin{aligned}\mathsf{minus} \stackrel{\text{def}}{=} \mathsf{fix}\ \mathsf{minus} := \lambda\, x\, y.\mathsf{case}\, x\, \mathsf{of}\ & \mid \mathsf{O} \Rightarrow x \\ & \mid \mathsf{S}\, x' \Rightarrow \mathsf{case}\ y\ \mathsf{of} \\ & \qquad \mid \mathsf{O} \Rightarrow \mathsf{S}\, x' \\ & \qquad \mid \mathsf{S}\, y' \Rightarrow \mathsf{minus}\, x'\, y'\end{aligned}$$

$$\begin{aligned}\mathsf{div} \stackrel{\text{def}}{=} \mathsf{fix}\ \mathsf{div} := \lambda\, x\, y.\mathsf{case}\, x\, \mathsf{of}\ & \mid \mathsf{O} \Rightarrow \mathsf{O} \\ & \mid \mathsf{S}\, x' \Rightarrow \mathsf{S}(\mathsf{div}(\mathsf{minus}\, x'\, y)\, y)\end{aligned}$$

The original guard predicate defined by Giménez accepts minus which can be defined by recursion on the first argument. The recursive call is performed on x' which satisfies $x' \prec \mathsf{S}\, x' \equiv x$. It can also be defined by recursion on the second argument using the same reasoning. On the other hand, div is defined by recursion on the first argument, but is not accepted, since the recursive call is performed with argument $\mathsf{minus}\, x'\, y$, which is not structurally smaller than x.

The definition of guard predicates has been extended over time in Coq to accept more recursive functions. Both functions given above are now accepted. In order to accept div, Coq needs to unfold the definition of minus to check that the recursive call is performed on a structurally smaller argument, i.e., $\mathsf{minus}\, x'\, y \prec \mathsf{S}\, x'$.

However, there is caveat. Consider the following alternative definition of subtraction:

$$\begin{aligned}\mathsf{minus}' \stackrel{\text{def}}{=} \mathsf{fix}\ \mathsf{minus}' := \lambda\, x\, y.\ \mathsf{case}\, x\, \mathsf{of}\ & \mid \mathsf{O} \Rightarrow \mathsf{O} \\ & \mid \mathsf{S}\, x' \Rightarrow\ \mathsf{case}\ y\ \mathsf{of} \\ & \qquad \mid \mathsf{O} \Rightarrow \mathsf{S}\, x' \\ & \qquad \mid \mathsf{S}\, y' \Rightarrow \mathsf{minus}'\, x'\, y'\end{aligned}$$

The only difference is in the highlighted case, where it was $\mathsf{O} \Rightarrow x$ before. Note that both definitions are operationally equivalent. However, if we write function div using this alternative definition of subtraction, Coq will not accept it because it cannot check that the recursive call $\mathsf{div}\, (\mathsf{minus}'\, x'\, y)\, y$ is structurally smaller than x. This example shows the fragility of guard predicates.

In practice, the guard predicate is not directly checked on the defined function, but on the normal form of the body of the recursive function. Thus, the typing rule looks like:

$$\frac{\Gamma(f : T) \vdash M : T}{\Gamma \vdash \mathsf{fix}_n\ f : T := M : T}\ \mathcal{G}^f_x(\mathrm{nf}(M))$$

This approach has two drawbacks. First, typing is inefficient since the term has to be reduced, creating a possibly large term. Second, SN is lost. Consider the following example:

$$\mathsf{fix}\ F\ (n : \mathsf{nat}) : \mathsf{nat} := (\lambda\_.\mathsf{O})\,(F\,n)$$

This function is accepted by Coq as terminating. The guard condition is not valid in the body given above, but it is valid in the normal form of it, which is simply O. However, the function is not strongly normalizing, since it accepts the following reduction sequence:

$$F\,\mathsf{O} \to (\lambda\_.\mathsf{O})\,(F\,\mathsf{O}) \to (\lambda\_.\mathsf{O})((\lambda\_.\mathsf{O})\,(F\,\mathsf{O})) \to \ldots$$

Designing a guard predicate that is not restrictive in practice, while at the same time being easy to understand and implement is a difficult task. While the guard predicate implemented in Coq have been used with success over the years, its limitations appear often in practice and the implementation is hard to maintain.

A significant amount of research has been devoted to develop techniques to write recursive (and co-recursive) functions that are accepted by the guard predicate of Coq (e.g., [24,27,77,78] to name a few).

1.2.2 Type-based termination

Type-based termination is an alternative approach to ensure termination. The basic idea is the use of sized types, i.e., types annotated with size information. Termination of recursive function is ensured by checking that recursive calls are performed on smaller arguments as evidenced by their types. We illustrate the approach with the example of natural numbers. The inductive definition of Peano numbers given above introduces a family of types of the form nat^s where s is a size (or stage). Intuitively, nat^s represents an approximation of the type of natural numbers. Concretely, the natural numbers whose size is smaller than s:

$$\mathsf{nat}^s = \{\mathsf{O}, \mathsf{S}\,\mathsf{O}, \ldots \underbrace{\mathsf{S}(\ldots(\mathsf{S}}_{s-1}\,\mathsf{O})\ldots)\}\ .$$

Hence nat^s is an approximation of the full type. Sizes are defined by a simple grammar:

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

where $\imath$, $\jmath$ are size variables, $\widehat{\cdot}$ represents a successor function on sizes and ∞ is used to represent full types.

These intuitions are expressed in the typing rules. For constructors, we have the following rules:

$$\frac{}{\Gamma \vdash \mathsf{O} : \mathsf{nat}^{\widehat{s}}} \qquad \frac{\Gamma \vdash M : \mathsf{nat}^s}{\Gamma \vdash \mathsf{S}\,M : \mathsf{nat}^{\widehat{s}}}$$

Size information is used to type recursive functions, as shown in the following (simplified) typing rule for fixpoint construction:

$$\frac{\Gamma(f : T^{\imath}) \vdash M : T^{\widehat{\imath}}}{\Gamma \vdash (\mathsf{fix}\ f : T := M) : T^s}\ \imath \text{ fresh}$$

The type T^ι can be seen as an approximation of a type T. To type-check the body M, we assume that f is in the approximation T^ι, and check that M is in the next approximation, $T^{\widehat{\iota}}$. This is enough to ensure that $\mathsf{fix}\ f : T := M$ is in any approximation of T.

This is a simplified version of the typing rule, sufficient to introduce the main concepts; the full version of the rule is given in the next chapter. To illustrate the advantages of this approach, let us revisit the example on division of natural numbers. Subtraction can be defined as follows:

$$\begin{array}{l}\mathsf{fix\ minus} : \mathsf{nat} \to \mathsf{nat} \to \mathsf{nat} := \lambda\ m\, n.\ \mathsf{case}\ m_{\mathsf{nat}(\widehat{\imath})}\ \mathsf{of} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathsf{O} \Rightarrow \mathsf{O} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathsf{S}\ m' \Rightarrow \mathsf{case}\ n\ \mathsf{of} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathsf{O} \Rightarrow \mathsf{S}\, m' \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathsf{S}\, n' \Rightarrow \mathsf{minus}\ m'_{\mathsf{nat}(\imath)}\ n\end{array}$$

The subscript type annotations are given for clarification purposes only, and are not needed in the actual syntax. For the sake of readability, we sometimes write $\mathsf{nat}(s)$ instead of nat^s.

There is no difference with the definition using guard predicates, except that with sized types, this function can be given a more precise type, namely $\mathsf{nat}^\iota \to \mathsf{nat}^\infty \to \mathsf{nat}^\iota$. This means that the size of the result is not greater than the size of the first arguments. This *size-preserving* type of minus is what allows to us to type-check div:

$$\begin{array}{l}\mathsf{fix\ div} : \mathsf{nat} \to \mathsf{nat} \to \mathsf{nat} := \lambda\ m\ n.\ \mathsf{case}\ m_{\mathsf{nat}(\widehat{\imath})}\ \mathsf{of} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathsf{O} \Rightarrow \mathsf{O} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathsf{S}\ m' \Rightarrow \mathsf{S}\ (\mathsf{div}\ (\mathsf{minus}\ m'_{\mathsf{nat}(\imath)}\ n)_{\mathsf{nat}(\imath)}\ n)\end{array}$$

Since m' has type nat^ι, $\mathsf{minus}\, m'\, n$ also has type nat^ι and the recursive call is valid. In this example, the advantage of type-based termination over guard predicates is that it is not necessary to look into the definition of minus. In particular, both definitions of minus given above can be used.

The type-based termination approach has several advantages over guard predicates.

- It is more expressive. As we show in Chapter 2, it accepts non-structurally recursive functions whose termination can not be ensured by guard predicates.
- It is more intuitive. Type-based termination relies on a semantically intuitive notion of size of inductive types. Therefore, it is easier to understand why functions are accepted or rejected.
- It is more efficient to implement. There is no need to consider recursive functions in normal form, and only typing information is needed to ensure termination.
- It is easier to implement. The guard condition implemented in Coq has been extended from the original formulation to account for more cases. It is one of the most delicate parts of the kernel of Coq which makes it difficult to extend and maintain. Although there are no mature implementations of type-based termination (as far as I am aware), prototype implementations [3,19] show that the type-based termination approach is much easier to implement than guard predicates.

For these reasons, we propose an extension of CIC with a type-based termination mechanism. The type-based termination approach has a long history, and we are not the first to propose such extensions of CIC; our work is directly based on the work of Barthe et al. [18,19]. We summarize our contribution in Sect. 1.4.

1.3 Pattern matching

Pattern matching is an essential feature of functional programming languages. It allows to define clear and concise functions by a sequence of equations. In the presence of dependent type families, it can be a powerful tool to write concise and safer functions. However, using the basic case analysis construction of CIC can be cumbersome. In this section, we identify the problem and propose a solution.

We use lists as a running example. In CIC, we can define it by

$$\mathsf{Ind}(\mathsf{list}(A : \mathsf{Type}) : \mathsf{Type} := \mathsf{nil} : \mathsf{list}\, A, \mathsf{cons} : A \to \mathsf{list}\, A \to \mathsf{list}\, A)$$

Standard functions operating on lists include head (that takes the first element of a non-empty list) and tail (that removes the first element of a non-empty list and returns the remaining part). In ML they are defined by

```
head (x::xs) = x
tail (x::xs) = xs
```

where the infix operator (::) denotes the cons constructor. Note that these functions are partial: they are not defined on the empty list. If one applies these to the empty list, a run-time exception occurs. Then, it is up to the programmer to ensure that this situation will not arise, or to properly handle the exceptional case.

As we mentioned, in CIC it is not possible to take the same approach. All functions defined must be total. One possibility for defining the above functions is to return a default value when the argument is outside the domain of the function. Another possibility is to use inductive families to get a more precise definition of the functions.

Inductive families are a generalization of inductive types. An inductive family defines a family of inductive types with the same structure, indexed by some arguments. Let us illustrate with a typical example: lists indexed by their length, usually called vectors. We can define it in CIC as follows:

$$\begin{aligned}\mathsf{Ind}(\mathsf{vec}(A : \mathsf{Type}) : \mathsf{nat} \to \mathsf{Type} := \ & \mathsf{vnil} : \mathsf{vec}\, A\, \mathsf{O}, \\ & \mathsf{vcons} : \Pi(n : \mathsf{nat}).A \to \mathsf{vec}\, A\, n \to \mathsf{vec}\, A\, (\mathsf{S}\, n))\end{aligned}$$

Note that $\mathsf{vec}\, A$ is not a type, but a type family, indexed by nat: $\mathsf{vec}\, A\, \mathsf{O}$, $\mathsf{vec}\, A\, (\mathsf{S}\, \mathsf{O})$, ..., are types. All the elements of this family are built using vnil and vcons, thus sharing the same structure. However, different types in the family have different elements. For example, $\mathsf{vec}\, A\, \mathsf{O}$ has only one (canonical) element, vnil. The elements of $\mathsf{vec}\, A\, (\mathsf{S}\, \mathsf{O})$ are of the form $\mathsf{vcons}\, \mathsf{O}\, x\, \mathsf{vnil}$, i.e., lists of length 1. In general, $\mathsf{vec}\, A\, n$ contains only lists of length n whose elements are in A.

Inductive families are useful for defining propositional types. For example, we can define an inductive family to represent if a natural number is even:

$$\begin{aligned}\mathsf{Ind}(\mathsf{even} : \mathsf{nat} \to \mathsf{Prop} := \ & \mathsf{even\_O} : \mathsf{even}\, \mathsf{O}, \\ & \mathsf{even\_SS} : \Pi(n : \mathsf{nat}).\mathsf{even}\, n \to \mathsf{even}\, (\mathsf{S}\, (\mathsf{S}\, n)))\end{aligned}$$

In the following we concentrate on computational types, although the development equally applies to propositional types like even.

Going back to the example of vectors, note that both lists and vectors are equivalent, in the sense that it is possible to define a bijection between objects of both types. However,

vectors have more information on their types. For example, the tail function on vectors can be given a more precise type

$$\mathsf{vtail} : \Pi(n : \mathsf{nat}).\mathsf{vec}\, A\, (\mathsf{S}\, n) \to \mathsf{vec}\, A\, n$$

thus ensuring that it can only be applied to non-empty vectors. If we define vtail by

$$\mathsf{vtail}\, n\, (\mathsf{vcons}\, n\, x\, xs) = xs$$

this definition should be total. Note that the case vnil does not need to be considered, since its type cannot be $\mathsf{vec}\, A\, (\mathsf{S}\, n)$. Furthermore, the function vtail has two arguments, the first is a natural number n and the second a vector. In the case vcons, the first argument should match the first argument of vtail, which is why we have a non-left-linear pattern.

Using inductive families to write clear and concise pattern-matching definitions is one of the main arguments in favor of dependently-typed programming [8]. The problem now becomes how to check that such definitions are actually programs, i.e., are total and terminating. In a seminal paper, Coquand [30] proposes an algorithm for this. It allows to eliminate impossible cases (like vnil above) as well as structural recursion. The algorithm was implemented in the ALF theorem prover [57].

Case analysis in CIC. In CIC, for a case analysis construction to be accepted, all cases should be considered. Let us consider the definition of vtail. The naive solution is to write $\mathsf{vtail}\, n\, v$ as

$$\mathsf{case}\ v\ \mathsf{of}\ \mathsf{vnil} \Rightarrow ?\ |\ \mathsf{vcons}\ k\ x\ t \Rightarrow t\ .$$

There are two problems with this definition. The first is that we need to complete the vnil branch with a term explicitly ruling out this case. The second is that the body of the vcons branch is not well-typed, since we are supposed to return a term of type $\mathsf{vec}\, n$, while t has type $\mathsf{vec}\, k$. Let us see how to solve them.

For the first problem, it should be possible to reason by absurdity: if v is a non-empty vector (as evidenced by its type), it cannot be vnil. More specifically, we reason on the indices of the inductive families, and the fact that the indices can determine which constructors were used to build the term (the inversion principle). In this case, v has type $\mathsf{vec}\, A\, (\mathsf{S}\, n)$, while vnil has type $\mathsf{vec}\, A\, \mathsf{O}$. Since distinct constructors build distinct objects (the "no confusion" property), we can prove that $\mathsf{O} \neq \mathsf{S}\, n$, and, as a consequence, v cannot reduce to vnil. This is translated to the definition of vtail by generalizing the type of each branch to include a proof of equality between the indices. The definition of vtail looks something like

$$\begin{aligned}\mathsf{case}\ v\ \mathsf{of}\ &|\ \mathsf{vnil} \Rightarrow \lambda(H : \mathsf{O} = \mathsf{S}\, n).\ \textit{here a proof of contradiction from } H\\ &|\ \mathsf{vcons}\, k\, x\, t \Rightarrow \lambda(H : \mathsf{S}\, k = \mathsf{S}\, n).t,\end{aligned}$$

where, in the vnil branch, we reason by absurdity from the hypothesis H.

We have solved the first problem, but we still suffer the second. Luckily, the same generalization argument used for the vnil branch provides a way out. Note that, in the vcons branch, we now have a new hypothesis H of type $\mathsf{S}\, k = \mathsf{S}\, n$. From it, we can prove that $k = n$, since the constructor S is injective (again, the no-confusion property). Then, we can use the obtained equality to build, from t, a term of type $\mathsf{vec}\, n$. In the end, the body of this branch is a term built from H and t that *changes* the type of t from $\mathsf{vec}\, A\, k$ to $\mathsf{vec}\, A\, n$.

This solves both problems, but the type of the function obtained is $\mathsf{S}\,n = \mathsf{S}\,n \to \mathsf{vec}\,A\,n$, which is not the desired one yet. So, all we need to do is just to apply the function to a trivial proof of equality for $\mathsf{S}\,n = \mathsf{S}\,n$.

It is important to notice that this function, as defined above, still has the desired computational behavior: given a term $v = \mathsf{vcons}\,n\,h\,t$, we have $\mathsf{vtail}\,n\,v \to^+ t$. In particular, in the body of the vcons branch, the extra equational burden necessary to change the type of t collapses to the identity. However, the definition is clouded with equational reasoning expressions that do not relate to the computational behavior of the function, but are necessary to convince the typechecker that the function does not compromise the type correctness of the system.

We propose an alternative case rule that allows to automatically omit impossible cases and propagate inversion constraints for possible cases. Our work is directly based on the dependently-programming languages Epigram [60] and Agda [69], two modern implementations of Coquand's proposal. We summarize this contribution in the next section.

Let us illustrate how to write the vtail function using the alternative case rule. One possibility is to define it

$$\mathsf{case}\ v\ \mathsf{in}\ [n_0]\,\mathsf{vec}(A, \mathsf{S}\,n_0)\ \mathsf{of}\ \mathsf{vcons}\,(k := n)\,x\,t \Rightarrow t$$

The intuition behind this definition is that, since v has type $\mathsf{vec}\,A\,(\mathsf{S}\,n)$, we can restrict the analysis to cases in the family of types $\mathsf{vec}\,A\,(\mathsf{S}\,n_0)$, for $n_0 : \mathsf{nat}$. Then, the case vnil can be omitted, since no term in this family can be constructed using vnil. Furthermore, in the case vcons, the first argument must be n, which gives t the correct type — there is no need to change its type.

1.4 Contribution

We propose two extensions of CIC that aim to improve the use of elimination rules and overcome the limitations mentioned in Sections 1.2.2 and 1.3. The first extension concerns termination of recursive functions, while the second concerns pattern matching.

Our long-term objective is that these extensions could be implemented in the kernel of Coq. A main concern is then ensuring the preservation of metatheoretical properties such as confluence, subject reduction (SR) and, most important, logical consistency. We prove these properties for both extensions.

Another concern is that these extensions could be implemented efficiently. As we noted before, one important aspect to consider in the implementation of a type-checker for type theory is the reduction engine, which is the basis for the conversion test. The possibility of implementing an efficient reduction engine design choices of the system we present in this work.

Termination of recursive functions. We propose the use of type-based termination as a mechanism for checking termination of recursive functions in CIC. We are not the first to consider extensions of dependent type theories with type-based termination. As we mentioned, type-based termination has a long history that can be traced back to the work of Mendler [64]. In this section, we only mention related works that are necessary to put our contribution in context. For further references on termination of recursive functions, and type-based termination in particular, see Sect. 2.7.

Our extension of CIC with type-based termination follows a line of research that started with an extension of the simply-typed λ-calculus with sized types by Barthe et al. [17], described in detail in Frade's thesis [35]. The extension, called $\lambda\hat{}$, enjoys several desired metatheoretical properties including subject reduction and SN.

An extension of system F, called $\mathsf{F}\hat{}$, is introduced in Barthe et al. [18]. The main feature of $\mathsf{F}\hat{}$ is a size inference algorithm, meaning that sizes are completely transparent to the user. SR and SN are also proved (see [20] for details).

Furthermore, based on the same ideas developed in $\lambda\hat{}$ and $\mathsf{F}\hat{}$, Barthe et al. [19] introduce an extension of CIC with a type-based termination mechanism. This extension, called CIC$\hat{}$, enjoys SR and size inference as in $\mathsf{F}\hat{}$. However, Logical Consistency is proved using a conjecture stating Strong Normalization.

While CIC$\hat{}$ seems a good candidate for replacing the core theory of Coq, the lack of a proof of Logical Consistency does not give enough assurance for a sound implementation. In this work we try to remedy this situation.

We present an extension of CIC with a type-based termination, called CIC$\hat{}_{-}$, based on the approach of CIC$\hat{}$ (Chapter 2). We prove that the extension, enjoys several metatheoretical properties including SR (Chapter 3). Our main contribution is the definition of a model of CIC$\hat{}_{-}$ based on Λ-sets [7,63]. As a consequence of soundness of the model, we can prove SN and LC (Chapter 4).

This contribution extends on previous work with Benjamin Grégoire [44].

Pattern matching. Building on previous work by Coquand [30] and the dependently-typed programming languages Epigram [60,62] and Agda [69], we propose an extension of CIC with a new pattern matching rule.

The new rule, which allows the user to write more direct and more efficient functions, combines explicit restriction of pattern-matching to inductive subfamilies, and translation of unification constraints into local definitions of the typing context (Chapter 6). We show that this extension satisfies metatheoretical properties such as SR. We also prove LC by a translation towards an extension of CIC$\hat{}_{-}$.

This contribution is based on previous work with Bruno Barras, Pierre Corbineau, Benjamin Grégoire, and Hugo Herbelin [16].

1.5 Overview of the Rest of the Thesis

The rest of the thesis is organized as follows. In Chapter 2 we introduce our calculus CIC$\hat{}_{-}$, an extension of CIC with a type-based termination mechanism in the style of CIC$\hat{}$. We give a complete formal presentation explaining its syntax and typing rules. The features of CIC$\hat{}_{-}$ are illustrated through a series of examples.

In Chapter 3 we show some metatheoretical properties of CIC$\hat{}_{-}$ including SR. Due to some technical difficulties related to impredicativity, we cannot prove SN directly on CIC$\hat{}_{-}$. We therefore introduce an annotated version of CIC$\hat{}_{-}$, called ECIC$\hat{}_{-}$, where applications and abstractions are fully annotated with domain and codomain. We prove that both presentations are equivalent and, in particular, we show that SN of CIC$\hat{}_{-}$ can be derived from SN of ECIC$\hat{}_{-}$.

The proof of SN of ECIC$\hat{}_{-}$ is developed in Chapter 4, the main contribution of this work. We use a Λ-set model based on the work of Altenkirch [7] and Melliès and Werner [63],

extended to cope with sized types.

In Chapter 5 we consider some possible extensions of $\mathrm{CIC}\widehat{\ }_{-}$. Namely, universe inclusion, equality, and coinductive types. For each extension, we sketch how to adapt the metatheory of Chapters 3 and 4. In particular, we discuss how to adapt the Λ-set model.

In Chapter 6 we develop an extension of $\mathrm{CIC}\widehat{\ }_{-}$ with a pattern-matching mechanism in the style of dependently-typed programming languages such as Epigram and Agda. We prove that the extension satisfies desired metatheoretical properties such as SR and show LC by a translation to an extension of $\mathrm{CIC}\widehat{\ }_{-}$.

Finally, we give some concluding remarks in Chapter 7.

Chapter 2

$\text{CIC}^{\widehat{}}_{-}$

$\text{CIC}^{\widehat{}}_{-}$ is an extension of the Calculus of Inductive Constructions (CIC) with a type-based termination mechanism. In this chapter we present the syntax and typing rules of $\text{CIC}^{\widehat{}}_{-}$. We illustrate its features through a series of examples.

2.1 Introduction

Before presenting the formal definition of the syntax and typing rules of $\text{CIC}^{\widehat{}}_{-}$, we give a short introduction describing the design choices we made. $\text{CIC}^{\widehat{}}_{-}$ can be seen as a relatively small restriction of $\text{CIC}^{\widehat{}}$ [19]. Along this chapter, we make clear the distinction between both systems.

Our long-term objective is to implement a type-based termination mechanism in Coq. This affects the design of both $\text{CIC}^{\widehat{}}$ and $\text{CIC}^{\widehat{}}_{-}$, as it is not desirable to make strong changes with respect to current implementations of Coq. First, sized types, the main component of the type-based termination approach, should be transparent for the user. Sized types are used by the type-checking algorithm to check termination. The user should not need to deal size with information (as far as possible). Second, it should be possible to give an efficient implementation of sized types. In particular, sizes should not be involved in computation (i.e., reduction). This way, reduction could be implemented as efficient as in CIC.

We present the main features of $\text{CIC}^{\widehat{}}_{-}$in the following.

Sized types. Sizes (or stages) define the annotations attached to inductive types. Following $\text{CIC}^{\widehat{}}$ and other approaches (e.g., [1]), size annotations in $\text{CIC}^{\widehat{}}_{-}$ are taken from a simple grammar.

Definition 2.1 (Stages)**.** *The syntax of stages is given by the following grammar, where $\mathcal{V}_{\mathcal{S}}$ denotes a denumerable set of stage variables.*

$$\mathcal{S} ::= \mathcal{V}_{\mathcal{S}} \mid \widehat{\mathcal{S}} \mid \infty \ .$$

We use $\imath, \jmath, \kappa$ to denote stage variables, and s, r to denote stages. The base of a stage expression is defined by $\lfloor \imath \rfloor = \imath$ and $\lfloor \widehat{s} \rfloor = \lfloor s \rfloor$ (the base of a stage containing ∞ is not defined).

More complex stage algebras can be considered. For example, including more operators on stages such as addition, maximum, minimum, etc. The advantage of using a more complex

stage algebra is the possibility of expressing more information of a function in the type. For example, the addition of natural numbers could be given type:

$$\mathsf{nat}^s \to \mathsf{nat}^r \to \mathsf{nat}^{s+r}$$

In turn, it would be possible to accept more functions as terminating, compared to the simple algebra we use. See Sect. 2.7 for some references. On the other hand, a richer size algebra would complicate the definition of the typing system. In particular, it would be more complicated to keep the size information hidden from the user.

One advantage of using this small size algebra is that it is possible to infer size information. As is shown in [19] for CIC$\hat{}$, there is a size inference algorithm that can reconstruct size information from a term without any size annotations. Given the close relationship between CIC$\hat{\sqsubseteq}$ and CIC$\hat{}$ (cf. Sect. 2.6), it is reasonable to believe that this algorithm can be adapted to our case.

Sizes as bounds. Stages define the size annotations of inductive types. Intuitively, they represent a bound on the number of constructors that can be applied. Given the interpretation of sizes as bounds, it is necessary to have a subtyping relation. This relation is derived from an order relation on stages.

Definition 2.2 (Substage)**.** *The relation s is a substage of s', written $s \sqsubseteq s'$, is defined by the rules:*

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

The complete definition of subtyping is given in Sect. 2.3.1. To illustrate the intuitive meaning of the subtyping relation, we can mention some instances that derive from the definition of the substage relation. Since $s \leq \infty$, we have $\mathsf{nat}^s \leq \mathsf{nat}^\infty$, for any stage s. In effect, nat^∞ represents the full type of natural numbers. Note that size variables are not comparable, therefore we have $\mathsf{nat}^\imath \not\leq \mathsf{nat}^\jmath$, when $\imath \neq \jmath$. Although these examples only involve nat, similar rules apply to all inductive types.

Note that the interpretation of stages as a bound on the size of elements is not strictly necessary in the type-based termination approach. A different interpretation, considered by Blanqui and Riba [26], and also Xi [85], consists in interpreting nat^α as the set of terms whose size is exactly α. Our interpretation can be recovered by the use of existential and constraints: $\exists\alpha(\alpha \leq s).\mathsf{nat}^\alpha$. We do not follow this approach, since the introduction of constraints would complicate the typing system.

Implicit stages declaration. The definition of stages and terms is separated. Stages are used to annotate types, but are not themselves terms. In particular, there are no constructions to introduce new stage variables or manipulate stages. Instead, stage variables are implicitly declared globally. For example, consider the following typing judgment:

$$(f : \mathsf{nat}^\imath \to \mathsf{nat}^\jmath)(x : \mathsf{nat}^\imath) \vdash f\,x : \mathsf{nat}^\jmath$$

Size variables $\imath$ and $\jmath$ are implicitly declared.

Abel [1,3] considers a different approach where sizes are declared as part of the language. Since we want to keep sizes hidden from the user, as much as possible, we do not consider this approach. See Sect. 2.7 for a detailed comparison between both alternatives.

However, the naive use of implicit stages does not satisfy Subject Reduction. Consider the term $M = \mathsf{fix}\ f : \mathsf{nat}^{\imath} \to \mathsf{nat}^{\infty} := \lambda x : \mathsf{nat}^{\widehat{\imath}}.x$. Using the simplified typing rule for fixpoint given in the previous chapter, M can be given the type $\mathsf{nat}^{s} \to \mathsf{nat}^{\infty}$, for any s. With the obvious typing rule for application, the following is a valid typing judgment:

$$(y : \mathsf{nat}^{\jmath}) \vdash M\,(\mathsf{S}\,y) : \mathsf{nat}^{\infty}$$

where $\mathsf{S}\,y$ has type $\mathsf{nat}^{\widehat{\jmath}}$. This term reduces to $(\lambda x : \mathsf{nat}^{\widehat{\imath}}.x)\,(\mathsf{S}\,y)$ which is not well-typed in the context $(y : \mathsf{nat}^{\jmath})$.

One solution could be to replace the stage variable in the body of fixpoints when reducing. Then, we would have the reduction:

$$M\,(\mathsf{S}\,y) = (\mathsf{fix}\ f : \mathsf{nat}^{\imath} \to \mathsf{nat}^{\infty} := \lambda x : \mathsf{nat}^{\widehat{\imath}}.x)\,(\mathsf{S}\,y) \to (\lambda x : \mathsf{nat}^{\widehat{\widehat{\jmath}}}.x.)\,(\mathsf{S}\,y)$$

where we replace $\imath$ with $\widehat{\jmath}$, since $\mathsf{S}\,y$ has type $\mathsf{nat}^{\widehat{\jmath}}$. This means that reduction and typing would depend on each other, since the size information of the argument of the fixpoint is obtained through its type. This approach would greatly complicate the metatheory, so we do not consider it. Another possibility could be to explicitly state the size information of the argument, as in $M\,(\widehat{\jmath})\,(\mathsf{S}\,y)$. This would mean that the user would have to manipulate sizes.

But there is another reason to not consider this approach: efficiency. In the above example, sizes are involved in the reduction, since a size substitution is needed to maintain Subject Reduction. Sizes are used to ensure termination, but should not be needed to compute with terms. The solution proposed in CIC$\widehat{\ }$, which we adapt here, is to combine *annotated terms* and *erased terms*. The latter class include terms where the size information is erased, while the former class include terms where types are annotated with sizes.

Erased terms are used in places where a type is expected. For example, in the case of abstractions. In CIC$\widehat{\ }_{-}$, an abstraction has the form $\lambda x : T^{\circ}.M$, where T° is an erased term. The typing rule for abstraction is

$$\frac{\Gamma(x : T) \vdash M : U}{\Gamma \vdash \lambda x : |T|.M : \Pi x : T.U}$$

where $|T|$ is an erased term obtained from T by removing all size annotations. Note that sizes are used to check the type of M but are removed from the term. For example, a valid typing judgment for the identity is $\vdash \lambda x : \mathsf{nat}.x : \mathsf{nat}^{s} \to \mathsf{nat}^{s}$, for any size s.

Let us go back to the example that motivated this discussion. Using erased terms we write $M = \mathsf{fix}\ f : \mathsf{nat} \to \mathsf{nat} := \lambda x : \mathsf{nat}.x$. Then

$$(y : \mathsf{nat}^{\jmath}) \vdash M\,(\mathsf{S}\,y) : \mathsf{nat}^{\widehat{\jmath}}$$

Note that $M\,(\mathsf{S}\,y) \to (\lambda x : \mathsf{nat}.x)\,(\mathsf{S}\,y)$. This last term can also be given type $\mathsf{nat}^{\widehat{\jmath}}$, so Subject Reduction is preserved. The result is that sizes are not involved in the computation, hence no size substitution is needed to reduce a fixpoint.

As we show in the next section, the reduction rules of CIC$\widehat{\ }_{-}$ are exactly the same as in CIC. In particular, sizes are not involved in the sense that reduction does not use size information. This means that reduction in CIC$\widehat{\ }_{-}$ could be implemented as efficiently as in CIC.

2.2 Syntax of CIC$\hat{}$

We begin by describing the constructions of CIC$\hat{}$. The language of CIC$\hat{}$ is based on CIC. All the constructions are taken from CIC, with some differences related to the use of sized types and implicit stages.

2.2.1 Basic Terms

As is the case with CIC and dependent type theory in general, there are no separate syntactic categories for terms and types. We describe the syntax of terms, in two parts: basic terms (including applications, abstractions, etc.) and inductive terms (including constructors and destructors of inductive types, i.e., case analysis and fixpoint functions).

In CIC$\hat{}$, inductive types are decorated with size information. As we mentioned in the previous section, size information is erased from some terms in order to obtain an efficient reduction and maintain subject reduction. We present the syntax of terms parameterized by the size expressions that inductive types carry, to account for different classes of terms. Three classes are of interest for us:

- bare terms (no annotations),
- position terms (either no annotation or a $\star$ used to indicate recursive arguments in fixpoint definitions), and
- sized terms (annotated with stage expressions).

We begin by describing the syntax of basic terms. This includes the constructions of the language that do not directly refer to inductive types.

Definition 2.3 (Basic terms)**.** *The generic set of basic terms over the set a is defined by the grammar:*

$$\begin{array}{rcll} \mathcal{T}[a] & ::= & \mathcal{V} & \textit{(term variable)} \\ & | & \mathsf{Prop} & \textit{(universe of propositions)} \\ & | & \mathsf{Type}_i \quad (i \geq 0) & \textit{(universes of computations)} \\ & | & \lambda x : \mathcal{T}^{\circ}.\mathcal{T}[a] & \textit{(abstraction)} \\ & | & \mathcal{T}[a]\,\mathcal{T}[a] & \textit{(application)} \\ & | & \Pi x : \mathcal{T}[a].\mathcal{T}[a] & \textit{(product)} \end{array}$$

where $\mathcal{V}$ is a denumerable set of term variables. The set of bare terms, position terms and sized terms are defined by $\mathcal{T}^{\circ} ::= \mathcal{T}[\epsilon]$, $\mathcal{T}^{\star} ::= \mathcal{T}[\{\epsilon, \star\}]$, and $\mathcal{T} ::= \mathcal{T}[\mathcal{S}]$, respectively. We also consider the class of sized terms with no size variables: $\mathcal{T}^{\infty} ::= \mathcal{T}[\infty]$.

The grammar of basic terms includes the constructions of the Calculus of Constructions with Universes. Note that there is no difference between classes of terms. The difference will appear once we extend this grammar with constructions related to inductive types.

The terms Prop and Type_i, for $i \geq 0$, are called *sorts* or *universes*. We use $\mathcal{U}$ to denote the set of sorts—i.e., $\mathcal{U} = \{\mathsf{Prop}\} \cup \{\mathsf{Type}_i : i \geq 0\}$.

We use M, N, P, T, U, C, a, b, p, q, t, to denote terms and x, y, z to denote variables. Bare terms are denoted with a superscript $\circ$ and position terms with a superscript $\star$, as in M° and $M^{\star}$. In $\Pi x : M.N$ and $\lambda x : M^{\circ}.N$, the variable x is bounded in N. We write $\mathrm{FV}(M)$ to denote the set of free *term variables* in M and $M\,[x := N]$ to denote the substitution of the free occurrences of x in M by N; we omit their definitions, since they are standard. Note that the substitution needs to erase the size annotations when replacing inside an erased term (e.g., the type of abstractions). To deal with sized and erased terms we use an erasure

function $|.| : \mathcal{T} \to \mathcal{T}^\circ$; again, we omit the definition. We write $M \to N$ as a synonym of $\Pi x : M.N$ if $x \notin \mathrm{FV}(N)$. We denote α-convertibility with $\equiv$.

We write $\vec{X}$ to denote a sequence of X and $\#\vec{X}$ to denote the length of the sequence $\vec{X}$. Alternatively, we write $\langle X_i \rangle_{i=1..n}$ to denote the sequence $X_1 X_2 \dots X_n$. We write $\langle X_i \rangle_{i=m..n}$ for the subsequence $X_m X_{m+1} \dots X_n$. Sometimes we write simply $\langle X_i \rangle_i$ when the index is not important, or known from the context. We denote the empty sequence with ε. We sometimes write commas between the elements of a sequence to avoid confusion with application, as in $T_1, T_2, \dots, T_n$ and $T_1, \vec{T}$.

We define contexts which are, basically, a sequence of declarations of the form $(x : T)$.

Definition 2.4 (Contexts)**.** *A* context *is a (finite) sequence of declarations of the form* $(x : T)$*, where* x *is a variable and* T *a (sized) term. The empty context is denoted by* $[]$*. Similarly, a* bare context *is a (finite) sequence of declarations of the form* $(x : T^\circ)$*, where* T° *is an erased term.*

We use Γ, Δ, Θ, ... to denote contexts. We write $\#\Gamma$ to denote the length of context Γ, $\mathrm{dom}\,(\Gamma)$ to denote the set of variables declared in Γ, $(x : T) \in \Gamma$ to mean that Γ contains the declaration $(x : T)$, and $\Gamma(x)$ with $x \in \mathrm{dom}\,(\Gamma)$ to mean the unique T (up to α-convertibility) such that $(x : T) \in \mathrm{dom}\,(\Gamma)$. We write $\mathrm{FV}(\Gamma)$ to denote the union of the sets of free term variables of all types declared in Γ. We extend the function FV to tuples: we write $\mathrm{FV}(X_1, \dots, X_n)$ to mean $\mathrm{FV}(X_1) \cup \dots \cup \mathrm{FV}(X_n)$, where X_i is either a term or a context for $i \in \{1, \dots, n\}$.

Given a context Γ and a sequence of variables $\vec{x}$ such that $\#\vec{x} = \#\Gamma$, we define the operation $(\vec{x} : \Gamma)$ that renames the declared variables of Γ with $\vec{x}$. It is defined by $(\varepsilon : []) = []$ and $(x_1\, \vec{x} : (y : T)\Gamma) = (x_1 : T)(\vec{x} : \Gamma\,[y := x_1])$.

We extend the syntax of abstractions and dependent products to deal with several variables using contexts. Given a context $\Gamma \equiv (x_1 : T_1) \dots (x_n : T_n)$ we write $\lambda\Gamma.M$ and $\Pi\Gamma.M$ to mean the terms $\lambda x_1 : T_1. \dots \lambda x_n : T_n.M$ and $\Pi x_1 : T_1. \dots \Pi x_n : T_n.M$ respectively.

2.2.2 Inductive types

We now extend the syntax of terms to include inductive types (and inductive families). Inductive types are defined by a signature, which is a (finite) sequence of inductive type declarations. Each declaration takes the form [1]

$$\mathsf{Ind}(I[\Gamma] : A := \langle C_i : T_i \rangle_i),$$

where I is the inductive type being defined, Γ is the context of parameters, A is the type of I, and $\langle C_i : T_i \rangle_i$ are the *constructors* of I with their corresponding types. In the types of constructors, $T_1, \dots, T_n$, a special variable $\mathcal{X}$ refers to I applied to the parameters. Let us illustrate with some examples. Natural numbers, defined by Peano axioms, are introduced by the declaration

$$\mathsf{Ind}(\mathsf{nat} : \mathsf{Type}_0 := (\mathsf{O} : \mathcal{X})(\mathsf{S} : \mathcal{X} \to \mathcal{X}))\ .$$

This means that we declare an inductive type nat whose type is itself Type_0, with two constructors O and S of types nat and $\mathsf{nat} \to \mathsf{nat}$ respectively. A slightly more complicated

1. We will extend this definition with polarities in order to allow more subtyping rules. They are not necessary now, so we omit them for the moment.

example is given by vectors (lists indexed by their length). They are defined by

$$\mathsf{Ind}(\mathsf{vec}[A : \mathsf{Type}_0] : \mathsf{nat} \to \mathsf{Type}_0 := (\mathsf{vnil} : \mathcal{X}\,\mathsf{O}) \\ (\mathsf{vcons} : \Pi(x : \mathsf{nat}).A \to \mathcal{X}\,n \to \mathcal{X}\,(\mathsf{S}(n)))) \ .$$

Given a type A, $\mathsf{vec}\,(A, N)$ is an inductive type, for N of type nat, where $\mathsf{vnil}(A)$ has type $\mathsf{vec}\,(A, \mathsf{O})$ and $\mathsf{vcons}(A, N\,M\,V)$ has type $\mathsf{vec}\,(A, \mathsf{S}(N))$ if N, M and V have type nat, A and $\mathsf{vec}\,(A, N)$ respectively. Constructors are always fully applied and we separate the parameters from the actual arguments.

Note that not all declarations of the form above are accepted. Besides the restriction to well-typed declarations, a valid declaration satisfies several positivity conditions ensuring monotonicity of the inductive type. We explain this restrictions in Sect. 2.3.

We extend the syntax of terms to include inductive types and constructors. We also need a way to analyze and operate with inductive types. Hence we extend the syntax with *destructors*. The destructors are embodied in two constructions of the language: case analysis (pattern matching) and recursive functions (fixpoint).

Definition 2.5 (Terms (continued))**.** *The grammar of terms over the set a is extended with the following rules:*

$$\begin{array}{rcll} T & ::= & \ldots & \\ & | & \mathcal{I}^a\left(\vec{\mathcal{T}[a]}, \vec{\mathcal{T}[a]}\right) & \textit{(inductive type)} \\ & | & \mathcal{C}(\vec{\mathcal{T}^\circ}, \vec{\mathcal{T}[a]}) & \textit{(constructor)} \\ & | & \mathsf{case}_{\mathcal{T}^\circ}\ \mathcal{V} := \mathcal{T}[a]\ \mathsf{in}\ \mathcal{I}\left(\vec{\mathcal{T}^\circ}, \vec{\mathcal{V}}\right)\ \mathsf{of}\ \langle \mathcal{C} \Rightarrow \mathcal{T}[a]\rangle & \textit{(case analysis)} \\ & | & \mathsf{fix}_n\ \mathcal{V} : \mathcal{T}^\star := \mathcal{T}[a] & \textit{(fixpoint)} \end{array}$$

Names of inductive types are taken from a denumerable set of names $\mathcal{I}$ and names of constructors are taken from a denumerable set of names $\mathcal{C}$. We assume that $\mathcal{V}_\mathcal{S}$ (size variables), $\mathcal{V}$ (term variables), $\mathcal{I}$ and $\mathcal{C}$ are mutually disjoint.

We define another erasure function to deal with position terms. The function $|.|^\imath : \mathcal{T} \to \mathcal{T}^\star$ replaces all stage annotations s with $\star$ if $\lfloor s \rfloor = \imath$, or by ϵ otherwise. (We omit the definition by induction on the structure of terms.) Given a term M, we write M^∞ to denote the term M where all size annotations are replaced with ∞, and $\mathrm{SV}(M)$ to denote the set of *stage variables* appearing in M. (Note that $\mathrm{SV}(M^\infty) = \emptyset$.)

We briefly explain the new constructions. A more detailed explanation is given in Sect. 2.3. Inductive types are decorated with size information and applied to parameters and arguments. We write I^s for $I^s[]$, for inductive types I with no parameters. Constructor terms are fully applied to the parameters and the proper arguments. Note that the parameters in constructors are formed by erased terms. In a term of the form

$$\mathsf{case}_{T^\circ}\ x := M\ \mathsf{in}\ I\left(\vec{p^\circ}, \vec{y}\right)\ \mathsf{of}\ \{C \Rightarrow N\},$$

M is the argument, T° is the return type, $I\left(\vec{p^\circ}, \vec{y}\right)$ is a pattern describing the type of M, and $\{C \Rightarrow N\}$ are the branches. T° and $I\left(\vec{p^\circ}, \vec{y}\right)$ are related to the typing rule and will be explained later. M has type I (applied to parameters and arguments), and for each constructor C of I there is a branch of the form $C \Rightarrow N$ that represents the value of the whole expression if M is a term headed by constructor C. This is the usual case analysis

construction found in Haskell or ML, but the typing rule is more complicated due to the presence of dependent types.

The last construction concerns the definition of recursive functions. A term of the form

$$\mathsf{fix}_n\ f : T^\star := M,$$

defines a recursive function f (bound in M) of type based on $T^\star$. The actual type of this function, let us call it T', satisfies $|T'|^\imath \equiv T^\star$, for some size variable $\imath$. The type $T^\star$ should satisfy several restrictions in order to obtain a terminating function. We define them properly in Sect. 2.3.

2.2.3 Reduction

The reduction rules define how to compute with terms. Before defining the actual rules, we introduce some general notation. Let R be a relation on terms. We write $\to_R$ for the compatible closure of R, $\leftarrow_R$ for the inverse of $\to_R$, $\to_R^*$ for the reflexive-transitive closure, $\approx_R$ for the equivalence closure (reflexive-transitive-symmetric closure), and $\downarrow_R$ for the associated joinability relation (i.e., $M \downarrow_R N$ iff $M \to_R^* P \leftarrow_R^* N$ for some P). We omit the definitions of the different closure operators, since they are standard.

Definition 2.6 (Reduction relation)**.** *We consider three reductions on terms: β-reduction (for function application), ι-reduction (for case expression), and μ-reduction (for fixpoint expressions). They are defined by the following rules:*

$$\begin{array}{rcl} (\lambda x : T^\circ.M)N & \beta & M\,[x := N] \\ \mathsf{case}_{T^\circ}\ x := C_j(\vec{q^\circ}, \vec{a})\ \mathsf{in}\ I\,(\vec{p^\circ}, \vec{y})\ \mathsf{of}\ \{C_i \Rightarrow N_i\} & \iota & N_j\,\vec{a} \\ F\,\langle N_i\rangle_{i=1..n-1}\,C(\vec{p^\circ}, \vec{a}) & \mu & M\,[f := F]\ \langle N_i\rangle_{i=1..n-1}\,C(\vec{p^\circ}, \vec{a}) \end{array}$$

where $F \equiv \mathsf{fix}_n\ f : T^\star := M$. We write $\to$ instead of $\to_{\beta\iota\mu}$; similarly for $\to^*$, $\leftarrow$, $\approx$, and $\downarrow$.

Note that the n in the fixpoint construction denotes the recursive argument. Only when applied to n arguments and the n-th argument is in constructor form we perform the reduction (by unfolding the recursive function in the body). Unrestricted unfolding of fixpoint immediately breaks Strong Normalization.

Reduction does not depend on size information; in the reduction rules there is no mention of stages. In particular, for fixpoint reduction, there is no size substitution involved. This is a design choice that, as we mentioned in the previous chapter, makes reduction more efficient. On the other hand, it is necessary to use erased terms to maintain Subject Reduction (cf. Sect. 2.1).

The reduction relation is confluent, as stated in the next lemma.

Lemma 2.7 (Confluence)**.** *If $M \approx N$, then $M \downarrow N$.*

Proof. Confluence follows easily from the diamond property for $\to^*$: if $M \to^* N_1$ and $M \to^* N_2$, then $N_1 \downarrow N_2$. The diamond property is proved using the Tait and Martin-Löf's method of defining parallel reduction. We briefly sketch how it works. The first step is to define a reduction called one-step parallel reduction (OSPR); we denote OSPR with $\Rightarrow$. The main rule of OSPR is

$$\frac{M_1 \Rightarrow M_2 \qquad N_1 \Rightarrow N_2}{(\lambda x : T^\circ.M_1)\,N_1 \Rightarrow M_2\,[x := N_2]}$$

OSPR is reflexive by definition. It is not difficult to see that OSPR satisfies the diamond property. Then, the diamond property for $\to^*$ follows from $\to \subseteq \Rightarrow \subseteq \to^*$.

We complete the definition of OSPR. The relevant rules are the application rule given above plus rules to take care of μ-reduction and ι-reduction:

$$\frac{F_1 \equiv \mathsf{fix}_n\ f : T_1^\star := M_1 \qquad F_2 \equiv \mathsf{fix}_n\ f : T_2^\star := M_2 \qquad T_1^\star \Rightarrow T_2^\star \qquad M_1 \Rightarrow M_2 \qquad \langle N_{1,i}\rangle_{i=1..n-1} \Rightarrow \langle N_{2,i}\rangle_{i=1..n-1} \qquad \vec{p_1^\circ} \Rightarrow \vec{p_2^\circ} \qquad \vec{a_1} \Rightarrow \vec{a_2}}{F_1\,\langle N_{1,i}\rangle_{i=1..n-1}\,(C_j(\vec{p_1^\circ}, \vec{a_1})) \Rightarrow M_2\,[f := F_2]\,\langle N_{2,i}\rangle_{i=1..n-1}\,(C_j(\vec{p_2^\circ}, \vec{a_2}))}$$

$$\frac{\vec{a_1} \Rightarrow \vec{a_2} \qquad N_{1,j} \Rightarrow N_{2,j}}{\mathsf{case}_{P^\circ}\ x := C_j(\vec{q_1^\circ}, \vec{a_1})\ \mathsf{in}\ I\left(\vec{p^\circ}, \vec{a}\right)\ \mathsf{of}\ \langle C_i \Rightarrow N_{1,i}\rangle_i \Rightarrow N_{2,j}\,\vec{a_2}}$$

The definition is completed with compatible closure rules. □

2.3 Typing rules

The typing rules define which terms are considered valid. We consider three typing judgments:

- $\mathsf{WF}(\Sigma)$ means that the signature Σ is well formed;
- $\mathsf{WF}_\Sigma(\Gamma)$ means that the context Γ is well formed under the signature Σ;
- $\Sigma; \Gamma \vdash M : T$ means that the term M has type T under context Γ and signature Σ.

They are defined inductively by the typing rules. Note that their definition is mutually recursive: in the last judgment, to check that a term M has type T we need to check that the signature Σ and context Γ are well-formed. Similarly to check that a context or a signature is well-formed, we need to check that their components (terms) are well-typed. For the sake of readability, we usually omit the signature in judgments.

This section is organized as follows. First, we extend the reduction relation defined in the previous section into a subtyping relation. Then, we define the positivity conditions for inductive types and the typing rules for them. Finally, we define the typing rules for contexts and terms.

2.3.1 Subtyping

Recall that the intuitive meaning of an inductive type is a monotone operator. The size annotations represent approximations of these operator (with ∞ representing the least fixed point of the operator). On the syntactic level, this intuition is represented by a subtyping relation, denoted $\leq$, derived from the substage relation (Def. 2.2).

The subtyping relation includes the usual contravariance rule for products. For inductive types, we assume that each inductive definition includes a declaration of *polarities* of parameters. A polarity is either positive, negative, or invariant:

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

For each inductive type I, we assume a vector of polarities, written $I.\vec{\nu}$, denoting the polarities of the parameters of I. Polarities are used to increase the subtyping relation in the case of inductive types, by allowing subtyping to occur in the parameters. For example, $\mathsf{list}^\infty\,(\mathsf{nat}^\imath) \leq \mathsf{list}^\infty\,\left(\widehat{\mathsf{nat}^\imath}\right)$.

$$\text{(st-conv)}\quad \frac{T_1\ R\ T_2}{T_1 \preceq_R T_2} \qquad \text{(st-prod)}\quad \frac{T_2 \preceq_R T_1 \qquad U_1 \preceq_R U_2}{\Pi x : T_1.U_1 \preceq_R \Pi x : T_2.U_2}$$

$$\text{(st-ind)}\quad \frac{s \sqsubseteq s' \qquad \vec{p_1} \preceq_R^{I,\vec{\nu}} \vec{p_2} \qquad \vec{a_1}\ R\ \vec{a_2}}{I^s\,(\vec{p_1}, \vec{a_1}) \preceq_R I^{s'}\,(\vec{p_2}, \vec{a_2})} \qquad \text{(vst-inv)}\quad \frac{T_1\ R\ U_1 \qquad \vec{T} \preceq_R^{\vec{\nu}} \vec{U}}{T_1, \vec{T} \preceq_R^{\circ,\vec{\nu}} U_1, \vec{U}}$$

$$\text{(vst-pos)}\quad \frac{T_1 \preceq_R U_1 \qquad \vec{T} \preceq_R^{\vec{\nu}} \vec{U}}{T_1, \vec{T} \preceq_R^{+,\vec{\nu}} U_1, \vec{U}}$$

$$\text{(vst-neg)}\quad \frac{U_1 \preceq_R T_1 \qquad \vec{T} \preceq_R^{\vec{\nu}} \vec{U}}{T_1, \vec{T} \preceq_R^{-,\vec{\nu}} U_1, \vec{U}} \qquad \text{(vst-conv)}\quad \frac{\vec{T}\ R\ \vec{U}}{\vec{T} \preceq_R^{\emptyset} \vec{U}}$$

$$\text{(st-trans)}\quad \frac{T_1 \preceq_R T_2 \qquad T_2 \preceq_R T_3}{T_1 \preceq_R T_3}$$

Figure 2.1: Subtyping relation

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. Let R be a relation on terms that is stable under substitution of terms and stages. We define the subtyping relation parametrized by R, denoted with $\preceq_R$. The rules are given in Fig. 2.1. The instances of R that we will use are convertibility ($\approx$) and α-equivalence ($\equiv$). We write simply $\leq$ to mean $\preceq_\approx$.

2.3.2 Positivity

We describe two notions of positivity. They are used in the definition of the typing rules for inductive types to ensure the soundness of the subtyping rules defined above.

We define the predicates $\imath$ pos T and $\imath$ neg T denoting that a stage variable $\imath$ appears positively in T and negatively in T, respectively. They are defined inductively by the rules given in Fig. 2.2. The goal of these definitions is to ensure the following property: if $\imath$ pos T (resp. $\imath$ neg T) and $s \sqsubseteq r$, then $T\,[\imath := s] \leq T\,[\imath := r]$ (resp. $T\,[\imath := r] \leq T\,[\imath := s]$). The proof of these properties follows easily by induction on the positivity predicate.

We briefly explain the rules. If a size variable does not appear on a term, it appears both positively and negatively. In the case of products, the rule follows that fact that subtyping is contravariant in the domain and covariant in the codomain. In the case of inductive types we have to check that the positivity is preserved with respect to the polarity declaration. Note that $\imath$ appears positively in $I^\imath$ (i.e., $\imath$ pos $I^\imath$).

We also define a similar notion of positivity of term variable. The predicates x pos T and x neg T state that x appears positively and negatively in T, respectively. The definition follows a similar pattern to that of positivity of stage variable.

The rules are given in Fig. 2.3. The main property we are interested is the following: if x pos T (resp. x neg T) and $U_1 \leq U_2$, then $T\,[x := U_1] \leq T\,[x := U_2]$ (resp. $T\,[x := U_2] \leq T\,[x := U_1]$). The proof follows easily by induction on the positivity predicates.

The definition of positivity is extended in a natural way to sequences of variables and

$$\frac{\imath \notin \mathrm{SV}(T)}{\imath\ \mathsf{pos}\ T} \qquad \frac{\imath \notin \mathrm{SV}(T)}{\imath\ \mathsf{neg}\ T}$$

$$\frac{\imath\ \mathsf{pos}\ T_1 \qquad \imath\ \mathsf{neg}\ T_2}{\imath\ \mathsf{neg}\ \Pi x : T_1.T_2} \qquad \frac{\imath\ \mathsf{neg}\ T_1 \qquad \imath\ \mathsf{pos}\ T_2}{\imath\ \mathsf{pos}\ \Pi x : T_1.T_2}$$

$$\frac{\imath\ \mathsf{pos}^{I.\vec{\nu}}\ \vec{p} \qquad \imath \notin \mathrm{SV}(\vec{a})}{\imath\ \mathsf{pos}\ I^s\ (\vec{p}, \vec{a})} \qquad \frac{\lfloor s \rfloor \neq \imath \qquad \imath\ \mathsf{neg}^{I.\vec{\nu}}\ \vec{p} \qquad \imath \notin \mathrm{SV}(\vec{a})}{\imath\ \mathsf{neg}\ I^s\ (\vec{p}, \vec{a})}$$

$$\frac{}{x\ \mathsf{pos}^{\varepsilon}\ \varepsilon} \qquad \frac{}{x\ \mathsf{neg}^{\vec{\nu}}\ \varepsilon\varepsilon}$$

$$\frac{\imath\ \mathsf{pos}\ T_1 \qquad \imath\ \mathsf{pos}^{\vec{\nu}}\ \vec{T}}{\imath\ \mathsf{pos}^{+,\vec{\nu}}\ T_1, \vec{T}} \qquad \frac{\imath\ \mathsf{neg}\ T_1 \qquad \imath\ \mathsf{pos}^{\vec{\nu}}\ \vec{T}}{\imath\ \mathsf{pos}^{-,\vec{\nu}}\ T_1, \vec{T}} \qquad \frac{\imath \notin \mathrm{SV}(T_1) \qquad \imath\ \mathsf{pos}^{\vec{\nu}}\ \vec{T}}{\imath\ \mathsf{pos}^{\circ,\vec{\nu}}\ T_1, \vec{T}}$$

$$\frac{\imath\ \mathsf{neg}\ T_1 \qquad \imath\ \mathsf{neg}^{\vec{\nu}}\ \vec{T}}{\imath\ \mathsf{neg}^{+,\vec{\nu}}\ T_1, \vec{T}} \qquad \frac{\imath\ \mathsf{pos}\ T_1 \qquad \imath\ \mathsf{neg}^{\vec{\nu}}\ \vec{T}}{\imath\ \mathsf{neg}^{-,\vec{\nu}}\ T_1, \vec{T}} \qquad \frac{\imath \notin \mathrm{SV}(T_1) \qquad \imath\ \mathsf{neg}^{\vec{\nu}}\ \vec{T}}{\imath\ \mathsf{neg}^{\circ,\vec{\nu}}\ T_1, \vec{T}}$$

Figure 2.2: Positivity and negativity of stage variables

$$\frac{x \notin \mathrm{FV}(T) \vee T \equiv x}{x\ \mathsf{pos}\ T} \qquad \frac{x \notin \mathrm{FV}(T)}{x\ \mathsf{neg}\ T}$$

$$\frac{x\ \mathsf{pos}\ T_1 \qquad x\ \mathsf{neg}\ T_2}{x\ \mathsf{neg}\ \Pi y : T_1.T_2} \qquad \frac{x\ \mathsf{neg}\ T_1 \qquad x\ \mathsf{pos}\ T_2}{x\ \mathsf{pos}\ \Pi y : T_1.T_2}$$

$$\frac{x\ \mathsf{pos}^{I.\vec{\nu}}\ \vec{p} \qquad x \notin \mathrm{FV}(\vec{a})}{x\ \mathsf{pos}\ I^s\ (\vec{p}, \vec{a})} \qquad \frac{x\ \mathsf{neg}^{I.\vec{\nu}}\ \vec{p} \qquad x \notin \mathrm{FV}(\vec{a})}{x\ \mathsf{neg}\ I^s\ (\vec{p}, \vec{a})}$$

$$\frac{}{x\ \mathsf{pos}^{\varepsilon}\ \varepsilon} \qquad \frac{}{x\ \mathsf{neg}^{\varepsilon}\ \varepsilon}$$

$$\frac{x\ \mathsf{pos}\ T_1 \qquad x\ \mathsf{pos}^{\vec{\nu}}\ \vec{T}}{x\ \mathsf{pos}^{+,\vec{\nu}}\ T_1, \vec{T}} \qquad \frac{x\ \mathsf{neg}\ T_1 \qquad x\ \mathsf{pos}^{\vec{\nu}}\ \vec{T}}{x\ \mathsf{pos}^{-,\vec{\nu}}\ T_1, \vec{T}} \qquad \frac{x \notin \mathrm{FV}(T_1) \qquad x\ \mathsf{pos}^{\vec{\nu}}\ \vec{T}}{x\ \mathsf{pos}^{\circ,\vec{\nu}}\ T_1, \vec{T}}$$

$$\frac{x\ \mathsf{neg}\ T_1 \qquad x\ \mathsf{neg}^{\vec{\nu}}\ \vec{T}}{x\ \mathsf{neg}^{+,\vec{\nu}}\ T_1, \vec{T}} \qquad \frac{x\ \mathsf{pos}\ T_1 \qquad x\ \mathsf{neg}^{\vec{\nu}}\ \vec{T}}{x\ \mathsf{neg}^{-,\vec{\nu}}\ T_1, \vec{T}} \qquad \frac{x \notin \mathrm{FV}(T_1) \qquad x\ \mathsf{neg}^{\vec{\nu}}\ \vec{T}}{x\ \mathsf{neg}^{\circ,\vec{\nu}}\ T_1, \vec{T}}$$

Figure 2.3: Positivity and negativity of term variables

contexts as follows:

$$\frac{x_1 \;\mathsf{pos}\; T_1 \qquad \vec{x}\;\mathsf{pos}^{\vec{\nu}}\;\Delta}{x_1, \vec{x}\;\mathsf{pos}^{+,\vec{\nu}}\;(y_1 : T_1)\Delta} \qquad \frac{x_1 \;\mathsf{neg}\; T_1 \qquad \vec{x}\;\mathsf{pos}^{\vec{\nu}}\;\Delta}{x_1, \vec{x}\;\mathsf{pos}^{-,\vec{\nu}}\;(y_1 : T_1)\Delta} \qquad \frac{x_1 \notin \mathrm{FV}(T_1) \qquad \vec{x}\;\mathsf{pos}^{\vec{\nu}}\;\Delta}{x_1, \vec{x}\;\mathsf{pos}^{\circ,\vec{\nu}}\;(y_1 : T_1)\Delta}$$

Remark: This is not the same definition as in CIC$\hat{}$. The actual definition in CIC$\hat{}$ is semantic in nature. Stage positivity is defined as follows: $\imath\;\mathsf{pos}_{\mathrm{CIC}\hat{}}\;T$ iff $\forall s \sqsubseteq r.\; T\,[\imath := s] \leq_{\mathrm{CIC}\hat{}} U\,[\imath := r]$. And similarly for the other notions. Our definition ensures the implication from left to right, but not in the opposite way.

Given the restrictions we impose on size variables (which are described in the next section), our definition is sufficient for CIC$\hat{}_{-}$. We prefer to define it this way to give a more syntactic definition, compared to CIC$\hat{}$. Note, however, that both definitions coincide for types in normal form. That is, if T is in normal form, then $\imath\;\mathsf{pos}_{\mathrm{CIC}\hat{}}\;T$ iff $\imath\;\mathsf{pos}\;T$.

2.3.3 Inductive Types

We present the complete definition of inductive types and the conditions that ensure their correctness. As we mentioned, parameters of inductive types are associated with polarities. We extend the definition given above to include strict positivity.

Definition 2.8 (Extended polarity)**.** *An extended polarity is either strict positivity, positivity, negativity, or invariant:*

$$\nu ::= \oplus \mid + \mid - \mid \circ \;.$$

Parameters of inductive types are actually associated with extended polarities. The notion of strict positivity plays an important rôle in the definition of inductive types, as explained below. The definitions given above (subtyping, positivity of stage variables, and positivity of term variables) extend immediately to extended polarities by giving $\oplus$ the same rôle as $+$. Extended polarities are introduced to allow an inductive type to be used recursively as argument of another inductive type (cf. Example 2.12).

An inductive type is introduced by a declaration of the form

$$\mathsf{Ind}(I[\Delta_p]^{\vec{\nu}} : A := \langle C_i : T_i \rangle_i) \qquad (*)$$

where I is the name of the inductive type, Δ_p is a context defining the parameters of I, $\vec{\nu}$ is a sequence of extended polarities of the parameters, A is the type of I, and $\langle C_i : T_i \rangle_i$ define the constructors of I. We use the variable $\mathcal{X}$ in the types of constructor to indicate recursive arguments. Given $x \in \mathrm{dom}\,(\Delta_p)$, we write $\vec{\nu}(x)$ to mean the extended polarity associated with x.

Inductive definitions are collected in a signature. Formally signatures are defined by the grammar:

$$\Sigma ::= [] \mid \Sigma(\mathsf{Ind}(I \ldots))$$

where $\mathsf{Ind}(I \ldots)$ is an inductive type declaration.

The conditions that ensure the correctness of an inductive declaration in CIC$\hat{}_{-}$ are derived from CIC [72] and CIC$\hat{}$ [19]. From CIC we use the concepts of arity, strict positivity, and constructor type.

Definition 2.9 (Arity)**.** *A term T is an* arity of sort u *if it satisfies one of the following conditions:*

– $T \equiv u$;
– $T \equiv \Pi x : U_1.U_2$ *and* U_2 *is an arity of sort* u

In a declaration of the form (*), A must be an arity of sort Type_k for some k.

Recall that an inductive definition defines a monotone operator. One way to ensure monotonicity is to restrict the variable $\mathcal{X}$ in the types of constructor to appear positively. However, in the case of CIC, this condition is not enough to ensure a correct definition [33] (see Example 2.13). The stronger condition of *strict positivity* is required.

Definition 2.10 (Strict Positivity)**.** *A variable* X *occurs* strictly positive *in a term* T*, written* X POS T*, if one of the following conditions is satisfied:*
– $X \notin \mathrm{FV}(T)$;
– $T \equiv X\, t_1 \ldots t_n$*, and* $X \notin \mathrm{FV}(t_1, \ldots, t_n)$;
– $T \equiv \Pi x : U_1.U_2$*,* $X \notin \mathrm{FV}(U_1)$*, and* X POS U_2;
– $T \equiv I^\infty\,(\langle p_i \rangle_i, \vec{a})$*, where* I *is an inductive definition in* Σ*, i.e.,*

$$\mathsf{Ind}(I[\Gamma]^{\vec{\nu}} : A := \Delta) \in \Sigma,$$

(with $\#\vec{p} = \#\vec{\Gamma}$*), and the following conditions hold:*
– $X \notin \mathrm{FV}(\vec{a})$,
– $X \notin \mathrm{FV}(p_k)$*, for all* k *such that* $\nu_k \neq \oplus$*, and*
– X POS p_k*, for all* k *such that* $\nu_k = \oplus$.
In other words, X *appears strictly positive in the strictly positive parameters.*

In a correct inductive declaration of the form (*), the variable $\mathcal{X}$ appears strictly positive in the arguments of the constructors. This condition is ensured by the notion of constructor type.

Definition 2.11 (Constructor type)**.** *A term* T *is a* constructor type of X*, denoted with* $\mathsf{Constr}_X(T)$*, if one of the following conditions is satisfied:*
– $T \equiv X\,\vec{t}$;
– $T \equiv \Pi x : U_1.U_2$*, where* $X \notin \mathrm{FV}(U_1)$ *and* $\mathsf{Constr}_X(U_2)$;
– $T \equiv U_1 \to U_2$*, where* X POS U_1 *and* $\mathsf{Constr}_X(U_2)$.

Note that if T is a constructor type of X, then $T \equiv \Pi\Delta.X\vec{t}$, where X POS Δ.

We are now ready to define the conditions that ensure that an inductive definition is valid. As mentioned, these conditions depend on the typing judgment for terms. Let Σ be a signature such that $\mathsf{WF}(\Sigma)$, and let $\mathcal{I}$ be the inductive definition

$$\mathsf{Ind}(I[\Delta_p]^{\vec{\nu}} : A := \langle C_i : \Pi\Delta_i.\mathcal{X}\,\vec{t_i} \rangle_{i=1..n}) \ .$$

(I1). The inductive definition is well-typed: there exists j such that

$$\Delta_p \vdash A : \mathsf{Type}_j$$

is a valid typing judgment with signature Σ.

(I2). The constructors are well-typed in the universe Type_k, i.e., for each $i = 1 \ldots n$,

$$\Delta_p(\mathcal{X} : A) \vdash \Pi\Delta_i.\mathcal{X}\,\vec{t_i} : \mathsf{Type}_k,$$

where $\mathcal{X}$ is a special variable representing the inductive type I applied to the parameters $\mathrm{dom}\,(\Delta_p)$.

(I3). There exists k such that A is an arity of sort Type_k. I.e., $A \equiv \Pi\Delta_a.\mathsf{Type}_k$. The sort Type_k is the sort (or the universe) of I. The context Δ_a is the context of arguments of I.

(I4). The types T_i, for $i = 1 \dots n$, satisfy $\mathsf{Constr}_{\mathcal{X}}(T_i)$.

(I5). Each occurrence of inductive types in Δ_p, Δ_a, and in Δ_i, $\vec{t_i}$ (for $i = 1 \dots n$), is annotated with ∞;

(I6). Each variable in Δ_p satisfies the polarity condition in the type of each constructor. This means $\mathrm{dom}\,(\Delta_p)\ \mathsf{pos}^{\vec{\nu}}\ \Delta_p$ and for $i = 1 \dots n$, $\mathrm{dom}\,(\Delta_p)\ \mathsf{pos}^{\vec{\nu}}\ \Delta_i$.

(I7). Positive and negative variables in Δ_p do not appear in $\vec{t_i}$, for $i = 1 \dots n$.

(I8). Each variable in Δ_p satisfies the polarity condition in the arguments of the inductive type. This means $\mathrm{dom}\,(\Delta_p)\ \mathsf{pos}^{\vec{\nu}}\ \Delta_a$.

(I9). Each strictly positive variable in Δ_p appears strictly positively in the constructors. This means if $x \in \mathrm{dom}\,(\Delta_p)$ with $\vec{\nu}(x) = \oplus$, then $x\ \mathsf{POS}\ \Delta_i$ for $i = 1 \dots n$.

The first four clauses are standard [72]. We require that all the components be well-typed —clauses (I1) and (I2)—, that the type A is an arity —clause (I3)— ensuring that I (suitable applied) is a type itself, and that recursive arguments in constructors are strictly positive —clause (I4). Note that constructors are well-typed in the same universe as the inductive type itself. This condition is important to ensure logical consistency.

The next four clauses are taken from CIC$^\frown$ [19]. Recall, however, that our definition of positivity for CIC$^\frown_-$ is slightly different than that of CIC$^\frown$. Clause (I5) ensures that constructors use previously defined inductive types, but not approximations of them. It is not useful in our current type system to use approximations (e.g., an argument of type $\mathsf{nat}^\imath$) since we cannot instantiate size variables. Consider, for example, an inductive type $\mathsf{Ind}(I : \mathsf{Type}_0 := C : (\mathsf{nat}^\imath \to \mathsf{nat}^\imath) \to \mathcal{X})$. Then, the term $M \equiv C(\lambda x : \mathsf{nat}.x)$ would have type I^∞. Given the global nature of size variables, this type would not be very useful if we cannot instantiate $\imath$ to something else. For example, we would not be able to use $\imath$ in fixpoints, because of the freshness condition we impose. On the other hand, we could allow instantiating $\imath$ to another stage. But this mean that stages would have to appear explicitly in terms. Since we want to keep stages hidden from the user, we decided not to pursue this approach.

Clauses (I6) and (I7) reflect the subtyping rules for inductive types, and are used in the proof of subject reduction.

Clause (I8) is inherited from CIC$^\frown$ where it is required to guarantee the completeness of type inference. [2] From subtyping rules, we have that $\vec{p_1} \leq^{I.\vec{\nu}} \vec{p_2}$ implies $I\,(\vec{p_1}, \vec{a}) \leq I\,(\vec{p_2}, \vec{a})$. We require $\mathrm{dom}\,(\Delta_p)\ \mathsf{pos}^{I.\vec{\nu}}\ \Delta_a$ to guarantee that if $I\,(\vec{p_1}, \vec{a})$ and all the components of p_2 are well typed, then $I\,(\vec{p_2}, \vec{a})$ will be well typed.

Lastly, clause (I9) is ensures the soundness of the strict positivity condition for parameters. The following example show the definitions of some commonly-used inductive types.

Example 2.12. *Natural numbers, lists, and trees can be defined as inductive types as follows:*

$$\mathsf{Ind}(\mathsf{nat} : \mathsf{Type}_0 := \mathsf{O} : \mathcal{X} \mid \mathsf{S} : \mathcal{X} \to \mathcal{X})$$
$$\mathsf{Ind}(\mathsf{list}[A^\oplus : \mathsf{Type}_0] : \mathsf{Type}_0 := \mathsf{nil} : \mathcal{X} \mid \mathsf{cons} : A \to \mathcal{X} \to \mathcal{X})$$
$$\mathsf{Ind}(\mathsf{tree}[A^\oplus : \mathsf{Type}_0] : \mathsf{Type}_0 := \mathsf{node} : A \to \mathsf{list}(\mathcal{X}) \to \mathcal{X})$$

2. In this work we do not cover type inference of CIC$^\frown_-$.

The definitions of natural numbers and lists are standard. A tree is formed by an element of type A and a list of trees. The definition is allowed because the parameter A in list *is strictly positive. Extended polarities are introduced precisely for this purpose.*

Logic connectives can also be defined as inductive types. For example, truth and contradiction can be represented as follows:

$$\mathsf{Ind}(\mathsf{True} : \mathsf{Type}_0 := I : \mathcal{X})$$
$$\mathsf{Ind}(\mathsf{False} : \mathsf{Type}_0 :=)$$

The following example shows that the restriction to strict positivity in recursive arguments is essential.

Example 2.13. *The condition of strict positivity is essential in clause (I4), as is shown in this example taken from [33]. Consider the inductive type*

$$\mathsf{Ind}(I : \mathsf{Type}_0 := \mathsf{in} : ((\mathcal{X} \to \mathsf{Prop}) \to \mathsf{Prop}) \to \mathcal{X})$$

Then $\mathcal{X}$ is positive in the type of in*, but not strictly positive. For readability, let us write $X \to \mathsf{Prop}$ as $\mathcal{P}(X)$. Intuitively, $\mathcal{P}(X)$ represents the type of subsets of X, or in set-theoretical terms, the powerset of X. In the following we assume that we have the usual impredicative encodings for equality, existential, logical negation and logical conjunction.*

Note that in *is an injective function of type $(\mathcal{P}(\mathcal{P}(I)) \to I$.*

It is not difficult to write an injective function of type $\mathcal{P}(I) \to \mathcal{P}(\mathcal{P}(I))$. For example, for any X, we can take

$$\mathsf{out} \equiv \lambda(x : X)(y : X).x = y : X \to \mathcal{P}(X)$$

Taking $X = \mathcal{P}(I)$ we have, where $\circ$ denotes function composition,

$$f \equiv \mathsf{in} \circ \mathsf{out} : \mathcal{P}(I) \to I,$$

which is an injective function. Following the intuitive meaning of $\mathcal{P}$, we just defined an injective function from the powerset of I to I. Since there is no such function in set theory, it is not surprising that we can derive a contradiction. More precisely, taking

$$P_0 \equiv (\lambda x : I.\exists(P : I \to \mathsf{Prop}).x = f\, P \wedge \neg(P\, x)) : I \to \mathsf{Prop}$$

we can easily conclude $P_0(f\, P_0)$ iff $\neg P_0(f\, P_0)$.

Note that our definition of positivity does not allow non-invariant parameters to have function types. For example, the following inductive type is not allowed:

$$\mathsf{Ind}(\mathsf{sigma}[(A : \mathsf{Type}_0)(B : A \to \mathsf{Type}_0)]^{+,+} := \mathsf{dpair} : \Pi(x : A).B\, a \to \mathcal{X})$$

However, the same definition with polarities $(+, \circ)$ is valid. The same situation occurs in CIC$\hat{}$.

Typing rules for signatures. We are now ready to define the typing rules for signatures. As mentioned, they depend on the typing judgment for terms. There are two rules, one for the empty signature, and another for adding an inductive type to a signature:

$$\text{(S-empty)} \quad \frac{}{\mathsf{WF}([])}$$

$$\text{(S-cons)} \quad \frac{\mathsf{WF}(\Sigma) \qquad I \text{ satisfies conditions (I1) to (I9)}}{\mathsf{WF}(\Sigma, I)}$$

We introduce some abbreviations to handle inductive types. They are used in the typing rules. Given an inductive definition

$$\mathsf{Ind}(I[\Delta_p]^{\vec{\nu}} : \Pi\Delta_a.u := \langle C_i : \Pi\Delta_i.\mathcal{X}\,\vec{t_i}\rangle_i)$$

we define

$$\begin{aligned}
\mathsf{params}(I) &= \Delta_p \\
\mathsf{sort}(I) &= u \\
\mathsf{typeInd}_I(\vec{p}) &= \Pi\Delta_a\left[\mathrm{dom}\left(\Delta_p\right) := \vec{p}\right].u \\
\mathsf{typeConstr}^s_{C_j}(\vec{p},\vec{a}) &= \widehat{I^s}\left(\vec{p}, t_j\left[\mathrm{dom}\left(\Delta_p\right) := \vec{p}\right]\left[\mathrm{dom}\left(\Delta_j\right) := \vec{a}\right]\right) \\
\mathsf{argsConstr}^s_{C_j}(\vec{p}) &= \Delta_j\left[\mathrm{dom}\left(\Delta_p\right) := \vec{p}\right]\left[\mathcal{X}\,\vec{u} := I^s\left(\vec{p},\vec{u}\right)\right] \\
\mathsf{indices}_{C_j}(\vec{p}) &= \vec{t_j}\left[\mathrm{dom}\left(\Delta_p\right) := \vec{p}\right] \\
\mathsf{branch}^s_{C_j}(\vec{p},\vec{y}.x.P) &= \Pi\mathsf{argsConstr}^s_{C_j}(\vec{p}).P\left[\vec{y} := \mathsf{indices}_{C_j}(\vec{p})\right]\ \left[x := C_j(|\vec{p}|, \mathrm{dom}\left(\Delta_i\right))\right] \\
\mathsf{caseType}^s_I(\vec{p},\vec{y},x) &= (\vec{y} : \Delta_a\left[\mathrm{dom}\left(\Delta_p\right) := \vec{p}\right])(x : I^s\left(\vec{p},\vec{y}\right))
\end{aligned}$$

in the definition of $\mathsf{argsConstr}$, the replacement $[\mathcal{X}\,\vec{u} := I^s\left(\vec{p},\vec{u}\right)]$ means to replace each occurrence of $\mathcal{X}\,\vec{u}$ by $I^s\left(\vec{p},\vec{u}\right)$. It only makes sense for well-typed inductive definitions. branch and $\mathsf{caseType}$ are used in the typing rule of the case construction to give the type of a branch and the return type, respectively (see Sect. 2.4).

2.3.4 Simple types

Up to this point, all the definitions given are also valid for CIC$^{\widehat{}}$ (except for positivity as we already remarked). In the typing rules for terms described below, we depart from CIC$^{\widehat{}}$, by making some restrictions on the use of size variables.

Basically, the restriction we make in CIC$^{\widehat{}}_{-}$ with respect to CIC$^{\widehat{}}$ is in the use of size variables. Intuitively, we only allow types of the form

$$\Pi x_1 : T_1.\Pi x_2 : T_2.\ldots\Pi x_n : T_n.T_{n+1}, \qquad (*)$$

where each T_i is of the form (*), or is of the form $I^s\left(\vec{p},\vec{a}\right)$ with $\mathrm{SV}(\vec{a}) = \emptyset$ (i.e., inductive types fully applied), or satisfies $\mathrm{SV}(T) = \emptyset$. We call these types *simple*. In particular, we do not allow strong eliminations that involve approximations of inductive types. For example, we cannot write a function f such that

$$f\,n = \mathsf{nat}^{\imath} \to \ldots \to \mathsf{nat}^{\imath},$$

but we can write a function g such that

$$g\,n = \mathsf{nat}^{\infty} \to \ldots \to \mathsf{nat}^{\infty}\ .$$

$$\text{(s-empty)}\quad \frac{\mathrm{SV}(T)=\emptyset}{\mathsf{simple}(T)}$$

$$\text{(s-prod)}\quad \frac{\mathsf{simple}(T_1)\qquad \mathsf{simple}(T_2)}{\mathsf{simple}(\Pi x:T_1.T_2)}$$

$$\text{(s-ind)}\quad \frac{I\in\Sigma\qquad \mathsf{simple}^{\vec{\nu}}(\vec{p})\qquad \mathrm{SV}(\vec{a})=\emptyset}{\mathsf{simple}(I^s\,(\vec{p},\vec{a}))}$$

$$\text{(vs-empty)}\quad \frac{}{\mathsf{simple}^{\varepsilon}(\varepsilon)}\qquad \text{(vs-inv)}\quad \frac{\mathrm{SV}(T)=\emptyset\qquad \mathsf{simple}^{\vec{\nu}}(\vec{T})}{\mathsf{simple}^{\circ,\vec{\nu}}(T,\vec{T})}$$

$$\text{(vs-ninv)}\quad \frac{\mathsf{simple}(T)\qquad \mathsf{simple}^{\vec{\nu}}(\vec{T})\qquad \nu\neq\circ}{\mathsf{simple}^{\nu,\vec{\nu}}(T,\vec{T})}$$

Figure 2.4: Simple types

The computation with *full* types (i.e., types with no size variables) is not restricted. This way, CIC$\hat{}_{-}$ is stronger than CIC, but weaker than CIC$\hat{}$. We discuss in more detail the relation between CIC$\hat{}$ and CIC$\hat{}_{-}$ in Sect. 2.6. In particular, we see that the restrictions are not so severe in practice. For example, functions like f above have little practical value in CIC$\hat{}$.

The main advantage of restricting to simple types is that the model construction is easier and we are able to adapt the known technique of Λ-sets [7] to the case of CIC$\hat{}_{-}$. See Chapter 4 for a detailed explanation.

Formally, we define a predicate simple on types that ensures that the form (*) is respected. We want to allow size variables to appear in parameters of inductive types, e.g. $\mathsf{list}^{\imath}(\mathsf{nat}^{\imath})$, or $\mathsf{list}^{\imath}(\mathsf{nat}^{\imath}\to\mathsf{nat}^{\imath})$/ Arguments, on the other hand, have no size variables.

We extend the predicate to a sequence of terms with polarities. They are defined in Fig. 2.4. We extend the predicate simple to contexts in the obvious way. Rules (s-empty) and (s-prod) are defined as expected. In rule (s-ind), we allow size variables in the parameters, only for non-invariant parameters. This corresponds to our notion of positivity, where non-invariant parameters of inductive types must be types themselves.

Note that, if $\mathsf{simple}(I^s\,(\vec{p},\vec{a}))$ and I is well-formed (it satisfies conditions (I1) to (I9)), then the arguments and constructors of I are also simple. I.e., $\mathsf{simple}(\mathsf{argsInd}_I(\vec{p}))$ and $\mathsf{simple}(\mathsf{argsConstr}^s_C(\vec{p}))$ for all constructors C of I.

2.4 Terms and contexts

To complete the presentation of the typing system, we present the typing rules for terms and contexts. From now on, we assume a valid fixed signature Σ and we omit to refer to it in typing judgments. The typing rules for terms and contexts are given in Fig. 2.5. The rules for typing sequences of terms are given in Fig. 2.6. For handling the typing of sorts and products, we use a specification in the style of Pure Type Systems [11]. The set Axiom is a

set of pairs that represents the typing relations between sorts. It is defined by

$$\mathsf{Axiom} = \{(\mathsf{Prop}, \mathsf{Type}_0)\} \cup \{(\mathsf{Type}_i, \mathsf{Type}_{i+1}) : i \in \mathbb{N}\} \ .$$

The set Rule is a set of triples of sorts that represents the typing constraints in the formation of products. It is defined by

$$\begin{aligned}\mathsf{Rule} = \ & \{(\mathsf{Type}_i, \mathsf{Prop}, \mathsf{Prop}) : i \in \mathbb{N}\} \cup \\ & \{(\mathsf{Type}_i, \mathsf{Type}_j, \mathsf{Type}_{\max(i,j)}) : i, j \in \mathbb{N}\} \ .\end{aligned}$$

Note that $\{\mathsf{Type}_k\}_k$ is predicative, while Prop is impredicative (cf. Sect. 1.1). These sets are used in rules (sort) and (prod) in Fig. 2.5.

The side conditions in rules (app), (abs), (ind), (constr), and (case) ensure that we restrict to simple types. Note that, if we remove these side conditions, the resulting system is that of CIC$^\wedge$.[3]

We explain in detail the typing rules of CIC$^\wedge_-$. First, let us state some invariants of the typing judgment that help to understand some design choices we made. Consider a valid judgment $\Gamma \vdash M : T$. Then the following holds:

- $\mathsf{simple}(\Gamma) \wedge \mathsf{simple}(T)$;
- $T \approx u \Rightarrow \mathsf{simple}(M)$;
- $T \not\approx u \Rightarrow \mathrm{SV}(M) = \emptyset$.

Looking at the second and third invariant we see that we always have $\mathsf{simple}(M)$. However, although the simple predicate is defined on terms, we reserve the application for terms that are actually types (such as in the second invariant). These invariants state that all types involved in a valid judgment are simple. Furthermore, terms that are not types do not have size variables. Recall that a term M is *full* if it does not contain size variables (i.e., $\mathrm{SV}(M) = \emptyset$). This corresponds to our intuition that we do not compute with approximations of inductive types.

Rules (empty) and (cons) deal with well-formedness of contexts and are standard. Rules (var), (sort), (prod), (abs), (app), and (conv) are also standard. Note from the definition of the set Rule that Prop is impredicative, while Type_j forms a predicative hierarchy. (We discuss universe inclusion below.) In rule (app), the type of the application is simple. This follows from the property: $\mathsf{simple}(T) \wedge \mathrm{SV}(M) = \emptyset \Rightarrow \mathsf{simple}(T\,[x := M])$. In rule (abs), the type of the abstraction is erased (the same as in CIC$^\wedge$). As a further restriction, we require the body of an abstraction and the argument of an application to be full. This ensures that we do not compute with approximations of inductive types.

In rule (ind), we require the inductive type to be simple. That implies that arguments are full, but parameters can have size variables. For example, $\mathsf{list}^i\,(\mathsf{nat}^i)$ is well-typed.

Rule (constr) defines the typing of constructors. Recall that constructors are fully applied to parameters and real arguments. The parameters are erased terms (as in CIC$^\wedge$), while we further require the arguments to be full. Note that the size annotation in the return type is a successor size, while recursive arguments to have a smaller size. This is also true for non-recursive constructors. For example, $\vdash \mathsf{O} : \mathsf{nat}^{\widehat{s}}$. Allowing O of type nat^i would destroy termination (see rule (fix) and the explanation below).

Rule (case) looks complicated due to the presence of dependent types. The type of the expression depends on the argument M, whose type is an inductive type. Since different

3. Except for the treatment of the return type in case expressions and the fact that inductive types are fully applied. However, these differences are not essential.

(empty) $$\frac{}{\mathsf{WF}([])}$$

(cons) $$\frac{\mathsf{WF}(\Gamma) \qquad \Gamma \vdash T : u}{\mathsf{WF}(\Gamma(x:T))}$$

(var) $$\frac{\mathsf{WF}(\Gamma) \qquad \Gamma(x) = T}{\Gamma \vdash x : T}$$

(sort) $$\frac{\mathsf{WF}(\Gamma) \qquad (u_1, u_2) \in \mathsf{Axiom}}{\Gamma \vdash u_1 : u_2}$$

(prod) $$\frac{\Gamma \vdash T : u_1 \qquad \Gamma(x:T) \vdash U : u_2 \qquad (u_1, u_2, u_3) \in \mathsf{Rule}}{\Gamma \vdash \Pi x : T.U : u_3}$$

(abs) $$\frac{\Gamma(x:T) \vdash M : U}{\Gamma \vdash \lambda x : |T|.M : \Pi x : T.U} \quad \mathrm{SV}(M) = \emptyset$$

(app) $$\frac{\Gamma \vdash M : \Pi x : T.U \qquad \Gamma \vdash N : T}{\Gamma \vdash M\, N : U\,[x := N]} \quad \mathrm{SV}(N) = \emptyset$$

(conv) $$\frac{\Gamma \vdash M : T \qquad \Gamma \vdash U : u \qquad T \leq U}{\Gamma \vdash M : U}$$

(ind) $$\frac{I \in \Sigma \qquad \Gamma \vdash \vec{p} : \mathsf{params}(I) \qquad \Gamma \vdash \vec{a} : \mathsf{argsInd}_I(\vec{p})}{\Gamma \vdash I^s\,(\vec{p}, \vec{a}) : \mathsf{sort}(I)} \quad \mathsf{simple}(I^s\,(\vec{p}, \vec{a}))$$

(constr) $$\frac{\begin{array}{c} I \in \Sigma \qquad \Gamma \vdash \vec{p} : \mathsf{params}(I) \\ \Gamma \vdash \vec{a} : \mathsf{argsConstr}^s_{C_i}(\vec{p}) \end{array}}{\Gamma \vdash C_i(|\vec{p}|, \vec{a}) : \mathsf{typeConstr}^s_{C_i}(\vec{p}, \vec{a})} \quad \mathsf{simple}(\mathsf{typeConstr}^s_{C_i}(\vec{p}, \vec{a}))$$

(case) $$\frac{\begin{array}{c} \Gamma \vdash M : \widehat{I^s}\,(\vec{p}, \vec{a}) \qquad I \in \Sigma \qquad \Gamma\,(\mathsf{caseType}^s_I(\vec{p}, \vec{y}, x)) \vdash P : u \\ \Gamma \vdash N_i : \mathsf{branch}^s_{C_i}(\vec{p}, \vec{y}.x.P) \end{array}}{\Gamma \vdash \left(\begin{array}{l} \mathsf{case}_{|P|}\ x := M\ \mathsf{in}\ I\,(|\vec{p}|, \vec{y}) \\ \mathsf{of}\ \{C_i \Rightarrow N_i\}_i \end{array} \right) : P\,[\vec{y} := \vec{a}]\,[x := M]} \quad \mathrm{SV}(N_i) = \emptyset$$

(fix) $$\frac{\begin{array}{c} T \equiv \Pi\Delta(x : I^\imath\,(\vec{p}, \vec{u})).U \qquad \imath\ \mathsf{pos}\ U \qquad \#\Delta = n - 1 \\ \imath \notin \mathrm{SV}(\Gamma, \Delta, \vec{p}, \vec{u}, M) \qquad \Gamma \vdash T : u \qquad \Gamma(f : T) \vdash M : T\,[\imath := \widehat{\imath}] \end{array}}{\Gamma \vdash \mathsf{fix}_n\ f : |T|^\imath := M : T\,[\imath := s]}$$

Figure 2.5: Typing rules of terms and contexts of CIC$\widehat{\ }_-$

$$\text{(c-empty)} \qquad \frac{\mathsf{WF}(\Gamma)}{\Gamma \vdash \varepsilon : []}$$

$$\text{(c-cons)} \qquad \frac{\Gamma \vdash M : T \qquad \Gamma \vdash \vec{N}\,[x := M] : \Delta\,[x := M]}{\Gamma \vdash M, \vec{N} : (x : T)\Delta}$$

Figure 2.6: Typing rules for sequences of terms

constructors can have different indices, the return type $|P|$ also depends on the indices. We use a syntax similar to that of Coq, where we make explicit the bound variables in $|P|$ (x for the argument of the case and $\vec{y}$ for the indices). Note that by the invariants of the typing judgment, we have $\mathsf{simple}(P)$. Also by the invariants, $\mathrm{SV}(\vec{a}) = \mathrm{SV}(M) = \emptyset$. Then the return type of the expression, satisfies $\mathsf{simple}(P\,[\vec{y} := \vec{a}]\,[x := M])$. We use the abbreviations $\mathsf{caseType}$ to give the type to the variables x and $\vec{y}$, and branch to give the type of a branch. Note that the size of argument is required to be a successor size. Recursive arguments in the branches can thus be given a smaller size. This is essential to allow recursive calls on recursive arguments in the definitions of fixpoints.

Finally, rule (fix) allows to type recursive functions. The natural number n indicates the recursive argument. The type of the function is of the form

$$T \equiv \Pi\Delta(x : I^{\imath}\,(\vec{p}, \vec{u})).U(\imath)$$

where $\#\Delta = n - 1$. The size variable $\imath$ is fresh ($\imath \notin \mathrm{SV}(\Gamma)$) and does not appear in Δ, $\vec{p}$, nor $\vec{u}$. It does not appear in M either. This last condition is used to ensure Subject Reduction (cf. Sect. 2.1). It can, however, appear (positively) in U. This is a sufficient condition to ensure termination. If we allow $\imath$ to appear in U without restrictions, then it is possible to type non-terminating functions, as shown by Abel [1].

Note that the type of M is $T\,[\imath := \widehat{\imath}]$, which is a type of the form

$$\Pi\Delta(x : I^{\widehat{\imath}}\,(\vec{p}, \vec{u})).U(\widehat{\imath})$$

where the recursive argument (the n-th argument) is of type I with size $\widehat{\imath}$. In M is possible to make recursive calls to f that has a type of the form

$$\Pi\Delta(x : I^{\imath}\,(\vec{p}, \vec{u})).U(\imath)$$

where the recursive argument has type I with size only $\imath$. Effectively, it is possible to make recursive calls only to smaller arguments. This condition together with $\imath\ \mathsf{pos}\ U$ ensure that we define a terminating function.

Note that we require $\imath$ not to appear in $\vec{p}$. This means that we cannot define, for example, fixpoints of type

$$\mathsf{list}^{\imath}(\mathsf{nat}^{\imath}) \to U(\imath)$$

Although this type would not introduce problems, the following example is not valid. Consider the inductive type I defined by

$$\mathsf{Ind}(I[A^{\oplus} : \mathsf{Type}] : \mathsf{Type} := C : (\mathsf{nat}^{\infty} \to A) \to \mathcal{X})$$

Then, the type

$$I^{\imath}(\mathsf{nat}^{\imath}) \to U(\imath)$$

with $\imath$ pos U is not valid for fixpoints. Note that the type $I^{s}(\mathsf{nat}^{s})$ is equivalent to the type of functions $\mathsf{nat}^{\infty} \to \mathsf{nat}^{s}$, i.e., there is a bijection between both types. The intuitive semantics of $\mathsf{nat}^{\infty} \to \mathsf{nat}^{s}$ is the set of functions $\mathbb{N} \to \{0, 1, \ldots, s-1\}$. Then, it is not true that $I^{\infty}(\mathsf{nat}^{\infty}) = \bigcup_{\imath} I^{\imath}(\mathsf{nat}^{\imath})$, and the soundness of the typing rule for fixpoint cannot be justified in this case. Although I is not recursive, we can add recursive constructors and the argument remains valid.

Finally, let us describe the use of position types in fixpoints. The position type $T^{\star}$ in the term satisfies $|T|^{\imath} \equiv T^{\star}$. The occurrences of $\star$ in $T^{\star}$ correspond with size expressions s in T whose base variables is $\imath$ (i.e., $\lfloor s \rfloor = \imath$). The $\star$ allows to mark which stages should be related to $\imath$ permitting size-preserving functions. For example, using position types of the form $\mathsf{nat}^{\star} \to \mathsf{nat}^{\star}$ and $\mathsf{nat}^{\star} \to \mathsf{nat} \to \mathsf{nat}^{\star}$ we can define fixpoints of type $\mathsf{nat}^{\imath} \to \mathsf{nat}^{\imath}$ and $\mathsf{nat}^{\imath} \to \mathsf{nat}^{\infty} \to \mathsf{nat}^{\imath}$, respectively.

With position types it is possible to ensure compact most general types. For example, consider the following valid judgments:

$$\vdash (\mathsf{fix}_1\ f : \mathsf{nat}^{\star} \to \mathsf{nat} := \lambda x : \mathsf{nat}.\mathsf{O}) : \mathsf{nat}^{\imath} \to \widehat{\mathsf{nat}^{\jmath}}$$
$$\vdash (\mathsf{fix}_1\ f : \mathsf{nat}^{\star} \to \mathsf{nat}^{\star} := \lambda x : \mathsf{nat}.\mathsf{O}) : \mathsf{nat}^{\imath} \to \mathsf{nat}^{\imath}$$

Note that the types are not comparable (since we cannot compare different size variables). Without the use of position types, we would need a form of union types to express the most general type of the above function.

Position types are used in CIC$\hat{}$ to generate compact general types [19]. Then, the extra burden in annotating position types is justified by the simplicity of the size-inference algorithm.

Although we do not consider size-inference in this work, the algorithm of [19] should be easily adapted to the case of CIC$\hat{}_{\sqsubseteq}$. Intuitively the algorithm works as follows: given an bare context Γ° and a bare term M°, it returns a tuple (Γ', M', T', C) where Γ' is an annotated context such that $|\Gamma'| \equiv \Gamma^{\circ}$, M' is an annotated term such that $|M'| \equiv M^{\circ}$, T' is an annotated type, and C is a set of constraints on size variables. Soundness of the algorithm means that given a substitution ρ from size variables to stages such that $C\rho$ is a valid set of constraints, then $\Gamma'\rho \vdash M'\rho : T'\rho$ is a valid judgment.

It seems that this algorithm can be readily adapted to CIC$\hat{}_{\sqsubseteq}$. All we need to do is add constraints that will ensure that types are simple. This means, constraints of the form $\infty \sqsubseteq \alpha$, where α is an annotation generated by the algorithm. This will ensure that α can only be instantiated to ∞.

2.5 Examples

We show several example that illustrate the features of the type-based termination approach. For the sake of readability, we omit some type annotations and inline the arguments of constructors in the branches of a case construction: we write $C\,\vec{x} \Rightarrow \ldots$, instead of $C \Rightarrow \lambda\vec{x}.\ldots$.

Division of natural numbers. We show how to type the function $\mathsf{div}\,x\,y = \lceil \frac{x}{y+1} \rceil$, defined by repeated subtraction. It is a simple example showing that it is possible to type non-structural recursive functions in CIC$\hat{}_-$. The definitions are the same as in the introduction to type-based termination (Sect. 1.2.2).

$$
\begin{array}{l}
\mathsf{minus} : \mathsf{nat}^s \to \mathsf{nat}^\infty \to \mathsf{nat}^s \\
\mathsf{minus} \stackrel{\text{def}}{=} \mathsf{fix}_1\ \mathsf{minus} : \mathsf{nat}^\star \to \mathsf{nat} \to \mathsf{nat}^\star := \\
\qquad \lambda\, x\, y.\ \mathsf{case}\, x_{\mathsf{nat}}\ \mathsf{of} \\
\qquad\quad \mid \mathsf{O} \Rightarrow \mathsf{O} \\
\qquad\quad \mid \mathsf{S}\ x' \Rightarrow \mathsf{case}_{\,\mathsf{nat}}\ y\ \mathsf{of} \\
\qquad\qquad\quad \mid \mathsf{O} \Rightarrow \mathsf{S}\, x' \\
\qquad\qquad\quad \mid \mathsf{S}\, y' \Rightarrow \mathsf{minus}\ x'_{\mathsf{nat}(\imath)}\ y
\end{array}
$$

Note the use of the position type $\mathsf{nat}^\star \to \mathsf{nat} \to \mathsf{nat}^\star$ to ensure that minus can have the type $\mathsf{nat}^s \to \mathsf{nat}^\infty \to \mathsf{nat}^s$ for any s. This is the key to allow the following definition:

$$
\begin{array}{l}
\mathsf{div} : \mathsf{nat}^s \to \mathsf{nat}^\infty \to \mathsf{nat}^s \\
\mathsf{div} \stackrel{\text{def}}{=} \mathsf{fix}_1\ \mathsf{div} : \mathsf{nat}^\star \to \mathsf{nat} \to \mathsf{nat}^\star := \\
\qquad \lambda\, x\, y.\ \mathsf{case}\ x_{\mathsf{nat}}\ \mathsf{of} \\
\qquad\quad \mid \mathsf{O} \Rightarrow \mathsf{O} \\
\qquad\quad \mid \mathsf{S}\ x' \Rightarrow \mathsf{S}\,(\mathsf{div}\,(\mathsf{minus}\, x'\, y)\, y)
\end{array}
$$

As explained in Chapter 1, the recursive call with first argument $\mathsf{minus}\, x'\, y$ is accepted because minus is given a size-preserving type. Note that x' and $\mathsf{minus}\, x'\, y$ have the same size.

Quicksort. Another example of non-structural recursive function is quicksort. It is a typical example in type-based termination approaches. It shows the benefits of allowing size-preserving functions. Consider the function that concatenates two lists:

$$
\begin{array}{l}
\mathsf{append} : \Pi(A : \mathsf{Type}_0).\mathsf{list}^\infty(A) \to \mathsf{list}^\infty(A) \to \mathsf{list}^\infty(A) \\
\mathsf{append} \stackrel{\text{def}}{=} \lambda\, A.\ \mathsf{fix}_1\ \mathsf{append} : \mathsf{list}^\star(A) \to \mathsf{list}(A) \to \mathsf{list}(A) := \\
\qquad \lambda\, l_1\, l_2.\ \mathsf{case}_{\,\mathsf{list}(A)}\ l_1\ \mathsf{of} \\
\qquad\quad \mid \mathsf{nil} \Rightarrow l_2 \\
\qquad\quad \mid \mathsf{cons}\, h\, t \Rightarrow \mathsf{cons}\, h\, (\mathsf{append}\, t\, l_2)
\end{array}
$$

We can only give to append the type $\mathsf{list}^\infty(A) \to \mathsf{list}^\infty(A) \to \mathsf{list}^\infty(A)$, or $\mathsf{list}^s(A) \to \mathsf{list}^r(A) \to \mathsf{list}^\infty(A)$. But it is not possible to express the more precise type $\mathsf{list}^s(A) \to \mathsf{list}^r(A) \to \mathsf{list}^{s+r}(A)$ since our size algebra does not include an addition operator.

Another standard function on lists is filter. It removes the elements of a list that do not satisfy a given predicate. The usual definition can be typed in CIC$\hat{}_-$, with a more precise type:

$$
\begin{array}{l}
\mathsf{filter} : \Pi A : \mathsf{Type}_0.(A \to \mathsf{bool}) \to \mathsf{list}^s(A) \to \mathsf{list}^s(A) \\
\mathsf{filter} \stackrel{\text{def}}{=} \lambda\, A\, (p : A \to \mathsf{bool}).\mathsf{fix}_1\ \mathsf{filter} : \mathsf{list}^\star(A) \to \mathsf{list}^\star(A) := \\
\qquad \lambda\, l.\ \mathsf{case}_{\,\mathsf{list}(A)}\ l\ \mathsf{of} \\
\qquad\quad \mid \mathsf{nil} \Rightarrow \mathsf{nil} \\
\qquad\quad \mid \mathsf{cons}\, h\, t \Rightarrow \mathsf{case}_{\,\mathsf{bool}}\ p\, h\ \mathsf{of} \\
\qquad\qquad\quad \mid \mathsf{true} \Rightarrow \mathsf{cons}\, h\, (\mathsf{filter}\, t) \\
\qquad\qquad\quad \mid \mathsf{false} \Rightarrow \mathsf{filter}\, t
\end{array}
$$

The precise type of filter allows to type-check the following definition of quicksort:

$$\begin{array}{l}\mathsf{qsort} : \Pi A : \mathsf{Type}_0.\mathsf{list}^s(A) \to \mathsf{list}^\infty(A)\\ \mathsf{qsort} \stackrel{\text{def}}{=} \lambda A.\mathsf{fix}_1\ \mathsf{qsort} : \mathsf{list}^\star(A) \to \mathsf{list}(A) :=\\ \qquad \lambda l.\ \mathsf{case}_{\mathsf{list}(A)}\ l\ \mathsf{of}\\ \qquad\quad \mid \mathsf{nil} \Rightarrow \mathsf{nil}\\ \qquad\quad \mid \mathsf{cons}\, h\, t \Rightarrow \mathsf{append}\,(\mathsf{filter}\, A\,(<h)\, t)\\ \qquad\qquad\qquad\qquad\qquad (\mathsf{cons}\, h\,(\mathsf{filter}\, A\,(\geq h)\, t))\end{array}$$

where $(<\ h)\, a$ is true when a is less than h and false otherwise, and $(\geq\ h)\, a$ is true when a is greater of equal than h and false otherwise. The trick to allow this function to type-check is the size-preserving type of filter. The use of the position type $\mathsf{list}^\star(A) \to \mathsf{list}^\star(A)$ is essential.

Note that qsort has type $\mathsf{list}^s(A) \to \mathsf{list}^\infty(A)$. We can not give a size-preserving type to qsort since we can not give a size-preserving type to append. In [21], Barthe et al. consider an extension of system F, where append can be given type $\mathsf{list}^s(A) \to \mathsf{list}^r(A) \to \mathsf{list}^{s+r}(A)$. They consider a richer size algebra that includes addition and 0. Non-recursive constructors are given size 0, e.g., nil has type $\mathsf{list}^0(A)$. Furthermore, the type system includes sized products, which are used to define the function partition of type:

$$(A \to \mathsf{bool}) \to \mathsf{list}^s(A) \to \mathsf{list}^{s_1}(A) \times^s \mathsf{list}^{s_2}(A)$$

The result of $\mathsf{partition}\, p\, l$ is a pair of lists (l_1, l_2) where l_1 contains the elements of l that satisfy p and l_2 the elements of l that do not satisfy p. The sized product $\mathsf{list}^{s_1}(A) \times^s \mathsf{list}^{s_2}(A)$ means that the relation $s_1 + s_2 = s$ is satisfied. As a consequence, qsort can be typed as a size-preserving function with type $\mathsf{list}^s(A) \to \mathsf{list}^s(A)$. However, this is possible at the expense of a complicated type system that has to keep track of constraints between sizes. Furthermore, some changes in the syntax are required: the body of fixpoints is restricted to be a case expression.

Trees and lists. Let us recall the definition of trees in terms of lists:

$$\mathsf{Ind}(\mathsf{tree}[A^\oplus : \mathsf{Type}_0] : \mathsf{Type}_0 := \mathsf{node} : A \to \mathsf{list}\,(\mathcal{X}) \to \mathcal{X})\ .$$

Defining trees this way means that we can reuse functions on lists. For example, we can define a mapping function on trees as follows:

$$\begin{array}{l}\mathsf{map\_tree} : \Pi(A : \mathsf{Type}_0)(B : \mathsf{Type}_0).(A \to B) \to \mathsf{tree}^s(A) \to \mathsf{tree}^s(B)\\ \mathsf{map\_tree} \stackrel{\text{def}}{=} \lambda A\, B\,(f : A \to B).\mathsf{fix}_1\ \mathsf{map\_tree} : \mathsf{tree}^\star(A) \to \mathsf{tree}^\star(B) :=\\ \qquad \lambda t.\ \mathsf{case}_{\mathsf{tree}(B)}\ t\ \mathsf{of}\\ \qquad\quad \mid \mathsf{node}\, x\, l \Rightarrow \mathsf{node}\,(f\, x)\,(\mathsf{map}\ \mathsf{map\_tree}\ l)\end{array}$$

where map is the usual mapping function on lists, which has the size-preserving type

$$\mathsf{map} : \Pi(A : \mathsf{Type}_0)(B : \mathsf{Type}_0).(A \to B) \to \mathsf{list}^s(A) \to \mathsf{list}^s(B)$$

In the recursive call $\mathsf{map}\,\mathsf{map\_tree}\, l$, the recursive function map_tree has type $\mathsf{tree}^\imath(A) \to \mathsf{tree}^\imath(B)$ where $\imath$ is the fresh size variable introduced in the typing rule (fix), and l has type $\mathsf{list}^\infty(\mathsf{tree}^\imath(A))$. Then, the expression $\mathsf{map}\,\mathsf{map\_tree}\, l$ has type $\mathsf{tree}^\imath(B)$, while the expression $\mathsf{node}\,(f\, x)\,(\mathsf{map}\,\mathsf{map\_tree}\, l)$ has type $\mathsf{tree}^{\widehat{\imath}}(B)$ as desired.

Ad-Hoc Predicate. We present an example based on the work of Bove [27]. The idea is to define a recursive function by induction on an ad-hoc predicate that defines the structure of recursive calls. This example is taken from [23]. We show that sized types simplify the use of this technique.

Consider the function that calculates the logarithm of base 2 of a natural number. In Haskell, we could write something like

```
log2 (S O) = O
log2 (S (S p)) = S (log2 (S (div2 p)))
```

where div2 divides a number by 2.

There are two things to notice. First, this is a partial function, since it is not defined for zero. Second, it is not structurally recursive. Furthermore, it cannot be directly written using sized types, since the argument of the recursive call is headed by a constructor. This means that its type has a successor size, hence the recursive call cannot be made.

This cannot be written directly in Coq either. The idea presented in [27] is to define an inductive predicate that represents the structure of recursive calls of a function. While this is not the only way of defining log2 in Coq, the approach illustrates a technique that is useful for defining function that have complicated recursion patterns. We will see how the use of this technique is simplified in CIC$\hat{}$.

To define the logarithm function given above, we define the following inductive type that reflects the recursion pattern:

$$\begin{aligned}\mathsf{Ind}(\mathsf{log\_domain} : \mathsf{nat} \to \mathsf{Type}_0 := \ & \mathsf{log\_domain1} : \mathcal{X}\,(\mathsf{S}\,\mathsf{O}),\\ & \mathsf{log\_domain2} : \Pi(p : \mathsf{nat}).\mathcal{X}\,(\mathsf{S}\,(\mathsf{div2}\,p)) \to \mathcal{X}\,(\mathsf{S}\,(\mathsf{S}\,p)))\end{aligned}$$

The function log2 can be defined by recursion on this type. The trick is to show the so-called *inversion lemmas*. In this case, there is only one inversion lemma with type

$$\mathsf{log\_domain\_inv} : \Pi(p : \mathsf{nat}^{\infty}).\mathsf{log\_domain}^{\widehat{s}}\,(\mathsf{S}\,(\mathsf{S}\,p)) \to \mathsf{log\_domain}^{s}\,(\mathsf{S}\,(\mathsf{div2}\,p))$$

Note that the size decreases in the return type. This allows us to make recursive call on the result of this function. Thus we can define

$$\begin{aligned}&\mathsf{log2} : \Pi(x : \mathsf{nat}^{\infty}).\mathsf{log\_domain}^{s}(x) \to \mathsf{nat}\\ &\mathsf{log2} \stackrel{\mathrm{def}}{=} \ \mathsf{fix}_1\ \mathsf{log2} : \Pi(x : \mathsf{nat}).\mathsf{log\_domain}^{\star}(x) \to \mathsf{nat} :=\\ &\qquad (\lambda(x : \mathsf{nat})(h : \mathsf{log\_domain}(x)).\\ &\qquad\quad \mathsf{case}_{\,\mathsf{log\_domain}(x_0)\to\mathsf{nat}}\ x_0 := x\ \mathsf{of}\\ &\qquad\quad \mid \mathsf{O} \Rightarrow \lambda(h' : \mathsf{log\_domain}(\mathsf{O})).\ldots.\\ &\qquad\quad \mid \mathsf{S}\,x' \Rightarrow \mathsf{case}_{\,\mathsf{log\_domain}(\mathsf{S}\,x_1)\to\mathsf{nat}}\ x_1 := x'\ \mathsf{of}\\ &\qquad\qquad\qquad \mid \mathsf{O} \Rightarrow \lambda\_.\,\mathsf{O}\\ &\qquad\qquad\qquad \mid \mathsf{S}\,p \Rightarrow \lambda(h' : \mathsf{log\_domain}(\mathsf{S}\,(\mathsf{S}\,p)).\\ &\qquad\qquad\qquad\qquad\quad \mathsf{S}\,(\mathsf{log2}\,(\mathsf{S}\,(\mathsf{div2}\,p)))\,(\mathsf{log\_domain\_inv}\,p\,h')\\ &\qquad)\ h\end{aligned}$$

Note that we do a case analysis on the term of type nat, following the pattern of the Haskell definition, but the recursion is done on the predicate. The definition follows a similar pattern to the Haskell definition, except for the extra bookkeeping related to the predicate log_domain. In the case of O, we can prove False from $\mathsf{log\_domain}(x)$ and $x = 0$. We omit the proof since it is not related to sized types.

In Coq, to make this definition type-check, we have to make explicit that we do the recursion on the predicate log_domain. Furthermore, the proof of the inversion lemma has to be done in a specific way in order to show that the result is structurally smaller than the argument. That is, $\mathsf{log\_domain\_inv}\, p\, h'$ must be structurally smaller than h'. To show termination, Coq unfolds the definition of log_domain_inv in the definition log2.

In CIC$\hat{}_{-}$, the situation is simplified. Only the type of log_domain_inv is necessary to type-check the definition of log2. The actual definition is not important for log2 to be well typed. The situation is similar to the div/minus example given above. Only the type of minus (log_domain_inv in this case) is necessary to ensure that div (log2 in this case) is well typed and therefore terminating.

For example, one way of defining log_domain_inv is the following:

$$
\begin{array}{l}
\mathsf{log\_domain\_inv} \stackrel{\text{def}}{=} \lambda\,(p : \mathsf{nat})\,(h : \mathsf{log\_domain}(\mathsf{S}\,(\mathsf{S}\,p))).\\
\qquad \mathsf{case}_{\,n_0=\mathsf{S}\,(\mathsf{S}\,p)\to\mathsf{log\_domain}(\mathsf{S}\,(\mathsf{div2}\,p))}\ h\ \mathsf{in}\ \mathsf{log\_domain}(n_0)\ \mathsf{of}\\
\qquad \mid \mathsf{log\_domain1} \Rightarrow\\
\qquad\quad \lambda\, H_{\mathrm{eq}} : \mathsf{S}\,\mathsf{O} = \mathsf{S}\,(\mathsf{S}\,p).\, \mathsf{case}_{\,\mathsf{log\_domain}(\mathsf{S}\,(\mathsf{div2}\,p))}\ M_0\ H_{\mathrm{eq}}\ \mathsf{of}\\
\qquad \mid \mathsf{log\_domain2}\, p\, H \Rightarrow\\
\qquad\quad \lambda\, H_{\mathrm{eq}} : \mathsf{S}\,(\mathsf{S}\,p_0) = \mathsf{S}\,(\mathsf{S}\,p).\, \mathsf{case}_{\,\mathsf{log\_domain}(y)}\ M_1\ H_{\mathrm{eq}}\ \mathsf{in}\ \mathsf{eq}(\mathsf{nat}, p_0, y)\ \mathsf{of}\\
\qquad\qquad\qquad\qquad \mid \mathsf{refl} \Rightarrow H
\end{array}
$$

where M_0 and M_1 have types $\mathsf{S}\,\mathsf{O} = \mathsf{S}\,(\mathsf{S}\,p) \to \mathsf{False}$ and $\mathsf{S}\,(\mathsf{S}\,p_0) = \mathsf{S}\,(\mathsf{S}\,p) \to p_0 = p$, respectively.

Remark: In Coq, we would define the predicate log_domain in Prop instead of Type_0. Although both approaches work for defining log2, however, using the extraction mechanism would produce different results. If defined as a propositional type, all traces of log_domain are removed as computationally irrelevant. The extracted function is essentially the same as the initial Haskell definition. This is not the case if log_domain is defined as a computational type.

Recall that in this work we do not consider inductive propositional types. We leave the study of this important extension for future work.

2.6 A Comparison Between CIC$\hat{}_{-}$ and CIC$\hat{}$

Let us come back to the differences between CIC$\hat{}_{-}$ and CIC$\hat{}$. Basically, CIC$\hat{}$ is the system presented in Sections 2.2 and 2.3, removing the side conditions in rules (abs), (app), (ind), (constr), and (case) in Fig. 2.5, that are related to simple types. CIC$\hat{}_{-}$ is thus a subsystem of CIC$\hat{}$; all valid typing judgments of CIC$\hat{}_{-}$ are also valid for CIC$\hat{}$. The opposite direction is not true. It is easy to find examples of typing derivations of CIC$\hat{}$ that are not valid in CIC$\hat{}_{-}$. A trivial example, although not interesting, is $\vdash (\lambda x : \mathsf{Type}_0.x)\mathsf{nat}^{\imath} : \mathsf{Type}_0$.

In this section we argue that, in practice, typing derivations of CIC$\hat{}$ can be translated to CIC$\hat{}_{-}$. For the purposes of this comparison, we assume that both systems have the same syntax. We write $\hat{\vdash}$ to denote the typing relation of CIC$\hat{}$.

Consider the following property, valid on both systems:

$$\Gamma \vdash M : T \Rightarrow \Gamma\,[\imath := s] \vdash M\,[\imath := s] : T\,[\imath := s]$$

This property is called Stage Substitution (see Lemma 3.17). This property says that we can substitute a stage variable with any stage expression in a well-typed judgment. Recall that stage variables are implicitly declared at the global level.

Assume a valid judgment $\Gamma \mathrel{\hat{\vdash}} M : T$. Let us try to translate it into a valid judgment for CIC$\hat{\ }_{-}$. By Stage Substitution, we can replace all size variables in Γ, M, T by ∞ obtaining a valid judgment $\Gamma^\infty \mathrel{\hat{\vdash}} M^\infty : T^\infty$. Note that Γ^∞ and T^∞ are simple (since they contain no size variables). However, this does not imply that the judgment is valid in CIC$\hat{\ }_{-}$, since there are size variables appear in the derivation $\Gamma^\infty \mathrel{\hat{\vdash}} M^\infty : T^\infty$, introduced by the fixpoint rule. If there are no fixpoint in Γ, M, nor T, then $\Gamma^\infty \vdash M^\infty : T^\infty$ is also valid in CIC$\hat{\ }_{-}$.

Let us consider the case where fixpoints appear, for example, in M. Let us assume that M contains a fixpoint of type T. If T is a valid type for fixpoint, and is in normal form, it has the form

$$\Pi\Delta(x : I^\imath(\vec{p}, \vec{a})).U(\imath)$$

for some size variable $\imath$ with $\imath \notin \mathrm{SV}(\Delta, \vec{p}, \vec{a})$ and $\imath\ \mathsf{pos}_{\mathrm{CIC}\hat{\ }}\ U$. If the fixpoint is not nested inside another, we can assume that $\mathrm{SV}(T) = \{\imath\}$, since all other size variables can be replaced by ∞. Since U is in normal form, we have $\imath\ \mathsf{pos}\ U$. The type T satisfies $\mathsf{simple}(T)$; it can be used as a valid type for fixpoint in CIC$\hat{\ }_{-}$.

This does not mean that the derivation of M can be translated to CIC$\hat{\ }_{-}$. In the derivation of the body of the fixpoint, the variable $\imath$ can be used in ways that to dot satisfy the simple predicate. However, if only subterms of T are used in the derivation of the body of the fixpoint, then the predicate simple would be satisfied. For example, if $T \equiv \mathsf{nat}^\infty \to \mathsf{list}^\imath(\mathsf{nat}^\infty) \to \mathsf{nat}^\infty \to \mathsf{nat}^\imath$, then the subterms nat^∞, $\mathsf{nat}^\imath$ and $\mathsf{list}^\imath(\mathsf{nat}^\infty)$ are all simple.

Typically, definitions by fixpoint follow this pattern. This is the case for the examples in Sect. 2.5. Also for all the examples in [17,18,19,20] which are valid in CIC$\hat{\ }_{-}$.

2.7 Related Work

The idea of using types to ensure termination has a long history that can be traced back to the work of Mendler [64]. In the context of functional programming languages, an extension of Haskell with type-based termination is introduced in Hughes et al. [48], developed in detail in Pareto's thesis [71].

Type-based termination in dependent type theory. Giménez [40] considers a simple type-based termination mechanism for an extension of CC with inductive and coinductive definitions. His systems allows the possibility of expressing size-preserving functions, but does not include an explicit representation of stages. Size-preserving functions are expressed through constraints. For example, the minus function has type $\forall X \leq \mathsf{nat}.X \to \mathsf{nat} \to X$.

Barras [13,14] uses a similar mechanism in his formalization of the metatheory of CIC.

Another important related work is that of Blanqui [25]. He develops an extension of CC with rewriting rules, and uses a type-based criterion to ensure termination. The main difference with our approach is the use of rewriting rules, instead of fixpoint and case analysis. The advantage of using rewriting rules is the possibility of including non-deterministic rules

such as

$$\begin{aligned} \mathsf{plus}\,\mathsf{O}\,y &\to y \\ \mathsf{plus}\,(\mathsf{S}\,x)\,y &\to \mathsf{S}\,(\mathsf{plus}\,x\,y) \\ \mathsf{plus}\,x\,\mathsf{O} &\to x \end{aligned}$$

This is not possible with the fixpoint+case approach. On the other hand, the use of rewriting requires the use of complex criteria to ensure that rewriting rules are confluent.

Explicit sizes. Abel [1] introduces an extension of F_ω, called $\mathsf{F}^{\widehat{}}_\omega$. Unlike the systems in the CIC$\widehat{\ }$ family, in $\mathsf{F}^{\widehat{}}_\omega$ stages are defined as part of the language. Sizes are similar to CIC$\widehat{\ }$, having a successor operator and ∞, but are defined as terms of a kind ord. The typing rules governing sizes are:

$$\frac{}{\Gamma \vdash \mathsf{ord}\ \mathsf{kind}} \qquad \frac{\Gamma \vdash s : \mathsf{ord}}{\Gamma \vdash \widehat{s} : \mathsf{ord}} \qquad \frac{}{\Gamma \vdash \infty : \mathsf{ord}}$$

Being a non-dependent theory (types cannot depend on values), there are separate syntactic categories for terms, types constructors and kinds. Abstractions are in Curry-style, i.e., do not include the type of the argument, as in $\lambda x.M$. Fixpoints do not include the type of the abstraction neither. The typing rule is the following:

$$\frac{A\ \mathsf{fix}_n\text{-}\mathsf{adm} \qquad \Gamma \vdash a : \mathsf{ord}}{\Gamma \vdash \mathsf{fix}_n : (\forall \imath{:}\mathsf{ord}.A\,\imath \to A\,\widehat{\imath}) \to A\,a}$$

where $\mathsf{fix}_n\text{-}\mathsf{adm}$ is a predicate that ensures that A is a valid type for recursion. Introduction and elimination of $\forall$ is implicit. We show the typing rule for the particular case of size quantification, although similar rules apply to quantification on any kind.

$$\frac{\Gamma(\imath : \mathsf{ord}) \vdash t : F\,\imath}{\Gamma \vdash t : \forall \imath{:}\mathsf{ord}.F}\ \imath \notin \mathrm{FV}(F) \qquad \frac{\Gamma \vdash t : \forall \imath{:}\mathsf{ord}.F \qquad \Gamma \vdash a : \mathsf{ord}}{\Gamma \vdash t : F\,a}$$

As an example of fixpoint, addition of natural numbers could be defined with type $\mathsf{nat}^{\imath} \to \mathsf{nat}^{\infty} \to \mathsf{nat}^{\infty}$ as follows:

$$\mathsf{fix}_1(\lambda \mathsf{plus}.\lambda x.\lambda y.\mathsf{case}\ x\ \mathsf{of}\ \mathsf{O} \Rightarrow y \mid \mathsf{S} \Rightarrow \lambda x'.\mathsf{S}\,(\mathsf{plus}\,x'\,y))$$

Since abstraction is Curry-style and size abstraction and application is implicit, there are no sizes appearing in terms. Hence, the problem we have in CIC$\widehat{\ }_{-}$ with Subject Reduction does not appear in this case.

In the case of dependent types, a Church-style abstraction is necessary, in order to have decidability of type-checking. Abel [3] proposes a programming language featuring dependent and sized types in the form of an implementation called MiniAgda. It features dependent types, Agda-style pattern-matching definitions, and sized inductive and coinductive types.

Sizes are explicit. They are defined by a built-in inductive type with two constructors: successor and infinity. However, it is not allowed to pattern-match on sizes and there are several restrictions that ensure that sizes are not needed in computation. In particular, MiniAgda features two different variants of products and abstractions: explicit and implicit. The former kind is just the usual constructions for dependent product. The latter is inspired by [15,67]

and ICC [65]. Implicit products and abstraction behave like their explicit counterparts, but their typing rules are more restricted to ensure that they can be removed in runtime without affecting computational behavior. As an example, we show how to define natural numbers in MiniAgda:

$$\begin{aligned}\mathsf{Ind}(\mathsf{nat} : \mathsf{size} \to \mathsf{Type} := \ & \mathsf{O} : \Pi[s : \mathsf{size}].\mathsf{nat}\,\widehat{s} \\ & \mathsf{S} : \Pi[s : \mathsf{size}].\mathsf{nat}\,s \to \mathsf{nat}\,\widehat{s})\end{aligned}$$

where implicit product is denoted by $\Pi[x : T].U$. If we remove the implicit product, the definition is similar to that of CIC$\widehat{\ }_{-}$. Functions like minus, div can be given type $\Pi[\imath : \mathsf{size}] \to \mathsf{nat}^{\imath} \to \mathsf{nat}^{\infty} \to \mathsf{nat}^{\imath}$.

Comparing $\mathrm{F}\widehat{\omega}$ and MiniAgda against CIC$\widehat{\ }$ and CIC$\widehat{\ }_{-}$, the main difference is the use of explicit sizes in the former systems. We see as an advantage the implicit nature of sizes in CIC$\widehat{\ }$, since they can be inferred and, thus, are completely transparent to the user. On the other hand, the advantage of having to explicitly declare sizes is the possibility to express precise types for functions that are not possible in CIC$\widehat{\ }$.

It is of interest to compare thoroughly the approach of implicit products of MiniAgda against the approach of CIC$\widehat{\ }$. It seems that, after deleting implicit arguments in MiniAgda, one obtain similar definitions to those of CIC$\widehat{\ }$. However, this would require a precise definition of MiniAgda and its termination checker (which seems to be a more complex than CIC$\widehat{\ }$). As far as I am aware, a formal description of MiniAgda has not been developed yet.

$\mathrm{F}\widehat{\omega}$ and MiniAgda allow a more relaxed notion of valid type for fixpoint than CIC$\widehat{\ }_{-}$. For example, the function that computes the maximum of two natural numbers can be given the type $\mathsf{nat}^{\imath} \to \mathsf{nat}^{\imath} \to \mathsf{nat}^{\imath}$. This is not possible in our definition of CIC$\widehat{\ }_{-}$. Semantically, such types are justified by the principle of semi-continuity [2]. In the case of CIC$\widehat{\ }_{-}$, although the type $\mathsf{nat}^{\imath} \to \mathsf{nat}^{\imath} \to \mathsf{nat}^{\imath}$ is not allowed for recursion, the model we present in Chapter 4 supports it. It seems fair to think that a similar criterion to Abel's semi-continuity can be defined for CIC$\widehat{\ }_{-}$.

Other approaches to termination. Wahlstedt [82] presents a Martin-Löf type theory with first-order datatypes, where termination of recursive functions is ensured by the size-change principle [50]. It is a powerful method that handles lexicographic orders and permuting arguments. However, it cannot handle the quicksort function.

Chapter 3

Metatheory of CIC$\widehat{\sqsubseteq}$

In this chapter we show some basic metatheoretical results about CIC$\widehat{\sqsubseteq}$. For technical reasons explained in Chapter 4, we introduce an annotated variant of CIC$\widehat{\sqsubseteq}$ called ECIC$\widehat{\sqsubseteq}$(Sect. 3.2). We prove the equivalence between both presentations in Sect. 3.2.4). As a consequence, we can prove transfer metatheoretical properties from ECIC$\widehat{\sqsubseteq}$ to CIC$\widehat{\sqsubseteq}$. In particular, SN for CIC$\widehat{\sqsubseteq}$ is a corollary of SN for ECIC$\widehat{\sqsubseteq}$. Chapter 4 is devoted to the proof of SN for ECIC$\widehat{\sqsubseteq}$.

The main properties about CIC$\widehat{\sqsubseteq}$ that we prove in this chapter are the following:

- Weakening (Lemma 3.16). It states that typing is preserved if we add hypotheses to the context: if $\Gamma \vdash M : T$ and $\Delta \leq \Gamma$, then $\Delta \vdash M : T$. (cf Def. 3.15).
- Stage substitution (Lemma 3.17). It states that typing is preserved under replacement of a size variable: if $\Gamma \vdash M : T$ then $\Gamma\,[\imath := s] \vdash M\,[\imath := s] : T\,[\imath := s]$.
- Substitution (Lemma 3.20). It states that typing is preserved under replacement of a term variable: if $\Gamma(x : T)\Delta \vdash M : U$, $\Gamma \vdash N : T$, and $\mathrm{SV}(N) = \emptyset$, then $\Gamma\Delta\,[x := N] \vdash M\,[x := N] : T\,[x := N]$.
- Subject reduction (Lemma 3.24). It states that typing is preserved under reduction: if $\Gamma \vdash M : T$ and $M \to M'$ then $\Gamma \vdash M' : T$.
- Strengthening (Lemma 3.26). It states that typing is preserved if we remove unused hypotheses from the context: if $\Gamma(x : T)\Delta \vdash M : U$ and $x \notin \mathrm{FV}(\Delta, M, U)$, then $\Gamma\Delta \vdash M : U$.
- Strong normalization (Corollary 3.66). It states that well-typed terms are strongly normalizing.
- Logical consistency (Corollary 3.67). It states that it is not possible to prove false statements in the empty context: there is no term M such that $\vdash M : \mathsf{False}$.

3.1 Basic Metatheory

In this section we prove the basic metatheory of CIC$\widehat{\sqsubseteq}$. We begin by proving some metatheoretical results on the subtyping relation.

Subtyping

The following auxiliary results on stages are easy to prove by induction.

Lemma 3.1. *If $\widehat{r} \sqsubseteq \widehat{s}$, then $r \sqsubseteq s$.*

Proof. By induction on the substage relation. □

Lemma 3.2. *If $r_1 \sqsubseteq r_2$, then $r_1[\imath := s] \sqsubseteq r_2[\imath := s]$.*

Proof. By induction on the substage relation. □

We prove a Generation Lemma and Substitution Lemma for the subtyping relation.

Lemma 3.3 (Generation lemma for subtyping)**.**

1. *If $T_1 \leq T_2$ and $T_2 \approx I^s(\vec{p_2}, \vec{a_2})$, then $T_1 \approx I^r(\vec{p_1}, \vec{a_1})$, $r \sqsubseteq s$, $\vec{p_1} \leq^{I.\vec{\nu}} \vec{p_2}$ and $\vec{a_1} \approx \vec{a_2}$.*
2. *If $T_1 \leq T_2$ and $T_1 \approx I^r(\vec{p_1}, \vec{a_1})$, then $T_2 \approx I^s(\vec{p_2}, \vec{a_2})$, $r \sqsubseteq s$, $\vec{p_1} \leq^{I.\vec{\nu}} \vec{p_2}$, and $\vec{a_1} \approx \vec{a_2}$.*
3. *If $T_1 \leq T_2$ and $T_2 \equiv \Pi x : A_2.B_2$, then $T_1 \approx \Pi x : A_1.B_1$, $A_2 \leq A_1$ and $B_1 \leq B_2$.*
4. *If $T_1 \leq T_2$ and $T_1 \equiv \Pi x : A_1.B_1$, then $T_2 \approx \Pi x : A_2.B_2$, $A_2 \leq A_1$ and $B_1 \leq B_2$.*

Proof. By induction on the subtyping derivation. □

Lemma 3.4 (Stage substitution for subtyping)**.** *If $T \leq U$, then $T[\imath := s] \leq U[\imath := s]$.*

Proof. By induction on the subtyping derivation. In rule (st-conv) we use the fact that reduction is preserved under stage substitution. In rule (st-ind) we use Lemma 3.2. □

Lemma 3.5. *If $T_1 \leq T_2$, then $|T_1| \approx |T_2|$.*

Proof. By induction on the subtyping derivation. In rule (st-conv) we use the fact that reduction is preserved under erasure. □

We give an alternative definition of subtyping that does not have a transitivity rule. It is used in Sect. 3.2.4, where we prove the equivalence between CIC$\hat{}_{-}$ and ECIC$\hat{}_{-}$.

Definition 3.6 (Subtyping (alternative))**.** *The alternative subtyping relation, denoted by $\leq_a$, is defined by the following rules:*

$$\textit{(ast-conv)}\quad \frac{T \approx U}{T \leq_a U}$$

$$\textit{(ast-ind)}\quad \frac{T \to^* I^s(\vec{p_1}, \vec{a_1}) \qquad U \to^* I^r(\vec{p_2}, \vec{a_2}) \qquad s \sqsubseteq r \qquad \vec{p_1} \leq_a^{I.\vec{\nu}} \vec{p_2} \qquad \vec{a_1} \approx \vec{a_2}}{T \leq_a U}$$

$$\textit{(ast-prod)}\quad \frac{T \to^* \Pi x : T_1.T_2 \qquad U \to^* \Pi x : U_1.U_2 \qquad U_1 \leq_a T_1 \qquad T_2 \leq_a U_2}{T \leq_a U}$$

$$\textit{(vast-pos)}\quad \frac{T_1 \leq_a U_1 \qquad \vec{T} \leq_a^{\vec{\nu}} \vec{U}}{T_1, \vec{T} \leq_a^{+\vec{\nu}} U_1, \vec{U}}$$

$$\textit{(vast-neg)}\quad \frac{U_1 \leq_a T_1 \qquad \vec{T} \leq_a^{\vec{\nu}} \vec{U}}{T_1, \vec{T} \leq_a^{-\vec{\nu}} U_1, \vec{U}}$$

$$\textit{(vast-inv)}\quad \frac{T_1 \approx U_1 \qquad \vec{T} \leq_a^{\vec{\nu}} \vec{U}}{T_1, \vec{T} \leq_a^{\circ\vec{\nu}} U_1, \vec{U}}$$

$$\textit{(vast-conv)}\quad \frac{\vec{T} \approx \vec{U}}{\vec{T} \leq_a^{\emptyset} \vec{U}}$$

We prove that both definitions are equivalent. Note that it is easy to prove that $T \leq_a U \Rightarrow T \leq U$ (by induction on the subtyping relation). For the other direction, we first show that $\leq_a$ is transitive. We use the following lemma.

Lemma 3.7. *If $T \approx T' \leq_a U' \approx U$, then $T \leq_a U$, with a derivation of the same height. If $\vec{T} \approx \vec{T'} \leq_a^{\vec{v}} \vec{U'} \approx \vec{U}$, then $\vec{T} \leq_a^{\vec{v}} \vec{U}$, with a derivation of the same height.*

Proof. We proceed by simultaneous induction on the subtyping derivation. We proceed by induction on the subtyping derivation. We consider the case of inductive type. $T' \leq_a U'$ is derived from $T' \to^* I^s\left(\vec{t_1}\right)\ \vec{u_1}$, $U' \to^* I^r\left(\vec{t_2}\right)\ \vec{u_2}$, $s \sqsubseteq r$, $\vec{t_1} \leq_a^{\vec{v}} \vec{t_2}$, and $\vec{u_1} \approx \vec{u_2}$. By confluence, there exists $\vec{t_1'}$, $\vec{t_2'}$, $\vec{u_1'}$, and $\vec{u_2'}$ such that $T, I^s\left(\vec{t_1}\right)\ \vec{u_1} \to^* I^s\left(\vec{t_1'}\right)\ \vec{u_1'}$, and $U, I^r\left(\vec{t_2}\right)\ \vec{u_2} \to^* I^r\left(\vec{t_2'}\right)\ \vec{u_2'}$. We have then $\vec{t_1} \to^* \vec{t_1'}$ and $\vec{t_2} \to^* \vec{t_2'}$. By IH, $\vec{t_1'} \leq_a^{\vec{v}} \vec{t_2'}$ with a derivation of the same height. Similarly, $\vec{u_1'} \approx \vec{u_2'}$. The result follows.

The case of product follows by a similar reasoning. The rest of the cases follows easily by IH and confluence. □

The previous lemma is used in the following proof of transitivity of the alternative subtyping relation.

Lemma 3.8 (Transitivity of $\leq_a$)**.** *If $T_1 \leq_a T_2$ and $T_2 \leq_a T_3$, then $T_1 \leq_a T_3$.*

Proof. We proceed by induction on the sum of the height of the derivations of $T_1 \leq_a T_2$, and $T_2 \leq_a T_3$, and case analysis in the last rule used. If one of the derivations end with the (ast-conv), the result follows from the previous lemma. We consider the case where both derivations end with rule (ast-prod). $T_1 \leq_a T_2$ is derived from $T_1 \to^* \Pi x : U_1.W_1$, $T_2 \to^* \Pi x : U_2.W_2$, $U_2 \leq_a U_1$, $W_1 \leq_a W_2$. $T_2 \leq_a T_3$ is derived from $T_2 \to^* \Pi x : U_2'.W_2'$, $T_3 \to^* \Pi x : U_3.W_3$, $U_3 \leq_a U_2'$, $W_2' \leq_a W_3$. By confluence, $U_2 \approx U_2'$ and $W_2 \approx W_2'$. Applying the previous lemma, we have $U_2' \leq_a U_1$ with a derivation of the same height as the derivation of $U_2 \leq_a U_1$. Similarly, we have a derivation of $W_1 \leq_a W_2'$. We can apply the IH, and obtain $U_3 \leq_a U_1$ and $W_1 \leq_a W_3$. The result follows by applying rule (ast-prod). □

The other direction in the proof of equivalence between both subtyping relations given follows easily using the previous lemma for the difficult case of transitivity.

Lemma 3.9. *If $T \leq U$, then $T \leq_a U$.*

Proof. By induction on the subtyping derivation, using the previous lemma for rule (st-trans). □

Substitutions

Substitutions generalize the concept of substitution of free variables of a term. They are not strictly necessary in this chapter, but will be useful in Chapter 6.

A *pre-substitution* is a function σ, from variables to terms, such that $\sigma(x) \neq x$ for a finite number of variables x. The set of variables for which $\sigma(x) \neq x$ is the *domain* of σ and is denoted $\mathrm{dom}\,(\sigma)$. Given a term N and a pre-substitution σ, we write $N\sigma$ to mean the term obtained by simultaneously substituting every free variable x of N with $\sigma(x)$. The set of free variables of a pre-substitution σ is defined as $\mathrm{FV}(\sigma) = \cup_{x \in \mathrm{dom}(\sigma)} \mathrm{FV}(x\sigma)$.

A *substitution* is an idempotent pre-substitution. I.e., a pre-substitution σ is a substitution if for every term N, $N\sigma \equiv N\sigma\sigma$; or equivalently, if $\mathrm{FV}(\sigma) \cap \mathrm{dom}(\sigma) = \emptyset$. We use $\sigma, \rho, \ldots$ to denote substitutions.

Substitutions can be composed: if σ and ρ are substitutions, $\sigma\rho$ denotes a substitution such that $x\sigma\rho = (x\sigma)\rho$. Two substitutions σ and ρ are convertible, written $\sigma \approx \rho$, if for every variable x, $x\sigma \approx x\rho$.

Given a substitution σ and contexts Γ, Δ, we say that σ is *well-typed from* Γ *to* Δ, written $\sigma : \Gamma \to \Delta$, if $\mathsf{WF}(\Gamma)$, $\mathsf{WF}(\Delta)$, $\mathrm{dom}(\sigma) \subseteq \mathrm{dom}(\Gamma)$, and for every $(x : T) \in \Gamma$, $\Delta \vdash x\sigma : T\sigma$.

We use $\Gamma \vdash \sigma : \Delta \to \Theta$ to denote a well-typed substitution from $\Gamma\Delta$ to $\Gamma\Theta$ such that $\mathrm{dom}(\sigma) \subseteq \mathrm{dom}(\Delta)$. We use $\Gamma \vdash \sigma : \Delta$ to denote $\Gamma \vdash \sigma : \Delta \to []$.

Typing judgment

We prove a series of lemmas on typing judgment leading to Subject Reduction (Lemma 3.24). These include useful lemmas such as Stage Substitution (Lemma 3.17) and Substitution (Lemma 3.20). We first prove the Generation Lemma.

Lemma 3.10 (Generation)**.**

1.	$\Gamma \vdash x : U$	$\Rightarrow (x : T) \in \Gamma \wedge T \leq U$
2.	$\Gamma \vdash u : U$	$\Rightarrow u' \leq U \wedge (u, u') \in \mathsf{Axiom}$
3.	$\Gamma \vdash \Pi x : T_1.T_2 : U$	$\Rightarrow \Gamma \vdash T_1 : u_1 \wedge \Gamma(x : T_1) \vdash T_2 : u_2 \wedge u_3 \leq U \wedge$ $(u_1, u_2, u_3) \in \mathsf{Rule}$
4.	$\Gamma \vdash \lambda x : T^\circ.M : U$	$\Rightarrow \Gamma(x : T') \vdash M : W \wedge \Pi x : T'.W \leq U \wedge$ $\Gamma \vdash \Pi x : T'.W : u \wedge \lvert T' \rvert = T^\circ \wedge \mathrm{SV}(M) = \emptyset$
5.	$\Gamma \vdash M\, N : U$	$\Rightarrow \Gamma \vdash M : \Pi x : T.W \wedge \Gamma \vdash N : T \wedge$ $W[x := N] \leq U \wedge \mathrm{SV}(N) = \emptyset$
6.	$\Gamma \vdash I^s(\vec{p}, \vec{a}) : U$	$\Rightarrow \Gamma \vdash \vec{p} : \mathsf{params}(I) \wedge \Gamma \vdash \vec{a} : \mathsf{argsInd}_I(\vec{p}) \wedge \mathsf{sort}(I) \leq U$
7.	$\Gamma \vdash C(\vec{p^\circ}, \vec{a}) : U$	$\Rightarrow \Gamma \vdash \vec{p'} : \mathsf{params}(I) \wedge \Gamma \vdash \vec{a} : \mathsf{argsConstr}^s_C(\vec{p}) \wedge$ $\lvert\vec{p'}\rvert \equiv \vec{p^\circ} \wedge \mathrm{SV}(\vec{a}) = \emptyset \wedge \mathsf{typeConstr}^s_C(\vec{p'}, \vec{a}) \leq U$
8.	$\Gamma \vdash \mathsf{case}_{P^\circ}\ x := M\ \mathsf{in}\ I(\vec{p^\circ}, \vec{y})$ $\mathsf{of}\langle C_i \Rightarrow N_i\rangle_i : U$	$\Rightarrow \Gamma \vdash M : I^{\widehat{s}}\left(\vec{p'}, \vec{a}\right) \wedge$ $\Gamma(\mathsf{caseType}^s_I(\vec{p'}, \vec{y}, x)) \vdash P' : u \wedge$ $\Gamma \vdash N_i : \mathsf{branch}^s_{C_i}(\vec{p}, \vec{y}.x.P) \wedge \lvert\vec{p'}\rvert \equiv \vec{p^\circ} \wedge$ $\lvert P' \rvert \equiv P^\circ \wedge P[\vec{y} := \vec{a}][x := M] \leq U$
9.	$\Gamma \vdash \mathsf{fix}_n\ f : W^\star := M : U$	$\Rightarrow W^\star = \lvert T \rvert^\imath$ *with* $T = \Pi\Delta.\Pi x : I^\iota\, \vec{u}.U \wedge \imath\ \mathsf{pos}\ U \wedge$ $\#\Delta = n - 1 \wedge \imath \notin \mathrm{SV}(\Gamma, \Delta, \vec{u}, M) \wedge \Gamma \vdash T : u \wedge$ $\Gamma(f : T) \vdash M : T[\imath := \widehat{\imath}] \wedge T[\imath := s] \leq U \wedge$ $\mathrm{SV}(M) = \emptyset$

Proof. By induction on the type derivation, using transitivity of subtyping. □

The following two lemmas state that contexts and subcontexts are well-formed in a typing judgment.

Lemma 3.11 (Well-formedness of context)**.** *If* $\Gamma \vdash M : T$*, then* $\mathsf{WF}(\Gamma)$*.*

Proof. By induction on the type derivation. □

Lemma 3.12. *If* $\Gamma_0(x : T)\Gamma_1 \vdash M : U$*, then there exists a subderivation* $\Gamma_0 \vdash T : u$ *for some sort* u*. If* $\mathsf{WF}(\Gamma_0(x : T)\Gamma_1)$*, then there exists a subderivation* $\Gamma_0 \vdash T : u$ *for some sort* u*.*

Proof. By induction on the type derivation. □

The following lemma states that we can add hypothesis to a context preserving valid judgments.

Lemma 3.13. *Let* $\Gamma \vdash M : T$ *and* Δ *a well-formed context that contains all hypotheses in* Γ*. Then* $\Delta \vdash M : T$*.*

Proof. By induction on the type derivation. In rules (abs) and (case) we use Lemma 3.12. □

Lemma 3.14. *If* $\mathsf{WF}(\Gamma)$ *and* $(x : T) \in \Gamma$*, then* $\Gamma \vdash T : u$ *for some sort* u*.*

Proof. Γ is of the form $\Gamma_0(x : T)\Gamma_1$. The result follows from Lemma 3.12 and Lemma 3.13. □

We define an order relation between contexts.

Definition 3.15 (Subcontext)**.** *We say that* Δ *is a subcontext of* Γ*, denoted* $\Delta \leq \Gamma$*, if, for all* $(x : T) \in \Gamma$*, there exists* T' *such that* $(x : T') \in \Delta$ *with* $T' \leq T$*.*

Lemma 3.16 (Context conversion)**.** *If* $\Gamma \vdash M : T$*,* $\mathsf{WF}(\Delta)$*, and* $\Delta \leq \Gamma$*, then* $\Delta \vdash M : T$*.*

Proof. By induction on the typing derivation. In rules (abs) and (case) we we use Lemma 3.12. In rule (var) we use Lemma 3.12 and apply rule (conv). □

In the following we prove several lemmas regarding substitutions. We prove a general lemma on substitutions (Lemma 3.18), from which the usual version of Substitution (Lemma 3.20) follows. We begin by proving that typing is preserved under stage substitution.

Lemma 3.17 (Stage substitution)**.** *If* $\Gamma \vdash M : T$ *then* $\Gamma\,[\imath := s] \vdash M\,[\imath := s] : T\,[\imath := s]$*.*

Proof. We prove first the following result:

(*) *if* $\Gamma \vdash M : T$ *and* $\jmath$ *is fresh in* Γ*,* M*, and* T*, then* $\Gamma\,[\imath := \jmath] \vdash M\,[\imath := \jmath] : T\,[\imath := \jmath]$ *with the same height.*

We proceed by induction on the type derivation. All cases are easy by applying the IH. In rule (fix) we use the IH twice if the size variable introduced in the rule is $\jmath$. In rule (conv) we use Lemma 3.4. Then we prove the following result:

(**) *if* $\Gamma \vdash M : T$ *and* s *does not contain* $\imath$*, then* $\Gamma\,[\imath := s] \vdash M\,[\imath := s] : T\,[\imath := s]$*.*

We proceed by induction on the height h of the type derivation. We only consider the rule (fix). For the rest of the cases the result follows easily by applying the IH. In rule (conv) we use Lemma 3.4.

We have $M \equiv \mathsf{fix}_n\ f : |T_1|^{\jmath} := M_1$, $T \equiv T_1\,[\jmath := r]$, and the following subderivations of height $h - 1$:

$$\frac{\Gamma(f : T_1) \vdash M_1 : T_1\,[\jmath := \widehat{\jmath}] \qquad \Gamma \vdash T_1 : u}{}$$

where $T_1 \equiv \Pi\Delta(x : I^{\jmath}\,(\vec{p}, \vec{u})).U$, and $\jmath \notin \mathrm{SV}(\Gamma, \Delta, \vec{p}, \vec{u}, M_1)$. Without loss of generality, we can assume that s does not contain $\jmath$. Otherwise, we take a size variable κ that does not appear in s and use (*) to substitute $\jmath$ with κ in the derivations of M_1 and T. Since this substitution preserves the height of the derivation, we can still apply the IH.

By IH, we have

$$\Gamma(f : T_1)\,[\imath := s] \vdash M_1\,[\imath := s] : T_1\,[\jmath := \widehat{\jmath}]\,[\imath := s]$$
$$\Gamma\,[\imath := s] \vdash T_1\,[\imath := s] : u$$

Since $\jmath$ does not appear in s, $T_1\,[\jmath := \widehat{\jmath}]\,[\imath := s] \equiv T_1\,[\imath := s]\,[\jmath := \widehat{\jmath}]$. Applying rule (fix) we have

$$\Gamma\,[\imath := s] \vdash (\mathsf{fix}_n\ f : |T_1|^{\jmath} := M_1\,[\imath := s]) : T_1\,[\imath := s]\,[\jmath := r\,[\imath := s]]$$

where $T_1\,[\imath := s]\,[\jmath := r\,[\imath := s]] \equiv T\,[\jmath := r]\,[\imath := s]$ as expected.

This proves (**). The main result now follows easily. We consider two cases: if s does not contain $\imath$, the result follows from (**). If s contains $\imath$, we apply (*) and substitute $\imath$ by a fresh size variable κ that does not appear in s. Then we apply (**) substituting κ by s. □

Lemma 3.18 (General Substitution)**.** *If* $\Gamma \vdash M : T$, $\sigma : \Gamma \to \Delta$, *and* $\mathrm{SV}(\sigma) = \emptyset$, *then* $\Delta \vdash M\sigma : T\sigma$.

Proof. By induction on the typing derivation. In rules (abs) and (case) we use Lemma 3.12. We consider rules (prod) and (ind).

(prod). $\Gamma \vdash M : T$ is $\Gamma \vdash \Pi x : U_1.U_2 : u$ derived from $\Gamma \vdash U_1 : u_1$, $\Gamma(x : U_1) \vdash U_2 : u_2$ and $(u_1, u_2, u) \in \mathsf{Rule}$.
By IH, $\Delta \vdash U_1\sigma : u_1$. Note that $\sigma : \Gamma(x : U_1) \to \Delta(x : U_1\sigma)$, where we assume that $x \notin \mathrm{dom}\,(\Delta)$. Then, by IH, $\Delta(x : U_1\sigma) \vdash U_2\sigma : u_2$. The result follows by applying rule (prod).

(ind). $\Gamma \vdash M : T$ is $\Gamma \vdash I^s\,(\vec{p}, \vec{a}) : \mathsf{sort}(I)$ derived from $\Gamma \vdash \vec{p} : \mathsf{params}(I)$, $\Gamma \vdash \vec{a} : \mathsf{argsInd}_I(\vec{p})$, and $\mathsf{simple}(I^s\,(\vec{p}, \vec{a}))$. By IH, $\Delta \vdash \vec{p}\sigma : \mathsf{params}(I)$ (note that $\mathrm{FV}(\mathsf{params}(I)) \cap \mathrm{FV}(\sigma) = \emptyset$) and $\Delta \vdash \vec{a}\sigma : \mathsf{argsInd}_I(\vec{p})\sigma$. By definition of $\mathsf{argsInd}$, $\mathsf{argsInd}_I(\vec{p})\sigma \equiv \mathsf{argsInd}_I(\vec{p}\sigma)$. Since $\mathrm{SV}(\sigma) = \emptyset$, we have $\mathsf{simple}(I^s\,(\vec{p}, \vec{a})\,\sigma)$. The result follows by applying rule (ind).

The case of rule (constr) is similar to (ind), and (fix) is similar to (prod). The rest of the rules follow easily by IH. □

Lemma 3.19 (Composition of substitutions)**.** *If* $\sigma : \Gamma \to \Delta$ *and* $\rho : \Delta \to \Theta$, *then* $\sigma\rho : \Gamma \to \Theta$.

Proof. It is clear that $\mathsf{WF}(\Gamma)$ and $\mathsf{WF}(\Theta)$. Let $(x : T) \in \Gamma$. Then, $\Delta \vdash x\sigma : T\sigma$, by definition of well-typed substitution. By Lemma 3.18, $\Theta \vdash x\sigma\rho : T\sigma\rho$. □

We now prove the usual version of the Substitution Lemma.

Lemma 3.20 (Substitution)**.** *If* $\Gamma(x : T)\Delta \vdash M : U$ *and* $\Gamma \vdash N : T$, *then* $\Gamma(\Delta\,[x := N]) \vdash M\,[x := N] : T\,[x := N]$.

Proof. We prove that $[x := N] : \Gamma(x : T)\Delta \to \Gamma(\Delta\,[x := N])$, from $\mathsf{WF}(\Gamma(x : T)\Delta)$ and $\Gamma \vdash N : T$. Then the result follows by General Substitution (Lemma 3.18). We proceed by induction on the length of Δ. For the base case $\#\Delta = 0$, then $[x := N] : \Gamma(x : T) \to \Gamma$ follows from $\Gamma \vdash N : T$. For the inductive case $\#\Delta = n + 1$, let $\Delta \equiv \Delta_0(y : T_0)$.

By IH, $[x := N] : \Gamma(x : T)\Delta_0 \to \Gamma(\Delta_0\,[x := N])$. From $\mathsf{WF}(\Gamma(x : T)\Delta)$, by Lemma 3.12, $\Gamma(x : T)\Delta_0 \vdash T_0 : u$ for some sort u. By Lemma 3.18, $\Gamma(\Delta_0\,[x := T]) \vdash T_0\,[x := N] : u$. Then, $\mathsf{WF}(\Gamma(\Delta_0\,[x := T])(y : T_0\,[x := N]))$ and the result follows from the definition of well-typed substitution. □

The following lemma states that the type of a well-typed term is itself well typed.

Lemma 3.21 (Type validity)**.** *If* $\Gamma \vdash M : T$*, then* $\Gamma \vdash T : u$*.*

Proof. By induction on the type derivation. In rule (abs), we use Lemma 3.14. In rule (fix), we use Lemma 3.17. In rule (case), we use Generation (Lemma 3.10) and Lemma 3.18. In rule (constr), we use Lemma 3.18. □

Because of the use of erased terms in type positions, uniqueness of types is only valid modulo erasure of size annotations.

Lemma 3.22 (Uniqueness of types)**.** *If* $\Gamma \vdash M : T_1$ *and* $\Gamma \vdash M : T_2$*, then* $|T_1| \approx |T_2|$*.*

Proof. By induction on the structure of M. We consider the most relevant cases. The cases not considered follow easily by applying Generation (Lemma 3.10).

Abstraction. $M \equiv \lambda x : U^\circ.M_1$. By Generation on both derivations, there exists U_1, U_2, W_1, W_2 such that $|U_1| \equiv |U_2| \equiv U^\circ$, $\Gamma(x : U_1) \vdash M_1 : W_1$, $\Gamma(x : U_2) \vdash M_1 : W_2$, $\Pi x : U_1.W_1 \leq T_1$, $\Pi x : U_2.W_2 \leq T_2$. By IH on the type derivations for M_1, $|W_1| \approx |W_2|$. Therefore, $|T_1| \approx |\Pi x : U_1.W_1| \approx |\Pi x : U_2.W_2| \approx |T_2|$, using Lemma 3.5.

Application. $M \equiv M_1\,N_1$. By Generation on both derivations, there exists U_1, U_2, W_1, W_2 such that, for $i = 1, 2$, $\Gamma \vdash M_1 : \Pi x : U_i.W_i$, $\Gamma \vdash N_1 : U_i$, and $U_i\,[x := N_i] \leq T_i$. By IH, $|U_1| \equiv |U_2|$, therefore $|T_1| \equiv |U_1\,[x := N_1]\,| \equiv |U_2\,[x := N_1]\,| \equiv |T_2|$.

Product. The result follows easily by IH, noting that Rule is injective, in the sense that $(u_1, u_2, u_3) \in \mathsf{Rule}$ and $(u_1, u_2, u_3') \in \mathsf{Rule}$, implies $u_3 = u_3'$.

Case. $M \equiv \mathsf{case}_{P^\circ}\ x := M_1\ \mathsf{in}\ I\,(\vec{p^\circ}, \vec{y})\ \mathsf{of}\ \langle C_i \Rightarrow N_i \rangle_i$. The result follows by applying Generation on both derivations and the IH on M_1.

□

In order to prove Subject Reduction, we need a technical lemma that we use in the case of fixpoint reduction.

Lemma 3.23. *Let* $\Gamma \vdash M : \Pi\Delta_0.T_0$ *and* $\Gamma \vdash M\,\vec{N} : U$ *be valid judgments, with* $\#\vec{N} = \#\Delta_0$*. Then, there exists* Δ *and* T *with* $|\Delta| \equiv |\Delta_0|$ *and* $|T| \equiv |T_0|$ *such that* $\Gamma \vdash M : \Pi\Delta.T$*,* $\Gamma \vdash \vec{N} : \Delta$*, and* $T\left[\mathrm{dom}\,(\Delta) := \vec{N}\right] \leq U$*.*

The statement of this lemma looks a bit strange. A more natural way of stating it would be:

let $\Gamma \vdash M : \Pi\Delta_0.T_0$ *and* $\Gamma \vdash M\,\vec{N} : U$ *be valid judgments, with* $\#\vec{N} = \#\Delta_0$*. Then there is a derivation of* $\Gamma \vdash \vec{N} : \Delta_0$*, and* $T_0\left[\mathrm{dom}\,(\Delta_0) := \vec{N}\right] \leq U$*.*

But this statement is not true. The problem is that type uniqueness is valid up-to erasure of size information. In other words, a term can have two non-comparable types (by $\leq$). For example, consider $M \equiv \lambda x : \mathsf{nat}.x$ and $\vec{N} \equiv y$; the following judgments are valid:

$$(y : \mathsf{nat}^{\infty}) \vdash (\lambda x : \mathsf{nat}.x) : \mathsf{nat}^{i} \to \mathsf{nat}^{i}$$
$$(y : \mathsf{nat}^{\infty}) \vdash (\lambda x : \mathsf{nat}.x)\, y : \mathsf{nat}^{\infty}$$

However, the judgment $(y : \mathsf{nat}^{\infty}) \vdash y : \mathsf{nat}^{i}$ is not valid.

Proof of Lemma 3.23. We proceed by induction on $n = \#\Delta$. If $n = 0$, the result follows immediately. We consider the case $n = k+1$. Let $\vec{N} \equiv \langle N_i \rangle_{i=1..k}\, N_{k+1}$, and $\Delta_0 \equiv \Delta_1(x : T_1)$. By Generation on the judgment $\Gamma \vdash M\, \vec{N} : U$, there exists U_1, U_2 such that

$$\Gamma \vdash M\, \langle N_i \rangle_{i=1..k} : \Pi x : U_1.U_2 \tag{3.1}$$
$$\Gamma \vdash N_{k+1} : U_1 \tag{3.2}$$
$$U_2\,[x := N_{k+1}] \leq U \ . \tag{3.3}$$

From the IH applied to $\langle N_i \rangle_{i=1..k}$, there exists Δ' and T' with $|\Delta'| \equiv |\Delta_1|$ and $|T'| \equiv |\Pi x : T_1.T_0|$ such that

$$\Gamma \vdash M : \Pi \Delta'.T' \tag{3.4}$$
$$\Gamma \vdash \langle N_i \rangle_{i=1..k} : \Delta' \tag{3.5}$$
$$T' \left[\mathrm{dom}\left(\Delta'\right) := \langle N_i \rangle_{i=1..k}\right] \leq \Pi x : U_1.U_2 \ . \tag{3.6}$$

Let $T' \equiv \Pi x : T'_1.T'_0$. Take $\Delta \equiv \Delta'(x : T'_1)$ and $T \equiv T'_0$. Let us prove that Δ and T satisfy the required conditions. It is clear that $|\Delta| \equiv |\Delta_0|$, $|T'_0| \equiv |T_0|$, and, from (3.4), $\Gamma \vdash M : \Pi\Delta.T$. From (3.6), $U_1 \leq T'_1\,[\mathrm{dom}\,(\Delta') := \langle N_i \rangle_{i=1..k}]$. Then, applying (conv) to (3.2), we have $\Gamma \vdash N_{k+1} : T'_1\,[\mathrm{dom}\,(\Delta') := \langle N_i \rangle_{i=1..k}]$. (The last type is well-typed by applying Type validity and Generation to (3.4)). Combined with (3.5), we have $\Gamma \vdash \langle N_i \rangle_{i=1..k+1} : \Delta'(x : T'_1)$. Finally, from (3.6) and (3.3), we have $T'_0\,[\mathrm{dom}\,(\Delta'(x : T'_1)) := \langle N_i \rangle_{i=1..k+1}] \leq U$.

□

Lemma 3.24 (Subject reduction)**.** *If* $\Gamma \vdash M : T$ *and* $M \to M'$ *then* $\Gamma \vdash M' : T$

Proof. By induction on the type derivation and case analysis on the reduction rule. The interesting cases are when reduction occurs at the head. For compatible closure rules, the result follows by applying the IH; in rules (abs), (prod), (case), and (fix) we use Lemma 3.12 and Context conversion (Lemma 3.16).

We consider the cases of rules (app) and (case).

(app). $\Gamma \vdash M : T$ is $\Gamma \vdash M_1\, M_2 : T_2\,[x := M_2]$ derived from $\Gamma \vdash M_1 : \Pi x : T_1.T_2$ and $\Gamma \vdash M_2 : T_1$. We do a case analysis on the reduction $M \to M'$.

I. $M_1 \equiv \lambda x : U^{\circ}.N$ *and* $M' \equiv N\,[x := M_2]$.

Applying Generation on the derivation of M_1 we obtain

$$\Gamma(x : U_1) \vdash N : U_2 \tag{3.7}$$
$$\Pi x : U_1.U_2 \leq \Pi x : T_1.T_2 \tag{3.8}$$
$$|U_1| \equiv U^{\circ} \tag{3.9}$$
$$\mathrm{SV}(N) = \emptyset \tag{3.10}$$

By applying Generation of subtyping in (3.8) we have $T_1 \leq U_1$ and $U_2 \leq T_2$. Using the (conv) rule we obtain $\Gamma \vdash M_2 : U_1$, and substituting in (3.7) we get $\Gamma \vdash N\,[x := M_2] : U_2\,[x := M_2]$. The result follows by applying the (conv) rule, noting that $U_2\,[x := M_2] \leq T_2\,[x := M_2]$.

II. $M_1 \equiv (\mathsf{fix}_n\ f : T_1^\star := N)\,\vec{b}$ *and* $M_2 \equiv C(\vec{p^\circ}, \vec{a})$. *Also,*

$$M' \equiv N\,[f := \mathsf{fix}_n\ f : T_1^\star := N]\ \ \vec{b}\ (C(\vec{q^\circ}, \vec{a}))$$

We have a subderivation $\Gamma \vdash \mathsf{fix}_n\ f : T_1^\star := N : W$. By Generation, we have

$$\Gamma(f : T_1) \vdash M : T_1\,[\imath := \widehat{\imath}], \tag{3.11}$$

$$\Gamma \vdash T_1 : u \tag{3.12}$$

where $T_1^\star \equiv |T_1|^\imath$, $T_1 \equiv \Pi\Delta.\Pi x : I^\imath\,(\vec{p}, \vec{u}).U$, $\imath$ pos U, $\imath \notin \mathrm{SV}(\Gamma, \Delta, \vec{p}, \vec{u}, M)$, and $T_1\,[\imath := s] \leq W$. By Lemma 3.23, we have

$$\Gamma \vdash \vec{b}\,(C(\vec{q^\circ}, \vec{a})) : \Delta(x : I^s\,(\vec{p}, \vec{u})) \tag{3.13}$$

$$U\,[\imath := s]\left[\mathrm{dom}\,(\Delta) := \vec{b}\right]\,[x := C(\vec{q^\circ}, \vec{a})] \leq T \tag{3.14}$$

By inversion on (3.13), we have $\Gamma \vdash C(\vec{q^\circ}, \vec{a}) : I^s\,(\vec{p}, \vec{u})\left[\mathrm{dom}\,(\Delta) := \vec{b}\right]$. By Generation, there exists a stage r such that $\widehat{r} \sqsubseteq s$, and

$$\Gamma \vdash C(\vec{q^\circ}, \vec{a}) : I^{\widehat{r}}\left(\vec{q}, \vec{t}\right) \tag{3.15}$$

with $I^{\widehat{r}}\left(\vec{q}, \vec{t}\right) \leq I^s\,(\vec{p}, \vec{u})\left[\mathrm{dom}\,(\Delta) := \vec{b}\right]$. Then there exists s_1 such that $\widehat{s_1} \sqsubseteq s$, and $I^{\widehat{r}}\left(\vec{q}, \vec{t}\right) \leq I^{\widehat{s_1}}\,(\vec{p}, \vec{u})\left[\mathrm{dom}\,(\Delta) := \vec{b}\right]$.
Applying rule (fix) to (3.11) and (3.12), we have

$$\Gamma \vdash \mathsf{fix}_n\ f : T_1^\star := M : T_1\,[\imath := s_1], \tag{3.16}$$

Applying Stage Substitution in (3.11) with $[\imath := s_1]$, and Substitution with (3.15) we obtain

$$\Gamma \vdash M\,[f := \mathsf{fix}_n\ f : T_1^\star := M] : T_1\,[\imath := \widehat{s_1}]. \tag{3.17}$$

Applying repeatedly rule (app), we obtain

$$\Gamma \vdash (M\,[f := \mathsf{fix}_n\ f : T_1^\star := M])\,\vec{b}\,(C(\vec{q^\circ}, \vec{a})) : U_1\,[\imath := \widehat{s_1}]\ \left[\mathrm{dom}\,(\Delta) := \vec{b}\right]\ [x := C(\vec{q^\circ}, \vec{a})]. \tag{3.18}$$

We conclude from $\imath$ pos U, (3.14), and applying rule (conv).

(case). $\Gamma \vdash t : T$ is $\Gamma \vdash \mathsf{case}_{|P|}\ x := M$ in $I\left(\vec{p^\circ}, \vec{y}\right)$ of $\{C_i \Rightarrow N_i\} : P\,[\vec{y} := \vec{a}]\,[x := M]$, derived from $\Gamma \vdash M : I^{\widehat{s}}\,(\vec{p}, \vec{a})$, $\Gamma(x : \Delta_a\,[\mathrm{dom}\,(\Delta_p) := \vec{p}])(x : I^{\widehat{s}}\,(\vec{p}, \vec{y})) \vdash P : u$, $\Gamma \vdash N_i : \mathsf{branch}^s_{C_i}(\vec{p}, \vec{y}.x.P)$.
We only consider the case of reduction at the head. So we have $M \equiv C_i(\vec{q}^\circ, \vec{u})$ and $t' = N_i\,\vec{u}$.

Applying Generation to the derivation of M, we obtain

$$\Gamma \vdash \vec{q} : \Delta_p \tag{3.19}$$

$$\Gamma \vdash \vec{u} : \mathsf{argsConstr}^r_{C_i}(\vec{q}) \tag{3.20}$$

$$\mathsf{typeConstr}^r_{C_i}(\vec{q}, \vec{u}) \leq I^{\widehat{s}}\,(\vec{p}, \vec{a}) \tag{3.21}$$

From (3.21) we have

$$\vec{q} \leq^{I.\nu} \vec{p} \tag{3.22}$$

$$\vec{t_i}\,[\mathrm{dom}\,(\Delta_p) := \vec{q}]\,[\mathrm{dom}\,(\Delta_i) := \vec{u}] \approx \vec{a} \tag{3.23}$$

and $\widehat{r} \sqsubseteq \widehat{s}$, which implies $r \sqsubseteq s$. Let us write $\vec{t_i}^*$ for $\vec{t_i}\,[\mathrm{dom}\,(\Delta_p) := \vec{q}]\,[\mathrm{dom}\,(\Delta_i) := \vec{u}]$. Then. $I^r\left(\vec{q}, \vec{t_i}^*\right) \leq I^s\,(\vec{p}, \vec{a})$. From clauses (I6), (I7), and (I8) of the conditions imposed to inductive types, we have

$$\mathsf{branch}^s_{C_i}(\vec{p}, \vec{y}.x.P) \leq \mathsf{branch}^r_{C_i}(\vec{q}, \vec{y}.x.P) \tag{3.24}$$

Then, $\Gamma \vdash N_i : \mathsf{branch}^r_{C_i}(\vec{q}, \vec{y}.x.P)$. Applying repeatedly rule (app) to $\vec{u}$ we obtain

$$\Gamma \vdash N_i\ \vec{u} : P\left[\vec{y} := \vec{t_i}\,[\mathrm{dom}\,(\Delta_p) := \vec{q}]\,[\mathrm{dom}\,(\Delta_i) := \vec{u}]\right][x := C(|q|, \vec{u})] \tag{3.25}$$

From (3.23) we obtain, applying rule (conv),

$$\Gamma \vdash N_i\,\vec{u} : P\,[\vec{y} := \vec{a}]\left[x := C(\vec{q^\circ}, \vec{u})\right]$$

which is the desired result.

□

Strengthening Lemma

The Strengthening Lemma can be seen as the opposite of the Weakening Lemma. It states that unused hypotheses in a typing judgment can be removed. Specifically, if $\Gamma(x : T)\Delta \vdash M : U$ and $x \notin \mathrm{FV}(\Delta, M, U)$, then $\Gamma\Delta \vdash M : U$. This result is, in some sense, more difficult to prove than the results we have seen in this chapter, since a straightforward induction on the typing derivations fails. Consider rule (app),

$$\frac{\Gamma(x : T)\Delta \vdash M : \Pi y : U_1.U_2 \qquad \Gamma(x : T)\Delta \vdash N : U_1}{\Gamma(x : T)\Delta \vdash M\,N : U_2\,[y := N]}$$

The hypothesis is $x \notin \mathrm{FV}(\Delta, M\,N, U_2\,[y := N])$. The IH cannot be directly applied since we do not know if $x \notin \mathrm{FV}(U_1)$. There is a "standard" way to solve this problem [36,56,81], which we follow here. The trick is to first prove a weaker version of the lemma.

Lemma 3.25. *Let $\Gamma(x : T)\Delta \vdash M : U$ be a valid judgment such that $x \notin \mathrm{FV}(\Delta, M)$. Then, there exists U' such that $\Gamma\Delta \vdash M : U'$ and $U' \leq U$. Let $\mathsf{WF}(\Gamma(x : T)\Delta)$ be a valid judgment such that $x \notin \mathrm{FV}(\Delta)$. Then, $\mathsf{WF}(\Gamma\Delta)$.*

Proof. We proceed by induction on the type derivation. We only treat the most relevant cases; the rest follow by IH.

(prod). M is $\Pi y : T_1.T_2$; we have the derivation

$$\frac{\Gamma(x:T)\Delta \vdash T_1 : u_1 \qquad \Gamma(x:T)\Delta(y:T_1) \vdash T_2 : u_2}{\Gamma(x:T)\Delta \vdash \Pi y : T_1.T_2 : u_3}$$

where $(u_1, u_2, u_3) \in \mathsf{Rule}$. By IH, there exists U_1 and U_2 such that $\Gamma\Delta \vdash T_1 : U_1$, $U_1 \leq u_1$, $\Gamma\Delta(y : T_1) \vdash T_2 : U_2$, and $U_2 \leq u_2$. Then, $U_1 \approx u_1$ and $U_2 \approx u_2$. The result follows by applying rule (conv) twice and rule (prod).

(app). M is $M_1\, M_2$; we have the derivation

$$\frac{\Gamma(x:T)\Delta \vdash M_1 : \Pi y : U_1.U_2 \qquad \Gamma(x:T)\Delta \vdash M_2 : U_1}{\Gamma(x:T)\Delta \vdash M_1\, M_2 : U_2\,[y := M_2]} \quad \mathrm{SV}(M_2) = \emptyset$$

By IH, there exists U' such that $\Gamma\Delta \vdash M_1 : U'$ and $U' \leq \Pi y : U_1.U_2$. Also by IH, there exists U'' such that $\Gamma\Delta \vdash M_2 : U''$ and $U'' \leq U_1$. By Type Validity (Lemma 3.21), U' and U'' are well typed under context $\Gamma\Delta$. By Generation of subtyping (Lemma 3.3), there exists W_1, W_2 such that $U' \to^* \Pi y : W_1.W_2$ and $U_1 \leq W_1$ and $W_2 \leq U_2$. By Subject Reduction, $\Pi y : W_1.W_2$ is well-typed under context $\Gamma\Delta$. Applying rule (conv), we have $\Gamma\Delta \vdash M_1 : \Pi y : W_1.W_2$ and $\Gamma\Delta \vdash M_2 : W_1$. Applying rule (app), we have $\Gamma\Delta \vdash M_1\, M_2 : W_2\,[y := M_2]$. The result follows by applying rule (conv) with $W_2\,[y := M_2] \leq U_2\,[y := M_2]$.

(case). M is $\mathsf{case}_{|P|}\ z := M_1\ \mathsf{in}\ I\,(|\vec{p}|, \vec{y})\ \mathsf{of}\ \langle C_i \Rightarrow N_i\rangle_i$; we have the derivation

$$\frac{\begin{array}{c}\Gamma(x:T)\Delta \vdash M_1 : I^{\widehat{s}}\,(\vec{p}, \vec{a}) \qquad I \in \Sigma \\ \Gamma(x:T)\Delta\,(\mathsf{caseType}^s_I(\vec{p}, \vec{y}, z)) \vdash P : u \\ \Gamma(x:T)\Delta \vdash N_i : \mathsf{branch}^s_{C_i}(\vec{p}, \vec{y}.z.P)\end{array}}{\Gamma(x:T)\Delta \vdash \mathsf{case}_{|P|}\ z := M\ \mathsf{in}\ I\,(|\vec{p}|, \vec{y})\ \mathsf{of}\ \{C_i \Rightarrow N_i\}_i : P\,[\vec{y} := \vec{a}]\,[z := M_1]}$$

The IH gives us: there exists U' such that $\Gamma\Delta \vdash M_1 : U'$ and $U' \leq I^{\widehat{s}}\,(\vec{p}, \vec{a})$; and for $i = 1, \ldots, n$, there exists U_i'' such that $\Gamma\Delta \vdash N_i : U_i''$ and $U_i'' \leq \mathsf{branch}^s_{C_i}(\vec{p}, \vec{y}.z.P)$. Also by IH and Confluence, $\Gamma\Delta\,(\mathsf{caseType}^s_I(\vec{p}, \vec{y}, z)) \vdash P : u$. Recall that $\mathsf{caseType}^s_I(\vec{p}, \vec{y}, z) \equiv (\vec{y} : \Delta_a\,[\mathrm{dom}\,(\Delta_p) := \vec{p}])(x : I^s\,(\vec{p}, \vec{y}))$. Then, $\Gamma\Delta \vdash I^s\,(\vec{p}, \vec{y}) : u'$. Applying rule (conv), we have $\Gamma\Delta \vdash M_1 : I^{\widehat{s}}\left(\vec{p'}, \vec{a'}\right)$.

We have $\Gamma\Delta \vdash \vec{p} : \Delta_p$; it is not difficult to show that $\Gamma\Delta \vdash \mathsf{branch}^s_{C_i}(\vec{p}, \vec{y}.z.P) : u$. The result then follows by applying rule (case).

□

Strengthening follows by applying the previous lemma twice.

Lemma 3.26 (Strengthening)**.** *Let* $\Gamma(x:T)\Delta \vdash M : U$ *be a valid judgment such that* $x \notin \mathrm{FV}(\Delta, M, U)$*. Then* $\Gamma\Delta \vdash M : U$*.*

Proof. Let $\Gamma(x:T)\Delta \vdash M : U$ be a valid judgment such that $x \notin \mathrm{FV}(\Delta, M, U)$. By the previous lemma, there exists U' such that $\Gamma\Delta \vdash M : U'$ and $U' \leq U$. By Type validity, $\Gamma(x:T)\Delta \vdash U : u$, for some sort u. Since $x \notin \mathrm{FV}(\Delta, U)$, by the previous lemma, there exists W such that $\Gamma\Delta \vdash U : W$ and $W \leq u$. Since u is well-typed, applying rule (conv) we have $\Gamma\Delta \vdash U : u$. The result follows by an application of rule (conv).

□

3.2 Annotated Version of CIC$\hat{}_{-}$

In this section we present a variant of CIC$\hat{}_{-}$ with explicit type annotations in some constructions. This is necessary to prove strong normalization in the presence of an impredicative universe [7,63] (cf. Chap. 4). We refer to the system in this section as ECIC$\hat{}_{-}$.

We present the syntax and typing rules of ECIC$\hat{}_{-}$. We also show the equivalence between both systems (Sect. 3.2.4).

3.2.1 Syntax and Typing Rules

We begin by describing the syntax of ECIC$\hat{}_{-}$. As in the case of CIC$\hat{}_{-}$, we present the syntax in two parts: basic terms and the inductive terms.

Definition 3.27 (Basic terms of ECIC$\hat{}_{-}$)**.** *The generic set of basic terms over a set a is defined by the grammar:*

$$\begin{array}{rcll} \mathcal{T}[a] & ::= & \mathcal{V} & \textit{(term variable)} \\ & | & \mathcal{U} & \textit{(universe)} \\ & | & \lambda^{\mathcal{T}^\circ}_{\mathcal{V}:\mathcal{T}^\circ}.\mathcal{T}[a] & \textit{(abstraction)} \\ & | & \mathsf{app}^{\mathcal{T}^\circ}_{\mathcal{V}:\mathcal{T}^\circ}(\mathcal{T}[a],\mathcal{T}[a]) & \textit{(application)} \\ & | & \Pi\mathcal{V}:\mathcal{T}[a].\mathcal{T}[a] & \textit{(product)} \end{array}$$

The set of bare terms, position terms and sized terms are defined by $\mathcal{T}^\circ ::= \mathcal{T}[\epsilon]$, $\mathcal{T}^\star ::= \mathcal{T}[\{\epsilon,\star\}]$, and $\mathcal{T} ::= \mathcal{T}[\mathcal{S}]$, respectively.

As in the case of CIC$\hat{}_{-}$, basic terms do not depend on the set a. Hence, there is no difference between classes of terms at this point. The difference appears once we introduce inductive terms.

The constructions are the same as in CIC$\hat{}_{-}$, except for abstraction and application that include type annotations for the domain and codomain. In $\lambda^{U^\circ}_{x:T^\circ}.M$ the variable x is bound in M and U°. In $\mathsf{app}^{U^\circ}_{x:T^\circ}(M,N)$, the variable x is bound in U°. The functions FV(.), SV(.), $|.|$, and $|.|^t$ are defined for terms in ECIC$\hat{}_{-}$ in the same way as for CIC$\hat{}_{-}$.

We introduce some notation to handle multiple applications and abstractions. We write $\mathsf{app}^{U^\circ}_{\Delta^\circ}\left(M,\vec{N}\right)$ and $\lambda^{U^\circ}_{\Delta^\circ}.M$ for the terms defined as follows:

$$\begin{aligned} \mathsf{app}^{U^\circ}_{[]}(M,\varepsilon) &= M \\ \mathsf{app}^{U^\circ}_{(x:T^\circ)\Delta^\circ}\left(M,N_1\vec{N}\right) &= \mathsf{app}^{U^\circ[x:=N_1]}_{\Delta^\circ[x:=N_1]}\left(\mathsf{app}^{\Pi\Delta^\circ.U^\circ}_{x:T^\circ}(M,N_1),\vec{N}\right) \\ \lambda^{U^\circ}_{[]}.M &= M \\ \lambda^{U^\circ}_{(x:T^\circ)\Delta^\circ}.M &= \lambda^{\Pi\Delta^\circ.U^\circ}_{x:T^\circ}.\lambda^{U^\circ}_{\Delta^\circ}.M \end{aligned}$$

We simply write $\mathsf{app}(M,N)$ instead of $\mathsf{app}^{U^\circ}_{x:T^\circ}(M,N)$ when x, T° and U° are not important.

We extend the syntax of basic terms with constructions related with inductive types. To avoid some of the burden of dealing with the type annotations in the case of fixpoints, we require them to be fully applied.

Definition 3.28 (Inductive terms of ECIC$\widehat{\ }_{_}$)**.** *The generic set of basic terms over a set a is defined by the grammar:*

$$\begin{array}{rcll}
\mathcal{T}[a] & ::= & \ldots & \\
& | & I^a\left(\vec{\mathcal{T}[a]}\right) & \textit{(inductive type)} \\
& | & \mathcal{C}(\vec{\mathcal{T}^\circ}, \vec{\mathcal{T}[a]}) & \textit{(constructor)} \\
& | & \mathsf{case}_{\mathcal{T}^\circ}\ \mathcal{V}_{\mathcal{I}(\vec{\mathcal{T}^\circ},\vec{\mathcal{T}^\circ})} := \mathcal{T}[a]\ \mathsf{in}\ \mathcal{I}\left(_, \vec{\mathcal{V}}\right)\ \mathsf{of}\ \langle \mathcal{C} \Rightarrow \mathcal{T}[a]\rangle & \textit{(case)} \\
& | & \mathsf{fix}_n\ \mathcal{V} : \mathcal{T}^\star := (\mathcal{T}[a], \langle \mathcal{T}_i \rangle_{i=1..n}) & \textit{(fixpoint)}
\end{array}$$

Case construction is extended with the type of the argument. In a term of the form $\mathsf{case}_{P^\circ}\ x_{I\left(\vec{p^\circ},\vec{a^\circ}\right)} := M\ \mathsf{in}\ I\left(\_, \vec{y}\right)\ \mathsf{of}\ \langle C_i \Rightarrow N_i\rangle_i$, the type of M is $I\left(\vec{p^\circ}, \vec{a^\circ}\right)$, while the pattern is of the form $I\left(\_, \vec{y}\right)$ to avoid duplicating the parameters. Fixpoint constructions are fully applied in ECIC$\widehat{\ }_{_}$: in a term of the form $\mathsf{fix}_n\ f : T^\star := (M, \vec{N})$, M denotes the body of the fixpoint and $\vec{N}$ of length n denote the arguments. The n-th argument is the recursive argument and its type must be an inductive type.

Contexts and inductive types are defined in the same way for ECIC$\widehat{\ }_{_}$, with the obvious adaptations of the syntax.

Reduction

We define a reduction relation for ECIC$\widehat{\ }_{_}$. It is called tight reduction, since type annotations in applications should agree. In the case of β-reduction,

$$\mathsf{app}^{U_2^\circ}_{x:T_2^\circ}\left(\lambda^{U_1^\circ}_{x:T_1^\circ}.M, N\right)$$

we require that types are α-convertible for the reduction rule to be applicable. I.e., $T_1^\circ \equiv T_2^\circ$ and $U_1^\circ \equiv U_2^\circ$. In the case of fixpoint reduction, since fixpoints are fully applied, we take care of η-expanding the fixpoint when substituting inside the body.

Definition 3.29 (Tight Reduction)**.** *Tight reduction, is defined as the compatible closure of the union of tight β-reduction, ι-reduction, and μ-reduction. They are defined by the following rules:*

$$\begin{array}{rcl}
\mathsf{app}^{U^\circ}_{x:T^\circ}\left(\lambda^{U^\circ}_{x:T^\circ}.M, N\right) & \beta_{\mathsf{t}} & M\left[x := N\right] \\
\mathsf{case}_{P^\circ}\ x_{I\left(\vec{p^\circ},\vec{u^\circ}\right)} := C_j(\vec{q^\circ}, \vec{a})\ \mathsf{in}\ I\left(_, \vec{y}\right)\ \mathsf{of}\ \langle C_i \Rightarrow N_i\rangle_i & \iota & \mathsf{app}^{P^*}_{\Delta_j{}^*}\left(N_j, \vec{a}\right) \\
\mathsf{fix}_n\ f : T^\star := (M, (\vec{N}, C(\vec{p^\circ}, \vec{a}))) & \mu & \mathsf{app}^{|U^\star|}_{|\Delta^\star|}\left(M\left[f := F\right], (\vec{N}, C(\vec{p^\circ}, \vec{a}))\right)
\end{array}$$

where $T^\star = \Pi\Delta^\star.U^\star$, $\#\Delta^\star = n$, and $\#\vec{N} = n - 1$, and

$$F \equiv \lambda^{|U^\star|}_{|\Delta^\star|}.\mathsf{fix}_n\ f : T^\star := (M, \mathrm{dom}\left(|\Delta^\star|\right))$$

In ι-reduction, we assume $\mathsf{Ind}(I[\Delta_p]^{\vec{\nu}} : \Pi\Delta_a.u := \langle C_i : \Pi\Delta_i.\mathcal{X}\,\vec{t_i}\rangle_i)$ and where we write $\Delta_j{}^*$ for $\Delta_j\left[\mathrm{dom}\left(\Delta_p\right) := \vec{p^\circ}\right]$ and P^* for $P^\circ\left[\vec{y} := |\vec{t_j}|\left[\mathrm{dom}\left(\Delta_p\right) := \vec{p^\circ}\right]\right]\left[x := C_j(\vec{p^\circ}, \mathrm{dom}\left(\Delta_j\right))\right]$.

Tight reduction is thus defined as $\rightarrow_{\beta_{\mathsf{t}}\,\iota\,\mu}$. We write $\rightarrow_{\mathsf{t}}$ instead of $\rightarrow_{\beta_{\mathsf{t}}\,\iota\,\mu}$; similarly we write $\leftarrow_{\mathsf{t}}$, ${\rightarrow_{\mathsf{t}}}^*$, and $\downarrow_{\mathsf{t}}$.

Tight reduction is not confluent on all terms. The main reason is that the β-rule is not left linear. The counterexamples of [49] can be easily adapted. Note that confluence is still valid for well-typed terms; the lack of confluence is not a problem for proving Subject Reduction.

Subtyping

Given that the reduction relation is not confluent, we cannot directly adapt the subtyping relation given in Chap. 2 to ECIC$\hat{}_{\sqsubset}$. Doing so would introduce some difficulties due to the presence of the transitivity rule. Instead we adapt the alternative definition of subtyping of Sect. 3.1.

Definition 3.30 (Subtyping)**.** *Given a relation R on terms, stable under stage and term substitution, the subtyping relations $\leq_R$ and $\leq_R^{\vec{\nu}}$ are simultaneously defined by the rules:*

$$\textit{(stt-conv)}\quad \frac{T_1 \downarrow_R T_2}{T_1 \leq_R T_2}$$

$$\textit{(stt-prod)}\quad \frac{T\ R^*\ \Pi x : T_1.T_2 \qquad U_1 \leq_R T_1 \qquad T_2 \leq_R U_2 \qquad U\ R^*\ \Pi x : U_1.U_2}{T \leq_R U}$$

$$\textit{(stt-ind)}\quad \frac{T\ R^*\ I^s\,(\vec{p_1}, \vec{a_1}) \qquad s \sqsubseteq r \qquad \vec{p_1} \leq_R^{I,\vec{\nu}} \vec{p_2} \qquad \vec{a_1} \downarrow_R \vec{a_2} \qquad U\ R^*\ I^r\,(\vec{p_2}, \vec{a_2})}{T \leq_R U}$$

$$\textit{(sttv-inv)}\quad \frac{T_1 \downarrow_R U_1 \quad \vec{T} \leq_R^{\vec{\nu}} \vec{U}}{T_1, \vec{T} \leq_R^{\circ,\vec{\nu}} U_1, \vec{U}}$$

$$\textit{(sttv-pos)}\quad \frac{T_1 \leq_R U_1 \quad \vec{T} \leq_R^{\vec{\nu}} \vec{U}}{T_1, \vec{T} \leq_R^{+,\vec{\nu}} U_1, \vec{U}}$$

$$\textit{(sttv-neg)}\quad \frac{U_1 \leq_R T_1 \quad \vec{T} \leq_R^{\vec{\nu}} \vec{U}}{T_1, \vec{T} \leq_R^{-,\vec{\nu}} U_1, \vec{U}}$$

$$\textit{(sttv-conv)}\quad \frac{\vec{T} \downarrow_R \vec{U}}{\vec{T} \leq_R^{\emptyset} \vec{U}}$$

We write $\leq_{\mathsf{t}}$ and $\leq_{\mathsf{t}}^{\vec{\nu}}$ for $\leq_{\to_{\mathsf{t}}}$ and $\leq_{\to_{\mathsf{t}}}^{\vec{\nu}}$, respectively.

The subtyping relation satisfies the following lemmas.

Lemma 3.31. *If $T \to_{\mathsf{t}} T'$, $U \to_{\mathsf{t}} U'$, and $T' \leq_{\mathsf{t}} U'$, then $T \leq_{\mathsf{t}} U$.*

Proof. By induction on the derivation of $T' \leq_{\mathsf{t}} U'$. □

Lemma 3.32. *If $T \leq_{\mathsf{t}} U$, then $|T| \downarrow_{\mathsf{t}} |U|$.*

Proof. By induction on the derivation of $T \leq_{\mathsf{t}} U$. □

Typing Rules

In Fig. 3.1 we show the typing rules of ECIC$\hat{}_{\sqsubset}$. They are directly adapted from the typing rules of CIC$\hat{}_{\sqsubset}$. In rules (app), (abs), and (case'), the type annotations define the type of the term. Note in rule (app) that we check that the type of the function is itself well typed. Adding this check simplifies some proofs. Note that the expressiveness of the system is not altered if we remove this condition, since type validity (Lemma 3.37) is satisfied. In rule (e-fix) we also check that the arguments of the fixpoint are well typed.

(empty) $$\frac{}{\mathsf{WF_t}([])}$$

(cons) $$\frac{\mathsf{WF_t}(\Gamma) \qquad \Gamma \vdash_{\mathsf{t}} T : u}{\mathsf{WF_t}(\Gamma(x:T))} \quad \mathsf{simple}(\mathrm{T})$$

(var) $$\frac{\mathsf{WF_t}(\Gamma) \qquad (x:T) \in \Gamma}{\Gamma \vdash_{\mathsf{t}} x : T}$$

(sort) $$\frac{\mathsf{WF_t}(\Gamma) \qquad (u_1, u_2) \in \mathsf{Axiom}}{\Gamma \vdash_{\mathsf{t}} u_1 : u_2}$$

(prod) $$\frac{\Gamma \vdash_{\mathsf{t}} T : u_1 \qquad \Gamma(x:T) \vdash_{\mathsf{t}} U : u_2 \qquad (u_1, u_2, u_3) \in \mathsf{Rule}}{\Gamma \vdash_{\mathsf{t}} \Pi x : T.U : u_3}$$

(abs) $$\frac{\Gamma(x:T) \vdash_{\mathsf{t}} M : U}{\Gamma \vdash_{\mathsf{t}} \lambda^{|U|}_{x:|T|}.M : \Pi x : T.U} \quad \mathrm{SV}(M) = \emptyset$$

(app) $$\frac{\begin{array}{c}\Gamma \vdash_{\mathsf{t}} \Pi x : T.U : u \\ \Gamma \vdash_{\mathsf{t}} M : \Pi x : T.U \qquad \Gamma \vdash_{\mathsf{t}} N : T\end{array}}{\Gamma \vdash_{\mathsf{t}} \mathsf{app}^{|U|}_{x:|T|}\,(M, N) : U\,[x := N]} \quad \mathrm{SV}(N) = \emptyset$$

(conv) $$\frac{\Gamma \vdash_{\mathsf{t}} M : T \qquad \Gamma \vdash_{\mathsf{t}} U : u \qquad T \leq_{\mathsf{t}} U}{\Gamma \vdash_{\mathsf{t}} M : U} \quad \mathsf{simple}(\mathrm{U})$$

(ind) $$\frac{I \in \Sigma \qquad \Gamma \vdash_{\mathsf{t}} \vec{p} : \mathsf{params}(I) \qquad \Gamma \vdash_{\mathsf{t}} \vec{a} : \mathsf{argsInd}_I(\vec{p})}{\Gamma \vdash_{\mathsf{t}} I^s\,(\vec{p}, \vec{a}) : \mathsf{sort}(I)} \quad \mathsf{simple}(I^s\,(\vec{p}, \vec{a}))$$

(constr) $$\frac{\begin{array}{c}I \in \Sigma \qquad \Gamma \vdash_{\mathsf{t}} \vec{p} : \mathsf{params}(I) \\ \Gamma \vdash_{\mathsf{t}} \vec{a} : \mathsf{argsConstr}^s_{C_i}(\vec{p})\end{array}}{\Gamma \vdash_{\mathsf{t}} C_i(|\vec{p}|, \vec{a}) : \mathsf{typeConstr}^s_{C_i}(\vec{p}, \vec{a})} \quad \mathsf{simple}(\mathsf{typeConstr}^s_{C_i}(\vec{p}, \vec{a}))$$

(case) $$\frac{\begin{array}{c}\Gamma \vdash_{\mathsf{t}} M : \widehat{I^s}\,(\vec{p}, \vec{a}) \qquad I \in \Sigma \qquad \Gamma\,(\mathsf{caseType}^s_I(\vec{p}, \vec{y}, x)) \vdash_{\mathsf{t}} P : u \\ \Gamma \vdash_{\mathsf{t}} N_i : \mathsf{branch}^s_{C_i}(\vec{p}, \vec{y}.x.P)\end{array}}{\Gamma \vdash_{\mathsf{t}} \left(\begin{array}{l}\mathsf{case}_{|P|}\ x_{I(|\vec{p}|,|\vec{a}|)} := M\ \mathsf{in}\ I\,(\_, \vec{y}) \\ \mathsf{of}\langle C_i \Rightarrow N_i\rangle_i\end{array}\right) : P\,[\vec{y} := \vec{a}]\,[x := M]} \quad \mathrm{SV}(N_i) = \emptyset$$

(fix) $$\frac{\begin{array}{c}T \equiv \Pi\Delta(x : I^\imath\,(\vec{p}, \vec{u})).U \qquad \imath\ \mathsf{pos}\ U \qquad \#\Delta = n - 1 \\ \imath \notin \mathrm{SV}(\Gamma, \Delta, \vec{u}, M) \qquad \Gamma \vdash_{\mathsf{t}} T : u \qquad \Gamma(f : T) \vdash_{\mathsf{t}} M : T\,[\imath := \widehat{\imath}] \\ \Gamma \vdash_{\mathsf{t}} \vec{N} : \Delta(x : I^s\,(\vec{p}, \vec{u}))\end{array}}{\Gamma \vdash_{\mathsf{t}} \mathsf{fix}_n\ f : |T|^\imath := (M, \vec{N}) : U\left[\mathrm{dom}\,(\Delta) := \vec{N}\right][\imath := s]} \quad \mathrm{SV}(\vec{N}) = \emptyset$$

Figure 3.1: Typing rules of terms and contexts of ECIC$\widehat{\ }_{_}$

3.2.2 Metatheory

We state some metatheoretical results for ECIC$\hat{}_{-}$ that we need in the next chapter. Most of the proof are done in the same way as for CIC$\hat{}_{-}$, sometimes easier by the presence of annotations. The exception is Subject Reduction, for which we give a more detailed proof.

Lemma 3.33. *Let $\Gamma \vdash M : T$ and Δ a well-formed context that contains all hypotheses in Γ. Then $\Delta \vdash M : T$.*

We define the order relation between contexts in the same way as for CIC$\hat{}_{-}$. We say that Δ is a subcontext of Γ, denoted $\Delta \leq_{\mathsf{t}} \Gamma$, if for all $(x : T) \in \Gamma$, there exists T' such that $(x : T') \in \Delta$ with $T' \leq_{\mathsf{t}} T$.

Lemma 3.34 (Context conversion)**.** *If $\Gamma \vdash_{\mathsf{t}} M : T$, $\mathsf{WF}_{\mathsf{t}}(\Delta)$, and $\Delta \leq_{\mathsf{t}} \Gamma$, then $\Delta \vdash_{\mathsf{t}} M : T$.*

Lemma 3.35 (Stage substitution)**.** *If $\Gamma \vdash t : T$ then $\Gamma[\imath := s] \vdash t[\imath := s] : T[\imath := s]$.*

Lemma 3.36 (Substitution)**.** *If $\Gamma(x : T)\Delta \vdash M : U$, and $\Gamma \vdash N : T$, and $\mathrm{SV}(N) = \emptyset$ then $\Gamma\Delta[x := N] \vdash M[x := N] : U[x := N]$.*

Lemma 3.37 (Type validity)**.** *If $\Gamma \vdash_{\mathsf{t}} M : T$, then $\Gamma \vdash_{\mathsf{t}} T : u$.*

A direct proof by induction of SR fails in the case of β_{t}-reduction, since tight subtyping is not transitive: if $\Pi x : T_1.U_1 \leq_{\mathsf{t}} \ldots \leq_{\mathsf{t}} \Pi x_n : T_n.U_n$, it is not immediately true that $T_n \leq_{\mathsf{t}} T_1$ and $U_1 \leq_{\mathsf{t}} U_n$. We prove that this result is valid in the particular case when $|\Pi x : T_1.U_1| \equiv |\Pi x : T_n.U_n|$ (cf. Lemma 3.43 and Lemma 3.45). This particular case is enough to handle β_{t}-reduction in the proof of SR.

We define *loose reduction* relation which will be useful in the rest of the chapter. In particular, to prove the equivalence between ECIC$\hat{}_{-}$ and CIC$\hat{}_{-}$.

Definition 3.38 (Loose reduction)**.** *Loose reduction, is defined as the compatible closure of the union of loose β-reduction, ι-reduction, and μ-reduction. Loose β-reduction is defined by the rule:*

$$\mathsf{app}^{U_2^\circ}_{x:T_2^\circ}\left(\lambda^{U_1^\circ}_{x:T_1^\circ}.M, N\right) \quad \beta_{\mathsf{l}} \quad M[x := N]$$

Loose reduction is thus defined as $\to_{\beta_{\mathsf{l}}\,\iota\,\mu}$. We write $\to_{\mathsf{l}}$ instead of $\to_{\beta_{\mathsf{l}}\,\iota\,\mu}$. Similarly, we write $\leftarrow_{\mathsf{l}}$, ${\to_{\mathsf{l}}}^*$, $\approx_{\mathsf{l}}$, and $\downarrow_{\mathsf{l}}$.

Note the difference between tight reduction and loose reduction. In loose reduction we do not impose any condition on the type annotations when reducing an application. The rules become left linear; therefore, loose reduction is confluent.

Lemma 3.39. *Loose reduction is confluent: if $M \approx_{\mathsf{l}} N$, then $M \downarrow_{\mathsf{l}} N$.*

Proof. The proof goes along the same lines as for Lemma 2.7. Since the rules are left linear, the problem with tight reduction is avoided. □

We can apply Def. 3.30 to loose reduction. We write $\leq_{\mathsf{l}}$ for $\leq_{\to_{\mathsf{l}}}$. We prove some lemmas about the relation $\leq_{\mathsf{l}}$. Since loose reduction is confluent, we can prove that $\leq_{\mathsf{l}}$ is transitive.

Lemma 3.40. *If $T \approx_{\mathsf{l}} T' \leq_{\mathsf{l}} U' \approx_{\mathsf{l}} U$, then $T \leq_{\mathsf{l}} U$, with a derivation of the same height. If $\vec{T} \approx_{\mathsf{l}} \vec{T'} \leq^{\vec{v}}_{\mathsf{l}} \vec{U'} \approx_{\mathsf{l}} \vec{U}$, then $\vec{T} \leq_{\mathsf{l}} \vec{U}$, with a derivation of the same height.*

Proof. By induction on the subtype derivation. Similar to Lemma 3.7. □

Lemma 3.41. *If* $T_1 \leq_{\mathsf{l}} T_2$ *and* $T_2 \leq_{\mathsf{l}} T_3$*, then* $T_1 \leq_{\mathsf{l}} T_3$*.*

Proof. By induction on the sum of the heights of the subtype derivations. Similar to Lemma 3.8. □

Lemma 3.42. *If* $T \leq_{\mathsf{t}} U$*, then* $T \leq_{\mathsf{l}} U$*.*

Proof. By induction on the type derivation. The proof follows from the fact that $\to_{\mathsf{t}} \subseteq \to_{\mathsf{l}}$. □

We prove some inversion lemmas for $\leq_{\mathsf{l}}$.

Lemma 3.43. *If* $\Pi x : T_1.T_2 \leq_{\mathsf{l}} \Pi x : U_1.U_2$*, then* $U_1 \leq_{\mathsf{l}} T_1$ *and* $T_2 \leq_{\mathsf{l}} U_2$*.*

Proof. By induction on the subtype derivation. There are two possible cases for the last rule used: (stt-conv) and (stt-prod). In the former case, the result follows from confluence. In the latter case, we use Lemma 3.40. □

Lemma 3.44. *If* $I^s\,(\vec{p_1}, \vec{a_1}) \leq_{\mathsf{l}} I^r\,(\vec{p_2}, \vec{a_2})$*, then* $s \sqsubseteq r$*,* $\vec{p_1} \leq_{\mathsf{l}}^{I.\vec{\nu}} \vec{p_2}$ *and* $\vec{a_1} \downarrow_{\mathsf{l}} \vec{a_2}$*.*

Proof. By induction on the subtype derivation. There are two possible cases for the last rule used: (stt-conv) and (stt-ind). The result follows the same reasoning as the previous lemma. □

Next we prove the lemma we need in the proof of Subject Reduction to handle tight β-reduction.

Lemma 3.45. *If* $T \leq_{\mathsf{l}} U$*,* $\mathsf{simple}(T, U)$*, and* $|T| \equiv |U|$*, then* $T \leq_{\mathsf{t}} U$*. If* $\vec{T} \leq_{\mathsf{l}} \vec{U}$*,* $\mathsf{simple}^{\vec{\nu}}(\vec{T}, \vec{U})$ *and* $|\vec{T}| \equiv |\vec{U}|$*, then* $\vec{T} \leq_{\mathsf{t}} \vec{U}$*.*

Proof. By simultaneous induction on the predicate $\mathsf{simple}(T)$ and $\mathsf{simple}(\vec{T})$.

- $T \equiv \Pi x : T_1.T_2$, $U \equiv \Pi x : U_1.U_2$, and $\mathsf{simple}(T_1, T_2, U_1, U_2)$. By Lemma 3.43, $U_1 \leq_{\mathsf{l}} T_1$ and $T_2 \leq_{\mathsf{l}} U_2$. By IH, $U_1 \leq_{\mathsf{t}} T_1$ and $T_2 \leq_{\mathsf{t}} U_2$. The result follows by rule (stt-prod).
- $T \equiv I^s\,(\vec{p_1}, \vec{a_2})$, $U \equiv I^r\,(\vec{p_2}, \vec{a_2})$, $\mathsf{simple}^{I.\vec{\nu}}(\vec{p_1})$, $\mathsf{simple}^{I.\vec{\nu}}(\vec{p_2})$, $\mathrm{SV}(\vec{a_1}) = \mathrm{SV}(\vec{a_2}) = \emptyset$. By Lemma 3.44, $s \sqsubseteq r$, $\vec{p_1} \leq_{\mathsf{l}}^{I.\vec{\nu}} \vec{p_2}$ and $\vec{a_1} \downarrow_{\mathsf{l}} \vec{a_2}$. Since $\vec{a_1}$ and $\vec{a_2}$ contain no size variable, $\vec{a_1} \equiv \vec{a_2}$. The result follows by the IH and rule (stt-ind).
- If T is neither a product nor an inductive type (and neither is U), we have $\mathrm{SV}(T) = \mathrm{SV}(U) = \emptyset$. Then $T \equiv U$, and the result follows.
- The cases on vectors follow easily by applying the IH.

□

Lemma 3.46 (Subject reduction)**.** *If* $\Gamma \vdash M : T$ *and* $M \to_{\mathsf{t}} M'$ *then* $\Gamma \vdash M' : T$*.*

Proof. By induction on the type derivation. The interesting case is when the last rule applied is the application rule and reduction is at the head. The rest of the cases follows in the same way as in Lemma 3.24. Fixpoint reduction follows easily since fixpoints are fully applied; there is no need to prove an analogue of Lemma 3.23.

We consider rule (app). We have $M \equiv \mathsf{app}^{U_2^\circ}_{x:U_1^\circ}\left(\lambda^{U_2^\circ}_{x:U_1^\circ}.M_1, M_2\right)$ and $M' \equiv M_1\,[x := M_2]$. Then $\Gamma \vdash M : T$ is derived from

$$\Gamma \vdash \lambda^{U_2^\circ}_{x:U_1^\circ}.M_1 : \Pi x : U_1'.U_2' \tag{3.26}$$

$$\Gamma \vdash M_2 : U_1' \tag{3.27}$$

We also know $\mathrm{SV}(M_2) = \emptyset$, $|\Pi x : U_1'.U_2'| \equiv \Pi x : U_1^\circ.U_2^\circ$, and $T \equiv U_2^\circ\,[x := M_2]$. By inversion on (3.26), there exists U_1'' and U_2'' such that

$$\Gamma(x : U_1'') \vdash M_1 : U_2'', \tag{3.28}$$

with $|U_1''| \equiv U_1^\circ$ and $|U_2''| \equiv U_2^\circ$. Also, $\Pi x : U_1''.U_2'' \leq_{\mathsf{t}} \ldots \leq_{\mathsf{t}} \Pi x : U_1'.U_2'$. By Lemma 3.42 and Lemma 3.41, $\Pi x : U_1''.U_2'' \leq_{\mathsf{l}} \Pi x : U_1'.U_2'$. By Lemma 3.43, $U_1' \leq_{\mathsf{l}} U_1''$ and $U_2'' \leq_{\mathsf{l}} U_2'$. We can apply Lemma 3.45 and obtain $U_1' \leq_{\mathsf{t}} U_1''$ and $U_2'' \leq_{\mathsf{t}} U_2'$. By rule (conv) on (3.27) and (3.28), $\Gamma \vdash M_2 : U_1''$ and $\Gamma(x : U_1'') \vdash M_1 : U_2'$. Finally, by Substitution, $\Gamma \vdash M_1\,[x := M_2] : U_2'\,[x := M_2]$. □

The following two lemmas, Type uniqueness and Strengthening, are proved by induction on the type derivation. The proofs are easier because of the added type annotations. In particular, in the case of Strengthening, the problem mentioned in Sect. 3.1 is not present in the case of ECIC$\hat{}_{\sqsubset}$.

Lemma 3.47 (Type uniqueness)**.** *If* $\Gamma \vdash M : T_1$ *and* $\Gamma \vdash M : T_2$*, then* $|T_1| \approx |T_2|$*.*

Lemma 3.48 (Strengthening)**.** *If* $\Gamma(x : T)\Delta \vdash M : U$ *and* $x \notin \mathrm{FV}(\Delta, M, U)$*, then* $\Gamma\Delta \vdash M : U$*.*

3.2.3 Strong Normalization and Logical Consistency

In the rest of the chapter, we make use of the following theorems, stating strong normalization of tight reduction, and logical consistency of ECIC$\hat{}_{\sqsubset}$:

Theorem 3.49. *If* $\Gamma \vdash_{\mathsf{t}} M : T$*, then* $M \in \mathrm{SN}_{\to_{\mathsf{t}}}$*.*

Theorem 3.50. *There is no term* M *such that* $\vdash_{\mathsf{t}} M : \mathsf{False}$*.*

The proof of these theorems is developed in Chapter 4.

3.2.4 From CIC$\hat{}_{\sqsubset}$ to ECIC$\hat{}_{\sqsubset}$

In this section we prove the equivalence between CIC$\hat{}_{\sqsubset}$ and ECIC$\hat{}_{\sqsubset}$. As a consequence, we can obtain SN of CIC$\hat{}_{\sqsubset}$ from SN of ECIC$\hat{}_{\sqsubset}$.

We need to take into account to main differences between CIC$\hat{}_{\sqsubset}$ and ECIC$\hat{}_{\sqsubset}$ in order to prove their equivalence: codomain annotations in abstractions and applications, and fully applied fixpoints. We introduce an intermediate system that lies half-way between CIC$\hat{}_{\sqsubset}$ and ECIC$\hat{}_{\sqsubset}$. The intermediate system, called CIC$\hat{}_{\eta}$, features fully applied fixpoints, but does not include codomain annotations.

We present CIC$\hat{}_{\eta}$, compared to CIC$\hat{}_{\sqsubset}$, since both systems are very close. The set of terms is similar, except for the fixpoint construction, that is fully applied:

$$\mathcal{T} ::= \ldots \mid \mathsf{fix}_n\ f : T^\star := (M, \langle N_i \rangle_{i=1..n})$$

The reduction rule for fixpoint is similar to that of ECIC$\hat{}_{\sqsubset}$:

$$\mathsf{fix}_n\ f : T^\star := (M, (\langle N_i \rangle_{i=1..n-1}, C(\vec{p^\circ}, \vec{a})))\ (\mu')$$
$$M\,[f := \lambda|\Delta^\star|.\mathsf{fix}_n\ f : T^\star := (M, \mathrm{dom}\,(\Delta^\star))]\,\langle N_i \rangle_{i=1..n-1} C(\vec{p^\circ}, \vec{a})$$

where $T^\star \equiv \Pi\Delta^\star.U^\star$, where $\#\Delta^\star = n$. The reduction relation of CIC$\widehat{\ }_\eta$ is defined as the compatible closure of $\beta\iota\mu'$-reduction. We denote it $\to'$. Reduction is confluent; the proof follows the same pattern as for CIC$\widehat{\ }_{_}$.

Subtyping is defined in the same way as for CIC$\widehat{\ }_{_}$, using the alternative definition (Def. 3.6). We use $\vdash_\eta$ for the typing judgment. Typing is defined by the rules of Fig. 2.5 and Fig. 2.6, except for rule (fix), which is replaced by the following rule:

$$\text{(fix-}\eta\text{)}\ \frac{\begin{array}{c} T \equiv \Pi\Delta(x : I^\imath\,(\vec{p},\vec{u})).U \qquad \imath \;\mathsf{pos}\; U \qquad \#\Delta = n-1 \\ \imath \notin \mathrm{SV}(\Gamma,\Delta,\vec{u},M) \qquad \Gamma \vdash_\eta T : u \\ \Gamma(f : T) \vdash_\eta M : T\,[\imath := \widehat{\imath}] \qquad \Gamma \vdash_\eta \vec{N} : \Delta(x : I^s\,(\vec{p},\vec{u})) \end{array}}{\Gamma \vdash_\eta \mathsf{fix}_n\ f : |T|^\imath := (M,\vec{N}) : U\left[\mathrm{dom}\,(\Delta), x := \vec{N}\right][\imath := s]}\quad \mathrm{SV}(\vec{N}) = \emptyset$$

We define the first translation, from CIC$\widehat{\ }_{_}$ to CIC$\widehat{\ }_\eta$. It is defined by a function on terms that η-expands fixpoints to make them fully applied. The function, denoted $\lceil . \rceil$, is defined as follows:

$$\begin{aligned}
\lceil x \rceil &= x \\
\lceil u \rceil &= u \\
\lceil \lambda x : T^\circ.M \rceil &= \lambda x : \lceil T^\circ \rceil.\lceil M \rceil \\
\lceil M\,N \rceil &= \lceil M \rceil\,\lceil N \rceil \\
\lceil \Pi x : T.U \rceil &= \Pi x : \lceil T \rceil.\lceil U \rceil \\
\lceil C(\vec{p^\circ},\vec{a}) \rceil &= C(\lceil \vec{p^\circ} \rceil, \lceil \vec{a} \rceil) \\
\lceil I^s\,(\vec{p},\vec{a}) \rceil &= I^s\,(\lceil \vec{p} \rceil, \lceil \vec{a} \rceil) \\
\lceil \mathsf{case}_{P^\circ}\ x := M\ \mathsf{in}\ I\,(\vec{p},\vec{y})\ \mathsf{of}\ \langle C_i \Rightarrow N_i \rangle_i \rceil &= \mathsf{case}_{\lceil P^\circ \rceil}\ x := \lceil M \rceil\ \mathsf{in}\ I\,(\lceil \vec{p} \rceil, \vec{y})\ \mathsf{of}\ \langle C_i \Rightarrow \lceil N_i \rceil \rangle_i \\
\lceil \mathsf{fix}_n\ f : T^\star := M \rceil &= \lambda|\Delta_1|.\mathsf{fix}_n\ f : \lceil T^\star \rceil := (\lceil M^\star \rceil, \mathrm{dom}\,(\Delta_1))
\end{aligned}$$

where $|T^\star| \equiv \Pi\Delta^\circ.U$, with $\#\Delta^\circ = n$, and $\Delta_1 = \lceil \Delta^\circ \rceil$.

The function is defined only for *well-defined* terms. A term is well-defined if all for all fixpoints of the form $\mathsf{fix}_n\ f : T^\star := M$, the type $T^\star$ is of the form $\Pi\Delta^\star.U^\star$ with $\#\Delta^\star = n$. For the rest of the chapter, we assume that all terms considered are well-defined. The translation function preserves reductions.

Lemma 3.51. *1.* $\lceil M\,[x := N] \rceil \equiv \lceil M \rceil\,[x := \lceil N \rceil]$.

2. If $M \to N$, *then* $\lceil M \rceil \to'^+ \lceil N \rceil$.

Proof. For part 1 we proceed by induction on the structure of terms.

For part 2 we proceed by induction on the reduction relation. The only interesting case is μ-reduction at the head. Let $M \equiv (\mathsf{fix}_n\ f : T^\star := M_1)\vec{N_1}C(\vec{p^\circ},\vec{a})$, with $T^\star \equiv \Pi\Delta^\star.U^\star$. Then,

$$\begin{aligned}
&\lceil (\mathsf{fix}_n\ f : T^\star := M_1)\vec{N_1}C(\vec{p^\circ},\vec{a}) \rceil \equiv \\
&\qquad (\lambda\lceil|\Delta^\star|\rceil.\mathsf{fix}_n\ f : \lceil T^\star \rceil := (\lceil M_1 \rceil, \mathrm{dom}\,(\Delta^\star)))\lceil \vec{N_1}, C(\vec{p^\circ},\vec{a}) \rceil\ \beta^+ \\
&\qquad\qquad \mathsf{fix}_n\ f : \lceil T^\star \rceil := (\lceil M_1 \rceil, \lceil \vec{N_1}, C(\vec{p^\circ},\vec{a}) \rceil)\ \mu' \\
&\qquad \lceil M_1 \rceil\,[f := \lambda\lceil|\Delta^\star|\rceil.\mathsf{fix}_n\ f : \lceil T^\star \rceil := (\lceil M_1 \rceil, \mathrm{dom}\,(\Delta^\star))]\,\lceil \vec{N_1}, C(\vec{p^\circ},\vec{a}) \rceil \equiv \\
&\qquad\qquad \lceil M_1\,[f := \mathsf{fix}_n\ f : T^\star := M_1]\,\vec{N_1}C(\vec{p^\circ},\vec{a}) \rceil
\end{aligned}$$

The last equality follows from part 1. □

Using the previous lemma we prove that the translation function preserves conversion, subtyping and typing.

Lemma 3.52. *1. If $M \approx N$, then $\lceil M \rceil \approx \lceil N \rceil$.*

2. If $M \leq N$, then $\lceil M \rceil \leq \lceil N \rceil$.

3. If $\Gamma \vdash M : T$, then $\lceil \Gamma \rceil \vdash_\eta \lceil M \rceil : \lceil T \rceil$.

Proof. Part 1 follows from confluence and the previous lemma. Parts 2 and 3 follow by induction on the subtyping derivation and the typing derivation, respectively. □

We prove the equivalence between CIC$\widehat{\ }_\eta$ and ECIC$\widehat{\ }_{-}$. There is one side of the equivalence that is easy to establish. We define a stripping map from terms of ECIC$\widehat{\ }_{-}$ to terms of CIC$\widehat{\ }_\eta$ that removes type annotations.

Definition 3.53. *We define the map $\lfloor . \rfloor$ from terms of ECIC$\widehat{\ }_{-}$ to terms of CIC$\widehat{\ }_\eta$ by the following rules:*

$$\begin{aligned}
\lfloor x \rfloor &= x \\
\lfloor u \rfloor &= u \\
\lfloor \lambda^{U^\circ}_{x:T^\circ}.M \rfloor &= \lambda x : \lfloor T^\circ \rfloor . \lfloor M \rfloor \\
\lfloor \mathsf{app}^{U^\circ}_{x:T^\circ}(M, N) \rfloor &= \lfloor M \rfloor \, \lfloor N \rfloor \\
\lfloor \Pi x : T.U \rfloor &= \Pi x : \lfloor T \rfloor . \lfloor U \rfloor \\
\lfloor C(\vec{p^\circ}, \vec{a}) \rfloor &= C(\lfloor \vec{p^\circ} \rfloor, \lfloor \vec{a} \rfloor) \\
\lfloor I^s(\vec{p}, \vec{a}) \rfloor &= I^s(\lfloor \vec{p} \rfloor, \lfloor \vec{a} \rfloor) \\
\lfloor \mathsf{case}_{P^\circ}\, x_{I(\vec{p^\circ}, \vec{a^\circ})} := M \,\mathsf{in}\, I(_, \vec{y}) \,\mathsf{of}\, \langle C_i \Rightarrow N_i \rangle_i \rfloor &= \mathsf{case}_{\lfloor P^\circ \rfloor}\, x := \lfloor M \rfloor \,\mathsf{in}\, I(\lfloor \vec{p} \rfloor, \vec{y}) \,\mathsf{of}\, \langle C_i \Rightarrow \lfloor N_i \rfloor \rangle_i \\
\lfloor \mathsf{fix}_n\ f : T^\star := (M, \vec{N}) \rfloor &= \mathsf{fix}_n\ f : \lfloor T^\star \rfloor := (\lfloor M \rfloor, \lfloor \vec{N} \rfloor)
\end{aligned}$$

The map $\lfloor . \rfloor$ is extended to contexts and signatures in the obvious way.

Reduction, subtyping and typing are preserved by the stripping map, as stated in the following lemma.

Lemma 3.54. *1. $M \downarrow_{\mathsf{t}} M' \Rightarrow \lfloor M \rfloor \approx \lfloor M' \rfloor$;*

2. $T \leq_{\mathsf{t}} U \Rightarrow \lfloor T \rfloor \leq \lfloor U \rfloor$;

3. if $\Gamma \vdash_t M : T$, then $\lfloor \Gamma \rfloor \vdash_\eta \lfloor M \rfloor : \lfloor T \rfloor$.

Proof. All proofs follow by straightforward induction on the relevant structure. □

In the rest of the chapter we prove the opposite direction: if $\Gamma \vdash_\eta M : T$, then there exists Γ^+, M^+ and T^+ such that $\lfloor \Gamma^+ \rfloor = \Gamma$, $\lfloor M^+ \rfloor = M$, $\lfloor T^+ \rfloor = T$, and $\Gamma^+ \vdash_{\mathsf{t}} M^+ : T^+$ (Lemma 3.65). As a consequence, can prove SN for CIC$\widehat{\ }_\eta$ and CIC$\widehat{\ }_{-}$(Corollary 3.66).

We begin by proving that reduction and subtyping are preserved from CIC$\widehat{\ }_{-}$ to ECIC$\widehat{\ }_{-}$. The following lemma states some simple properties of loose reduction.

Lemma 3.55. *1. $M \approx_{\mathsf{l}} M' \Rightarrow \lfloor M \rfloor \approx \lfloor M' \rfloor$;*

2. if $\lfloor M \rfloor \to' N$, then there exists P such that $M \to_{\mathsf{l}} P$ and $\lfloor P \rfloor \equiv N$.

Proof. Both parts are proved by induction on the reduction rules definition. □

Some notation on reductions will be useful in the rest of the chapter. Given a relation R on terms and a term M, we write M is in R-nf or $M \in \mathrm{NF}(R)$, to mean that M is in normal form with respect to R, i.e., that there is no N such that $M\ R\ N$.

Lemma 3.56. *Let* $\mathsf{app}^{U_2^\circ}_{x:T_2^\circ}\left(\lambda^{U_1^\circ}_{x:T_1^\circ}.M, N\right)$ *be a well-typed term in some context* Γ*. Then* $\Pi x : T_1^\circ.U_1^\circ \approx_{\mathsf{l}} \Pi x : T_2^\circ.U_2^\circ$.

Proof. By inversion on the type derivation, the term $\lambda^{U_1^\circ}_{x:T_1^\circ}.M$ has type W_1 with $|W_1| \equiv \Pi x : T_1^\circ.U_1^\circ$, and it also has type W_2 with $|W_2| \equiv \Pi x : T_2^\circ.U_2^\circ$ and $W_1 \leq_{\mathsf{t}} \ldots \leq_{\mathsf{t}} W_2$. Therefore, $|W_1| \approx_{\mathsf{l}} |W_2|$. □

Lemma 3.57. *Let* M *and* M' *be two well-typed terms under context* Γ*. If* $M \approx_{\mathsf{l}} M'$ *and* M *and* M' *are in* $\to_{\mathsf{t}}$*-nf, then* $M \equiv M'$.

Proof. By mutual induction on the size of M and M'. We prove first that M and M' are in $\to_{\mathsf{l}}$-nf. Assume that there is a loose redex in M.

The redex is of the form

$$\mathsf{app}^{U_2^\circ}_{x:T_2^\circ}\left(\lambda^{U_1^\circ}_{x:T_1^\circ}.N, P\right) \ .$$

Then by Lemma 3.56, $T_1^\circ \approx_{\mathsf{l}} T_2^\circ$ and $U_1^\circ \approx_{\mathsf{l}} U_2^\circ$. Since T_1^∞ and T_2^∞ are well-typed in context Γ^∞ (by Stage Substitution), by IH, $T_1^\infty \equiv T_2^\infty$. Hence, $T_1^\circ \equiv T_2^\circ$. Similarly, $U_1^\circ \equiv U_2^\circ$, since U_1^∞ and U_2^∞ are well-typed under context $\Gamma^\infty(x : T_1^\infty)$. But then, the above redex is actually a tight redex, which contradicts the hypothesis that M is in $\to_{\mathsf{t}}$-nf.

We have $M \approx_{\mathsf{l}} M'$ and both terms are in $\to_{\mathsf{l}}$-nf. Since $\to_{\mathsf{l}}$ is confluent, we conclude that $M \equiv M'$. □

Corollary 3.58. *Let* M *and* M' *be two well-typed terms under context* Γ*. If* $M \approx_{\mathsf{l}} M'$*, then* M *and* M' *have a common unique normal form.*

Proof. Let N and N' be normal forms (for $\to_{\mathsf{t}}$) of M and M' respectively. Then, $N \approx_{\mathsf{l}} N'$. By Subject Reduction, N and N' are well-typed in Γ. Applying the previous lemma, we obtain $N \equiv N'$. □

Corollary 3.59. *Let* M *be a well-typed term. If* M *is in* $\to_{\mathsf{t}}$*-nf, then* M *is in* $\to_{\mathsf{l}}$*-nf.*

Proof. Following the same reasoning as in Lemma 3.57, if M has a loose redex, we can show that is in fact a tight redex. □

Corollary 3.60. *Let* M *be a well-typed term in* $\to_{\mathsf{t}}$*-nf. Then* $\lfloor M \rfloor$ *is in* $\to$*-nf.*

Proof. From Cor. 3.59 and Lemma 3.55. □

We prove that conversion in ECIC$\widehat{\ }_{-}$ can be obtained from conversion on CIC$\widehat{\ }_{\eta}$. That is, if $\lfloor M_1 \rfloor \approx \lfloor M_2 \rfloor$, then $M_1 \downarrow_{\mathsf{t}} M_2$. The idea is to reduce M_1 and M_2 to normal form, and prove that they are equal (up to α-convertibility).

Lemma 3.61. *Assume* $\Gamma_1 \vdash M_1 : T_1$ *and* $\Gamma_2 \vdash M_2 : T_2$*, with* $|\Gamma_1| = |\Gamma_2|$*. If* $\lfloor M_1 \rfloor \approx \lfloor M_2 \rfloor$ *and* M_1 *and* M_2 *are in* $\to_{\mathsf{t}}$*-nf, then* $M_1 \equiv M_2$.

Proof. By Cor. 3.60, $\lfloor M_1 \rfloor$ and $\lfloor M_2 \rfloor$ are in $\to$-nf. Therefore, $\lfloor M_1 \rfloor \equiv \lfloor M_2 \rfloor$. We proceed by induction on the size of M_1 and M_2, and case analysis on the shape of M_1 and M_2.

Abstraction. $M_1 \equiv \lambda^{U_1^\circ}_{x_1:T_1^\circ}.N_1$ and $M_2 \equiv \lambda^{U_2^\circ}_{x_2:T_2^\circ}.N_2$. Since M_1 and M_2 are in $\to_{\mathsf{I}}$-nf, then $T_1^\circ \approx_{\mathsf{I}} T_2^\circ$, $U_1^\circ \approx_{\mathsf{I}} U_2^\circ$ and $N_1 \approx_{\mathsf{I}} N_2$.

From IH on T_1^∞ and T_2^∞ (well-typed on context Γ_1^∞), we obtain that $T_1^\infty \equiv T_2^\infty$ (and $T_1^\circ \equiv T_2^\circ$). Similarly, applying IH to U_1^∞ and U_2^∞ (well-typed on context $\Gamma_1^\infty(x_1 : T_1^\infty)$), we obtain $U_1^\circ \equiv U_2^\circ$.

There exists W_i, for $i = 1, 2$ such that $|W_i| = T_i{}^\circ$, and N_i is well-typed under context $\Gamma_i(x_i : W_i)$. The IH applies, and $N_1 \equiv N_2$.

Application. $M_1 \equiv \mathsf{app}^{U_1^\circ}_{x_1:T_1^\circ}(N_1, P_1)$ and $M_2 \equiv \mathsf{app}^{U_2^\circ}_{x_2:T_2^\circ}(N_2, P_2)$. By IH, $P_1 \equiv P_2$ and $N_1 \equiv N_2$. In context Γ_1^∞ ($\equiv \Gamma_2^\infty$), N_1 has types $\Pi x : T_1^\infty.U_1^\infty$ and $\Pi x : T_2^\infty.U_2^\infty$. The IH applies, and both types are equal.

□

Corollary 3.62. *Let M_1 and M_2 be well-typed terms under context Γ. If $\lfloor M_1 \rfloor \approx \lfloor M_2 \rfloor$, then $M_1 \downarrow_{\mathsf{t}} M_2$.*

Proof. By applying the previous lemma to the normal forms of M_1 and M_2. □

Lemma 3.63. *Let T_1 and T_2 be well-typed terms under context Γ. If $\lfloor T_1 \rfloor \leq \lfloor T_2 \rfloor$, then $T_1 \leq_{\mathsf{t}} T_2$.*

Proof. We prove the lemma using the alternative subtyping relation (Def. 3.6). We proceed by induction on the height of the subtyping relation, and case analysis on the last rule.

(ast-prod) $\lfloor T_1 \rfloor \leq \lfloor T_2 \rfloor$ is derived from $\lfloor T_1 \rfloor \to^* \Pi x : U_1.W_1$, $\lfloor T_2 \rfloor \to^* \Pi x : U_2.W_2$, $U_2 \leq_a U_1$, and $W_1 \leq_a W_2$.

Let N_1 and N_2 be normal forms of T_1 and T_2, respectively. By Cor. 3.60, $\lfloor N_1 \rfloor$ is in $\to$-nf. Also $\lfloor N_1 \rfloor \approx \Pi x : U_1.W_1$ and, by confluence, $\Pi x : U_1.W_1 \to^* \lfloor N_1 \rfloor$ and $\lfloor N_1 \rfloor \equiv \Pi x : U_1'.W_1'$ for some U_1', W_1'. Then $N_1 \equiv \Pi x : U_1^+.W_1^+$ for some U_1^+, W_1^+. Similarly, $N_2 \equiv \Pi x : U_2^+.W_2^+$, for some U_2^+, W_2^+.

We have, $\lfloor U_2^+ \rfloor \approx U_2 \leq_a U_1 \approx \lfloor U_1^+ \rfloor$. By Lemma 3.7, $\lfloor U_2^+ \rfloor \leq_a \lfloor U_1^+ \rfloor$ with a derivation of the same height as $U_2 \leq_a U_1$. We can apply the IH, and obtain $U_2^+ \leq_a U_1^+$. Similarly, $W_1^+ \leq_a W_2^+$. The result follows by applying rule (stt-prod).

(ast-ind) Similarly to the previous case.

(ast-conv) By Cor. 3.62.

□

In the following we prove that typing is preserved from CIC$\widehat{\ }_\eta$ to ECIC$\widehat{\ }_{_}$. First we prove a result on preservation of reduction. Preservation of reduction is proved for well-typed terms, since it is necessary to check that domain and codomain annotations agree.

Lemma 3.64. *Let M^+ be a well-typed term. If $\lfloor M^+ \rfloor \to N$, then there exists N^+ such that $M^+ \to_{\mathsf{t}}{}^+ N^+$ and $\lfloor N^+ \rfloor \equiv N$.*

Proof. We proceed by induction on $M \to N$. The interesting cases are when the reduction is at the head.

β-redex M^+ is of the form $\mathsf{app}^{U_2^\circ}_{x_2:T_2^\circ}\left(\lambda^{U_1^\circ}_{x_1:T_1^\circ}.M_1, M_2\right)$ and $N \equiv \lfloor M_1\,[x_1 := M_2]\rfloor$. By Lemma 3.56, $T_1^\circ \approx_{\mathsf{I}} T_2^\circ$ and $U_1^\circ \approx_{\mathsf{I}} U_2^\circ$. By Cor. 3.58, there exists T_3° and U_3° such that $T_1^\circ, T_2^\circ \to_{\mathsf{t}}{}^* T_3^\circ$ and $U_1^\circ, U_2^\circ \to_{\mathsf{t}}{}^* U_3^\circ$. Then, $M \to_{\mathsf{t}}{}^* \mathsf{app}^{U_3^\circ}_{x_2:T_3^\circ}\left(\lambda^{U_3^\circ}_{x_1:T_3^\circ}.M_1, M_2\right) \to_{\mathsf{t}} M_1\,[x_1 := M_2]$. (We can assume that $x_1 = x_2$.) We take $N^+ \equiv M_1\,[x_1 := M_2]$.

μ-redex M^+ is of the form $\mathsf{fix}_n\ f : T^\star := (M_1, \vec{P}\,C(\vec{p^\circ}, \vec{a}))$. We take

$$N^+ \equiv \mathsf{app}^{|U^\star|}_{|\Delta^\star|}\left(M_1\,[f := \lambda|\Delta^\star|.\mathsf{fix}_n\ f : T^\star := (M_1, \mathrm{dom}\,(\Delta^\star))], \vec{P}C(\vec{p^\circ}, \vec{a})\right),$$

where $T^\star \equiv \Pi\Delta^\star.U^\star$.

ι-redex M^+ is of the form $\mathsf{case}_{P^\circ}\ x := C_j(\vec{q^\circ}, \vec{a})\ \mathsf{in}\ I\left(\vec{p^\circ}, \vec{y}\right)\ \mathsf{of}\ \{C_i \Rightarrow N_i\}$. We take $N^+ \equiv \mathsf{app}^{P^*}_{\Delta_j{}^*}(N_j, \vec{a})$.

□

The following lemma states the relation between CIC$\widehat{\ }_\eta$ and ECIC$\widehat{\ }_-$. It is the final step in the proof of equivalence between CIC$\widehat{\ }_-$ and ECIC$\widehat{\ }_-$.

Lemma 3.65. *If $\Gamma \vdash_\eta M : T$, then there exists a judgment $\Gamma^+ \vdash_{\mathsf{t}} M^+ : T^+$ such that $\lfloor\Gamma^+\rfloor = \Gamma$, $\lfloor M^+\rfloor = M$ and $\lfloor T^+\rfloor = T$.*

Proof. We proceed by induction on the derivation of $\Gamma \vdash_\eta M : T$. We consider the case of rule (conv). The rest of the cases follow easily by IH and Lemma 3.34. The judgment $\Gamma \vdash_\eta M : T$ is derived from $\Gamma \vdash_\eta M : U$, $\Gamma \vdash_\eta T : u$, $\mathsf{simple}(U)$, and $U \le T$. By IH, there exists Γ^+, M^+, U^+ such that $\Gamma^+ \vdash_{\mathsf{t}} M^+ : U^+$. Also by IH, there exists Δ^+, T^+ such that $\Delta^+ \vdash_{\mathsf{t}} T^+ : u$. Since $\lfloor\Gamma^+\rfloor \equiv \lfloor\Delta^+\rfloor$, we have $\Gamma^+ \downarrow_{\mathsf{t}} \Delta^+$, and $\Gamma^+ \vdash_{\mathsf{t}} T^+ : u$, by Lemma 3.34. By Lemma 3.63, $U^+ \le_{\mathsf{t}} T^+$, and the result follows by an application of rule (conv).

The rest of the cases follows similarly, using Lemma 3.34 and Corollary 3.62. □

We can derive SN and LC for CIC$\widehat{\ }_-$ from the same results for ECIC$\widehat{\ }_-$ and the equivalence between both presentations.

Corollary 3.66 (Strong normalization of CIC$\widehat{\ }_-$)**.** *If $\Gamma \vdash M : T$, then $M \in \mathrm{SN}_\to$.*

Proof. Let us assume an infinite reduction sequence starting with M: $M \equiv M_0 \to M_1 \to M_2 \to \ldots$.

By Subject Reduction for CIC$\widehat{\ }_-$ (Lemma 3.24) and Lemma 3.52, $\lceil\Gamma\rceil \vdash_\eta \lceil M_i\rceil : \lceil T\rceil$, for $i = 0, 1, \ldots$. By part 1 of Lemma 3.51, we have an infinite reduction sequence starting with $\lceil M_0\rceil$: $\lceil M_0\rceil \to'^+ \lceil M_1\rceil \to'^+ \lceil M_2\rceil \to'^+ \ldots$.

By Lemma 3.65, there exists Γ^+, M_0^+ and T^+ such $\lfloor\Gamma^+\rfloor = \lceil\Gamma\rceil$, $\lfloor M_0^+\rfloor = \lceil M_0\rceil$, $\lfloor T^+\rfloor = \lceil T\rceil$, and $\Gamma^+ \vdash_{\mathsf{t}} M_0^+ : T_0^+$. By Lemma 3.64, there exists an infinite reduction sequence starting from M_0^+: $M_0^+ \to_{\mathsf{t}}{}^+ M_1^+ \to_{\mathsf{t}}{}^+ M_2^+ \ldots$, such that $\lfloor M_i^+\rfloor \equiv \lceil M_i\rceil$.

Since ECIC$\widehat{\ }_-$ is strongly normalizing (Theorem 3.49), no such sequence exists, and the result follows. □

Corollary 3.67 (Logical consistency of CIC$\widehat{\ }_-$)**.** *There is no term M such that $\vdash M : \mathsf{False}$.*

Proof. By Lemma 3.52, Lemma 3.65, and Theorem 3.50. □

Chapter 4

Strong Normalization

In this chapter we prove the main technical contribution of this thesis: strong normalization and logical consistency of ECIC$^{\frown}_{_}$. The proof is based on Λ-sets as introduced by Altenkirch in his PhD thesis [7], and later used by Melliès and Werner [63] to prove strong normalization for Pure Type Systems. Our development follows more closely the latter.

The chapter is organized as follows. First, we show an overview of the proof containing only informal explanations but with enough details to give some intuition on the steps that we follow in the rest of the chapter (Sect. 4.1). Second, we introduce formally all the concepts and definitions needed (Sect. 4.2). Third, we introduce the interpretation of terms and types and show some basic properties (Sect. 4.3). Finally, we prove that the interpretation is sound (Sect. 4.4). As a consequence, we obtain Strong Normalization and Logical Consistency (Sect. 4.5).

4.1 Overview of the Proof

As mentioned, the proof of SN is obtained from a model of the theory based on Λ-sets. This model can be seen as a realizability interpretation modified for proving normalization. We give a short overview of the proof referring to the definitions and lemmas on the rest of the chapter.

Let us begin by introducing informally the main concept used in the proof: Λ-sets. A Λ-set X is a pair $(X_\circ, \Vdash)$, where $X_\circ$ is a set called the *carrier-set*, and $\Vdash \subseteq \mathrm{SN} \times X_\circ$ is a relation called the *realizability relation* (SN denotes the set of strongly normalizing terms). Types are interpreted as Λ-sets, whereas terms are interpreted as elements in the interpretation of their types. The interpretation is in part set-theoretical: product (Π) types are interpreted by (dependent) function spaces, abstractions (λ) are interpreted as functions, applications are interpreted as set-theoretical function application, and so on. Let us denote the interpretation of a term M by $[M]_\gamma$, where γ is the interpretation of the free variables of M. The soundness theorem says that if $\Gamma \vdash M : T$ and γ is an adequate interpretation of the free variables of M, then

$$[M]_\gamma \in X_\circ,$$

where $[T]_\gamma$ is a Λ-set $(X_\circ, \Vdash)$. Intuitively, we say that γ is adequate if $\gamma(x) \in [U]$ for each $(x : U) \in \Gamma$. One of the advantages of interpreting the constructions of the language in set-theoretical terms is that it gives a clear intuition of their meaning.

As a consequence of soundness it is possible to prove SN for well-typed terms: if $\Gamma \vdash M : T$, then

$$M \models [M]_\gamma,$$

for some adequate γ where $[T]_\gamma = (X_\circ, \models)$. SN follows from the fact that realizers in $\models$ are, by definition, strongly normalizing.

Inductive types

The Λ-set model can be adapted to the case of inductive types. Altenkirch [7] proves that CC extended with trees at the impredicative level satisfies SN, by extending the Λ-set model. In our case, we consider inductive types only at the predicative level.

Intuitively, the meaning of an inductive type is the smallest set that is closed under application of constructors. For example, let us consider the case of nat. We interpret constructors with tags using natural numbers: $\{\mathsf{O} \mapsto 0, \mathsf{S} \mapsto 1\}$. The semantic meaning of nat is given by

$$[\mathsf{nat}] = \{(0,\emptyset), (1,(0,\emptyset)), (1,(1,(0,\emptyset))), \ldots\} \ .$$

This set is obtained as the least fixed point of a monotone operator derived from the definition of nat:

$$\phi(X) = \{(0,\emptyset)\} \cup \{(1,x) : x \in X\} \ .$$

We need transfinite recursion to reach a fixpoint in the case of higher-order inductive types. It is defined as follows. Given a monotone operator ϕ and an ordinal $\mathfrak{a}$ we define the set $\phi^{\mathfrak{a}}$ by the following equations:

$$\begin{aligned} \phi^0 &= \emptyset \\ \phi^{\mathfrak{a}+1} &= \phi(\phi^{\mathfrak{a}}) \\ \phi^{\mathfrak{b}} &= \bigcup_{\mathfrak{a}<\mathfrak{b}} \phi^{\mathfrak{a}}, \text{ where } \mathfrak{b} \text{ is a limit ordinal.} \end{aligned}$$

For example, in the case of nat, we reach a fixpoint after ω iterations (where ω is the smallest infinite ordinal). In the case of False (which has no constructors), the monotone operator derived is $\phi_{\mathsf{False}}(X) = \emptyset$. Hence, the interpretation of the type is $[\mathsf{empty}] = \emptyset$.

For proving SN, empty types are not allowed in the interpretation. Otherwise, we will not be able to ensure SN in inconsistent contexts. For example, in the judgment $(x : \mathsf{False}) \vdash M : T$ there is no valid interpretation of M since we cannot instantiate x.

To interpret inductive types as a Λ-set we take the set-theoretical interpretation given above and add extra elements to avoid having empty interpretations. The realizability relation is extended in a natural way. The intuitive interpretation of nat is given by a pair $(\mathcal{N}, \models_{\mathsf{nat}})$ where

$$\mathcal{N} = \{(0,\emptyset), (1,(0,\emptyset)), (1,(1,(0,\emptyset))), \ldots\} \cup \{\bot, (1,\bot), (1,(1,\bot)), \ldots\}$$

and $\bot$ is an element that is included in the interpretation of all inductive definitions. The realizability relation states is defined such that, for example,

$$\begin{aligned} M \models_{\mathsf{nat}} (0,\emptyset) &\iff M \to^* \mathsf{O} \\ M \models_{\mathsf{nat}} (1,\alpha) &\iff M \to^* \mathsf{S}(N) \wedge N \models_{\mathsf{nat}} \alpha \end{aligned}$$

We omit the definition of the relation for elements contained $\bot$, since it is not important at this point. In Sect. 4.3.3 we present the formal definition of the interpretation of inductive types as the least fixed point of a monotone operator.

4.1.1 The case of CIC$\hat{}$.

For proving SN for CIC$\hat{}$, we adapt the basic Λ-set model described above. The main aspects of the type-based termination approach have a direct and intuitive semantics in the Λ-sets model. Sized types are interpreted as approximations of the monotone operators described above, where stages are interpreted by ordinals. For example, the interpreation of a type nat^s is given by $[\mathsf{nat}^s] = \phi^{[s]}_{\mathsf{nat}}$, where the interpretation of a stage s is an ordinal. The semantic interpretation of subtyping is subset inclusion: $\mathsf{nat}^s \leq \mathsf{nat}^r \Rightarrow [\mathsf{nat}^s] \subseteq [\mathsf{nat}^r]$. The stage ∞ is interpreted by an ordinal big enough to ensure that the least fixed point of ϕ_{nat} is reached. In the case of nat is sufficient to set $[\infty] = \omega$, but for higher-order types bigger ordinals are necessary.

Although sized types have a semantically intuitive interpretation, there are other aspects that do not fall directly into this model. We discuss them briefly and then show our proposed solution.

Contravariance rule. Recall that the Λ-set model can be seen as a set-theoretical model extended with a realizability relation. A natural way of interpreting the type $\mathsf{nat}^\imath$ is $\phi^{[\imath]}_{\mathsf{nat}}$, i.e., iterating the associated monotone operator of nat iterated $[\imath]$ times, where $[\imath]$ is the interpretation of the stage $\imath$. Consider for example the subtyping relation $\mathsf{nat}^\infty \to T \leq \mathsf{nat}^\imath \to T$, for some term T. In the Λ-set model, the interpretation of these terms would be $[\mathsf{nat}^\infty \to T] = \phi^{\omega}_{\mathsf{nat}} \to [T]$ and $[\mathsf{nat}^\imath \to T] = \phi^{[\imath]}_{\mathsf{nat}} \to T$, where the arrows on the rhs of the equalities denote Λ-set function space. Note that the carrier-set of a Λ-set function space is just the set-theoretical function space.

Then, it is not true that $[\mathsf{nat}^\infty \to T] \subseteq [\mathsf{nat}^\imath \to T]$, since the domains of the functions are different (functions are interpreted as sets of pairs). In other words, the natural way of interpreting subtyping in the set-theoretical model as inclusion, is not directly valid.

While it is not true that $[\mathsf{nat}^\infty \to T] \subseteq [\mathsf{nat}^\imath \to T]$, there is, a coercion from one set to the other. We could interpret subtyping by a coercions functions. However, this means that the interpretation would depend, not only of the term being interpreted, but also on the type derivation of the term. Consider a simplified derivation of the form

$$\frac{\Gamma \vdash M : \mathsf{nat}^\infty \to T \qquad \mathsf{nat}^\infty \to T \leq \mathsf{nat}^\imath \to T}{\Gamma \vdash M : \mathsf{nat}^\imath \to T}\ (\text{conv})$$

To know if we interpret M as an element of $[\mathsf{nat}^\infty \to T]$ or as an element of $[\mathsf{nat}^\imath \to T]$, we need to look at a type derivation of M. Different type derivation would lead to different interpretations. We do not pursue this approach because it would add many complications in the definition of the interpretation and the proof of soundness. Our solution, described below, follows a different path.

Lack of annotations. As we mentioned, the lack of size annotations in types has some advantages and disadvantages. The principal advantage is an efficient reduction mechanism, since we do not have to care about substitution of stage variables. However, in the interpretation, we encounter a similar problem as the one described above. Consider a term of the form $\lambda x : \mathsf{nat}.M$. This term would have a type of the form $\mathsf{nat}^s \to T$. Since s does not appear on the term itself, we cannot give a correct interpretation.

Our solution. We propose to define two interpretations. The first is used for terms that do not contain size variables; therefore the issues mentioned above are avoided. In particular, for erased terms, we assume that the missing annotations are ∞. To ensure the soundness of this interpretation and, in particular, to ensure that the size information is respected, we define a second (relational) interpretation on types.

Let us illustrate the concept with an example. Consider the type $\mathsf{nat}^s \to \mathsf{nat}^r$. We define two interpretations for this term. The first interpretation, denoted with $[\,]$, does not consider the size annotation, so we simply define $[\mathsf{nat}^s \to \mathsf{nat}^r] = [\mathsf{nat}^\infty \to \mathsf{nat}^\infty] = \phi^\omega_{\mathsf{nat}} \to \phi^\omega_{\mathsf{nat}}$. For the second interpretation, we take the size information (s and r) into account. Recall that $\phi^{[s]}_{\mathsf{nat}}$ is, intuitively, the set

$$\{0, (1,0), (1,1,0), \ldots, \underbrace{(1,1,1,\ldots(1,0))}_{[s]}\ldots)$$

We define $[\![\mathsf{nat}^s]\!]$ as a (partial) relation on the set $\phi^\omega_{\mathsf{nat}}$ (the full interpretation of nat); concretely

$$[\![\mathsf{nat}^s]\!] = \{(\alpha, \alpha) : \alpha \in \phi^{[s]}_{\mathsf{nat}}\}$$

That is, two elements are related in $[\![\mathsf{nat}^s]\!]$ if they are equal and respect the size s. The relational interpretation for products takes related elements in the domain to related elements in the codomain. For example,

$$\begin{aligned}[\![\mathsf{nat}^s \to \mathsf{nat}^r]\!] &= \{(f_1, f_2) : \phi^\omega_{\mathsf{nat}} \to \phi^\omega_{\mathsf{nat}} \mid (\alpha,\alpha) \in [\![\mathsf{nat}^s]\!] \Rightarrow (f_1(\alpha), f_2(\alpha)) \in [\![\mathsf{nat}^r]\!]\} \\ &= \{(f_1, f_2) : \phi^\omega_{\mathsf{nat}} \to \phi^\omega_{\mathsf{nat}} \mid \alpha \in \phi^{[s]}_{\mathsf{nat}} \Rightarrow f_1(\alpha) = f_2(\alpha) \in \phi^{[r]}_{\mathsf{nat}}\}\end{aligned}$$

If a type T contains variables, the relational interpretations of T takes two interpretations of free variables: $[\![T]\!]_{\gamma_1,\gamma_2}$. We extend the interpretation to contexts to ensure that γ_1 and γ_2 respect sizes.

Then, the soundness result says that if $\Gamma \vdash M : T$ and $(\gamma_1, \gamma_2) \in [\![\Gamma]\!]$, then $([M]_{\gamma_1}, [M]_{\gamma_2}) \in [\![T]\!]_{\gamma_1,\gamma_2}$. As usual the proof proceeds by induction on the type derivation.

Note that the relational interpretation solves both problems mentioned above. First, the contravariance rule is sound under this interpretation. Consider a stage $s' \sqsubseteq s$; the relational interpretation of $\mathsf{nat}^{s'} \to \mathsf{nat}^r$ gives

$$[\![\mathsf{nat}^{s'} \to \mathsf{nat}^r]\!] = \{(f_1, f_2) \in \mathbb{N} \to \mathbb{N} : \alpha \in \phi^{[s']}_{\mathsf{nat}} \Rightarrow f_1(\alpha) = f_2(\alpha) \in \phi^{[r]}_{\mathsf{nat}}\}$$

Since $[s'] \leq [s]$, we have $[\![\mathsf{nat}^s \to \mathsf{nat}^r]\!] \subseteq [\![\mathsf{nat}^{s'} \to \mathsf{nat}^r]\!]$.

As for the lack of annotations, consider the following derivation:

$$\frac{\Gamma(x : \mathsf{nat}^s) \vdash M : \mathsf{nat}^r}{\Gamma \vdash \lambda x : \mathsf{nat}.M : \mathsf{nat}^s \to \mathsf{nat}^r}$$

We interpret the term $\lambda x : \mathsf{nat}.M$ as a function with domain $\phi^\omega_{\mathsf{nat}}$; specifically $\alpha \in \phi^\omega_{\mathsf{nat}} \mapsto [M]_{\gamma,\alpha}$. Soundness of this interpretations states

$$(\alpha \in \phi^\omega_{\mathsf{nat}} \mapsto [M]_{\gamma_1,\alpha}, \alpha \in \phi^\omega_{\mathsf{nat}} \mapsto [M]_{\gamma_2,\alpha}) \in [\![\mathsf{nat}^s \to \mathsf{nat}^r]\!]$$

The proof of soundness reduces to proving that if $\alpha \in \phi^{[s]}_{\mathsf{nat}}$, then $[M]_{\gamma_1,\alpha} = [M]_{\gamma_2,\alpha} \in \phi^{[r]}_{\mathsf{nat}}$. This is exactly what the IH for the judgment $\Gamma(x : \mathsf{nat}^s) \vdash M : \mathsf{nat}^r$ says.

The formal definition of the relational interpretation is given in Sect. 4.3.5.

4.2 Preliminary Definitions

In this section we introduce the concepts necessary to define the interpretation of terms. Namely, saturated sets, Λ-sets, and inaccessible cardinals. Saturated sets a usual tool in proofs of SN. They are not strictly needed in our case, but we introduce them to relate with the definition of Λ-sets. We also present some notions of set theory. In particular, the notion of *inaccessible cardinal* which is used in the interpretation of universes.

Saturated sets

A saturated set is a set of strongly normalizing terms that satisfy some closure properties with respect to reduction. We define them in terms of elimination contexts.

Definition 4.1 (Elimination context)**.** *An elimination context is a term with a "hole", denoted by* $[]$*, that belongs to the following grammar:*

$$\begin{array}{rcl} E[] & ::= & [] \\ & | & \mathsf{app}^{\mathcal{T}^\circ}_{\mathcal{V}:\mathcal{T}^\circ}\,(E[], \mathcal{T}) \\ & | & \mathsf{case}_{\mathcal{T}^\circ}\ \mathcal{V}_{I(\vec{\mathcal{T}^\circ}, \vec{\mathcal{T}^\circ})} := E[]\ \mathsf{in}\ \mathcal{I}\left(\_, \vec{\mathcal{V}}\right)\ \mathsf{of}\ \langle C_i \Rightarrow \mathcal{T}_i\rangle_i \\ & | & \mathsf{fix}_n\ \mathcal{V} : \mathcal{T}^\star := (\mathcal{T}, \langle \mathcal{T}_i\rangle_{i=1..(n-1)} E[]) \end{array}$$

We write $E[M]$ *for the term obtained by replacing the hole of* $E[]$ *with* M*.*

Note that, in the case of fixpoint, the function is applied to n arguments, the last one being an elimination context. We define weak-head reduction in terms of elimination contexts.

Definition 4.2 (Weak-head reduction)**.** *Weak-head reduction, denoted* $\to_{\mathrm{wh}}$*, is defined by the rule*

$$E[M] \to_{\mathrm{wh}} E[N] \iff M\ \beta\iota\mu\ N\ .$$

Note on the rhs that we do not take the compatible closure of the relation, but only reduction at the head. Furthermore, note that we use loose β-reduction.

We say that a term M is in *weak-head normal form* (whnf), if there is no N such that $M \to_{\mathrm{wh}} N$. Note that a term in whnf is not necessarily in normal form. We define the notion of *atomic term* which are terms in whnf, where weak-head reduction is stopped by a variable.

Definition 4.3 (Atomic term)**.** *A term is* atomic *if it is of the form* $E[x]$*. In such case,* x *is the* head variable *of* $E[x]$*. We use* AT *to denote the set of atomic terms.*

Atomic terms have the following properties: if M is an atomic term, and $M \to N$, then N is also atomic; if $M \equiv E[x]$ (an atomic term with head variable x) and $y \neq x$, then $M\,[y := N]$ is also an atomic term with head variable x.

We formally define the notion of strongly normalizing term.

Definition 4.4 (Strongly normalizing term)**.** *Let* R *be a relation on terms. A* M *term is strongly normalizing with respect to* R*, if there is no infinite reduction sequence,* $M \equiv M_0\ R\ M_1\ R\ M_2 \ldots$*. We denote with* $\mathrm{SN}(R)$ *the set of strongly normalizing terms with respect to* R*.*

If R_1 and R_2 are two relations on terms such that $R_1 \subseteq R_2$, then $\mathrm{SN}(R_2) \subseteq \mathrm{SN}(R_1)$. Let $M \in \mathrm{SN}(R_2)$. If there exists an infinite reduction sequence $M \equiv M_0\ R_1\ M_1\ R_1\ M_2\ R_1 \ldots$,

then we would have $M \equiv M_0\ R_2\ M_1\ R_2\ M_2\ R_2\ \ldots$. Since $M \in \mathrm{SN}(R_2)$, this is impossible. Therefore, there is no infinite reduction sequence on R_1 and $M \in \mathrm{SN}(R_1)$.

By the above, $\mathrm{SN}(\to_{\mathsf{l}}) \subseteq \mathrm{SN}(\to_{\mathsf{t}})$. We are interested in strong normalization with respect to $\to_{\mathsf{l}}$. Hence, when we say strongly normalizing term, we mean strongly normalizing with respect to $\to_{\mathsf{l}}$. To simplify the notation, we write SN instead of $\mathrm{SN}(\to_{\mathsf{l}})$.

Below is the formal definition of a saturated set.

Definition 4.5 (Saturated set)**.** *A set of terms $X \subseteq \mathrm{SN}$ is saturated iff it satisfies the following conditions:*

(S1) $\mathrm{AT} \cap \mathrm{SN} \subseteq X$*;*

(S2) if $M \in \mathrm{SN}$ *and* $M \to_{\mathrm{wh}} M'$ *and* $M' \in X$*, then* $M \in X$*.*

In other words, a set of strongly-normalizing terms is saturated if it contains all strongly normalizing atomic terms and is closed by weak-head expansion. We denote with SAT the set of saturated sets. Note that SAT is a complete lattice ordered by inclusion. The greatest element is SN, and the least element is $\overline{\mathrm{AT}}$, that is the closure of AT under weak-head expansion.

Λ-sets

This is the main tool we use in the proof of SN. Our definition of Λ-sets is slightly different than the usual [7,63], as explained below.

Definition 4.6 (Λ-set)**.** *A Λ-set is a triple $X = (X_\circ, \models_X, \bot_X)$ where $X_\circ$ is a non-empty set, witnessed by $\bot_X \in X_\circ$, and $\models_X \subseteq \mathcal{T} \times X_\circ$.*

$X_\circ$ is the *carrier-set* and the elements of $X_\circ$ are called the *carriers* of X. The terms M such that $M \models_X \alpha$ for some $\alpha \in X_\circ$ are called the *realizers* of α. The element $\bot_X$ is called the *atomic element* of X. We write $\alpha \sqsubset X$ for $\alpha \in X_\circ$. A Λ-set X is included in a Λ-set Y, written $X \subseteq Y$, if $X_\circ \subseteq Y_\circ$, $\models_X \subseteq \models_Y$, and $\bot_X = \bot_Y$.

The notion of *saturated Λ-set* is necessary for proving SN.

Definition 4.7 (Saturated Λ-set)**.** *A Λ-set X is said to be* saturated *if*

1. *every realizer is strongly normalizable;*
2. *the atomic element $\bot_X$ is realized by any atomic strongly normalizable term;*
3. *for every $\alpha \in X_\circ$, if $N \models_X \alpha$, and $M \to_{\mathrm{wh}} N$ with $M \in \mathrm{SN}$, then $M \models_X \alpha$ (i.e., the set of realizers is closed under weak-head expansion).*

The relation between saturated Λ-sets and saturated sets is the following: given a saturated Λ-set X, the set of all realizers of X is a saturated set.

The difference between the definition in [7,63] and ours is that the atomic element of a Λ-set is explicit in the definition. The use of the atomic element will be evident in the definition of the interpretation of terms. However, the difference in the definition is not essential in our proof.

A saturated set that will be useful in the rest of the chapter is

$$\bot = (\{\emptyset\}, \mathrm{AT} \cap \mathrm{SN} \times \{\emptyset\}, \emptyset),$$

that is, $\bot$ is a saturated Λ-set, whose carrier is the singleton $\{\emptyset\}$ and the realizers of its only element are the atomic strongly normalizing terms.

Given a set X, we define $\overline{X}$ as the Λ-set $\overline{X} = (X \cup \{\emptyset\}, \mathrm{SN} \times X, \emptyset)$. We often use this operation in the case where $\emptyset \in X$. In such case, we write X to mean both X proper and $\overline{X}$; which one we refer will be clear from the context.

We define a product operation on Λ-sets that is used to interpret the product construction on terms.

Definition 4.8 (Product)**.** *Let X be a Λ-set and $\{Y_\alpha\}_{\alpha \in X_\circ}$ an $X_\circ$-indexed family of Λ-sets. We define the Λ-set $\Pi(X, Y)$ by:*

- $\Pi(X,Y)_\circ \stackrel{\text{def}}{=} \{f \in X_\circ \to \bigcup_{\alpha \in X_\circ} (Y_\alpha)_\circ : \forall \alpha \in X_\circ . f(\alpha) \in (Y_\alpha)_\circ\}$;
- $M \models_{\Pi(X,Y)} f \stackrel{\text{def}}{=} \forall \alpha \in X_\circ .\ N \models_X \alpha.$
  $\forall T^\circ, U^\circ, x.\ \mathsf{app}^{U^\circ}_{x:T^\circ}(M,N) \models_{Y_\alpha} f(\alpha)$;
- $\bot_{\Pi(X,Y)} \stackrel{\text{def}}{=} \alpha \in X_\circ \mapsto \bot_{Y_\alpha}$.

The following lemma state that saturated Λ-sets are closed under product.

Lemma 4.9. *If X and every $\{Y_\alpha\}_{\alpha \in X_\circ}$ are saturated Λ-sets, so is $\Pi(X, Y)$.*

We define a notion of morphism between Λ-sets.

Definition 4.10 (Λ-morphism)**.** *Let X and Y be Λ-sets. A Λ-morphism f from X to Y is a function $f : X_\circ \to Y_\circ$, such that $M \models_X \alpha \Rightarrow M \models_Y f(\alpha)$, and $f(\bot_X) = \bot_Y$.*

Definition 4.11 (Λ-iso)**.** *Let X and Y be Λ-sets. A Λ-iso f from X to Y is a Λ-morphism such that f is a one-to-one function, and $M \models_X \alpha \iff M \models_Y f(\alpha)$.*

The following operation is useful for defining the relational interpretation of types.

Definition 4.12. *Let X be a Λ-set. Then X^2 is a Λ-set where*

- $(X^2)_\circ = \{(\alpha, \alpha) : \alpha \sqsubset X\}$
- $M \models_{X^2} (\alpha, \alpha) \iff M \models_X \alpha$
- $\bot_{X^2} = (\bot_X, \bot_X)$

The following lemma states a relation between X and X^2.

Lemma 4.13. *Let X be a Λ-set. Then X and X^2 are isomorphic.*

To interpret contexts, it is useful to consider a small generalization of Λ-sets, where the realizability relation is defined on sequences of terms of a fixed length.

Definition 4.14 (Λ^n-set)**.** *A Λ^n-set is a triple $X = (X_\circ, \models_X, \bot_X)$ where $X_\circ$ is a non-empty set, witnessed by $\bot_X \in X_\circ$, and $\models_X \subseteq \mathcal{T}^n \times X_\circ$, where $\mathcal{T}^n$ denotes the set of sequences of terms of length n.*

Note that, when $n = 1$, we reduce to a Λ-set. For $n = 0$, the realizability relation only considers the empty sequence of terms, ε. We write $\emptyset$ to denote the Λ^0-set $(\{\emptyset\}, (\varepsilon, \emptyset), \emptyset)$.

The notion of saturated Λ^n-set is defined as the obvious generalization of the notion of saturated Λ-set.

Definition 4.15 (Saturated Λ^n-set)**.** *A Λ^n-set X is said to be* saturated *if*

1. *every realizer is in* SN^n*;*
2. *the atomic element* $\bot_X$ *is realized by all elements of* $\mathrm{AT}^n \cap \mathrm{SN}^n$*;*
3. *for every* $\alpha \in X_\circ$*, if* $\vec{N} \models_X \alpha$*, and* $\vec{M} \to_{\mathrm{wh}} \vec{N}$ *with* $\vec{M} \in \mathrm{SN}^n$*, then* $\vec{M} \models_X \alpha$

We write $\vec{M} \to_{\mathrm{wh}} \vec{N}$ when $\#\vec{M} = \#\vec{N}$ to mean that some element in $\vec{M}$ weak-head reduces to the corresponding element in $\vec{N}$. In other words, if $\vec{M} = M_1 \dots M_n$ and $\vec{N} = N_1 \dots N_n$, then $\vec{M} \to_{\mathrm{wh}} \vec{N}$ if $M_i \to_{\mathrm{wh}} N_i$ for some $1 \leq i \leq n$ and $M_j \equiv N_j$ for $j \neq i$.

Definition 4.16 (Sum)**.** *Let* X *be a* Λ^n*-set and* $\{Y_\alpha\}_{\alpha \in X_\circ}$ *a* $X_\circ$*-indexed family of* Λ^m*-sets. We define the* Λ^{n+m}*-set* $\Sigma(X, Y)$ *by:*

- $\Sigma(X,Y)_\circ = \sum_{\alpha \sqsubset X} (Y_\alpha)_\circ = \{(\alpha, \beta) : \alpha \sqsubset X \wedge \beta \sqsubset Y_\alpha\}$*;*
- $\vec{M}, \vec{N} \models_{\Sigma(X,Y)} (\alpha, \beta) \iff \vec{M} \models_X \alpha \wedge \vec{N} \models_{Y_\alpha} \beta$*;*
- $\bot_{\Sigma(X,Y)} = (\bot_X, \bot_{Y_{(\bot_X)}})$.

Saturated Λ-sets are closed under the sum operation.

Lemma 4.17. *If* X *is a saturated* Λ^n*-set and* $\{Y_\alpha\}_{\alpha \in X_\circ}$ *is a* $X_\circ$*-indexed family of saturated* Λ^m*-sets, then* $\Sigma(X, Y)$ *is a saturated* Λ^{n+m}*-set.*

Given a set X, we write $\mathcal{L}(X)$ to denote the set of saturated Λ-sets whose carrier set is an element of X. Similarly, $\mathcal{L}^n(X)$ denotes the set of saturated Λ^n-sets whose carrier set is an element of X, and $\mathcal{L}^*(X)$ denotes $\bigcup_n \mathcal{L}^n(X)$. We use this definitions in the case when X is the interpretation of a universe.

Notions of set theory

We need some notions of set theory for the interpretation of terms. In particular, we use inaccessible cardinals for the interpretations of universes. For the interpretation of inductive types we need some concepts from ordinal and cardinal theory. All these notions are standard and can be found in many texts on set theory (e.g. [54]).

We begin by defining the cumulative hierarchy of sets, and strongly inaccessible cardinals.

Definition 4.18 (Cumulative hierarchy)**.** *The* cumulative hierarchy *for an ordinal* α*, denoted* V_α*, is defined by transfinite induction by the following equations:*

$$\begin{aligned} V_0 &= \emptyset \\ V_{\alpha+1} &= \mathcal{P}(V_\alpha) \\ V_\lambda &= \bigcup_{\alpha<\lambda} V_\alpha \quad \textit{(for limit } \lambda\textit{)} \end{aligned}$$

Definition 4.19 (Strongly inaccessible cardinal)**.** *A cardinal number* κ *is said to be* (strongly) inaccessible *iff it satisfies the following conditions:*

(i) $\kappa > \aleph_0$ *(i.e.,* κ *is uncountable);*

(ii) for all $\lambda < \kappa$*,* $2^\lambda < \kappa$ *(i.e.,* κ *is a strong limit cardinal);*

(iii) if S *is a set of ordinals less than* κ *and if* $\mathrm{card}(S) < \kappa$*, then the ordinal* $\sup(S)$ *is less than* κ *(i.e.,* κ *is regular).*

There is a connection between the cumulative hierarchy and inaccessible cardinals that justifies their definition: if κ is an inaccessible cardinal, the set V_κ provides a model of ZF set theory. Hence, if ZF set theory is consistent, the existence of inaccessible cardinals cannot be proved inside the theory.

The main property of inaccessible cardinals that we use is that they are closed under dependent products.

Lemma 4.20. *Assume a inaccessible cardinal κ. Let $X \in V_\kappa$ and for each $x \in X$, let $Y_x \in V_\kappa$. Then $\prod_{x\in X} Y_x \in V_\kappa$.*

We use this property to justify the soundness of the product rule of CIC$\widehat{\ }$. Note that the inaccessibility of κ is not a necessary condition in the previous lemma. For example, for $\kappa = \omega$ the property is valid. The set V_ω is the set of *hereditarily finite sets*. In CC, where there are only two universes, Prop and Type, the set V_ω can be used as the interpretation of Type. Using that model, we can prove that it is not possible to define provably infinite types in CC.

In our case, given that we have a predicative hierarchy of universes, we assume the existence of inaccessible cardinals. This might seem radical, since their existence is not provable from the axioms of ZF set theory. However, their use is standard in many models of type theory with universes [41,51,56,83].

One might wonder if the use of inaccessible cardinals is unavoidable. In [83], Werner defines a set-theoretical model of CIC using inaccessible cardinals and a formalization of set theory inside CIC. He shows that there is an equivalence between the predicative universes and inaccessible cardinals, giving some evidence to the fact that inaccessible cardinals might be necessary. On the other hand, Melliès and Werner [63] avoid the use of inaccessible cardinals in their proof of SN of Pure Type Systems, by defining a finer structure on Λ-sets using relations. However, this greatly complicates the proof and, as the authors point out, it is not clear that this construction can be carried on in the presence of inductive types.

For our purposes, we assume an increasing infinite sequence of inaccessible cardinals, $\{\lambda_i\}_i$. We define $\mathcal{U}_i$ to be the set of saturated Λ-sets whose carrier set are in V_{λ_i}, i.e., $\mathcal{U}_i = \mathcal{L}(V_{\lambda_i})$. Similarly, we define $\mathcal{U}_i^n = \mathcal{L}^n(V_{\lambda_i})$. As mentioned before, we write $\mathcal{U}_i$ to mean both $\mathcal{U}_i$ and also $\overline{\mathcal{U}_i}$.

Following [51,56,83], we will interpret the sort hierarchy using these large universes. That is, the sequence of sorts $\{\mathsf{Type}_i\}_{i\in\mathbb{N}}$ is interpreted with the sequence $\{\mathcal{U}_i\}_{i\in\mathbb{N}}$, as explained below.

Note that the rule $\mathsf{Type}_0 : \mathsf{Type}_1 : \mathsf{Type}_2 \ldots$ (given by the Axiom set) is sound in this interpretation using inaccessible cardinals, i.e., $\mathcal{U}_0 \in \mathcal{U}_1 \in \mathcal{U}_2 \in \ldots$. Furthermore, we also have

$$\mathcal{U}_0 \subseteq \mathcal{U}_1 \subseteq \mathcal{U}_2 \ldots$$

meaning that the universe inclusion rule $\mathsf{Type}_0 \leq \mathsf{Type}_1 \leq \mathsf{Type}_2 \ldots$ is also sound in this interpretation [56]. However, recall that we do not consider universe inclusion in CIC$\widehat{\ }$ (cf. Sect. 5.1).

Monotone operators. Inductive definitions are interpreted by monotone operators in the universe they are defined. We state some properties that are sufficient to ensure the existence of a least fixed point of a monotone operator. This section is based on [4].

Let us consider an inductive definition $\mathsf{Ind}(I[\Delta_p]^{\vec{\nu}} : \Pi\Delta_a.\mathsf{Type}_k := \langle C_i : T_i\rangle_i)$. The interpretation of I will be the least fixed point of a monotone operator in the set $\mathcal{A} \to \mathcal{U}_k$, where $\mathcal{A}$ is the interpretation of the arguments, Δ_a. We give sufficient conditions to ensure that monotone operators defined on a set $\mathcal{A} \to \mathcal{U}$ have a least fixed point. The order relation on the set $\mathcal{A} \to \mathcal{U}$ is point-wise inclusion: given $X, Y \in \mathcal{A} \to \mathcal{U}$, we say that $X \subseteq Y$ when $X(\alpha) \subseteq Y(\alpha)$ for all $\alpha \in \mathcal{A}$.

We define transfinite iteration for an operator in the set $\mathcal{A} \to \mathcal{U}$.

Definition 4.21. *Given an operator $\phi : (\mathcal{A} \to \mathcal{U}) \to (\mathcal{A} \to \mathcal{U})$, we define the sequence ϕ_α, where α is an ordinal, by transfinite recursion, as follows:*

$$\begin{aligned} \phi_0 &= \bot \\ \phi_{\alpha+1} &= \phi(\phi_\alpha) \\ \phi_\lambda &= \bigcup_{\alpha<\lambda} \phi_\alpha \end{aligned}$$

where $\bot$ is an element of $\mathcal{A} \to \mathcal{U}$ we choose to start iterating. When $\mathcal{U}$ is a universe, we take $\bot = \alpha \in \mathcal{A} \mapsto \emptyset$.

Note that, if ϕ is a monotone operator, then the sequence $\{\phi^\alpha\}_\alpha$ is increasing. In the following, we prove a sufficient condition to ensure that the sequence $\{\phi^\alpha\}_\alpha$ will reach a fixed point.

Given $X \in \mathcal{A} \to \mathcal{U}$ we define $\mathrm{card}(X)$ as $\mathrm{card}(\bigcup_{\alpha\in\mathcal{A}} X(\alpha)_\circ)$. A monotone operator ϕ is α-based if

$$x \in \phi(X) \Rightarrow \exists Y \subseteq X. x \in \phi(Y) \wedge \mathrm{card}(Y) < \alpha$$

Lemma 4.22. *Let ϕ be an α-based monotone operator, where α is regular. Then $\phi_{\alpha+1} \subseteq \phi_\alpha$.*

Since $\{\phi_\alpha\}_\alpha$ is increasing, this means that ϕ_α is a fixed point of ϕ. In fact, it is the least one.

We also use the following property of cardinal numbers: for any cardinal $\mathfrak{a}$, there exists a greater regular cardinal (i.e., there exists $\mathfrak{b} \geq \mathfrak{a}$ such that $\mathfrak{b}$ is regular). We also assume the axiom of choice. In particular, this implies that $\mathfrak{a} + \mathfrak{b} = \mathfrak{a} \cdot \mathfrak{b} = \max\{\mathfrak{a}, \mathfrak{b}\}$, for cardinals $\mathfrak{a}$, $\mathfrak{b}$ such that both are non-zero and one of them is infinite.

4.3 The Interpretation

In this section we define the interpretation function and prove some properties such as soundness of Weakening, Substitution and Subject Reduction. In Sect. 4.4 we show the main soundness theorem.

This section is organized as follows. We begin by explaining how to treat impredicativity in the model (Sect. 4.3.1). Then we define the interpretation of terms and context that do not contain size variables (i.e., the set of free size variables is empty), and prove some properties. Then we define the relational interpretation of types and prove some properties. In this case, types are allowed to contain size variables by the simple predicate; we also define the interpretation of stages.

4.3.1 Impredicativity

Impredicativity is a delicate issue in dependent type theory [29]. It is well-known that system F polymorphism is not set-theoretical [73], in the sense that only a trivial model is sound. Since CIC contains system F as a subsystem, in a set-theoretical model, the impredicative universe is interpreted as a two-element set, representing the truth value of a proposition. By a set-theoretical model we mean a model where abstractions, applications and products are interpreted by their set-theoretical counterparts. Let us recall the product rule for propositions:

$$\frac{\Gamma \vdash T : \mathsf{Type}_k \qquad \Gamma(x:T) \vdash U : \mathsf{Prop}}{\Gamma \vdash \Pi x : T.U : \mathsf{Prop}}$$

Then Prop can only be interpreted as the set $\{\emptyset, \{\bullet\}\}$, where $\emptyset$ represents a false proposition and $\bullet$ is a set representing a true proposition. In the case of propositional types, the interpretation of $\Pi x : T.U$, denoted $[\Pi x : T.U]$, cannot be directly defined as the set-theoretical product

$$\prod_{\alpha \in [T]} [U]_\alpha$$

since it should be an element of $\{\emptyset, \{\bullet\}\}$.

But there is a natural way of interpreting an impredicative product. Note that, for $\alpha \in [T]$, $[U]_\alpha$ is either $\emptyset$, or $\{\bullet\}$. If $[U]_\alpha = \emptyset$ for some α, then $\prod_{\alpha\in[T]}[U]_\alpha = \emptyset$, and we can define $[\Pi x : T.U] = \emptyset$. Otherwise, if $[U]_\alpha = \bullet$ for all α, then $\prod_{\alpha\in[T]}[U]_\alpha$ is a singleton $\{\alpha \in [T] \mapsto \bullet\}$. In the latter case, $[\Pi x : T.U]$ is collapsed to $\{\bullet\}$.

This model is proof-irrelevant since all proofs are interpreted by the same element. Miquel and Werner [66] show that some care needs to be taken when defining this model. It is important to separate proofs from computation. Therefore, it is not trivial to extend this model in the presence of a subtyping rule of the form $\mathsf{Prop} \leq \mathsf{Type}_0$. This is the reason we do not consider this rule in our proof (cf. Sect. 5.1).

A set-theoretical model is useful to prove Logical Consistency, but not to prove Strong Normalization. For the latter, proofs can not be collapsed into the same element (ffl in our case). A relatively simple model of CC (and ECC) where the interpretation of impredicativity is not trivial is provided by the ω-set model [55,56]. A ω-set is a pair $(A, \models)$ where A is a set and $\models \subseteq \omega \times A$. The impredicative universe is then interpreted as a category of PER. The Λ-set model is just a modification of the ω-set where realizers are not natural numbers, but strongly normalizing λ-terms. Altenkirch [7] interprets impredicativity in the Λ-set model as a category of PER. In our case, we follow [63] and interpret the impredicative universe as the set of degenerated Λ-sets.

Definition 4.23. *A Λ-set X is* degenerated*, if the carrier-set $X_\circ$ is a singleton $\{A\}$, where A is a saturated set, $M \models_X A$ iff $M \in A$, and $\bot_X = A$.*

We write $\overline{A}$ for the degenerated Λ-set corresponding to a saturated set A.

Consider again the product rule for propositions:

$$\frac{\Gamma \vdash T : \mathsf{Type}_k \qquad \Gamma(x:T) \vdash U : \mathsf{Prop}}{\Gamma \vdash \Pi x : T.U : \mathsf{Prop}}$$

In this case $[T]$ is a Λ-set X, and $[U]_\alpha$ is a degenerated Λ-set Y_α for each $\alpha \sqsubset [T]$. Let $Y_\alpha = \{y_\alpha\}$. The interpretation of $\Pi x : T.U$ cannot be $\Pi([T], [U]\_)$, since it should be a degenerated

Λ-set. However, note that the carrier-set of $\Pi([T],[U]_{\_})$ is the singleton $\{\alpha \sqsubset X \mapsto y_\alpha\}$. Then, there is a canonical representation of $\Pi(X,Y)$ given by the degenerated Λ-set corresponding to the saturated set

$$\downarrow(X,Y) = \{M \in \mathrm{SN} : N \models_X \alpha \Rightarrow \mathsf{app}^{U^\circ}_{x:T^\circ}(M,N) \models_{Y_\alpha} y_\alpha\} \ .$$

Note that there is a Λ-iso $p_{(X,Y)} : \Pi(X,Y) \to \overline{\downarrow(X,Y)}$.

We use the isomorphism $p(X,Y)$ to convert between the interpretation of a proof term as an element of $\Pi(X,Y)$ (if it is needed as a function), or as the canonical proof $\downarrow(X,Y)$. For this reason it is necessary to have the domain and codomain annotations in abstractions and application. For example, let us consider a proof term of the form $\mathsf{app}^{U^\circ}_{x:T^\circ}(M,N)$. In this case M is also a proof, and its interpretation $[M]$ is an element of a degenerated Λ-set. Then, the interpretation of $\mathsf{app}^{U^\circ}_{x:T^\circ}(M,N)$ cannot be defined as the function application $[M][N]$, since $[M]$ is not a function. We use the above isomorphism to transform $[M]$ into a function, more precisely, a function in $[\Pi x : T^\circ.U^\circ]$.

This transformation is only needed in the case of proofs. Predicative products are interpreted by the product operation on Λ-sets. For convenience, we define the functions $\Pi_{\Gamma\vdash T}(X,Y)$, $\downarrow^{\Gamma\vdash T}_{\Pi(X,Y)}$, $\uparrow^{\Gamma\vdash T}_{\Pi(X,Y)}$ that give the interpretation of products, abstractions, and applications (respectively), depending if the type T is a proposition or a computational type. In the case of proposition, these functions convert between $\Pi(X,Y)$ and the canonical representation $\downarrow(X,Y)$. Otherwise, there is no conversion needed. The relation between $\Pi_{\Gamma\vdash T}(X,Y)$, $\downarrow^{\Gamma\vdash T}_{\Pi(X,Y)}$, $\uparrow^{\Gamma\vdash T}_{\Pi(X,Y)}$ is given by the following:

$$\downarrow^{\Gamma\vdash T}_{\Pi(X,Y)} : \Pi(X,Y) \to \Pi_{\Gamma\vdash T}(X,Y)$$
$$\uparrow^{\Gamma\vdash T}_{\Pi(X,Y)} : \Pi_{\Gamma\vdash T}(X,Y) \to \Pi(X,Y)$$

where $\downarrow^{\Gamma\vdash T}_{\Pi(X,Y)}$ and $\uparrow^{\Gamma\vdash T}_{\Pi(X,Y)}$ are Λ-isomorphisms between $\Pi(X,Y)$ and $\Pi_{\Gamma\vdash T}(X,Y)$ which is the interpretation of a product. The definition depends on whether T is a proposition or a computational type, and is given by:

- if $\Gamma^\infty \vdash T^\infty : \mathsf{Prop}$, then $\Pi_{\Gamma\vdash T}(X,Y) = \overline{\downarrow(X,Y)}$, $\downarrow^{\Gamma\vdash T}_{\Pi(X,Y)} = p(X,Y)$, and $\uparrow^{\Gamma\vdash T}_{\Pi(X,Y)} = p^{-1}(X,Y)$;
- if $\Gamma^\infty \vdash T^\infty : \mathsf{Type}_i$, then $\Pi_{\Gamma\vdash T}(X,Y) = \Pi(X,Y)$ and $\downarrow^{\Gamma\vdash T}_{\Pi(X,Y)} = \uparrow^{\Gamma\vdash T}_{\Pi(X,Y)} = \mathrm{id}_{\Pi(X,Y)}$.

To avoid cluttering, we write $\uparrow^{\Gamma\vdash \Pi x:T.U}_{\gamma}$ to mean $\uparrow^{\Gamma\vdash \Pi x:T.U}_{\Pi([\Gamma\vdash T](\gamma),[\Gamma(x:T)\vdash U](\gamma,\_))}$; similarly for the inverse operation.

4.3.2 Interpretation of Terms and Contexts

We present the first interpretation of terms. As we mentioned, it only makes sense for full terms (i.e., terms with no size variables).

Given a bare context Γ, a full term M (with no size variables) and an interpretation γ of the free variables of M, the interpretation of M under Γ is denoted $[\Gamma \vdash M]_\gamma$. The interpretation of a bare context Γ is denoted $[\Gamma]$. In the rest of the chapter, we assume a valid signature Σ. The interpretation of terms and contexts, depends on the interpretation of the inductive types defined in Σ (Sect. 4.3.3). For the moment, we assume a function $\mathcal{I}$ such that $\mathcal{I}(I)$ is the interpretation of I, for $I \in \Sigma$. The definition of $\mathcal{I}$ is given in Sect. 4.3.3. The interpretation of I is a function that takes as arguments an interpretation of the parameters and arguments and returns a Λ-set. The elements in the carrier-set of this Λ-set are of the form

– (j, ρ), where j is a tag corresponding to the j-th constructor of I and ρ is the interpretation of the arguments of the constructor;
– or $\emptyset$ which is the atomic element.

We define the interpretation by induction on the structure of terms and contexts. As usual in proofs of SN, the interpretation is defined as a partial function. Only in the proof of soundness it is ensured that for well-typed terms the interpretation is well-defined.

Terms

Definition 4.24 (Interpretation of terms)**.** *We define the interpretation function by induction on the structure of terms.*

– $[\Gamma \vdash \mathsf{Type}_i]_\gamma = \mathcal{U}_i$
– $[\Gamma \vdash \mathsf{Prop}]_\gamma = \{X : X \text{ is a degenerated } \Lambda\text{-set}\}$
– $[\Gamma \vdash x]_\gamma = \gamma_i$
  – *if* $\Gamma = (x_1 : T_1) \dots (x_n : T_n)$, $\gamma = \gamma_1, \dots, \gamma_n$, *and* $x = x_i$;
– $[\Gamma \vdash \Pi x : T.U]_\gamma = \Pi_{\Gamma \vdash \Pi x:T.U}([\Gamma \vdash T]_\gamma, [\Gamma(x : T) \vdash U]_{\gamma,\_})$
  – *if* $[\Gamma \vdash T]_\gamma$ *is defined,*
  – $[\Gamma(x : T) \vdash U]_{\gamma,\alpha}$ *is defined for any* $\alpha \sqsubset [\Gamma \vdash T]_\gamma$, *and*
  – $\Gamma \vdash \Pi x : T.U : u$ *for some sort* u;
– $[\Gamma \vdash \lambda^{U^\circ}_{x:T^\circ}.M]_\gamma = \downarrow_\gamma^{\Gamma \vdash \Pi x:T^\circ.U^\circ}([\Gamma(x : T^\infty) \vdash M]_{\gamma,\_})$
  – *if* $[\Gamma \vdash T^\infty]_\gamma$ *is defined,*
  – $[\Gamma(x : T^\infty) \vdash M]_{\gamma,\alpha}$ *is defined for all* $\alpha \sqsubset [\Gamma \vdash T^\infty]_\gamma$, *and*
  – $\Gamma^\infty \vdash \Pi x : T^\infty.U^\infty : u$ *for some sort* u;
– $[\Gamma \vdash \mathsf{app}^{U^\circ}_{x:T^\circ}(M, N)]_\gamma = \uparrow_\gamma^{\Gamma \vdash \Pi x:T^\circ.U^\circ}([\Gamma \vdash M]_\gamma)([\Gamma \vdash N]_\gamma)$
  – *if* $[\Gamma \vdash M]_\gamma$ *and* $[\Gamma \vdash N]_\gamma$ *are defined;*
– $[\Gamma \vdash I^\infty(\vec{p}, \vec{a})]_\gamma = \mathcal{I}(I)([\Gamma \vdash \vec{p}]_\gamma)([\Gamma \vdash \vec{a}]_\gamma)$
  – *if* $[\Gamma \vdash \vec{p}]_\gamma$ *and* $[\Gamma \vdash \vec{a}]_\gamma$ *are defined;*
– $[\Gamma \vdash C_j(\vec{p^\circ}, \vec{a})]_\gamma = (j, [\Gamma \vdash \vec{a}]_\gamma)$, *if* $[\Gamma \vdash \vec{a}]_\gamma$ *is defined, where* C_j *is the* j*-th constructor in an inductive type* $I \in \Sigma$.
– $[\Gamma \vdash \mathsf{case}_{P^\circ}\ x_{I(\vec{p^\circ}, \vec{a^\circ})} := M \text{ in } I(\_, \vec{y}) \text{ of } \langle C_i \Rightarrow N_i \_ i \rangle]_\gamma =$
  – $\uparrow_\gamma^{\Gamma \vdash \mathsf{branch}^s_{C_j}(\vec{p}, \vec{y}.x.P)}[\Gamma \vdash N_j]_\gamma(\rho)$, *if* $[\Gamma \vdash M]_\gamma = (j, \rho)$ *is defined,* $[\Gamma \vdash N_j]_\gamma(\rho)$ *is defined, and* $\uparrow_\gamma^{\Gamma \vdash \mathsf{branch}^s_{C_j}(\vec{p}, \vec{y}.x.P)}$ *is defined;*
  – $\bot_{[\Gamma(\mathsf{caseType}^s_I(\vec{p}, \vec{y}, x)) \vdash P](\gamma, \alpha, \bot)}$, *if* $[\Gamma \vdash M]_\gamma = \emptyset$ *is defined, and* $[\Gamma(\mathsf{caseType}^s_I(\vec{p}, \vec{y}, x)) \vdash P]_{\gamma,\alpha,\bot}$ *is defined.*
– $[\Gamma \vdash \mathsf{fix}_n\ f : T^\star := (M, \vec{N})]_\gamma = \uparrow \epsilon(F, P)([\Gamma \vdash \vec{N}]_\gamma)$, *if* $[\Gamma^\infty \vdash T^\infty]_\gamma$ *is defined, where*
  – $T^\star \equiv \Pi \Delta^\star.U^\star$, *with* $\#\Delta^\star = n$; *let* $\Delta^\star = \Delta'^\star(x : I^\star(\vec{p})\,\vec{a})$;
  – $F \sqsubset [\Gamma \vdash T^\infty]_\gamma$;
  – P *is the conjunction of the following properties:*

  (F1) $\uparrow F(\alpha, \emptyset) = \bot_{[U^\infty](\gamma,\alpha,\emptyset)}$;

  (F2) $\uparrow F(\alpha, \beta) = \uparrow([\Gamma(f : |T|) \vdash M]_{\gamma,F})(\alpha, \beta)$, *for all* $(\alpha, \beta) \sqsubset [\Gamma \vdash \Delta^\infty(I^\infty(\vec{p}, \vec{a}))]_\gamma$ *and* $\beta \neq \emptyset$

  – *and* $\uparrow = \uparrow_\gamma^{\Gamma \vdash |T^\star|}$.

We write $[\Gamma\Delta \vdash M]_{\gamma,\_}$ *as a short hand for* $\delta \sqsubset [\Gamma \vdash \Delta]_\gamma \mapsto [\Gamma\Delta \vdash M]_{\gamma,\delta}$.

The interpretation of basic terms follows [63]. In the case of product, abstraction and application we check if we are in the propositional universe, or in a computational universe. For abstraction, we interpret the body as a function on the parameter of the function. Using function $\downarrow$ we convert this function into an element of the interpretation of the product $\Pi x : T^\infty.U^\infty$. If this product is in a computational universe, no conversion is needed. Otherwise, the interpretation of the product is a degenerated set, so we convert the function to the only element in the carrier-set.

In the application we have the opposite case. The interpretation of the term M is an element of the interpretation of the product $\Pi x : T^\infty.U^\infty$. We use function $\uparrow$ to convert this element back into a function and we apply to the interpretation of the argument N. The interpretation is partial, since at this point we do not know if N is in the domain of the function.

In the inductive type case, we just apply the interpretation of the type (which we assume to exists) to the parameters and arguments. The interpretation of inductive types is given in the next section.

In the interpretation of case expression we see the reason we explicitly add the atomic element as part of the definition of Λ-set. If the interpretation of M is $\emptyset$, we cannot choose any branch. In this case it is safe to return the atomic element of the type. Otherwise, the interpretation of M is (j, ρ), hence we apply the interpretation of the corresponding branch to the arguments of the constructor ρ. As in the application case, we use function $\uparrow$ to transform the interpretation of the branch in an actual function.

Finally, in the fixpoint case we use Hilbert's choice operator to choose a function on the interpretation of T^∞ that satisfy the property of being invariant under computation. We refer to the equations that define the properties of the function as the *fixpoint equations*. Soundness of subject reduction (SSR) states that the interpretation is invariant under reductions: if $[\Gamma \vdash M]_\gamma T$ is defined and $M \to N$, then $[\Gamma \vdash N]_\gamma$ is defined and $[\Gamma \vdash M]_\gamma = [\Gamma \vdash N]_\gamma$. The properties we require for the interpretation of a fixpoint ensure directly SSR (proved as Lemma 4.33).

Note that the invariance under reduction is only required when the recursive argument is (the interpretation of) a constructor. If the recursive argument is the atomic element, we return the atomic element of the type (similar to the situation with case expressions mentioned above). For example, consider the following two terms (for the sake of readability, we omit the codomain annotations):

$$M_1 \equiv \lambda x : \mathsf{nat}.\mathsf{O}$$
$$M_2 \equiv \lambda x : \mathsf{nat}.\mathsf{fix}_1\ f : \mathsf{nat}^\star \to \mathsf{nat} := (\lambda x : \mathsf{nat}.\mathsf{O}, x)$$

Both represent the constant function returning zero (with the natural numbers as domain). However, their interpretation is different:

$$[M_1] = \alpha \sqsubset \mathcal{I}(\mathsf{nat}) \mapsto 0$$
$$[M_2] = \alpha \sqsubset \mathcal{I}(\mathsf{nat}) \mapsto \begin{cases} \emptyset & \text{if } \alpha = \emptyset \\ 0 & \text{if } \alpha \neq \emptyset \end{cases}$$

where $[\mathsf{O}] = 0$, and $\emptyset$ is the atomic element of $\mathcal{I}(\mathsf{nat})$. The difference in the interpretation corresponds to the different behavior of M_1 and M_2 when applied to an atomic element. For example, $\mathsf{app}\,(M_1, x) \to \mathsf{O}$, while $\mathsf{app}\,(M_2, x)$ is in normal form.

The use of Hilbert's choice operator is related to the soundness of the fixpoint reduction rule. In the main soundness theorem, we prove that there is only one function satisfying the fixpoint equations, showing the correctness of the definition. Without the use of Hilbert's operator, we would not be able to show separately the SSR and soundness of the interpretation. This suggests that we can avoid the use of Hilbert's operator if we use a judgmental equality instead of a conversion rule. In the judgmental presentation of type theory, we replace the conversion rule with a judgment as follows:

$$\frac{\Gamma \vdash M : T \qquad \Gamma \vdash T \approx U : u}{\Gamma \vdash M : U}$$

The judgment $\Gamma \vdash T \approx U : u$ ensures that T is convertible of U and that both have type u. Furthermore, the judgmental equality ensures that the conversion between T and U is performed on well-typed terms. Then, the proof of soundness of the typing judgment and the judgmental equality are done at the same time, since the definition is mutual. This way, we would have enough information to prove that the invariance of fixpoint reduction, since we would be dealing with well-typed terms.

Conversion rule and judgmental equality are two ways of presenting a dependent type theory. Examples of systems using the conversion presentation include the systems we mentioned so far (CC, ECC, CIC, Coq), as well as Pure Type Systems [11], and the Logical Framework [45]. In particular, the conversion presentation is easier and more efficient to implement since, to check conversion between two types, it is not necessary to type-check all the intermediate terms. On the other hand, in the presentation using judgmental equality, defining models is relatively easier. Examples of systems using the judgmental equality are Martin-Löf's Type Theory [68] and Luo's UTT [56].

Since both presentations essentially define the same system, a natural question is whether both presentation are equivalent. One way of proving equivalence is as a consequence of Strong Normalization. Adams [6] gave a syntactic proof of equivalence for functional Pure Type Systems. Recently, Siles and Herbelin [75,76] extended Adams's proof all Pure Type Systems.

Contexts

We interpret contexts as Λ^n-sets, where n is the length of the context.

Definition 4.25 (Interpretation of contexts)**.** *The interpretation of contexts is defined by the following rules.*

- $[\Gamma \vdash []]_\gamma = \emptyset$;
- $[\Gamma \vdash \Delta(x : T)]_\gamma = \Sigma([\Gamma \vdash \Delta]_\gamma, [\Gamma\Delta \vdash T]_{\gamma,\_})$;

We write $[\Gamma]$ *for* $[\vdash \Gamma]$.

We prove some lemmas about the interpretation.

Lemma 4.26. *Let* $T \equiv \Pi\Delta.U$ *be a term and* Γ *a context. Assume that* $[\Gamma \vdash T]_\gamma$ *is defined,* $[\Gamma \vdash \Delta]_\gamma$ *is defined and* $[\Gamma\Delta \vdash U]_{\gamma,\delta}$ *is defined for any* $\delta \sqsubset [\Gamma \vdash \Delta]_\gamma$*. Then* $[\Gamma \vdash T]_\gamma \cong \Pi([\Gamma \vdash \Delta]_\gamma, [\Gamma\Delta \vdash U]_{\gamma,\_})$.

Proof. By induction on the length of Δ. □

Lemma 4.27. *Let Γ, Δ_1, Δ_2 be contexts. Assume that $[\Gamma \vdash \Delta_1]_\gamma$ is defined and $[\Gamma\Delta_1 \vdash \Delta_2]_{\gamma,\delta}$ is defined for any $\delta \sqsubset [\Gamma \vdash \Delta_1]_\gamma$. Then $[\Gamma \vdash \Delta_1\Delta_2]_\gamma \cong \Sigma([\Gamma \vdash \Delta_1]_\gamma, [\Gamma\Delta_1 \vdash \Delta_2]_{\gamma,\_})$.*

Proof. By induction on the length of Δ_1. □

We use the above equivalences without explicit mention to the morphisms.

4.3.3 Interpretation of Inductive Types

Inductive types are interpreted as the least fixed point of monotone operators defined in the universe they live in. To simplify the presentation, we assume that inductive types are defined in a specific way that we describe in the following. Let $\mathsf{Ind}(I[\Delta_p]^{\vec{\nu}} : \Pi\Delta_a.u := \langle C_i : \Pi\Delta_i.\mathcal{X}\,\vec{t_i}\rangle_i)$ be an inductive definition in some signature Σ. We assume that, in each constructor, non-recursive argument appear first, followed by the recursive arguments. That is, each Δ_i can be divided in two parts: $\Delta_i \equiv \Delta_i^{\mathrm{nr}}\,\Delta_i^{\mathrm{r}}$, where Δ_i^{nr} contains the non-recursive arguments and Δ_i^{r} contains the recursive arguments. Sometimes we write $C_i(\vec{p^\circ}, \vec{N}, \vec{R})$ to separate the non-recursive arguments, $\vec{N}$, from the recursive arguments, $\vec{R}$.

Recall that the variable $\mathcal{X}$ appears strictly positive in the recursive arguments. We assume that, in each recursive argument, the recursive variable $\mathcal{X}$ appears as argument of an inductive type. That is, each recursive argument is of the form $\Pi\Theta.J^\infty[\mathcal{X}\,\vec{u}]$, where J is an inductive type defined in Σ and $\vec{u}$ are the arguments of I. For this definition to be valid, J must have only one parameter which is strictly positive.

These assumptions do not change the expressiveness of the system. For a given inductive type, arguments of constructors can be reordered so that recursive arguments appear last (recall that recursive arguments are not dependent), while non-recursive arguments appear first (reordering can be done preserving dependencies). For example, we can define the natural numbers as $\mathsf{Ind}(\mathsf{nat} : \mathsf{Type}_0 := \mathsf{O} : \mathcal{X}, \mathsf{S} : \mathsf{id}\,(\mathcal{X}) \to \mathcal{X})$, where id is an inductive type defined as $\mathsf{Ind}(\mathsf{id}(A^\oplus : \mathsf{Type}_0) : \mathsf{Type}_0 := \mathsf{in} : A \to \mathcal{X})$. Note that $\mathsf{id}\,(A)$ and A are equivalent.

The interpretation. Let Σ be a valid signature, and Φ a function that assigns an interpretation Φ_I for each inductive type $I \in \Sigma$. For each $\mathsf{Ind}(I[\Gamma]^{\vec{\nu}} : A := \Theta) \in \Sigma$ we assume

- $[\Gamma]$ is defined;
- $[\Gamma \vdash A]_\gamma$ is defined for any $\gamma \sqsubset [\Gamma]$;
- given $\gamma \sqsubset [\Gamma]$, $\Phi_I(\gamma) \sqsubset [\Gamma \vdash A]_\gamma$.

The function Φ_I takes an interpretation of the parameters of I and gives the interpretation in the type of I.

Let I be an inductive definition under signature Σ given by

$$\mathsf{Ind}(I[\Delta_p]^{\vec{\nu}} : \Pi\Delta_a.\mathsf{Type}_k := \langle C_i : T_i\rangle_i)$$

where $T_i \equiv \Pi\Delta_i.\mathcal{X}\,\vec{t_i}$. In the following, we define the interpretation of I as an operator ϕ_I.

Following the assumptions mentioned above, we assume that each Δ_i can be divided in two parts, containing the non-recursive and recursive arguments: $\Delta_i \equiv \Delta_i^{\mathrm{nr}}\,\Delta_i^{\mathrm{r}}$. Furthermore, $\Delta_i^{\mathrm{r}} \equiv \langle y_{ij} : \Pi\Theta_{ij}.J_{ij}\,(\mathcal{X}\,\vec{u_{ij}})\rangle_j$, where $\langle J_{ij}\rangle_{i,j}$ are inductive types defined in Σ.

We assume that all the components of I are defined. Concretely, we assume:

i. $[\Delta_p]$ is defined;

ii. $[\Delta_p \vdash \Delta_a]_\delta$ is defined for $\delta \sqsubset [\Delta_p]$;

iii. $[\Delta_p \vdash \Delta_i^{\mathrm{nr}}]_\delta$ is defined and belongs to $\mathcal{U}_k^m$ for $\delta \sqsubset [\Delta_p]$ where $m = \#\Delta_i^{\mathrm{nr}}$;

iv. $[\Delta_p \Delta_i^{\mathrm{nr}} \vdash \Theta_{ij}]_{\delta,\nu}$ is defined and belongs to $\mathcal{U}_k^m$ for $\delta, \nu \sqsubset [\Delta_p \Delta_i^{\mathrm{nr}}]$, where $m = \#\Theta_{ij}$;

v. $[\Delta_p \Delta_i^{\mathrm{nr}} \Theta_{ij} \vdash \vec{u_{ij}}]_{\delta,\nu,\rho}$ is defined and belongs to $[\Delta_p \vdash \Delta_a]_\delta$, for $\delta, \nu, \rho \sqsubset [\Delta_p \Delta_i^{\mathrm{nr}} \Theta_{ij}]$;

vi. $[\Delta_p \Delta_i^{\mathrm{nr}} \vdash \vec{t_i}]_{\delta,\nu}$ is defined and belongs to $[\Delta_p \vdash \Delta_a]_\delta$ for $\delta, \nu \sqsubset [\Delta_p \Delta_i^{\mathrm{nr}}]$.

We use the following notation in the rest of the section: if X is a term, a context, or a sequence of terms, we write $[X]_\gamma$ instead of $[\Gamma \vdash X]_\gamma$, when the context Γ is clear.

The interpretation of I is, as usual, a monotone operator. In our case, in $\mathcal{U}_k$, i.e. in the (interpretation of the) universe where I is defined. We define the operator, denoted with ϕ_I, prove that is monotone, and show that, under certain assumptions, it has a least fixed point in the universe. These conditions are satisfied by well-typed inductive definitions (see Sect. 4.4). Recall that, in this interpretation, we do not consider size variables. The interpretation of I is thus directly the least fixed point of ϕ_I. In the definition of the relational interpretation we also consider approximations of this operator.

The definition of ϕ_I. Let us take $\delta \sqsubset [\Delta_p]$. We write $\mathcal{A}$ for $[\Delta_a]_\delta$. We define ϕ_I as an operator on $\Pi(\mathcal{A}, \mathcal{U}_k)$.

Let $X \in \Pi(\mathcal{A}, \mathcal{U}_k)$, and $\alpha \sqsubset \mathcal{A}$. $\phi_I(X)(\alpha)$ is a Λ-set where

- $\phi_I(X)(\alpha)_\circ = \{\emptyset\} \cup \bigcup_{i=1\ldots n} \{(i, \nu, \langle \rho_j \rangle_j) : \nu \sqsubset [\Delta_i^{\mathrm{nr}}]_\delta \wedge \rho_j \sqsubset \Pi([\Theta_{ij}]_{\delta,\nu}, \Phi_{J(i,j)}(X([\vec{u_{ij}}]_{\delta,\nu,\_}))) \wedge [\vec{t_i}]_{\delta,\nu} = \alpha\}$
- $M \models_{\phi_I(X)(\alpha)} \beta$, with $M \in \mathrm{SN}$ iff
  - $\beta = \emptyset$ and $M \to_{\mathrm{wh}}^* N \in \mathrm{AT}$
  - $\beta = (i, \nu, \langle \rho_j \rangle_j)$ and $M \to_{\mathrm{wh}}^* C_i(\vec{p}, \vec{N}, \langle R_j \rangle_j)$, where $\vec{N} \models_{[\Delta_i^{\mathrm{nr}}](\delta)} \nu$ and $R_j \models_{\Pi([\Theta_{ij}](\delta,\nu), \Phi_{J(i,j)}(X([\vec{u_{ij}}](\delta,\nu,\_))))} \rho_j$.
- $\bot_{\phi_I(X)(\alpha)} = \emptyset$.

We prove that the function ϕ_I is monotone, and that it has a least fixed point. We keep using the assumptions i-vi given above. Monotonicity follows easily from the strict positivity condition for $\mathcal{X}$.

Lemma 4.28 (Monotonicity of ϕ_I)**.** *The function ϕ_I is monotone on $\mathcal{A} \to \mathcal{U}_k$.*

Proof. Let $X, Y \in \mathcal{A} \to \mathcal{U}_k$ such that $X \subseteq Y$, and $(i, \nu, \langle \rho_j \rangle_j) \in \phi_I(X)(\alpha)$. By definition, $\rho_j \sqsubset \Pi([\Theta_{ij}]_{\delta,\nu}, \Phi_{J(i,j)}(X([\vec{u_{ij}}]_{\delta,\nu,\_})))$. Since the only parameter of J_{ij} is strictly positive, then $\Phi_{J(i,j)}(X([\vec{u_{ij}}]_{\delta,\nu,\alpha})) \subseteq \Phi_{J(i,j)}(X([\vec{u_{ij}}]_{\delta,\nu,\alpha}))$ for $\alpha \sqsubset [\Theta_{ij}]_{\delta,\nu}$. By definition of product, $\Pi([\Theta_{ij}]_{\delta,\nu}, \Phi_{J(i,j)}(X([\vec{u_{ij}}]_{\delta,\nu,\_}))) \subseteq \Pi([\Theta_{ij}]_{\delta,\nu}, \Phi_{J(i,j)}(Y([\vec{u_{ij}}]_{\delta,\nu,\_})))$. Then, $(i, \nu, \langle \rho_j \rangle_j) \in \phi_I(Y)(\alpha)$ as desired. □

To prove that ϕ_I has a (least) fixed point in $\mathcal{U}_k$, we make use of Lemma 4.22. We need to find a cardinal $\mathfrak{a}$ such that ϕ_I is $\mathfrak{a}$-based. We directly compute $\mathfrak{a}$ from the interpretation of the components of I.

Lemma 4.29. *There exists a regular cardinal $\mathfrak{a}$ such that ϕ_I is $\mathfrak{a}$-based.*

Proof. Let $x \in \Phi_I(X)(\alpha)$. We want to find a set Y such that $Y \subseteq X$, $x \in \phi_I(Y)(\alpha)$ and $\mathrm{card}(Y) < \mathfrak{a}$. We have two cases. The first case is $x = \emptyset$, then any set Y will do. For example, taking $Y = \emptyset = \alpha \in \mathcal{A} \mapsto \emptyset$, we have $\mathrm{card}(Y) = 1$.

The second case is $x = (i, \nu, \langle \rho_{ij} \rangle_j)$. For each j,

$$\rho_{ij} \sqsubset \Pi([\Theta_{ij}]_{\delta,\nu}, \Phi_{J(i,j)}(X([\vec{u_{ij}}]_{\delta,\nu,\_}))) \ .$$

Let us first consider $\Phi_{J(i,j)} = \mathsf{id}$ for all i, j. Then taking $Y(\alpha) = \mathrm{ran}(\rho_{ij})$ we have $\rho_{ij} \sqsubset \Pi([\Theta_{ij}]_{\delta,\nu}, \Phi_{J(i,j)}(Y([\vec{u_{ij}}]_{\delta,\nu,\_})))$. We generalize to all j and i; we bound the cardinal of $\bigcup_{i,j} \mathrm{ran}(\rho_{ij})$. Note that $\mathrm{card}(\mathrm{ran}(\rho_{ij})) \leq \mathrm{card}([\Theta_{ij}]_{\delta,\nu})$. Take $\mathfrak{a}_0 = \max_{ij}\{\mathrm{card}([\Theta_{ij}]_{\delta,\nu})\}$. Then, for $\mathfrak{a}$, we can take a regular cardinal greater $\mathfrak{a}_0$.

It remains the case where $\Phi_{J(i,j)}$ is not necessarily id for all i, j. We show that $\Phi_{J(i,j)}$ is $\mathfrak{b}$-based with respect to its (only) parameter, in the following sense: if $x \sqsubset \Phi_{J(i,j)}(X)$, then there exists $Y \subseteq X$, with $\mathrm{card}(Y) < \mathfrak{b}$ such that $x \sqsubset \Phi_{J(i,j)}(Y)$. We assume that the all constructors of J_{ij} have one non-recursive argument and one recursive arguments. I.e., $\mathsf{Ind}(J_{ij}[A^{\oplus} : \mathsf{Type}] : \mathsf{Type} := \langle C_k : (\Pi\Delta_{ijk}.A) \to (\Pi\Theta_{ijk}.\mathcal{X}) \to \mathcal{X} \rangle_k)$. The general case follows in a similar way.

We consider two cases. The first case is $x = \emptyset$, which follows easily.

The second case is $x = (k, \nu, \rho)$. We follow a similar reasoning as in the previous case. Take $Y = \bigcup_k \mathrm{ran}([\Delta_{ijk}]) \cup \mathrm{ran}([\Theta_{ijk}])$. Then $x \sqsubset \Phi_{J_{ij}}(Y)$. Note that $\mathrm{card}(Y) \leq \max_k\{\mathrm{card}([\Delta_{ijk}]), \mathrm{card}([\Theta_{ijk}])\} = \mathfrak{a}_0$. For $\mathfrak{a}$, we can take a cardinal number greater than $\mathfrak{a}_0$. □

Let us illustrate the previous lemma in some examples. Let us consider the case of ordinal numbers, defined by

$$\mathsf{Ind}(\mathsf{ord} : \mathsf{Type}_0 := \mathsf{zero} : \mathcal{X}, \mathsf{succ} : \mathsf{id}(\mathcal{X}) \to \mathcal{X}, \mathsf{lim} : (\mathsf{nat} \to \mathsf{id}(\mathcal{X})) \to \mathcal{X})$$

We have two recursive arguments. From the previous lemma, we have $\mathfrak{a}_0$ is the maximum between $\mathrm{card}(\{\emptyset\})$ and $\mathrm{card}([\mathsf{nat}])$; i.e. $\mathfrak{a}_0 = \aleph_0$. We can take $\mathfrak{a} = \aleph_1$ (or ω_1 as an ordinal).

In the case of lists, $\mathsf{Ind}(\mathsf{list}[A : \mathsf{Type}_0] : \mathsf{Type}_0 := \mathsf{nil} : \mathcal{X}, \mathsf{cons} : A \to \mathsf{id}(\mathcal{X}) \to \mathcal{X})$. we have only one recursive argument; applying the previous lemma we obtain $\mathfrak{a}_0 = 1$. Then the monotone operator derived from list, ϕ_{list}, is ω-based. The least fixed point is reached after ω iterations, independently of the type A.

On other inductive types, the cardinal required might depend on the parameters. For example, for the type $\mathsf{Ind}(I[A : \mathsf{Type}_k] : \mathsf{Type}_k := (A \to \mathsf{id}(\mathcal{X})) \to \mathcal{X})$, the cardinal required to reach a fixed point depends on the cardinal of $[A]$.

Let I be an inductive type defined in Type_k. We write $o(I)$ for the ordinal λ_k. Iterating ϕ_I up to $o(I)$ is guaranteed to reach the least fixed point.

Let X be the least fixed point of ϕ_I, i.e., $X = \phi_I^{o(I)}(\delta)(\alpha)$, where δ and α are the interpretation of the parameters and arguments, respectively. The elements of $X_\circ$ are either $\emptyset$, or elements of the form (i, ν, ρ) corresponding to the interpretation of the i-th constructor of I. Since ϕ_I is monotone, for each element of $\beta \in X_\circ$, there exists a smallest ordinal $\mathfrak{b}$ such that $\beta \in \phi_I^{\mathfrak{b}}(\delta)(\alpha)_\circ$. We call this ordinal the *order* of β, and denote it $o(\beta)$. For $\beta = \emptyset$, we have $o(\beta) = 0$. For β of the form (i, ν, ρ), we show that the order is always a successor ordinal.

Lemma 4.30. *Let $(i, \nu, \rho) \sqsubset \phi_I^{\mathfrak{a}}(\delta)(\alpha)$. Then $o(i, \nu, \rho)$ is a successor ordinal.*

Proof. We prove that $o(i, \nu, \rho)$ is not a limit ordinal. Let $(i, \nu, \rho) \sqsubset \phi_I^{\mathfrak{b}}(\delta)(\alpha)$ for a limit ordinal $\mathfrak{b}$. But $\phi_I^{\mathfrak{b}}(\delta)(\alpha)$ is defined as $\bigcup_{\mathfrak{b}' < \mathfrak{b}} \phi_I^{\mathfrak{b}'}(\delta)(\alpha)$. Therefore, there exists $\mathfrak{b}' < \mathfrak{b}$ such that $(i, \nu, \rho) \sqsubset \phi_I^{\mathfrak{b}'}(\delta)(\alpha)$. Then $\mathfrak{b}$ cannot be the order of (i, ν, ρ). Since the order is not zero, it must be a successor ordinal. □

4.3.4 Properties of the Interpretation

In this section we prove some simple properties of the interpretation just defined. Namely, soundness of weakening, substitution and subject reduction.

Lemma 4.31 (Soundness of weakening)**.** *1. If* $[\Gamma \vdash \Delta_0\Delta_1]_\gamma$ *is defined, and* $z \notin \mathrm{FV}(\Delta_1)$*, then* $[\Gamma \vdash \Delta_0(z : T^\circ)\Delta_1]_\gamma$ *is defined, and*

$$[\Gamma \vdash \Delta_0(z : T^\circ)\Delta_1]_\gamma \cong \Sigma([\Gamma \vdash \Delta_0(z : T^\circ)]_\gamma, [\Gamma\Delta_0 \vdash \Delta_1]_{\gamma,\_}) \ .$$

2. if $[\Gamma\Delta \vdash M]_{\gamma,\delta}$ *is defined, and* $z \notin \mathrm{FV}(\Delta, M)$*, then* $[\Gamma(z : T^\circ)\Delta \vdash M]_{\gamma,\alpha,\delta}$ *is defined, and*

$$[\Gamma(z : T^\circ)\Delta \vdash M]_{\gamma,\alpha,\delta} = [\Gamma\Delta \vdash M]_{\gamma,\delta} \ .$$

Proof. By induction on the constructions of $[\Gamma\Delta]$ and $[\Gamma\Delta \vdash M]_{\gamma,\delta}$.

Context extension. We know that $[\Gamma \vdash \Delta_0\Delta_1]_\gamma$ is defined. Then, either Δ_1 is empty and $[\Gamma \vdash \Delta_0\Delta_1]_\gamma = [\Gamma \vdash \Delta_0]_\gamma$, or $\Delta_1 = \Delta_1'(y : U^\circ)$ and $[\Gamma \vdash \Delta_0\Delta_1'(y : U^\circ)]_\gamma$ is defined as

$$[\Gamma \vdash \Delta_0\Delta_1'(y : U^\circ)]_\gamma = \Sigma([\Gamma \vdash \Delta_0\Delta_1]_\gamma, [\Gamma\Delta_0\Delta_1 \vdash U]_{\gamma,\_}) \ .$$

In the former case, the result follows trivially. In the latter, by IH we have

$$[\Gamma \vdash \Delta_0(z : T^\circ)\Delta_1']_\gamma \cong \Sigma([\Gamma \vdash \Delta_0(z : T^\circ)]_\gamma, [\Gamma\Delta_0 \vdash \Delta_1]_{\gamma,\_})$$
$$[\Gamma\Delta_0(z : T^\circ)\Delta_1' \vdash U^\infty]_{\gamma,\delta_0,\nu,\delta_1} = [\Gamma\Delta_0\Delta_1 \vdash U^\infty]_{\gamma,\delta_0,\delta_1}$$

By definition,

$$[\Gamma \vdash \Delta_0(z : T^\circ)\Delta_1'(y : U^\circ)]_\gamma = \Sigma([\Gamma \vdash \Delta_0(z : T^\circ)\Delta_1']_\gamma, [\Gamma\Delta_0(z : T^\circ)\Delta_1' \vdash U^\infty]_{\gamma,\_}) \ .$$

The result follows from the IH and Lemma 4.27.

Variable. $M = x$, then $[\Gamma\Delta]$ is defined and $(\gamma, \delta) \in [\Gamma\Delta]$. By IH, $[\Gamma(z : T)\Delta]$ is also defined, and, since z is different from x,

$$[\Gamma(z : T)\Delta \vdash x]_{\gamma,\alpha,\delta} = [\Gamma\Delta \vdash x]_{\gamma,\delta} \ .$$

Abstraction. $M \equiv \lambda^{W^\circ}_{x:U^\circ}.N$. The result follows from the IH. Note that by Weakening of the typing judgment (Lemma 3.13), if $\Gamma\Delta \vdash \Pi x : U^\infty.W^\infty : u$ then $\Gamma(z : T^\infty)\Delta \vdash \Pi x : U^\infty.W^\infty : u$. Hence, $\downarrow^{\Gamma\Delta\vdash\Pi x:U^\circ.U^\circ}_{\gamma,\delta} = \downarrow^{\Gamma(z:T^\circ)\Delta\vdash\Pi x:U^\circ.W^\circ}_{\gamma,\alpha,\delta}$.

The rest of the cases are similar. In the case of application, case, and fixpoints, we use Lemma 3.13 to ensure that functions $\uparrow$ and $\downarrow$ remain the same, in a similar way as in the abstraction case. □

Lemma 4.32 (Soundness of substitution)**.** *If* $\Gamma(x : T)\Delta \vdash M : U$ *and* $\Gamma \vdash N : T$ *are derivable,* $\mathrm{SV}(N) = \emptyset$*,* $\gamma \in [\Gamma]$*,* $\nu = [\Gamma \vdash N]_\gamma$ *is defined,* $(\gamma, \nu, \delta) \in [\Gamma(x : T)\Delta]$*, and* $[\Gamma(x : T)\Delta \vdash M]_{\gamma,\nu,\delta}$ *is defined, then* $[\Gamma, \Delta\,[x := N] \vdash M\,[x := N]]_{\gamma,\delta}$ *is defined, and*

$$[\Gamma, \Delta\,[x := N] \vdash M\,[x := N]]_{\gamma,\delta} = [\Gamma(x : T)\Delta \vdash M]_{\gamma,\nu,\delta} \ .$$

Proof. By induction on the structure of M.

Variable. $M \equiv x$. We want to prove that $[\Gamma, \Delta\,[x := N] \vdash N]_{\gamma,\delta}$ is defined, and that

$$[\Gamma, \Delta\,[x := N] \vdash N]_{\gamma,\delta} = [\Gamma(x : T)\Delta \vdash x]_{\gamma,\nu,\delta} \ .$$

The rhs is, by definition, equal to $\nu = [\Gamma \vdash N]_\gamma$. The result follows by applying Lemma 4.31 as many times as the length of Δ.

Fixpoint. $M \equiv \mathsf{fix}_n\ f : W^\star := (M_1, \vec{N_1})$. Since the interpretation of M is defined, there exists a unique function F in $[\Gamma(x : T)\Delta \vdash W^\infty]_{\gamma,\nu,\delta}$ satisfying the fixpoint equations. By IH,

$$\begin{gathered}
[\Gamma\Delta\,[x := N] \vdash W^\infty\,[x := N]]_{\gamma,\delta} = [\Gamma(x : T)\Delta \vdash W^\infty]_{\gamma,\nu,\delta} \\
[\Gamma\Delta\,[x := N]\,(f : |W\,[x := N]\,| \vdash M_1\,[x := N]]_{\gamma,\delta} = [\Gamma(x : T)\Delta(f : |W|) \vdash M_1]_{\gamma,\nu,\delta} \\
[\Gamma\Delta\,[x := N] \vdash \vec{N_1}\,[x := N]]_{\gamma,\delta} = [\Gamma(x : T)\Delta \vdash \vec{N_1}]_{\gamma,\nu,\delta}
\end{gathered}$$

Also, using Substitution of the typing judgment (Lemma 3.36), the function $\uparrow$ remains the same. Then, the same function satisfies the fixpoint equations for $\Gamma\Delta\,[x := N] \vdash \mathsf{fix}_n\ f : W^\star\,[x := N] := (M_1\,[x := N], \vec{N_1}\,[x := N])$, and the result follows.

The rest of the cases are similar. We use Lemma 3.36 for the cases of abstraction, application and case expressions, in a similar way as for fixpoints. □

The following lemma states that the interpretation is invariant under reduction. Recall the use of Hilbert's choice operator in the interpretation of fixpoints: the conditions imposed ensure that μ-reduction is sound.

Lemma 4.33 (Soundness of subject reduction)**.** *Let $\Gamma \vdash M : T$ and $M \to N$. If $[\Gamma \vdash M]_\gamma$ is defined, then $[\Gamma \vdash N]_\gamma$ is defined and*

$$[\Gamma \vdash M]_\gamma = [\Gamma \vdash N]_\gamma$$

Proof. By induction on $M \to N$. The key cases are when M is a redex.

β-redex $M \equiv \mathsf{app}^{U^\circ}_{x:T^\circ}\left(\lambda^{U^\circ}_{x:T^\circ}.M_1, M_2\right)$ and $N \equiv M_1\,[x := M_2]$. We have two cases: if $\Gamma \vdash \Pi x : T^\circ.U^\circ : \mathsf{Prop}$, or $\Gamma \vdash \Pi x : T^\circ.U^\circ : \mathsf{Type}_i$. Note that, in both cases, $\uparrow^{\Gamma\vdash\Pi x:T^\circ.U^\circ}_\gamma \circ \downarrow^{\Gamma\vdash\Pi x:T^\circ.U^\circ}_\gamma = \mathrm{id}_{\Pi([\Gamma\vdash T^\infty](\gamma),[\Gamma(x:T^\circ)\vdash U^\infty](\gamma,\_)}$.
Then, $[\Gamma \vdash M]_\gamma = [\Gamma(x : T^\circ) \vdash M_1]_{\gamma,\_}([\Gamma \vdash M_2]_\gamma)$. By Lemma 4.32, this is equal to $[\Gamma \vdash M_1\,[x := M_2]]_\gamma$ which is $[\Gamma \vdash N]_\gamma$.

ι-redex $M \equiv \mathsf{case}_{P^\circ}\ x_{I(\vec{p^\circ},\vec{a^\circ})} := C_j(\_, \vec{u})\ \mathsf{in}\ I(\_, \vec{y})\ \mathsf{of}\ \langle C_i \Rightarrow N_i\rangle$ and

$$N \equiv \mathsf{app}^{\mathsf{branch}_{C_j}(\vec{p^\circ},\vec{y}.x.P^\circ)}\,(N_j, \vec{u})$$

The result follows directly by definition of the interpretation.

μ-redex $M \equiv \mathsf{fix}_n\ f : T^\star := (M_1, (\vec{N_1}, C(\vec{p^\circ}, \vec{a})))$ and

$$N \equiv \mathsf{app}^{|U^\star|}_{|\Delta^\star|}\left(M_1\left[f := \lambda^{|U^\star|}_{|\Delta^\star|}.\mathsf{fix}_n\ f : T^\star := (M_1, \mathrm{dom}\,(|\Delta^\star|))\right], (\vec{N_1}, C(\vec{p^\circ}, \vec{a}))\right)$$

Let $M_2 \equiv \lambda^{|U^\star|}_{|\Delta^\star|}.\mathsf{fix}_n\ f : T^\star := (M_1, \mathrm{dom}\,(|\Delta^\star|))$.

Then $[\Gamma \vdash M]_\gamma = \uparrow(F)[\Gamma \vdash \vec{N_1}, C(\vec{p^\circ}, \vec{a^\circ})]_\gamma$, where F satisfies the fixpoint equations, and $[\Gamma \vdash N]_\gamma = \uparrow([\Gamma \vdash M_1\,[f := M_2]]_\gamma)[\Gamma \vdash \vec{N_1}, C(\vec{p^\circ}, \vec{a^\circ})]_\gamma$.
By Soundness of Substitution, $[\Gamma \vdash M_1\,[f := M_2]]_\gamma = [\Gamma(f : |T^\star|) \vdash M_1]_{\gamma,[\Gamma\vdash M_2](\gamma)}$. The result follows if we prove that $[\Gamma \vdash M_2]_\gamma = F$, using the fixpoint equations.
By definition, $[\Gamma \vdash M_2]_\gamma = \downarrow(\delta \sqsubset [\Gamma \vdash \Delta]_\gamma \mapsto \uparrow(F)(\delta))$. But this is equal to F, since, for $\delta \sqsubset [\Gamma \vdash \Delta]_\gamma$, $\uparrow([\Gamma \vdash M_2]_\gamma)(\delta) = \uparrow(\downarrow(\delta \sqsubset [\Gamma \vdash \Delta]_\gamma \mapsto \uparrow(F)(\delta)))(\delta) = \uparrow F(\delta)$.

The rest of the cases concern the compatible closure of $\to$. They follow easily by direct applications of the IH. □

4.3.5 Interpretation of simple Types

Recall that types can have size variables, respecting the simple predicate. The relational interpretation of types (RI) is used to ensure that terms respect size information given by their types. The RI of a type T is a Λ-set whose carrier set is composed of pairs. The intuition is given by the following definition.

Definition 4.34 (Relational type)**.** *Let X_1 and X_2 be Λ-sets. A* relational type for X_1 and X_2 *is a Λ-sets X such that $X_\circ \subseteq X_{1\circ} \times X_{2\circ}$, and $\bot_X = (\bot_{X_1}, \bot_{X_2})$. We denote it by $X \in \mathcal{R}(X_1, X_2)$.*

The RI of a type T is denoted $[\![\Gamma \vdash T]\!]^\pi_{\gamma_1 \sim \gamma_2}$ where Γ is a context, π is an interpretation of stage variables and γ_1 and γ_2 are two interpretation of free term variables. Stages are interpreted by ordinals, so π is just a function from stage variables to ordinals. The RI is also extended to contexts. We denote wtih $[\![\Gamma]\!]^\pi$ for the RI of a context Γ, where π is an interpretation of stage variables.

We define some notation for dealing with elements of the carrier-set in relational interpretations. We write $\alpha_1 \sim \alpha_2 \sqsubset [\![\Gamma \vdash T]\!]^\pi_{\gamma_1 \sim \gamma_2}$ and $\gamma_1 \sim \gamma_2 \sqsubset [\![\Gamma]\!]^\pi$ for $(\alpha_1, \alpha_2) \sqsubset [\![\Gamma \vdash T]\!]^\pi_{\gamma_1 \sim \gamma_2}$ and $(\gamma_1, \gamma_2) \sqsubset [\![\Gamma]\!]^\pi$, respectively. We sometimes write

To give some intuition on the use of the RI, we state informally the main soundness theorem: if $\Gamma \vdash T : u$ and $\gamma_1 \sim \gamma_2 \sqsubset [\![\Gamma]\!]^\pi$, then

$$[\![\Gamma \vdash T]\!]^\pi_{\gamma_1 \sim \gamma_2} \in \mathcal{R}([\Gamma \vdash T^\infty]_{\gamma_1}, [\Gamma \vdash T^\infty]_{\gamma_2}) \ .$$

In the following we define the RI for types are contexts. Recall that a simple type is either a product, an inductive type, or a term with no size variables. We exploit this structure in the definition of the RI. First, we define a product operation for relational types. Then, we define the relational interpretation of inductive types, as a monotone operator in the universe they live in; we proceed in the a similar way as in the previous section. Finally, we define the RI for simple types and prove some properties.

Relational product

We define a relational interpretation for products. Let $\Pi(X_i, \alpha_i \sqsubset X_i \mapsto Y_{i,\alpha_i})$ for $i = 1, 2$ be two product spaces of Λ-sets. Let X be a Λ-set such that $X \in \mathcal{R}(X_1, X_2)$. For each $(\alpha_1, \alpha_2) \sqsubset X$, let Y_{α_1,α_2} be a Λ-set such that $Y(\alpha_1, \alpha_2) \in \mathcal{R}(Y_{1,\alpha_1}, Y_{2,\alpha_2})$. We define a relational type $\Pi^+(X, Y)$ for $\Pi(X_1, Y_1)$ and $\Pi(X_2, Y_2)$:

$$\Pi^+(X, Y) \in \mathcal{R}(\Pi(X_1, Y_1), \Pi(X_2, Y_2)),$$

by the following rules.

– $\Pi^+(X,Y)_\circ = \{(f_1,f_2) \sqsubset \Pi(X_1,Y_1) \times \Pi(X_2,Y_2) \mid (\alpha_1,\alpha_2) \sqsubset X \Rightarrow (f_1\alpha_1, f_2\alpha_2) \sqsubset Y_{\alpha_1,\alpha_2}\}$;

– $M \models_{\Pi^+(X,Y)} (f_1,f_2) \iff \forall (\alpha_1,\alpha_2) \sqsubset X.\ N \models_X (\alpha_1,\alpha_2).\ \forall T^\circ, U^\circ, x.\ \mathsf{app}^{U^\circ}_{x:T^\circ}(M,N) \models_{Y(\alpha_1,\alpha_2)} (f_1\alpha_1, f_2\alpha_2)$;

– $\bot_{\Pi^+(X,Y)} = (\bot_{\Pi(X_1,Y_1)}, \bot_{\Pi(X_2,Y_2)})$

The carrier set of a relational product is formed by pairs of functions that take related elements in the domain to related elements in the codomain.

In the following we write $[\![\Gamma(x:T_1) \vdash T_2]\!]^\pi_{\gamma_1,\_\sim\gamma_2,\_}$ to mean

$$(\alpha_1,\alpha_2) \sqsubset [\![\Gamma \vdash T_1]\!]^\pi_{\gamma_1\sim\gamma_2} \mapsto [\![\Gamma(x:|T_1|) \vdash T_2]\!]^\pi_{\gamma_1,\alpha_1\sim\gamma_2,\alpha_2}$$

Inductive types

To define RI of inductive types, we proceed in a similar way as for the term interpretation. We define a monotone operator and prove that, under certain conditions, the operator has a least fixed point.

Let I be an inductive definition

$$\mathsf{Ind}(I[\Delta_p]^{\vec{\nu}} : \Pi\Delta_a.\mathsf{Type}_k := \langle C_i : T_i\rangle_i)\ .$$

As before, we assume that non-recursive arguments appear first: $T_i \equiv \Pi\Delta_i^{\mathrm{nr}}\Delta_i^{\mathrm{r}}.\mathcal{X}\,\vec{t_i}$, where $\Delta_i^{\mathrm{r}} \equiv \langle y_{ij} : \Pi\Theta_{ij}.J_{ij}\,(\mathcal{X}\,\vec{u_{ij}})\rangle_j$, where $\langle J_{ij}\rangle_{i,j}$ are inductive types defined in Σ. We follow the same path as in the interpretation of inductive types. We define a monotone operator $\Phi_{I(\vec{p})}$ in the universe where I is defined (Type_k in the above example).

In this case, the interpretation depends on two valuations of free variables (γ_1 and γ_2) and a stage valuation π. We write X^* to mean $X\,[\mathrm{dom}\,(\Delta_p) := \vec{p}]$, where X is a term or a context.

We assume the following:

i. $[\Gamma \vdash (\Delta_a^*)^\infty]_{\gamma_i}$ is defined, for $i = 1, 2$;

ii. $[\![\Gamma \vdash \Delta_a^*]\!]^\pi_{\gamma_1\sim\gamma_2}$ is a relational type for $([\Gamma \vdash (\Delta_a^*)^\infty]_{\gamma_1})$ and $([\Gamma \vdash (\Delta_a^*)^\infty]_{\gamma_2})$;

iii. $[\![\Gamma \vdash \Delta_i^*]\!]^\pi_{\gamma_1\sim\gamma_2}$ is defined

iv. $[\![\Gamma\Delta_i^* \vdash \Theta_i^*]\!]^\pi_{\gamma_1,\delta_1\sim\gamma_2,\delta_2}$ is defined for $(\delta_1,\delta_2) \sqsubset [\![\Gamma \vdash \Delta_i^*]\!]^\pi_{\gamma_1\sim\gamma_2}$

v. $[\Gamma\Delta_i^*\Theta_i^* \vdash \vec{u_i}]_{\gamma_1,\delta_1,\tau_1\sim\gamma_2,\delta_2,\tau_2}$ is defined and belongs to $[\![\Gamma \vdash \Delta_a^*]\!]^\pi_{\gamma_1\sim\gamma_2}$ for $((\delta_1,\tau_1),(\delta_2,\tau_2)) \sqsubset [\![\Gamma \vdash \Delta_i^*\Theta_i^*]\!]^\pi_{\gamma_1\sim\gamma_2}$

For readability, we omit Γ, γ_1, and γ_2, in the following.

The definition of $\mathbf{\Phi}_{I(\vec{p})}$. The operator $\mathbf{\Phi}_{I(\vec{p})}$ is defined on the carrier-set of

$$\Pi^+([\![(\Delta_a^*)^\infty]\!]^\pi, \mathcal{U}_k)$$

Let X be a function on $\Pi^+([\![(\Delta_a^*)^\infty]\!]^\pi, \mathcal{U}_k)$ Let $(\delta_1,\delta_2) \sqsubset ([\![(\Delta_a^*)^\infty]\!]^\pi)$. We define $\mathbf{\Phi}_{I(\vec{p})}(X)(\delta_1,\delta_2)$ as a Λ-set where

–

$$X = \{(\emptyset,\emptyset)\} \cup \{((j,\nu_1,\langle\rho_{1j}\rangle),(j,\nu_2,\langle\rho_{2j}\rangle)) : (\nu_1,\nu_2) \sqsubset [\![\Delta_i^*]\!]^\pi \wedge (\rho_{1j},\rho_{2j}) \sqsubset \Pi([\![\Theta_{ij}^*]\!]^\pi_{\nu_1\sim\nu_2}, X([\vec{u_{ij}}^*]_{\gamma_1,\rho_1}, [\vec{u_{ij}}^*]_{\gamma_2,\rho_2})) \wedge \forall i = 1,2.\ [\vec{t_j}]_{\gamma_i,\rho_i} = \delta_i\}$$

- $M \models (\beta_1, \beta_2)$, with $M \in \mathrm{SN}$, iff
  - $(\beta_1, \beta_2) = (\emptyset, \emptyset)$ and $M \to_{\mathrm{wh}}^* N \in \mathrm{AT}$
  - $(\beta_1, \beta_2) = ((j, \nu_1, \rho_1), (j, \nu_2, \rho_2))$ and $M \to_{\mathrm{wh}}^* C_j(\vec{p^\circ}, \vec{N}, \langle R_j \rangle_j)$, $\vec{N} \models_{[\![\Delta_j]\!]^\pi} (\nu_1, \nu_2)$, and $R_j \models_{\Pi([\![\Theta_{ij}^*]\!](\nu_1 \sim \nu_2, \pi), X([u_{ij}^{\vec{}}{}^*](\gamma_1, \rho_1), [u_{ij}^{\vec{}}{}^*](\gamma_2, \rho_2)))} (\rho_1, \rho_2)$
- $\bot = (\emptyset, \emptyset)$

In a similar way to the previous section, we prove that the operator is monotone and it has a least fixed point.

Lemma 4.35 (Monotonicity of $\mathbf{\Phi}_{I(\vec{p})}$)**.** *The function $\mathbf{\Phi}_{I(\vec{p})}$ is monotone on the Λ-set $\Pi([\![\Gamma \vdash (\Delta_a^*)^\infty]\!]^\pi_{\gamma_1 \sim \gamma_2}, \mathcal{U}_k)$.*

Proof. Similar to Lemma 4.28. □

To prove that $\mathbf{\Phi}_{I(\vec{p})}$ has a (least) fixed point in $\mathcal{U}_k$, we make use of Lemma 4.22. We need to find a cardinal $\mathfrak{a}$ such that $\mathbf{\Phi}_{I(\vec{p})}$ is $\mathfrak{a}$-based. We directly compute $\mathfrak{a}$ from the interpretation of the components of I.

Lemma 4.36. *There exists $\mathfrak{a}$ such that $\mathbf{\Phi}_{I(\vec{p})}$ is $\mathfrak{a}$-based.*

Proof. We can construct $\mathfrak{a}$ following the same reasoning as in Lemma 4.29. □

In the rest of chapter we assume a fixed signature Σ. Recall that $o(I)$ is an ordinal that ensure that the least fixed point is reached by iterating the interpretation of I. Let $\mathfrak{M} = \max\{o(I) : I \in \Sigma\}$. Then $\mathfrak{M}$ ensures that the least fixed point is reached for any $I \in \Sigma$.

Stages. Recall that stages represent approximations of the monotone operators defined for inductive types. Stages are interpreted by ordinals up to the smallest ordinal we need to reach the least fixed point for the inductive types in Σ. In our case, this ordinal is $\mathfrak{M}$.

Definition 4.37 (Stage interpretation)**.** *A stage assignment π is a function from $\mathcal{V}_\mathcal{S}$ to $\mathfrak{a}$. Given a stage assignment π, the interpretation of a stage s under π, written $(\!|s|\!)_\pi$, is defined by:*

$$
\begin{aligned}
(\!|\imath|\!)_\pi &= \pi(\imath) \\
(\!|\infty|\!)_\pi &= \mathfrak{M} \\
(\!|\widehat{s}|\!)_\pi &= \begin{cases} (\!|s|\!)_\pi + 1 & \text{if } (\!|s|\!)_\pi < \mathfrak{M} \\ \mathfrak{M} & \text{if } (\!|s|\!)_\pi = \mathfrak{M} \end{cases}
\end{aligned}
$$

We use ∞ to denote the stage assignment such that $\infty(\imath) = \mathfrak{M}$ for all $\imath$.

We prove some lemmas about the interpretation of stages.

Lemma 4.38 (Stage substitution)**.** *Let s, r be stages, and π a stage assignment. Then $(\!|r\,[\imath := s]|\!)_\pi = (\!|r|\!)_{\pi(\imath := (\!|s|\!)_\pi)}$.*

Proof. By induction on the structure of r. □

Lemma 4.39 (Stage monotonicity)**.** *For any stages s, r, such that $s \sqsubseteq r$, and any stage assignment π, $(\!|s|\!)_\pi \leq (\!|r|\!)_\pi$.*

Proof. By induction on $s \sqsubseteq r$. Note that, for any s and π, $(\!|s|\!)_\pi \leq \mathfrak{M}$. □

Relational interpretation

Now we are ready to define the relational interpretation. We proceed by induction on the definition of simple types. Let T be a type such that $\mathsf{simple}(T)$.

- If $T \equiv \Pi x : T_1.T_2$, we consider two cases
  - $\Gamma^\infty \vdash T^\infty : \mathsf{Type}_i$. Then we define

$$[\![\Gamma \vdash T]\!]^\pi_{\gamma_1 \sim \gamma_2} = \Pi^+([\![\Gamma \vdash T_1]\!]^\pi_{\gamma_1 \sim \gamma_2}, [\![\Gamma(x : T_1) \vdash T_2]\!]^\pi_{\gamma_1,\_ \sim \gamma_2,\_})$$

  - $\Gamma^\infty \vdash T^\infty : \mathsf{Prop}$. In this case, $[\Gamma \vdash T^\infty]_{\gamma_1}$ and $[\Gamma \vdash T^\infty]_{\gamma_2}$ are degenerated Λ-sets. We define
    - $([\![\Gamma \vdash T]\!]^\pi_{\gamma_1 \sim \gamma_2})_\circ = ([\Gamma \vdash T^\infty]_{\gamma_1})_\circ \times ([\Gamma \vdash T^\infty]_{\gamma_2})_\circ$
    -

$$M \models_{[\![\Gamma \vdash T]\!](\gamma_1 \sim \gamma_2, \pi)} (\phi_1, \phi_2) \iff N \models_{[\![\Gamma \vdash T_1]\!](\gamma_1 \sim \gamma_2, \pi)} (\alpha_1, \alpha_2) \Rightarrow \mathsf{app}^{U^\circ}_{x:T^\circ}(M, N) \models_{[\![\Gamma(x:T_1) \vdash T_2]\!](\gamma_1, \alpha_1 \sim \gamma_2, \alpha_2, \pi)} (\uparrow_1(\phi_1)\alpha_1, \uparrow_2(\phi_2)\alpha_2)$$

      where $\uparrow_i = \uparrow^{\Gamma \vdash T}_{\Pi([\Gamma \vdash T_1](\gamma_i), [\Gamma(x:T_1) \vdash T_2](\gamma_i,\_))}$, for $i = 1, 2$;
    - $(\bot_{[\Gamma \vdash T^\infty](\gamma_1)}, \bot_{[\Gamma \vdash T^\infty](\gamma_2)})$.
- If $T \equiv I^s\ (\vec{p})\ \vec{a}$, then

$$[\![\Gamma \vdash T]\!]^\pi_{\gamma_1 \sim \gamma_2} = \mathbf{\Phi}^{\mathfrak{a}}_{I(\vec{p})}(\bot)([\Gamma \vdash \vec{a}]_{\gamma_1 \sim \gamma_2})$$

  where $\mathfrak{a} = (\!|s|\!)_\pi$.
- Otherwise, $\mathrm{SV}(T) = \emptyset$. We define

$$[\![\Gamma \vdash T]\!]^\pi_{\gamma_1 \sim \gamma_2} = ([\Gamma \vdash T]_{\gamma_1})^2$$

Note that the RI of a propositional type is not a degenerated Λ-set. A Λ-set is *relationally degenerated* if its carrier-set is a singleton whose element is a pair of saturated sets.

Relational interpretation of contexts. As in the interpretation of terms, we extend the relational interpretation to contexts, by the following rules:

$$[\![\Gamma \vdash []]\!]^\pi_{\gamma_1 \sim \gamma_2} = \emptyset^2 = ((\emptyset, \emptyset), AT \cap SN \times \{(\emptyset, \emptyset)\}, (\emptyset, \emptyset))$$
$$[\![\Gamma \vdash (x : T)\Delta]\!]^\pi_{\gamma_1 \sim \gamma_2} = \Sigma([\![\Gamma \vdash T]\!]^\pi_{\gamma_1 \sim \gamma_2}, [\![\Gamma(x : T) \vdash \Delta]\!]^\pi_{\gamma_1,\_ \sim \gamma_2,\_})$$

In the definition of the RI, more than one clause might be applicable for one given type (e.g, a product with no size variables). In the following lemma, we prove that the definition is the same no matter which clause is used.

Lemma 4.40. *Let T be a term such that $\mathsf{simple}(T)$ and $\mathrm{SV}(T) = \emptyset$. If $[\![\Gamma \vdash T]\!]^\pi_{\gamma_1 \sim \gamma_2}$ is defined and $[\Gamma \vdash T]_{\gamma_1} = [\Gamma \vdash T]_{\gamma_2}$, then $[\![\Gamma \vdash T]\!]^\pi_{\gamma_1 \sim \gamma_2} = ([\Gamma \vdash T]_{\gamma_1})^2$.*

Proof. If $\Gamma^\infty \vdash T^\infty : \mathsf{Prop}$, the result follows by definition. Otherwise, we proceed by induction on the structure of T.

I. $T \equiv \Pi x : T_1.T_2$.
Since $[\![\Gamma \vdash T]\!]^\pi_{\gamma_1 \sim \gamma_2}$ is defined, we have that $[\![\Gamma \vdash T_1]\!]^\pi_{\gamma_1 \sim \gamma_2}$ is defined, and for every $(\alpha_1, \alpha_2) \in [\![\Gamma \vdash T_1]\!]^\pi_{\gamma_1 \sim \gamma_2}$, $[\![\Gamma(x : |T_1|) \vdash T_2]\!]^\pi_{(\gamma_1,\alpha_1)\sim(\gamma_2,\alpha_2)}$ is defined.
We write $[\![T]\!]$ for $[\![\Gamma \vdash T]\!]^\pi_{\gamma_1 \sim \gamma_2}$, $[\![T_1]\!]$ for $[\![\Gamma \vdash T_1]\!]^\pi_{\gamma_1 \sim \gamma_2}$ and $[\![T_2]\!]_{\alpha_1,\alpha_2}$ for $[\![\Gamma(x : |T_1|) \vdash T_2]\!]^\pi_{(\gamma_1,\alpha_1)\sim(\gamma_2,\alpha_2)}$.
By IH, $[\![T_1]\!] = ([\Gamma \vdash T_1]_{\gamma_1})^2$. Hence, $(\alpha_1, \alpha_2) \sqsubset [\![T_1]\!]$ iff $\alpha_1 = \alpha_2 \sqsubset [\Gamma \vdash T_1]_{\gamma_1}$.
Again by IH, $[\![T_2]\!]_{\alpha,\alpha} = ([\Gamma(x : |T_1|) \vdash T_2]_{\gamma_1,\alpha})^2$ for every $\alpha \sqsubset [\Gamma \vdash T_1]_{\gamma_1}$.
Let $(\phi_1, \phi_2) \sqsubset ([\Gamma \vdash T]_{\gamma_1})^2 \times [\Gamma \vdash T^\infty]_{\gamma_2}$ iff for all $\alpha \sqsubset [\![T_1]\!], (\phi_1(\alpha), \phi_2(\alpha)) \sqsubset [\![T_2]\!]_{\alpha,\alpha}$.
Hence, $\phi_1 = \phi_2$. Checking the other conditions is easy.

□

To give a better understanding of the relational interpretation, we consider the particular case where the type to be interpreted is an inductive type applied to parameters and arguments containing no size variables. It shows that the RI reduces to the identity (relation) on elements of order less than the size annotation of the inductive type.

Lemma 4.41. *Let $\gamma \in [\Gamma]$, $\vec{p}$ and $\vec{a}$ sequences of terms with no size variables. If $[\![\Gamma \vdash I^s(\vec{p}, \vec{a})]\!]^\pi_{\gamma\sim\gamma}$ is defined for all stage valuations π, then*

$$[\![\Gamma \vdash I^s(\vec{p}, \vec{a})]\!]^\pi_{\gamma\sim\gamma} = \left(\Phi_I^{(\!|s|\!)_\pi}([\Gamma \vdash \vec{p}]_\gamma)([\Gamma \vdash \vec{a}]_\gamma)\right)^2$$

where Φ_I is the term interpretation of I.

Proof. By induction on $(\!|s|\!)_\pi$. □

4.3.6 Properties of the Relational Interpretation

In the following we prove the following properties of the relational interpretation: soundness of weakening, substitution, subject reduction, subtyping, and stage monotonicity.

Lemma 4.42 (Soundness of weakening)**.** *If $[\![\Gamma\Delta \vdash U]\!]^\pi_{(\gamma_1,\delta_1)\sim(\gamma_2,\delta_2)}$ is defined, then $[\![\Gamma(z : T)\Delta \vdash U]\!]^\pi_{(\gamma_1,\alpha_1,\delta_1)\sim(\gamma_2,\alpha_2,\delta_2)}$ is defined and both are equal.*

Proof. We consider two cases

If $\Gamma^\infty\Delta^\infty \vdash U^\infty : \mathsf{Prop}$ The result follows by Lemma 4.32.

If $\Gamma^\infty\Delta^\infty \vdash U^\infty : \mathsf{Type}_k$ We proceed by induction on the structure of U. We consider two cases.

$U \equiv \Pi x : U_1.U_2$**.** By IH, $[\![\Gamma(z : T)\Delta \vdash U_1]\!]^\pi_{(\gamma_1,\alpha_1,\delta_1)\sim(\gamma_2,\alpha_2,\delta_2)}$ is defined and

$$[\![\Gamma(x : T)\Delta \vdash U_1]\!]^\pi_{(\gamma_1,\alpha_1,\delta_1)\sim(\gamma_2,\alpha_2,\delta_2)} = [\![\Gamma\Delta \vdash U_1]\!]^\pi_{(\gamma_1,\delta_1)\sim(\gamma_2,\delta_2)} \ .$$

Also, for every $(\nu_1, \nu_2) \sqsubset [\![\Gamma\Delta \vdash U_1]\!]^\pi_{(\gamma_1,\delta_1)\sim(\gamma_2,\delta_2)}$,

$$[\![\Gamma(z : T)\Delta(x : U_1) \vdash U_2]\!]^\pi_{(\gamma_1,\alpha_1,\delta_1,\nu_1)\sim(\gamma_2,\alpha_2,\delta_2,\nu_2)}$$

is defined and

$$[\![\Gamma(z : T)\Delta(x : U_1) \vdash U_2]\!]^\pi_{(\gamma_1,\alpha_1,\delta_1,\nu_1)\sim(\gamma_2,\alpha_2,\delta_2,\nu_2)} = [\![\Gamma\Delta(x : U_1) \vdash U_2]\!]^\pi_{(\gamma_1,\delta_1,\nu_1)\sim(\gamma_2,\delta_2,\nu_2)} \ .$$

The result follows easily from the definition of the relational interpretation.

$U \equiv I^s\,(\vec{p}, \vec{a})$. Since $[\![\Gamma\Delta \vdash U]\!]^\pi_{(\gamma_1,\delta_1)\sim(\gamma_2,\delta_2)}$ is defined, we know that the interpretation of the constructors, parameter and arguments of I are defined. We can apply the IH; the result then follows since the definition of $[\![\Gamma(z:T)\Delta \vdash U]\!]^\pi_{(\gamma_1,\alpha_1,\delta_1)\sim(\gamma_2,\alpha_2,\delta_2)}$ is the same as $[\![\Gamma\Delta \vdash U]\!]^\pi_{(\gamma_1,\delta_1)\sim(\gamma_2,\delta_2)}$.

In the rest of the cases, since U is simple, we have $\mathrm{SV}(U) = \emptyset$. The result follows by Lemma 4.32.

□

Lemma 4.43 (Soundness of stage substitution)**.** *If* $[\![\Gamma \vdash T\,[\imath := s]]\!]^\pi_{\gamma_1\sim\gamma_2}$ *is defined, then* $[\![\Gamma \vdash T]\!]^{\pi(\imath:=(\!|s|\!)_\pi)}_{\gamma_1\sim\gamma_2}$ *is defined, and both are equal.*

Proof. If $\Gamma^\infty\Delta^\infty \vdash T^\infty : \mathsf{Prop}$, we have to consider the case when T is a product, or it has no size variables. In the former, the result follows from IH, while in the latter, the result follows trivially.

If $\Gamma^\infty\Delta^\infty \vdash T^\infty : \mathsf{Type}_i$, we have three subcases. If T is a product, the result follows from the IH. If T is an inductive type, the result follows from the IH, using Lemma 4.38. The last subcase is when T has no size variables; the result follows trivially. □

Lemma 4.44 (Soundness of substitution)**.** *Let* $\Gamma(x:U)\Delta \vdash T : W$ *and* $\Gamma \vdash N : U$ *and* $\mathrm{SV}(N) = \emptyset$. *Let* $\gamma_i \in [\Gamma]$, $\nu_i \equiv [\Gamma \vdash N]_{\gamma_i}$, $(\gamma_i, \nu_i, \delta_i) \in [\Gamma(x:U)\Delta]$, *for* $i = 1, 2$, *and* $[\![\Gamma(x:U)\Delta \vdash T]\!]^\pi_{\gamma_1,\nu_1,\delta_1\sim\gamma_2,\nu_2,\delta_2}$ *are defined. Then*

$$[\![\Gamma\Delta\,[x := N] \vdash T\,[x := N]]\!]^\pi_{(\gamma_1,\delta_1)\sim(\gamma_2,\delta_2)} = [\![\Gamma(x:U)\Delta \vdash T]\!]^\pi_{(\gamma_1,\nu_1,\delta_1)\sim(\gamma_2,\nu_2,\delta_2)}$$

Proof. We follow the same scheme as in the two previous proofs. All cases follow easily, using Lemma 4.32. □

Lemma 4.45 (Soundness of subject reduction)**.** *Let* $\Gamma \vdash T : u$ *and* $T \to U$, *with* $\mathsf{simple}(T)$. *If* $[\![\Gamma \vdash T]\!]^\pi_{\gamma_1\sim\gamma_2}$ *is defined, then* $[\![\Gamma \vdash U]\!]^\pi_{\gamma_1\sim\gamma_2}$ *is defined and*

$$[\![\Gamma \vdash T]\!]^\pi_{\gamma_1\sim\gamma_2} = [\![\Gamma \vdash U]\!]^\pi_{\gamma_1\sim\gamma_2}$$

Proof. By induction on the structure of T, using Lemma 4.33. □

The following lemma states that the RI is sound with respect to subtyping.

Lemma 4.46 (Soundness of subtyping)**.** *Assume* $\Gamma \vdash T, U : u$, *and* $\mathsf{simple}(T, U)$ *and* $T \le U$. *If* $[\![\Gamma \vdash T]\!]^\pi_{\gamma_1\sim\gamma_2}$ *and* $[\![\Gamma \vdash U]\!]^\pi_{\gamma_1\sim\gamma_2}$ *are defined, then* $[\![\Gamma \vdash T]\!]^\pi_{\gamma_1\sim\gamma_2} \subseteq [\![\Gamma \vdash U]\!]^\pi_{\gamma_1\sim\gamma_2}$

Proof. We proceed by induction on the derivation of $T \le_{\mathsf{t}} U$.

(stt-conv) Follows directly from Lemma 4.45.

(stt-prod) We have the derivation

$$\frac{T \to_{\mathsf{t}}{}^* \Pi x : T_1.T_2 \qquad U_1 \le_{\mathsf{t}} T_1 \qquad T_2 \le_{\mathsf{t}} U_2 \qquad U \to_{\mathsf{t}}{}^* \Pi x : U_1.U_2}{T \le_{\mathsf{t}} U}$$

By Lemma 4.45, $[\![\Gamma \vdash T]\!]^\pi_{\gamma_1\sim\gamma_2} = [\![\Gamma \vdash \Pi x : T_1.T_2]\!]^\pi_{\gamma_1\sim\gamma_2}$ and $[\![\Gamma \vdash U]\!]^\pi_{\gamma_1\sim\gamma_2} = [\![\Gamma \vdash \Pi x : U_1.U_2]\!]^\pi_{\gamma_1\sim\gamma_2}$. In particular, since $T^\infty \downarrow_{\mathsf{t}} U^\infty$ (by Lemma 3.32), we have $[\Gamma \vdash T]_{\gamma_1\sim\gamma_2} = [\Gamma \vdash U]_{\gamma_1\sim\gamma_2}$.

The IH gives us $[\![\Gamma \vdash U_1]\!]^\pi_{\gamma_1\sim\gamma_2} \subseteq [\![\Gamma \vdash T_1]\!]^\pi_{\gamma_1\sim\gamma_2}$ and $[\![\Gamma(x:T_1) \vdash T_2]\!]^\pi_{\gamma_1,\alpha_1\sim\gamma_2,\alpha_2} \subseteq [\![\Gamma(x:U_1) \vdash U_2]\!]^\pi_{\gamma_1,\alpha_1\sim\gamma_2,\alpha_2}$, for any $(\alpha_1, \alpha_2) \sqsubset [\![\Gamma \vdash U_1]\!]^\pi_{\gamma_1\sim\gamma_2}$.

We consider two subcases.

1. $\Gamma \vdash T^\infty : \mathsf{Type}_k$. We have $[\![\Gamma \vdash T]\!]^\pi_{\gamma_1 \sim \gamma_2} = \Pi^+([\![T_1]\!]^\pi_{\gamma_1 \sim \gamma_2}, [\![T_2]\!]^\pi_{\gamma_1, \_ \sim \gamma_2, \_})$, and $[\![\Gamma \vdash U]\!]^\pi_{\gamma_1 \sim \gamma_2} = \Pi^+([\![U_1]\!]^\pi_{\gamma_1 \sim \gamma_2}, [\![\Gamma(x : U_1)]\!]^\pi_{U_2 \sim \gamma_1, \_} \gamma_2, \_)$.
   Note that the carrier set of $[\![T]\!]^\pi_{\gamma_1 \sim \gamma_2}$ is formed by pairs of functions, (ϕ_1, ϕ_2), where ϕ_1 has domain $[T_1^\infty]_{\gamma_1}$ and ϕ_2 has domain $[T_1^\infty]_{\gamma_2}$. Since $[T_1^\infty]_{\gamma_1} = [U_1^\infty]_{\gamma_1}$, and similarly for γ_2, the functions in the carrier set of $[\![U]\!]^\pi_{\gamma_1 \sim \gamma_2}$ have the same domain. This is the reason why contravariance is sound.
   In particular, let $(\phi_1, \phi_2) \sqsubset [\![T]\!]^\pi_{\gamma_1 \sim \gamma_2}$. Let $(\alpha_1, \alpha_2) \sqsubset [\![U_1]\!]^\pi_{\gamma_1 \sim \gamma_2}$. By IH, $(\alpha_1, \alpha_2) \sqsubset [\![T_1]\!]^\pi_{\gamma_1 \sim \gamma_2}$. By definition $(\phi_1\alpha_1, \phi_2\alpha_2) \sqsubset [\![T_2]\!]^\pi_{\gamma_1, \alpha_1 \sim \gamma_2, \alpha_2}$. Applying again the IH, $(\phi_1\alpha_1, \phi_2\alpha_2) \sqsubset [\![U_2]\!]^\pi_{\gamma_1, \alpha_1 \sim \gamma_2, \alpha_2}$. Then $(\phi_1, \phi_2) \sqsubset [\![U]\!]^\pi_{\gamma_1 \sim \gamma_2}$.
   Following a similar reasoning, we can prove that $M \models_{[\![T]\!](\gamma_1 \sim \gamma_2, \pi)} (\phi_1, \phi_2)$ implies $M \models_{[\![U]\!](\gamma_1 \sim \gamma_2, \pi)} (\phi_1, \phi_2)$.
2. $\Gamma \vdash T^\infty : \mathsf{Prop}$. The carrier set and atomic element are the same. It remains to prove that $M \models_{[\![\Gamma \vdash T]\!](\gamma_1 \sim \gamma_2, \pi)} (\phi_1, \phi_2)$ implies $M \models_{[\![\Gamma \vdash U]\!](\gamma_1 \sim \gamma_2, \pi)} (\phi_1, \phi_2)$; it follows by the same reasoning than in the previous case.

(stt-ind) We have the derivation

$$\frac{T \to_{\mathsf{t}} I^s\,(\vec{p_1}, \vec{a_1}) \qquad s \sqsubseteq r \qquad \vec{p_1} \leq^{I.\vec{\nu}}_{\mathsf{t}} \vec{p_2} \qquad \vec{a_1} \downarrow_{\mathsf{t}} \vec{a_2} \qquad U \to_{\mathsf{t}} I^r\,(\vec{p_2}, \vec{a_2})}{T \leq_{\mathsf{t}} U}$$

Let $I = \mathsf{Ind}(I[\Delta_p]^{\vec{\nu}} : \Pi\Delta_a.\mathsf{Type}_k := \langle C_i : \Pi\Delta_i^{\mathrm{nr}}\Delta_i^{\mathrm{r}} \to \mathcal{X}\,\vec{t_i}\rangle_i)$. From $\vec{a_1} \downarrow_{\mathsf{t}} \vec{a_2}$, we have $[\Gamma \vdash \vec{a_1}]_{\gamma_1 \sim \gamma_2} = [\Gamma \vdash \vec{a_2}]_{\gamma_1 \sim \gamma_2}$.
From the conditions imposed to valid inductive types, we have $\mathrm{dom}\,(\Delta_p)\ \mathsf{pos}^{\vec{\nu}}\ \Delta_i^{\mathrm{nr}}$, and $\mathrm{dom}\,(\Delta_p)\ \mathsf{neg}^{\vec{\nu}}\ \Delta_i^{\mathrm{r}}$. Using this property, we prove that $\mathbf{\Phi}^{\mathfrak{a}}_{I(\vec{p_1})} \subseteq \mathbf{\Phi}^{\mathfrak{a}}_{I(\vec{p_2})}$. The proof proceeds by induction on $\mathfrak{a}$. The cases when $\mathfrak{a} = 0$ or $\mathfrak{a}$ is a limit cardinal follow easily. In the case $\mathfrak{a} = \mathfrak{b} + 1$, we need to reinforce the main IH to be able to apply it to the constructors. Consider a constructor C_i whose arguments are $\Delta_i^{\mathrm{nr}}\Delta_i^{\mathrm{r}}$, where $\Delta_i^{\mathrm{r}} \equiv \langle \Pi\Theta_{ij}.J_{ij}(\mathcal{X}\,u_{ij}^{\vec{}})\rangle_j$. We reinforce the IH to be able to deduce, from $\mathrm{dom}\,(\Delta_p)\ \mathsf{pos}^{\vec{\nu}}\ \Delta_i^{\mathrm{nr}}$, that $[\![\Gamma \vdash \Delta_i^{\mathrm{r}}\,[\mathrm{dom}\,(\Delta_p) := \vec{p_1}]]\!]^\pi_{\gamma_1 \sim \gamma_2} \subseteq [\![\Gamma \vdash \Delta_i^{\mathrm{r}}\,[\mathrm{dom}\,(\Delta_p) := \vec{p_2}]]\!]^\pi_{\gamma_1 \sim \gamma_2}$. Similarly for the recursive arguments, where we use the fact that the interpretation of J_{ij} is monotone on its parameter (this follows by IH). From the definition of the RI, the inner result follows.
Then we have $[\![\Gamma \vdash I^s\,(\vec{p_1}, \vec{t_1})]\!]^\pi_{\gamma_1 \sim \gamma_2} \subseteq [\![\Gamma \vdash I^s\,(\vec{p_2}, \vec{t_2})]\!]^\pi_{\gamma_1 \sim \gamma_2}$. From $s \sqsubseteq r$ and Lemma 4.39, we have $[\![\Gamma \vdash I^s\,(\vec{p_2}, \vec{t_2})]\!]^\pi_{\gamma_1 \sim \gamma_2} \subseteq [\![\Gamma \vdash I^r\,(\vec{p_2}, \vec{t_2})]\!]^\pi_{\gamma_1 \sim \gamma_2}$, and the result follows.

□

The following lemma is used in the proof of soundness of the fixpoint rule.

Lemma 4.47 (Soundness of stage monotonicity)**.** *Let s, r be stages such that $s \sqsubseteq r$. If $[\![\Gamma \vdash T]\!]^\pi_{\gamma_1 \sim \gamma_2}$ is defined, and $\imath\ \mathsf{pos}\ T$, then $[\![\Gamma \vdash T]\!]^{\pi(\imath := s)}_{\gamma_1 \sim \gamma_2} \subseteq [\![\Gamma \vdash T]\!]^{\pi(\imath := r)}_{\gamma_1 \sim \gamma_2}$*

Proof. From $\imath\ \mathsf{pos}\ T$ we know that $T\,[\imath := s] \leq T\,[\imath := r]$. The result follows from the previous lemma. □

4.4 Soundness

In this section we prove our main theorem: soundness of the Λ-set model. The proof proceeds by induction on the type derivation. The most interesting case is the fixpoint rule.

In particular, we need to prove that there exists only one function satisfying the fixpoint equations of Def. 4.24, therefore showing that the definition given using Hilbert's choice operator is sound.

Let us recall the intuitive statement of the soundness theorem: if $\Gamma \vdash M : T$ and $\gamma_1 \sim \gamma_2 \sqsubset [\![\Gamma]\!]^\pi$, then $[\Gamma \vdash M]_{\gamma_1} \sim [\Gamma \vdash M]_{\gamma_2} \sqsubset [\![\Gamma \vdash T]\!]^\pi_{\gamma_1 \sim \gamma_2}$. In the following, we write $[\Gamma \vdash M]_{\gamma_1 \sim \gamma_2}$ for $([\Gamma \vdash M]_{\gamma_1}, [\Gamma \vdash M]_{\gamma_2})$.

Before stating and proving the soundness theorem, we analyze the fixpoint rule. We show that under certain conditions, later to be ensured in the soundness theorem, the existence of a unique function satisfying the fixpoint equations is assured.

Fixpoint. For the following lemmas, let us assume a typing derivation of the form

$$\frac{\begin{array}{c} T \equiv \Pi\Delta(x : I^\imath\,(\vec{p}, \vec{u})).U \qquad \imath \;\mathsf{pos}\; U \qquad \#\Delta = n-1 \\ \imath \notin \mathrm{SV}(\Gamma, \Delta, \vec{u}, M) \qquad \Gamma \vdash_{\mathsf{t}} T : u \\ \Gamma(f : T) \vdash_{\mathsf{t}} M : T\,[\imath := \widehat{i}] \qquad \Gamma \vdash_{\mathsf{t}} \vec{N} : \Delta(x : I^s\,(\vec{p}, \vec{u})) \end{array}}{\Gamma \vdash_{\mathsf{t}} \mathsf{fix}_n\ f : |T|^\imath := (M, \vec{N}) : U\left[\mathrm{dom}\,(\Delta) := \vec{N}\right][\imath := s]} \quad \mathrm{SV}(\vec{N}) = \emptyset$$

Furthermore, we assume:

1. $[\Gamma]$ is defined with $\gamma \in [\Gamma]$;
2. $[\Gamma \vdash T^\infty]_\gamma$ is defined;
3. $[\Gamma(f : T) \vdash M]_{\gamma,\phi}$ is defined for any ϕ such that $\gamma, \phi \in [\Gamma(f : T)]$;
4. $[\![\Gamma]\!]^\infty$ is defined;
5. $[\![\Gamma \vdash T]\!]^{\infty(\imath:=\mathfrak{a})}_{\gamma\sim\gamma}$ is defined for any ordinal $\mathfrak{a}$;
6. $[\Gamma(f : T) \vdash M]_{\gamma_1,\phi_1 \sim \gamma_2,\phi_2} \sqsubset [\![\Gamma \vdash T\,[\imath := \widehat{i}]]\!]^{\infty(\imath:=\mathfrak{a})}_{\gamma\sim\gamma}$, for any $\phi_1 \sim \phi_2 \sqsubset [\![\Gamma \vdash T]\!]^{\infty(\imath:=\mathfrak{a})}_{\gamma\sim\gamma}$.

Given the above assumptions, we prove that there is a unique function satisfying the fixpoint equations of Def. 4.24. For readability, we omit the context in the interpretations as well as γ. We write $\uparrow$ for $\uparrow^u_{\Pi([\Delta](\gamma),[U](\gamma,\_))}$, and $\imath := \mathfrak{a}$ for $\infty(\imath := \mathfrak{a})$.

Lemma 4.48. *Let $(\phi_1, \phi_2) \sqsubset [\![T]\!]^{\imath:=\mathfrak{a}}$. Then, for all $\delta \sqsubset [\Delta^\infty]$ and α such that $o(\alpha) < \mathfrak{a}$, $\uparrow(\phi_1)(\delta, \alpha) = \uparrow(\phi_2)(\delta, \alpha)$.*

Proof. Recall that $T \equiv \Pi\Delta(I^\imath\,(\vec{p}, \vec{a})).U$. Note that $[\![\Delta]\!]^{\imath:=\mathfrak{a}} = [\![\Delta^\infty]\!]^\infty = ([\Delta^\infty])^2$. Given $\delta \sqsubset [\Delta^\infty]$, we write $\mathcal{I}^{\mathfrak{a}}_\delta$ for $\Phi^{\mathfrak{a}}_I([\vec{p}^\infty]_\delta)([\vec{a}^\infty]_\delta)$.

By Lemma 4.41, $[\![I^s\,(\vec{p}, \vec{a})]\!]^\pi_{\delta\sim\delta} = (\mathcal{I}^{(\!|s|\!)_\pi}_\delta)^2$. Consider $\delta \sqsubset [\Gamma \vdash \Delta^\infty]_\gamma$ and $\alpha \sqsubset \mathcal{I}^{(\!|s|\!)_\pi}_\delta$. Then

$$(\uparrow(\phi_1)(\delta, \alpha), \uparrow(\phi_2)(\delta, \alpha)) \sqsubset [\![U]\!]^{\imath:=\mathfrak{a}}_{\delta,\alpha\sim\delta,\alpha}$$

Since $\imath \;\mathsf{pos}\; U$, $[\![U]\!]^{\imath:=\mathfrak{a}}_{\delta,\alpha\sim\delta,\alpha} \subseteq [\![U]\!]^\infty_{\delta,\alpha\sim\delta,\alpha}$. Hence, $\uparrow(\phi_1)(\delta, \alpha) = \uparrow(\phi_2)(\delta, \alpha)$. □

We give the definition of a function satisfying the fixpoint equations. Intuitively, the function is constructed by iterating the body of the fixpoint.

Given a function $\phi \sqsubset \Pi([\Delta^\infty(x : I^\infty\,(\vec{p}^\infty, \vec{a}^\infty))], [U^\infty]\_)$, we define the function $\phi_\perp$ as follows:

$$\phi_\perp(\delta, \alpha) = \begin{cases} \perp_{[U^\infty](\delta,\alpha)} & \text{if } \alpha = \emptyset \\ \phi(\delta, \alpha) & \text{otherwise} \end{cases}$$

We extend this operation to set of functions: given $\Phi \subseteq ([T^\infty])_\circ$, we define $\Phi_\perp = \{\phi_\perp : \phi \in \Phi\}$. Given a set of functions $\Phi \subseteq ([T^\infty])_\circ$, we write $[M]_\Phi$ for the set $\{[M]_\phi : \phi \in \Phi\}$.

We define a sequence of sets $\{\Phi^{\mathfrak{a}}\}$ for ordinals $\mathfrak{a}$, as follows.

1. $\Phi^0 = \{\phi_\bot : \phi \sqsubset [T^\infty]\}$;
2. $\Phi^{\mathfrak{a}+1} = ([M]_{\Phi^{\mathfrak{a}}})_\bot$;
3. $\Phi^{\mathfrak{b}} = \bigcap_{\mathfrak{a}<\mathfrak{b}} \Phi^{\mathfrak{a}}$, where $\mathfrak{b}$ is a limit ordinal.

The intuition is that the functions in $\Phi^{\mathfrak{a}}$ define the values of the fixpoint for elements of order less than $\mathfrak{a}$. The next lemma formalizes this intuition; we prove that all functions in $\Phi^{\mathfrak{a}}$ coincide in elements of order less than $\mathfrak{a}$.

Lemma 4.49. *For all $\mathfrak{a}$, $\Phi^{\mathfrak{a}} \sim \Phi^{\mathfrak{a}} \sqsubset [\![T]\!]^{\imath:=\mathfrak{a}}$. This means, for all $\phi_1, \phi_2 \in \Phi^{\mathfrak{a}}$, $\phi_1 \sim \phi_2 \sqsubset [\![T]\!]^{\imath:=\mathfrak{a}}$.*

Proof. Note that, since $\imath \notin \mathrm{SV}(\Delta)$, $[\![\Delta]\!]^{\imath:=\mathfrak{a}} = ([\Delta^\infty])^2$. Then, $\delta_1 \sim \delta_2 \sqsubset [\![\Delta]\!]^{\imath:=\mathfrak{a}}$ iff $\delta_1 = \delta_2$ and $\delta_1 \sqsubset [\Delta^\infty]$.

To prove the lemma, we proceed by transfinite induction on $\mathfrak{a}$.

$\mathfrak{a} = 0$. We need to prove that for all $\phi_1, \phi_2 \sqsubset [T^\infty]$, $(\phi_1)_\bot \sim (\phi_2)_\bot \sqsubset [\![T]\!]^{\imath:=0}$. This means, for all $\delta \sqsubset [\Delta^\infty]$, $\uparrow\phi_{1\bot}(\delta, \emptyset) \sim \uparrow\phi_{2\bot}(\delta, \emptyset) \sqsubset [\![U]\!]^{\imath:=0}$ Since, $\uparrow\phi_{1\bot}(\delta, \emptyset) = \uparrow\phi_{2\bot}(\delta, \emptyset) = \bot_{[U](\delta,\emptyset)}$, the result follows.

$\mathfrak{a} = \mathfrak{b} + 1$. The result follows by assumption 6.

$\mathfrak{a}$ is a limit ordinal. Let $\phi_1, \phi_2 \sqsubset \Phi^{\mathfrak{a}}$. Let $\delta \sqsubset [\Delta^\infty]$ and $\alpha \sqsubset [I^\imath\,(\vec{p}^\infty, \vec{a}^\infty)]_{\imath:=\mathfrak{a}}$. Then, there exists $\mathfrak{b} < \mathfrak{a}$ such that $\delta \sqsubset [\Delta^\infty]$ and $\alpha \sqsubset [I^\imath\,(\vec{p}^\infty, \vec{a}^\infty)]_{\imath:=\mathfrak{b}}$. By IH, $\uparrow\phi_{1\bot}(\delta, \alpha) \sim \uparrow\phi_{2\bot}(\delta, \alpha) \sqsubset [\![U]\!]^{\imath:=\mathfrak{b}}$. The result follows from $\imath$ pos U and Lemma 4.46.

□

In the next lemma, we prove that the sequence of sets defined above is decreasing. Each step of the sequence defines another element of the function.

Lemma 4.50. *The sequence $\{\Phi^{\mathfrak{a}}\}_{\mathfrak{a}}$ is decreasing. That is, if $\mathfrak{b} \leq \mathfrak{a}$, then $\Phi^{\mathfrak{a}} \subseteq \Phi^{\mathfrak{b}}$.*

Proof. By transfinite induction on $\mathfrak{b}$. □

After $o(I)$ steps of iterating the function $\alpha \sqsubset [T^\infty] \mapsto [M]_\alpha$, the fixpoint should be completely defined. In other words, there should be just one function in the set $\Phi^{o(I)}$. Note that there is at most one function in $\Phi^{o(I)}$, since $[\![\Delta^\infty(x : I^\imath\,(\vec{p}^\infty, \vec{a}^\infty))]\!]^{\imath:=o(I)} = ([\Delta^\infty(x : I^\imath\,(\vec{p}^\infty, \vec{a}^\infty))])^2$. In this Λ-set, two elements are related iff they are equal.

We give an explicit definition of a function satisfying the fixpoint equations. Let $F' \in \Pi([\Delta^\infty(x : I^s\,(\vec{p}^\infty, \vec{a}^\infty))], [U])$ be defined as follows: for each $(\delta, \alpha) \sqsubset [\Delta^\infty(x : I^\imath\,(\vec{p}^\infty, \vec{a}^\infty))]$, we define $F'(\delta, \alpha) = \uparrow\phi(\delta, \alpha)$,for some $\phi \in \Phi^{o(\alpha)}$.

Note that F' is well defined: from Lemmas 4.49 and 4.50, it does not matter which function ϕ in $\Phi^{o(\alpha)}$ we choose, since all such functions have the same value.

Let $F = \downarrow F'$. We prove that F satisfies the fixpoint equations; this follows a simple consequence of the following two lemmas.

Lemma 4.51. *For all $\mathfrak{a}$, $\Phi^{\mathfrak{a}} \sim F \sqsubset [\![T]\!]^{\imath:=\mathfrak{a}}$. This means, for all $\phi \in \Phi^{\mathfrak{a}}$, $\phi \sim F \sqsubset [\![T]\!]^{\imath:=\mathfrak{a}}$.*

Proof. Let $\phi \in \Phi^{\mathfrak{a}}$. It suffices to prove that for $\delta \sqsubset [\Delta^\infty]$ and $\alpha \sqsubset \Phi^{\mathfrak{a}}_I([\vec{p}^\infty]_\delta)([\vec{a}^\infty]_\delta)$,

$$\uparrow(\phi)(\delta, \alpha) \sim \uparrow(F)(\delta, \alpha) \sqsubset [\![U]\!]^{\imath:=\mathfrak{a}}_{\delta,\alpha\sim\delta,\alpha}$$

By definition, $\uparrow(F)(\delta, \alpha) = \uparrow(\phi')(\delta, \alpha)$ for some $\phi' \in \Phi^{o(\alpha)}$. Since $o(\alpha) \leq \mathfrak{a}$, by Lemma 4.50, $\phi \in \Phi^{o(\alpha)}$. By Lemma 4.49, $\phi \sim \phi' \sqsubset [\![T]\!]^{\imath:=o(\alpha)}$. Then, $\uparrow(\phi)(\delta, \alpha) \sim \uparrow(\phi')(\delta, \alpha) \sqsubset [\![U]\!]^{\imath:=o(\alpha)}_{\delta,\alpha\sim\delta,\alpha}$. The result follows from $\imath$ pos U and Soundness of Subtyping (Lemma 4.46). □

Lemma 4.52. *For all* $\mathfrak{a}$, $F \sim ([M]_F)_\perp \sqsubset [\![T]\!]^{\imath:=\mathfrak{a}}$.

Proof. Let $\delta \sqsubset [\Delta^\infty]$ and $\alpha \sqsubset \Phi^{\mathfrak{a}}_I([\vec{p}^\infty]_\delta)([\vec{a}^\infty]_\delta)$. If $\alpha = \emptyset$, then $\uparrow(F)(\delta,\alpha) = \perp_{[U](\delta,\alpha)}$ and the result follows immediately. We consider the case when $\alpha \neq \emptyset$. By definition, $\uparrow(F)(\delta,\alpha) = \uparrow(\phi)(\delta,\alpha)$ for some $\phi \in \Phi^{o(I)}$. Since $\alpha \neq \emptyset$, $o(I)$ is a successor cardinal, let us say, $\mathfrak{b}+1$. Then, $\phi = ([M]_{\phi'})_\perp$ for some $\phi' \in \Phi^{\mathfrak{b}}$. From the previous lemma, $\phi' \sim F \sqsubset [\![T]\!]^{\imath:=\mathfrak{b}}$, and by the assumption 6, $([M]_{\phi'})_\perp \sim ([M]_F)_\perp \sqsubset [\![T]\!]^{\imath:=\mathfrak{b}+1}$. Then, $\uparrow([M]_{\phi'})_\perp(\delta,\alpha) \sim \uparrow([M]_F)_\perp(\delta,\alpha) \sqsubset [\![U]\!]^{\imath:=\mathfrak{b}+1}$. Note that the lhs is equal to $\uparrow(F)(\delta,\alpha)$. The result follows from $\imath$ pos U and Lemma 4.46. □

As a consequence of the above lemma and Lemma 4.49, F satisfies the fixpoint equations. It suffices to instantiate the above lemma with $\mathfrak{a} = o(I)$.

In the main soundness theorem to be defined below, assumptions 1- 6 are satisfied by the inductive hypothesis. Then the existence of a function satisfying the fixpoint equations is ensured.

The soundness theorem. The statement of the soundness theorem is more complex than the intuitive meaning given above.

Lemma 4.53 (Soundness)**.** *1. If* $\mathsf{WF}(\Gamma)$*, then* $[\![\Gamma]\!]^\pi$ *is defined for any stage valuation* π*. Furthermore, if* $(\gamma_1,\gamma_2) \sqsubset [\![\Gamma]\!]^\pi$*, then* $(\gamma_1,\gamma_1) \sqsubset [\![\Gamma]\!]^\infty$*.*

2. If $\Gamma \vdash M : T$*, with* $\mathrm{SV}(M) = \emptyset$*, and* $(\gamma_1,\gamma_2) \in [\![\Gamma]\!]^\pi$*, then* $[\Gamma \vdash M]_{\gamma_1,\gamma_2}$*,* $[\![\Gamma \vdash T]\!]^\pi_{\gamma_1\sim\gamma_2}$ *are defined and*

$$[\Gamma \vdash M]_{\gamma_1,\gamma_2} \sqsubset [\![\Gamma \vdash T]\!]^\pi_{\gamma_1\sim\gamma_2} \ .$$

3. If $\Gamma \vdash T : \mathsf{Type}_i$*,* $\mathsf{simple}(T)$*, and* $(\gamma_1,\gamma_2) \in [\![\Gamma]\!]^\pi$*, then* $[\![\Gamma \vdash T]\!]^\pi_{\gamma_1\sim\gamma_2}$ *is defined, and*

$$[\![\Gamma \vdash T]\!]^\pi_{\gamma_1\sim\gamma_2} \sqsubset \mathcal{U}_i \ .$$

4. If $\Gamma \vdash T : \mathsf{Prop}$*,* $\mathsf{simple}(T)$*, and* $(\gamma_1,\gamma_2) \in [\![\Gamma]\!]^\pi$*, then* $[\![\Gamma \vdash T]\!]^\pi_{\gamma_1\sim\gamma_2}$ *is a defined relationally degenerated* Λ*-set.*

Proof. We prove all statements simultaneously, by induction on the type derivation, and case analysis on the last rule. Except for rules (ind) and (prod), clauses 3 and 4 follow easily from clause 2 and Lemma 4.40. Hence, we only consider clause 2, except for rules (ind) and (prod).

(empty) Trivial.

(cons) We have the derivation

$$\frac{\mathsf{WF}(\Gamma) \qquad \Gamma \vdash T : u}{\mathsf{WF}(\Gamma(x:T))}$$

Let π be a stage valuation. By IH, $[\![\Gamma]\!]^\pi$ is defined, and $(\gamma_1,\gamma_1) \sqsubset [\![\Gamma]\!]^\infty$ for any $(\gamma_1,\gamma_2) \sqsubset [\![\Gamma]\!]^\pi$. Also by IH, $[\![\Gamma \vdash T]\!]^\pi_{\gamma_1\sim\gamma_2}$ is defined, and $[\![\Gamma \vdash T]\!]^\infty_{\gamma_1\sim\gamma_1}$ is equal to $([\Gamma \vdash T^\infty]_{\gamma_1})^2$. The result follows by IH.

(var) Follows directly from $(\gamma_1,\gamma_2) \in [\![\Gamma]\!]^\pi$.

(sort) By definition of the interpretation of universes.

(prod) We need to consider two cases: impredicative and predicative product. Both cases follow directly from the definition of the interpretation. In the predicative case, we use the closure properties of the interpretation of the predicative universes.

(ind) By definition of the interpretation of inductive types. It is essential that constructors are defined in the same universe to ensure that the monotone operator is defined in the same universe.

(conv) Follows easily by IH and soundness of subtyping (Lemma 4.46).

(abs) M is $\lambda x : |T|.M_1|U|$; we have the derivation

$$\frac{\Gamma(x : T) \vdash M_1 : U}{\Gamma \vdash \lambda x : |T|.M_1|U| : \Pi x : T.U}$$

We consider two cases.

I Assume $\Gamma^\infty \vdash \Pi x : T^\infty.U^\infty : \mathsf{Type}_i$. By IH, $\gamma_1 \sim \gamma_1 \sqsubset [\![\Gamma]\!]^\infty$. Take $\alpha \sqsubset [\Gamma \vdash T]_{\gamma_1}$. Then $(\alpha, \alpha) \sqsubset [\![\Gamma \vdash T]\!]^\infty_{\gamma_1 \sim \gamma_1}$, and $\gamma_1, \alpha \sim \gamma_1, \alpha \sqsubset [\![\Gamma]\!]^\infty$. By IH, $[\Gamma(x : T) \vdash M]_{\gamma_1,\alpha}$ is defined. Similarly, $[\Gamma(x : T) \vdash M]_{\gamma_2,\alpha}$ is defined for any $\alpha \sqsubset [\Gamma \vdash T]_{\gamma_2}$.

II Assume $\Gamma^\infty \vdash \Pi x : T^\infty.U^\infty : \mathsf{Prop}$.

(app) M is $\mathsf{app}^{|U|}_{x:|T|}(M_1, M_2)$; we have the derivation

$$\frac{\begin{array}{c}\Gamma \vdash \Pi x : T.U : u \\ \Gamma \vdash M_1 : \Pi x : T.U \qquad \Gamma \vdash M_2 : T\end{array}}{\Gamma \vdash \mathsf{app}^{|U|}_{x:|T|}(M_1, M_2) : U\,[x := M_2]}\ \mathrm{SV}(M_2) = \emptyset$$

We consider two cases.

I Assume $\Gamma^\infty \vdash \Pi x : T^\infty.U^\infty : \mathsf{Type}_i$. By IH, $[\Gamma \vdash M_1]_{\gamma_1,\gamma_2} \sqsubset [\![\Gamma \vdash \Pi x : T.U]\!]^\pi_{\gamma_1 \sim \gamma_2}$ and $[\Gamma \vdash M_2]_{\gamma_1,\gamma_2} \sqsubset [\![\Gamma \vdash T]\!]^\pi_{\gamma_1 \sim \gamma_2}$. By definition of the relational interpretation, $([\Gamma \vdash M_1]_{\gamma_1}[\Gamma \vdash M_2]_{\gamma_1}, [\Gamma \vdash M_1]_{\gamma_2}[\Gamma \vdash M_2]_{\gamma_2}) \sqsubset [\![\Gamma(x : T_1) \vdash U]\!]^\pi_{\gamma_1,[\Gamma\vdash M_2](\gamma_1)\sim\gamma_2,[\Gamma\vdash M_2](\gamma_2)}$ By Lemma 4.44, the latter is equal to $[\![\Gamma \vdash U\,[x := M_2]]\!]^\pi_{\gamma_1 \sim \gamma_2}$.

(constr) M is $C(|\vec{p}|, \vec{N}, P)$; we have the derivation

$$\frac{\begin{array}{c}I \in \Sigma \qquad \Gamma \vdash \vec{p} : \Delta_p \\ \Gamma \vdash \vec{N} : \Delta_i\,[\mathrm{dom}\,(\Delta_p) := \vec{p}] \qquad \Gamma \vdash P : \Pi\Theta_i\,[\mathrm{dom}\,(\Delta_p) := \vec{p}]\,.I^s\,(\vec{p}, \vec{u_i})\end{array}}{\Gamma \vdash C_i(|\vec{p}|, \vec{N}, P) : \mathsf{typeConstr}^s_{C_i}(\vec{p}, \vec{N})}$$

where $\mathsf{typeConstr}^s_{C_i}(\vec{p}, \vec{N}) = \widehat{I^s}\,(\vec{p}, \vec{a})$, and $\vec{a} = \vec{t_i}\,[\mathrm{dom}\,(\Delta_p) := \vec{p}]\,\left[\mathrm{dom}\,(\Delta_i) := \vec{N}\right]$. Note that, since positive and negative parameters do not appear in $\vec{t_i}$, we have $\mathrm{SV}(\vec{t_i}) = \emptyset$. The result follows easily from the IH.

(case) M is $\mathsf{case}_{|P|}\ x := M\ \mathsf{in}\ I\,(|\vec{p}|, \vec{y})\ \mathsf{of}\ \langle C_i \Rightarrow N_i \rangle_i$; we have the derivation

$$\frac{\begin{array}{c}\Gamma \vdash M : \widehat{I^s}\,(\vec{p}, \vec{a}) \qquad I \in \Sigma \qquad \Gamma\,(\mathsf{caseType}^s_I(\vec{p}, \vec{y}, x)) \vdash P : u \\ \Gamma \vdash N_i : \mathsf{branch}^s_{C_i}(\vec{p}, \vec{y}.x.P)\end{array}}{\Gamma \vdash \mathsf{case}_{|P|}\ x := M_1\ \mathsf{in}\ I\,(|\vec{p}|, \vec{y})\ \mathsf{of}\ \langle C_i \Rightarrow N_i \rangle_i : P\,[\vec{y} := \vec{a}]\,[x := M]}$$

Let $I = \mathsf{Ind}(I[\Delta_p]^{\vec{\nu}} : \Pi\Delta_a.u := \langle C_i : \Pi\Delta_i.\mathcal{X}\,\vec{t_i} \rangle_i)$. By IH, $[\Gamma \vdash M]_{\gamma_1,\gamma_2} \sqsubset [\![\Gamma \vdash \widehat{I^s}\,(\vec{p}, \vec{a})]\!]^\pi_{\gamma_1 \sim \gamma_2}$. If $[\Gamma \vdash M]_{\gamma_1,\gamma_2} = (\emptyset, \emptyset)$, the result follows immediately. We consider the

case $[\Gamma \vdash M]_{\gamma_1,\gamma_2} = ((j,\rho_1),(j,\rho_2))$. By IH, $[\Gamma \vdash N_j]_{\gamma_1,\gamma_2} \sqsubset [\![\Gamma \vdash \mathsf{branch}^s_{C_i}(\vec{p},\vec{y}.x.P)]\!]^\pi_{\gamma_1\sim\gamma_2}$. By definition of branch and RI of product,

$$(\uparrow_1[\Gamma \vdash N_j]_{\gamma_1}(\rho_1), \uparrow_2[\Gamma \vdash N_j]_{\gamma_2}(\rho_2)) \sqsubset \\ [\![\Gamma \vdash P\left[\vec{y} := \mathsf{indices}_{C_j}(\vec{p})\right]\ [x := C_j(|\vec{p}|, \mathrm{dom}\,(\Delta_i))]]\!]^\pi_{\gamma_1,\rho_1\sim\gamma_2,\rho_2}$$

where $\uparrow_i = \uparrow^{\Gamma\vdash\mathsf{branch}^s_{C_j}(\vec{p},\vec{y}.x.P)}_{\gamma_i}$. Recall that $\mathsf{indices}_{C_j}(\vec{p}) = \vec{t_j}\,[\mathrm{dom}\,(\Delta_p) := \vec{p}]$. By definition of RI of inductive types, $[\Gamma\mathsf{argsConstr}^s_{C_j}(\vec{p}) \vdash \vec{t_j}\,[\mathrm{dom}\,(\Delta_p) := \vec{p}]]_{\gamma_i,\rho_i} = [\Gamma \vdash \vec{a}]_{\gamma_i}$, for $i = 1, 2$. The result follows from Lemma 4.44.

(fix) M is $\mathsf{fix}_n\ f : |T|^\imath := M_1\vec{N}$; we have the derivation

$$\frac{\begin{array}{c} T \equiv \Pi\Delta(x : I^\imath\,(\vec{p},\vec{u})).U \qquad \imath\ \mathsf{pos}\ U \qquad \#\Delta = n-1 \\ \imath \notin \mathrm{SV}(\Gamma,\Delta,\vec{u},M_1) \\ \Gamma \vdash T : u \qquad \Gamma(f : T) \vdash M_1 : T\,[\imath := \widehat{i}] \qquad \Gamma \vdash \vec{N} : \Delta(x : I^s\,(\vec{p},\vec{u})) \end{array}}{\Gamma \vdash \mathsf{fix}_n\ f : |T|^\imath := (M_1,\vec{N}) : U\left[\mathrm{dom}\,(\Delta) := \vec{N}\right][\imath := s]}\ \mathrm{SV}(\vec{N}) = \emptyset$$

Let $\gamma_1 \sim \gamma_2 \sqsubset [\![\Gamma]\!]^\pi$. We are in conditions of apply Lemmas 4.49-4.52, for γ_1 and γ_2. Then $[\Gamma \vdash M]_{\gamma_1}$ and $[\Gamma \vdash M]_{\gamma_2}$ are defined and satisfy the fixpoint equations of Def. 4.24. That is, we have $\phi_1 \sqsubset [\Gamma \vdash T^\infty]_{\gamma_1}$ and $\phi_2 \sqsubset [\Gamma \vdash T^\infty]_{\gamma_2}$ satisfying the fixpoint equations of Def. 4.24.

We prove $\phi_1 \sim \phi_2 \sqsubset [\![\Gamma \vdash T]\!]^\pi_{\gamma_1\sim\gamma_2}$.

We proceed by induction on $\pi(\imath)$.

- $\pi(\imath) = 0$ Consider $\delta_1,\alpha_1 \sim \delta_2,\alpha_2 \sqsubset [\![\Delta(x : I^\imath\,(\vec{p},\vec{a}))]\!]^\pi$. Then $\alpha_1 = \alpha_2 = \emptyset$. By definition, we have $\phi_i(\delta_i,\emptyset) = \bot_{[U](\delta_i,\emptyset)}$. Finally, $(\bot_{[U](\delta_1,\emptyset)}, \bot_{[U](\delta_2,\emptyset)}) \sqsubset [\![U]\!]^\pi_{\delta_1,\emptyset\sim\delta_2,\emptyset}$.
- $\pi(\imath) = \mathfrak{a}+1$ By IH, $\phi_1 \sim \phi_2 \sqsubset [\![\Gamma \vdash T]\!]^{\pi(\imath:=\mathfrak{a})}_{\gamma_1\sim\gamma_2}$. Then, $[\Gamma(f : T) \vdash M_1]_{\gamma_1,\phi_1\sim\gamma_2,\phi_2} \sqsubset [\![\Gamma \vdash T\,[\imath := \widehat{i}]]\!]^{\pi(\imath:=\mathfrak{a})}_{\gamma_1\sim\gamma_2}$. By Soundness of Stage Substitution (Lemma 4.43), $[\![\Gamma \vdash T\,[\imath := \widehat{i}]]\!]^{\pi(\imath:=\mathfrak{a})}_{\gamma_1\sim\gamma_2} = [\![\Gamma \vdash T]\!]^{\pi(\imath:=\mathfrak{a}+1)}_{\gamma_1\sim\gamma_2}$. The result follows easily, since ϕ_1 and ϕ_2 satisfy the fixpoint equations. Consider $\delta_1,\alpha_1 \sim \delta_2,\alpha_2 \sqsubset [\![\Delta(x : I^\imath\,(\vec{p},\vec{a}))]\!]^{\pi(\imath:=\mathfrak{a}+1)}$. If $\alpha_1 = \alpha_2 = \emptyset$, the result follows as in the previous case. Otherwise, for $i = 1, 2$, $\uparrow(\phi_i)(\delta_i,\alpha_i) = \uparrow([M]_{\phi_i})(\delta_i,\alpha_i)$, from the fixpoint equations, and the result follows since $[\Gamma(f : T) \vdash M_1]_{\gamma_1,\phi_1\sim\gamma_2,\phi_2} \sqsubset [\![\Gamma \vdash T\,[\imath := \widehat{i}]]\!]^{\pi(\imath:=\mathfrak{a})}_{\gamma_1\sim\gamma_2}$.
- $\pi(\imath)$ **is a limit ordinal** Let $\pi(\imath) = \mathfrak{b}$. Consider

$$\delta_1,\alpha_1 \sim \delta_2,\alpha_2 \sqsubset [\![\Delta(x : I^\imath\,(\vec{p},\vec{a}))]\!]^\pi\ .$$

  Then there exists $\mathfrak{a} < \mathfrak{b}$ such that $\alpha_1 \sim \alpha_2 \sqsubset [\![I^\imath(\vec{p},\vec{u})]\!]^{\pi(\imath:=\mathfrak{b})}_{\delta_1\sim\delta_2}$. By IH, $\uparrow(\phi_1)(\delta_1,\alpha_1) \sim \uparrow(\phi_2)(\delta_2,\alpha_2) \sqsubset [\![U]\!]^{\pi(\imath:=\mathfrak{b})}_{\delta_1,\alpha_1\sim\delta_2,\alpha_2}$. Since $\imath\ \mathsf{pos}\ U$, the result follows from Soundness of Subtyping (Lemma 4.46).

The conclusion follows easily.

□

4.5 Strong Normalization

In this section we prove our main result: strong normalization of ECIC$\widehat{\ }_{_}$. This result follows as a consequence of Lemma 4.55 given below, stating that a term, under a suitable substitution, realizes its interpretation. In other words, given $\Gamma \vdash M : T$, $(\gamma_1, \gamma_2) \sqsubset [\![\Gamma]\!]^\pi$, and a substitution θ that "agrees" with (γ_1, γ_2), then $M\theta \models [\Gamma \vdash M]_{\gamma_1 \sim \gamma_2}$ in the interpretation of the type T. We define precisely the notion of "agrees".

Definition 4.54. *Let $(\gamma_1, \gamma_2) \in [\![\Gamma]\!]^\pi$. We define $\theta \models^\Gamma_\pi (\gamma_1, \gamma_2)$ by the following clauses:*

$$\frac{}{\varepsilon \models^\Gamma_\pi []} \qquad \frac{\theta \models^\Gamma_\pi (\gamma_1, \gamma_2) \qquad M \models_{[\![\Gamma \vdash T]\!](\gamma_1 \sim \gamma_2, \pi)} (\alpha_1, \alpha_2)}{\theta(x \mapsto M) \models^{\Gamma(x:T)}_\pi (\gamma_1, \alpha_1), (\gamma_2, \alpha_2)}$$

Lemma 4.55. *If $\Gamma \vdash M : T$ and $(\gamma_1, \gamma_2) \in [\![\Gamma]\!]^\pi$ and $\theta \models^\Gamma_\pi (\gamma_1, \gamma_2)$, then*

$$M\theta \models_{[\![\Gamma \vdash T]\!](\gamma_1 \sim \gamma_2, \pi)} [\Gamma \vdash M]_{\gamma_1, \gamma_2}$$

Proof. We proceed by induction on the derivation of $\Gamma \vdash M : T$. We consider only the most relevant cases.

(app) M is $\mathsf{app}^{|U|}_{x:|T|}(M_1, M_2)$; we have the derivation

$$\frac{\begin{array}{c}\Gamma \vdash \Pi x : T.U : u \\ \Gamma \vdash M_1 : \Pi x : T.U \qquad \Gamma \vdash M_2 : T\end{array}}{\Gamma \vdash \mathsf{app}^{|U|}_{x:|T|}(M_1, M_2) : U\,[x := M_2]}\ \mathrm{SV}(M_2) = \emptyset$$

By IH, $M_1\theta \models_{[\![\Gamma \vdash \Pi x:T.U]\!](\gamma_1 \sim \gamma_2, \pi)} [\Gamma \vdash M_1]_{\gamma_1, \gamma_2}$, $M_2\theta \models_{[\![\Gamma \vdash T]\!](\gamma_1 \sim \gamma_2, \pi)} [\Gamma \vdash M_2]_{\gamma_1, \gamma_2}$, and $\Pi x : T.U\theta \models_{[\![\Gamma \vdash u]\!](\gamma_1 \sim \gamma_2, \pi)} [\Gamma \vdash \Pi x : T.U]_{\gamma_1, \gamma_2}$. The latter implies that $\Pi x : T.U\theta \in \mathrm{SN}$. By definition of the RI for products,

$$\mathsf{app}^{|U|}_{x:|T|}(M_1, M_2)\,\theta \models_{[\![\Gamma(x:T) \vdash U]\!](\gamma_1, \alpha_1 \sim \gamma_2, \alpha_2, \pi)} [\Gamma \vdash \mathsf{app}^{|U|}_{x:|T|}(M_1, M_2)]_{\gamma_1, \gamma_2}$$

where $(\alpha_1, \alpha_2) = [\Gamma \vdash M_2]_{\gamma_1, \gamma_2}$. This is valid when u is Prop and Type_k for some k. The result follows from Lemma 4.44.

(case) M is $\mathsf{case}_{|P|}\ x := M\ \mathsf{in}\ I\,(|\vec{p}|, \vec{y})\ \mathsf{of}\ \langle C_i \Rightarrow N_i \rangle_i$; we have the derivation

$$\frac{\begin{array}{c}\Gamma \vdash M : \widehat{I^s}\,(\vec{p}, \vec{a}) \qquad I \in \Sigma \qquad \Gamma\,(\mathsf{caseType}^s_I(\vec{p}, \vec{y}, x)) \vdash P : u \\ \Gamma \vdash N_i : \mathsf{branch}^s_{C_i}(\vec{p}, \vec{y}.x.P)\end{array}}{\Gamma \vdash \mathsf{case}_{|P|}\ x := M_1\ \mathsf{in}\ I\,(|\vec{p}|, \vec{y})\ \mathsf{of}\ \langle C_i \Rightarrow N_i \rangle_i : P\,[\vec{y} := \vec{a}]\,[x := M]}$$

Let $I = \mathsf{Ind}(I[\Delta_p]^{\vec{\nu}} : \Pi\Delta_a.u := \langle C_i : \Pi\Delta_i.\mathcal{X}\,\vec{t_i} \rangle_i)$. We write X^* for $X\,[\mathrm{dom}\,(\Delta_p) := \vec{p}]$, for a term or context X.

Let $\theta \models^\Gamma_\pi (\gamma_1, \gamma_2)$. We want to prove that $M\theta \models_{[\![\Gamma \vdash P[\vec{y} := \vec{a}][x := M]]\!](\gamma_1 \sim \gamma_2, \pi)} [\Gamma \vdash M]_{\gamma_1 \sim \gamma_2}$. The IH gives us

1 $M_1\theta \models_{[\![I^{\widehat{s}}(\vec{p}, \vec{a})]\!](\gamma_1 \sim \gamma_2, \pi)} [\Gamma \vdash M_1]_{\gamma_1 \sim \gamma_2}$;

2 $N_i\theta \models_{[\![\mathsf{branch}^s_{C_i}(\vec{p}, \vec{y}.x.P)]\!](\gamma_1 \sim \gamma_2, \pi)} [\Gamma \vdash N_i]_{\gamma_1 \sim \gamma_2}$.

By definition, $[\![I^{\widehat{s}}\,(\vec{p},\vec{a})]\!]^{\pi}_{\gamma_1\sim\gamma_2} = \mathbf{\Phi}^{\mathfrak{a}+1}_{I(\vec{p})}([\Gamma\vdash\vec{a}]_{\gamma_1\sim\gamma_2})$, where $\mathfrak{a} = (\!|s|\!)_\pi$. We write $\mathbf{\Phi}^{\mathfrak{a}}$ for $\mathbf{\Phi}^{\mathfrak{a}+1}_{I(\vec{p})}$.
We have two subcases.

1. $[\Gamma\vdash M_1]_{\gamma_1\sim\gamma_2} = ((\alpha,\emptyset),(\alpha,\emptyset))$. In this case, $M_1\theta \to^*_{\mathrm{wh}} R \in \mathrm{AT}$. Then $M \to^*_{\mathrm{wh}} \mathsf{case}_{|P|}\ x := R\ \mathsf{in}\ I\,(|\vec{p}|,\vec{y})\ \mathsf{of}\ \langle C_i \Rightarrow N_i\rangle_i \in \mathrm{AT}$. By definition of the interpretation, $[\Gamma\vdash M]_{\gamma_1\sim\gamma_2} = \bot_{[\Gamma\,(\mathsf{caseType}^s_I(\vec{p},\vec{y},x))\vdash P](\gamma,\alpha,\emptyset)}$. The result follows since atomic terms realize the atomic element of a Λ-set and realizers are closed under wh-expansion.
2. $[\Gamma\vdash M_1]_{\gamma_1\sim\gamma_2} = ((\alpha_1,(j,\nu_1,\rho_1)),(\alpha_2,(j,\nu_2,\rho_2)))$. In this case, $M_1\theta \to^*_{\mathrm{wh}} C_j(\vec{p^\circ},\vec{Q}\,R)$ where $\vec{Q}\,R \models_{[\![\Delta_i{}^*]\!](\gamma_1\sim\gamma_2|\mathcal{X}\mapsto\mathbf{\Phi}^{\mathfrak{a}},\pi)} (\nu_1,\rho_1),(\nu_2,\rho_2)$.
   We have, $[\Gamma\vdash M]_{\gamma_1\sim\gamma_2} = ([N_1]_{\gamma_1}(\nu_1,\rho_1),[N_2]_{\gamma_2}(\nu_2,\rho_2))$. Also, by Substitution of the RI,

   $$[\![\Gamma\vdash P\,[\vec{y}:=\vec{a}]\,[x:=M]]\!]^{\pi}_{\gamma_1\sim\gamma_2} = [\![\Gamma\,(\mathsf{caseType}^s_I(\vec{p},\vec{y},x))\vdash P]\!]^{\pi}_{\gamma_1,\alpha_1,(j,\nu_1,\rho_1)\sim\gamma_2,\alpha_2,(j,\nu_2,\rho_2)}$$

   Then, by IH 2,

   $$\mathsf{app}^{|P|}_{\Delta_j{}^*}\left(N_j,\vec{Q}\,R\right) \models_{[\![P]\!](\gamma_1,\alpha_1,(j,\nu_1,\rho_1)\sim\gamma_2,\alpha_2,(j,\nu_2,\rho_2),\pi)} ([N_1]_{\gamma_1}(\nu_1,\rho_1),[N_2]_{\gamma_2}(\nu_2,\rho_2))$$

   The result follows, since $M\theta \to^*_{\mathrm{wh}} \mathsf{app}^{|P|}_{\Delta_j{}^*}\left(N_j,\vec{Q}\,R\right)$.

(fix) M is $\mathsf{fix}_n\ f : |T|^\imath := (M',\vec{N})$; we have the derivation

$$\frac{\begin{array}{c}T \equiv \Pi\Delta(x : I^\imath\,(\vec{p},\vec{u})).U \qquad \imath\ \mathsf{pos}\ U \qquad \#\Delta = n-1 \\ \imath \notin \mathrm{SV}(\Gamma,\Delta,\vec{u},M') \\ \Gamma\vdash T : u \qquad \Gamma(f:T)\vdash M' : T\,[\imath:=\widehat{\imath}] \qquad \Gamma\vdash\vec{N} : \Delta(x : I^\imath\,(\vec{p},\vec{u}))\end{array}}{\Gamma\vdash \mathsf{fix}_n\ f : |T|^\imath := (M',\vec{N}) : U\,[\imath:=s]}\ \mathrm{SV}(\vec{N}) = \emptyset$$

We write Δ^+ for $\Delta(x : I^\imath\,(\vec{p},\vec{u}))$. Let $(\gamma_1,\gamma_2) \sqsubset [\![\Gamma]\!]^\pi$ and $\theta \models^\Gamma_\pi (\gamma_1,\gamma_2)$. We want to prove that

$$M\theta \models_{[\![\Gamma\vdash T[\imath:=s]]\!](\gamma_1\sim\gamma_2,\pi)} [\Gamma\vdash M]_{\gamma_1\sim\gamma_2}\ . \tag{4.1}$$

Let us write ϕ_i for $\epsilon(F_i,P)$, for $i = 1,2$, where P denotes the fixpoint equations of Def. 4.24. Also, we write $\uparrow_i$ for $\uparrow^u_{\Pi([\Delta^+](\gamma_i),[U](\gamma_i,\_))}$. Recall that the interpretation of M is defined by $[\Gamma\vdash M]_{\gamma_i} = \uparrow_i\phi_i([\Gamma\vdash\vec{N}]_{\gamma_i})$.
We first prove that if $\theta \models^\Gamma_\pi (\gamma_1,\gamma_2)$

$$(\lambda^{|U|}_{|\Delta^+|}.\mathsf{fix}_n\ f : |T|^\imath := (M',\mathrm{dom}\left(\Delta^+\right)))\theta \models_{[\![\Gamma\vdash T]\!](\gamma_1\sim\gamma_2,\pi)} (\phi_1,\phi_2)\ .$$

We write M^+ for $\lambda^{|U|}_{|\Delta^+|}.\mathsf{fix}_n\ f : |T|^\imath := (M',\mathrm{dom}\,(\Delta^+))$.
Recall that $\Delta^+ = \Delta(x : I^\imath\,(\vec{p},\vec{u}))$. Then $[\![\Gamma\vdash\Delta^+]\!]^\pi_{\gamma_1\sim\gamma_2} \cong \Sigma([\![\Gamma\vdash\Delta]\!]^\pi_{\gamma_1\sim\gamma_2},[\![\Gamma\Delta\vdash I^\imath\,(\vec{p})\ \vec{u}]\!]^\pi_{\gamma_1,\_\sim\gamma_2,\_})$. Then each element of its carrier set is of the form $((\delta_1,\alpha_1),(\delta_2,\alpha_2))$, with $(\delta_1,\delta_2) \sqsubset [\![\Delta]\!]^\pi_{\gamma_1\sim\gamma_2}$ and $\alpha \sqsubset [\![I^\imath\,(\vec{p})\ \vec{u}]\!]^\pi_{\gamma_1,\delta_1\sim\gamma_2,\delta_2}$.
We proceed by transfinite induction on $\pi(\imath)$.

$\pi(\imath) = 0$ Let $\theta \models^{\Gamma}_{\pi(\imath:=0)} (\gamma_1, \gamma_2)$. Let $\vec{t}\,u \models_{[\![\Delta^+]\!](\gamma_1 \sim \gamma_2, \pi)} (\delta_1, \alpha_1), (\delta_2, \alpha_2)$. By definition of the relational interpretation of I, $\alpha_1 = \alpha_2 = \emptyset$, and $u \to^*_{\mathrm{wh}} R \in \mathrm{AT}$. On the other hand, by definition of ϕ_1, $\uparrow\phi_1(\delta_1, \emptyset) = \bot_{[U](\gamma_1, \delta_1, \emptyset)}$; similarly for ϕ_2. Then,

$$\mathsf{app}^{|U\theta|}_{|\Delta^+\theta|}\left(M^+\theta, \vec{t}\,u\right) \to^* \mathsf{fix}_n\ f : T^\star\theta := (M'\theta, \vec{t}\,R) \models_{[\![\Gamma \vdash U]\!](\gamma_1, \delta_1, \emptyset \sim \gamma_2, \delta_2, \emptyset, \pi)} (\bot_{[U](\gamma_1, \delta_1, \emptyset)}, \bot_{[U](\gamma_2, \delta_2, \emptyset)}),$$

since $\mathsf{fix}_n\ f : T^\star\theta := (M'\theta, \vec{t}\,R)$ is an atomic term. The result follows.

$\pi(\imath) = \mathfrak{a} + 1$ Let $\theta \models^{\Gamma}_{\pi(\imath:=\mathfrak{a}+1)} (\gamma_1, \gamma_2)$.

The IH says that, if $\theta \models^{\Gamma}_{\pi(\imath:=\mathfrak{a})} (\gamma_1, \gamma_2)$ then

$$M^+\theta \models_{[\![T]\!](\gamma_1 \sim \gamma_2, \pi(\imath:=\mathfrak{a}))} (\phi_1, \phi_2)$$

The outer IH says that

$$\theta' \models^{\Gamma(f:T)}_{\pi(\imath:=\mathfrak{a})} (\gamma_1, \phi_1), (\gamma_2, \phi_2) \Rightarrow M'\theta' \models_{[\![T[\imath:=\widehat{\imath}]]\!](\gamma_1 \sim \gamma_2, \pi(\imath:=\mathfrak{a}))} [\Gamma(f : T) \vdash M']_{\gamma_1, \phi_1 \sim \gamma_2, \phi_2}$$

Let $\theta' = \theta(f \mapsto M^+\theta)$. Then, $\theta' \models^{\Gamma(f:T)}_{\pi(\imath:=\mathfrak{a})} (\gamma_1, \phi_1), (\gamma_2, \phi_2)$. By the outer IH,

$$M'\theta' \equiv M'\theta\left[f := M^+\theta\right] \models_{[\![T[\imath:=\widehat{\imath}]]\!](\gamma_1 \sim \gamma_2, \pi(\imath:=\mathfrak{a}))} [\Gamma(f : T) \vdash M']_{\gamma_1, \phi_1 \sim \gamma_2, \phi_2} \quad (4.2)$$

Let $\vec{t}\,u \models_{[\![\Delta^+]\!](\gamma_1 \sim \gamma_2, \pi(\imath:=\mathfrak{a}+1))} (\delta_1, \alpha_1), (\delta_2, \alpha_2)$. We consider two subcases. The first subcase, when $\alpha_1 = \alpha_2 = \emptyset$; we proceed as in the case $\pi(\imath) = 0$. The second subcase, when $\alpha_1 = (j, \nu_1, \rho_1)$ and $\alpha_2 = (j, \nu_2, \rho_2)$. By definition of the RI for I, $u \to^*_{\mathrm{wh}} C_j(\vec{p^\circ}, \vec{a})$. Then,

$$\mathsf{app}^{|U|}_{|\Delta^+|}\left(M^+\theta, \vec{t}\,u\right) \to^*_{\mathrm{wh}} \mathsf{fix}_n\ f : T^\star\theta := (M'\theta, \vec{t}\,u) \to_{\mathrm{wh}} \mathsf{app}^{|U|\theta}_{|\Delta^+\theta|}\left(M'\theta\left[f := M^+\theta\right], (\vec{t}(C_j(\vec{p^\circ}, \vec{a})))\right) \equiv F$$

Recall that $[\Gamma(f : T) \vdash M']_{\gamma_1, \phi_1 \sim \gamma_2, \phi_2} = (\phi_1, \phi_2)$. Then, from 4.2,

$$F \models_{[\![U]\!](\gamma_1, \delta_1, \alpha_1 \sim \gamma_2, \delta_2, \alpha_2, \pi(\imath:=\mathfrak{a}+1))} \uparrow\phi_1(\delta_1, \alpha_1), \uparrow\phi_2(\delta_2, \alpha_2);$$

the result follows, since realizers are closed under wh-expansion.

$\pi(\imath) = \mathfrak{a}$**, for** $\mathfrak{a}$ **limit** The IH says that, if $\theta \models^{\Gamma}_{\pi(\imath:=\mathfrak{b})} (\gamma_1, \gamma_2)$ then

$$M^+\theta \models_{[\![T]\!](\gamma_1 \sim \gamma_2, \pi(\imath:=\mathfrak{b}))} (\phi_1, \phi_2)$$

for any $\mathfrak{b} < \mathfrak{a}$.

Let $\vec{t}\,u \models_{[\![\Delta^+]\!](\gamma_1 \sim \gamma_2, \pi(\imath:=\mathfrak{a}))} (\delta_1, \alpha_1), (\delta_2, \alpha_2)$. There exists $\mathfrak{b} < \mathfrak{a}$ such that $(\alpha_1, \alpha_2) \sqsubset [\![I^\imath\ (\vec{p})\ \vec{a}]\!]^{\pi(\imath:=\mathfrak{b})}_{\gamma_1, \delta_1 \sim \gamma_2, \delta_2}$. By IH,

$$\mathsf{app}^{|U|\theta}_{|\Delta^+|\theta}\left(M^+\theta, \vec{t}, u\right) \models_{[\![U]\!](\gamma_1, \delta_1, \alpha_1 \sim \gamma_2, \delta_2, \alpha_2, \pi(\imath:=\mathfrak{b}))} \uparrow\phi_1(\delta_1, \alpha_1), \uparrow\phi_2(\delta_2, \alpha_2)$$

From $\imath$ pos U, $[\![U]\!]^{\pi(\imath:=\mathfrak{b})}_{\gamma_1, \delta_1, \alpha_1 \sim \gamma_2, \delta_2, \alpha_2} \subseteq [\![U]\!]^{\pi(\imath:=\mathfrak{a})}_{\gamma_1, \delta_1, \alpha_1 \sim \gamma_2, \delta_2, \alpha_2}$. The result follows.

Now we prove the main result: if $\theta \models^{\Gamma}_{\pi} (\gamma_1, \gamma_2)$, then

$$M\theta \models_{[\![\Gamma \vdash U[\imath:=s]]\!](\gamma_1 \sim \gamma_2, \pi)} [\Gamma \vdash M]_{\gamma_1 \sim \gamma_2}$$

Let $\mathfrak{a} = (\!|s|\!)_\pi$. By IH, $\vec{N}\theta \models_{[\![\Gamma \vdash \Delta^+]\!](\gamma_1 \sim \gamma_2, \pi(\imath:=\mathfrak{a}))} [\Gamma \vdash \vec{N}]_{\gamma_1 \sim \gamma_2}$. Let $[\Gamma \vdash \vec{N}]_{\gamma_1 \sim \gamma_2} = ((\delta_1, \alpha_1), (\delta_2, \alpha_2))$. Following a similar reasoning than in the above, we consider two subcases. First, $\alpha_1 = \alpha_2 = \emptyset$. The result follows using a similar reasoning. Second, $\alpha_1 = (\tau_1, (j, \nu_1, \rho_1))$ and $\alpha_2 = (\tau_2, (j, \nu_2, \rho_2))$, with $o(\alpha_1, \alpha_2) = \mathfrak{b}{+}1$. Note that $\mathfrak{b}{+}1 \leq \mathfrak{a}$. Let $\vec{N}\theta = \vec{P}\, R$, with $\#\vec{P} = n - 1$. Then, $R \to^*_{\text{wh}} C_j(\vec{p^\circ}, \vec{a})$ and

$$M\theta \to^*_{\text{wh}} \mathsf{app}^{|U\theta|}_{|\Delta^+\theta|}\left(M'\theta\left[f := \lambda^{|U\theta|}_{|\Delta^+\theta|}.\mathsf{fix}_n\ f : T^\star\theta := (M'\theta, \mathrm{dom}\left(\Delta^+\right))\right], \vec{P}\left(C_j(\vec{p^\circ}, \vec{a})\right)\right) \equiv F$$

By the above result,

$$M'\theta\left[f := \lambda^{|U\theta|}_{|\Delta^+\theta|}.\mathsf{fix}_n\ f : T^\star\theta := (M'\theta, \mathrm{dom}\left(\Delta^+\right))\right] \models_{[\![\Gamma \vdash T]\!](\gamma_1 \sim \gamma_2, \pi(\imath:=\mathfrak{b}+1))} (\phi_1, \phi_2)$$

Then $F \models_{[\![U]\!](\gamma_1, \delta_1, \alpha_1 \sim \gamma_2, \delta_2, \alpha_2, \pi(\imath:=\mathfrak{b}+1))} (\uparrow\phi_1(\delta_1, \alpha_1), \uparrow\phi_2(\delta_2, \alpha_2))$. The result follows since $\imath\ \mathsf{pos}\ U$ and $\mathfrak{b} + 1 \leq \mathfrak{a}$.

□

Corollary 4.56. *If $\Gamma \vdash M : T$ then M is strongly normalizing.*

Proof. Note that the identity substitution, id, satisfies $\mathrm{id} \models^{\Gamma}_{\infty} \bot_{[\Gamma]}$, since variables are atomic terms. From the previous theorem, $M\varepsilon \equiv M \models_{[\![\Gamma \vdash T]\!](\gamma_1 \sim \gamma_2, \infty)} [\Gamma \vdash M]_{\bot_{[\Gamma]} \sim \bot_{[\Gamma]}}$. The result follows, since all realizers are strongly normalizing by definition. □

Logical consistency states that it is not possible to prove false propositions in the empty context. We prove it using SN and the following lemma on canonical forms.

Lemma 4.57 (Canonicity)**.** *Let M be a term in normal form such that $\vdash M : T$ for some type T.*

- *If $|T| \approx \Pi x : T_1{}^\circ.T_2{}^\circ$, then $M \equiv \lambda^{U_2{}^\circ}_{x:U_1{}^\circ}.M_1$, for some $U_1{}^\circ$, $U_2{}^\circ$.*
- *If $|T| \approx I\left(\vec{p^\circ}, \vec{a^\circ}\right)$, then $M \equiv C(\vec{q^\circ}, \vec{u})$, for some $\vec{q^\circ}$, $\vec{u}$.*

Proof. By induction on the type derivation and case analysis on the last rule used. The interesting cases are (app), (case), and (fix). We have to show that these cases contradict the hypothesis that M is in normal form.

If the last rule used is (app), then M is of the form $\mathsf{app}\,(M_1, M_2)$. The type of M_1 is a product; by IH, M_1 is an abstraction. Then M is a β_{I}-redex.

If the last rule used is (case), then M is of the form $\mathsf{case}\ x := M_1\ \ldots$, where M_1 is the argument. The type of M_1 is an inductive type; by IH, M_1 is in constructor form. Then M is a ι-redex.

Finally, if the last rule used is (fix), then M is of the form $\mathsf{fix}_n\ f : T^\star := (M_1, \vec{N})$. We proceed as in the previous case. By IH the n-th argument is in constructor form, then M is a μ-redex. □

Finally, LC is an easy consequence of SN and the previous lemma.

Theorem 4.58 (Logical Consistency)**.** *There is no term M such that $\vdash M : \mathsf{False}$.*

Proof. Assume there exits an M such that $\vdash M : \Pi X : \mathsf{Type}_0.X$. By strong normalization, we can assume that M is in normal form. By lemma 4.57, M is in constructor form. This is a contradiction, since False has no constructor. □

Chapter 5

Extensions

In this chapter, we discuss some extensions to CIC$\hat{}$. We consider universe inclusion, equality, and coinductive types. We sketch how to extend the Λ-set model of the previous chapter to cope with each extension.

5.1 Universe Inclusion

CIC$\hat{}$ has one impredicative universe Prop and a predicative universe hierarchy $\{\mathsf{Type}_i\}_i$, following ECC, CIC and Coq. The impredicative universe is at the lowest level of the hierarchy of universes, as witnessed by the typing rules

$$\mathsf{Prop} : \mathsf{Type}_0 : \mathsf{Type}_1 : \dots$$

In ECC, CIC, and Coq, this hierarchical view is further enhanced by the *universe inclusion* rules, expressed as a subtyping relation between universes:

$$(*)\ \mathsf{Prop} \leq \mathsf{Type}_0, \qquad (**)\ \mathsf{Type}_i \leq \mathsf{Type}_{i+1}, \forall i = 0, 1, \dots \ .$$

Then, using rule (**), any type in Type_0 can be seen as a type in Type_1, Type_2, and so on. In particular, any function of type $\mathsf{Type}_1 \to \mathsf{Type}_1$ can be applied to, for example, nat which has Type_0. Using rule (*), any proposition can be injected in the computational universes.

Universe inclusion is present in the conversion rule for products:

$$\frac{T_1 \approx T_2 \qquad U_1 \leq U_2}{\Pi x : T_1.T_2 \leq \Pi x : U_1.U_2}$$

Note that universe inclusion is allowed in the codomain (in a covariant way), while in the domain, only conversion is allowed. The reason is that contravariance is not set-theoretical (cf. Sect. 4.1); including contravariance in the rule would complicate the definition of set-theoretical models. Miquel [65] is able to allow contravariance for the Implicit Calculus of Constructions, but using a complex model based on domain theory.

In the case of the Λ-set model of Chap. 4, one crucial property we use is the separation of proofs and computations: by looking at a term we know if its type is in Prop or $\{\mathsf{Type}_k\}_k$. This property would still hold in the presence of rule (**), and the extension of the conversion rule for products. For example, Werner [84] uses this property in the definition of a proof-irrelevant version of ECC.

In the case of CIC$\hat{}_{-}$, we could add rule (**) without major technical problems in the model definition. The complications of adding this rule come from the use of type annotations in abstractions and applications, and the lack of type uniqueness. For example, the term $\lambda^{\mathsf{Type}_0}_{x:\mathsf{Type}_0}.x$ can be given type $\mathsf{Type}_0 \to \mathsf{Type}_1$, using the extended conversion rule for products. Type annotations would no longer reflect the type of a term, since types are not unique (although Minimal Types exists [56]). Then, the term $\mathsf{app}^{\mathsf{Type}_1}_{\mathsf{Type}_0}\left(\lambda^{\mathsf{Type}_0}_{x:\mathsf{Type}_0}.x, \mathsf{nat}\right)$ is well typed, but it cannot be reduced using tight β-reduction, since the type annotations do not match.

One way to adapt the model in order to avoid these problems is to change the syntax of ECIC$\hat{}_{-}$ to include two variants of abstractions and applications: one for proofs, and another for computations. The former would contain domain and codomain annotations (as they are needed to handle impredicativity; cf. Sect. 4.3.1), while the latter would not contain any extra type annotations. The separation property of proofs and computations would be important in the proof of equivalence between CIC$\hat{}_{-}$ and ECIC$\hat{}_{-}$.

We chose not to include rule (**) in CIC$\hat{}_{-}$ since the complications introduced in the metatheoretical study are mostly technical, and do not add significant value to the proofs.

On the other hand, we cannot easily adapt our Λ-set model to handle rule (*). In the presence of this rule, the separation property of proofs and computations is no longer valid.

To adapt our development to rule (*) we could follow the approach of Luo [56]. He defines a set theoretical model of ECC using a ω-set semantics, where proofs and computations are separated using the property of Minimal Types of ECC. He defines the semantics of a term by considering a canonical derivation that gives the minimal type of a term. For example, a term M with a computational type, let us say $\Gamma \vdash M : T : \mathsf{Type}_i$, can be given the semantics of a proof if the minimal type T' of M satisfies $\Gamma \vdash T' : \mathsf{Prop}$.

In a recent paper, Lee and Werner [51] define a set-theoretical model of CIC with rule (*). They rely on a coding for functions proposed by Aczel [5] and a judgmental equality to prove soundness. It is not clear if the approach can be extended to the Λ-set model.

We leave this line of research for future work.

5.2 Equality

In this section we consider an extension of CIC$\hat{}_{-}$ with equality. This extension is useful for programming with dependent types; it will be used in the metatheoretical study in next chapter. We explain the reasons behind the inclusion of this extension and describe it in detail.

For simplicity, in CIC$\hat{}_{-}$ we do not consider inductive types in Prop. Recall that this universe is used to represent logical properties of programs in the computational universes. Examples of logical propositions that are usually defined as inductive types in Prop include logical conjunction, existential, binary relations, and equality.

In some cases, it is possible to encode some logical properties directly in the language, without the need for inductive definitions. In these cases, inductive types are a convenience, but we can still do many developments without them.

These encodings of logical properties take advantage of the impredicativity of Prop, and are therefore called *impredicative encodings*. The idea is to encode a logical property as its own induction principle. For example, logical conjunction of two propositions P and Q can be encoded as

$$\mathsf{and}\, P\, Q \stackrel{\text{def}}{=} \Pi(R : \mathsf{Prop}).(P \to Q \to R) \to R$$

This is the induction principle given by Coq when logical conjunction is defined using the following inductive definition:

$$\mathsf{Ind}(\mathsf{and}(P\,Q : \mathsf{Prop}) := \mathsf{conj} : P \to Q \to \mathsf{and}\,P\,Q)$$

Logical inductive types in Coq have some restrictions compared with computational inductive types. To maintain the intuition that Prop is proof-irrelevant, case analysis on logical inductive types is not allowed to generate a term in a computational universe. Without this restriction, proof-irrelevance would be lost, since we would be able to define a computational object depending on how a particular proposition is proved.

However, there is a subclass of logical inductive types for which case analysis towards a computational universe is allowed. If a logical inductive type has only one canonical inhabitant, then elimination towards a computational universe is allowed. Since there is only one inhabitant, the proof-irrelevant meaning of Prop remains valid. One particular example where this is allowed is equality, defined as an inductive type as follows:

$$\mathsf{Ind}(\mathsf{eq}(A : \mathsf{Type})(x : A) : A \to \mathsf{Prop} := \mathsf{refl} : \mathsf{eq}\,A\,x\,x)$$

Note that the only canonical proof of a proposition $\mathsf{eq}\,A\,x\,y$ is refl.

The possibility of eliminating towards a computational universe implies that the definition using an inductive type and the definition using an impredicative encoding are not equivalent. While it is possible to prove that both encodings define an equivalence relation, the former comes associated with the following reduction rule:

$$\mathsf{eq\_rect}\,A\,x\,P\,p\,y\,(\mathsf{eq\_refl}\,A'\,x') \to p$$

where eq_rect is the induction principle associated with eq. (The typing rules ensure that A and A' are convertible, as well as x, y and x'.)

As we see in the next chapter, this computation rule is very important for programming with dependent types. Hence, it is necessary to consider a similar inductive definition of equality for CIC$\hat{\ }$. We can proceed by adding equality as an inductive type in the computational universes:

$$\mathsf{Ind}(\mathsf{eq}(A : \mathsf{Type}_0)(x : A) : A \to \mathsf{Type}_0 := \mathsf{refl} : \mathsf{eq}\,A\,x\,x)$$

Proceeding this way in Coq would mean that the extraction mechanism [53] would not remove equality, as it could be computationally relevant. Since we do not consider extraction of programs in this work, we allow ourselves to define equality in the computational universes. However, as we explain in the following, this notion of equality is not expressive enough to use for programming with dependent types

Homogeneous and Heterogeneous Equality. The equality defined above is called *homogeneous equality*, since it relates elements of the same type. A more general and powerful version of equality is *heterogeneous equality*, introduced by McBride [59]. In CIC$\hat{\ }$, it is defined by

$$\mathsf{Ind}(\mathsf{heq}[(A : \mathsf{Type}_0)(x : A)] : \Pi(B : \mathsf{Type}_0).\ B \to \mathsf{Type}_0 := \mathsf{heq\_refl} : \mathcal{X}\,A\,x)$$

The advantage of heterogeneous equality over homogeneous equality is that the former allows to easily express equalities between contexts. For example, given $(n_1 : \mathsf{nat})(v_1 : \mathsf{vec}(A, n_1))$ and $(n_2 : \mathsf{nat})(v_2 : \mathsf{vec}(A, n_2))$, we can write the following equalities:

$$\mathsf{heq}(\mathsf{nat}, n_1, \mathsf{nat}, n_2), \qquad \mathsf{heq}((\mathsf{vec}(A, n_1)), v_1, (\mathsf{vec}(A, n_2)), v_2)\ .$$

If we want to express $n_1 = n_2$ and $v_1 = v_2$ using homogeneous equality, we would need to rewrite the type of v_1 to match the type of v_2.

In general, given a context $\Delta \equiv (x_1 : T_1) \dots (x_n : T_n)$ and two sequences of terms $\langle M_i \rangle_i, \langle N_i \rangle_i : \Delta$, we write $\mathsf{heq}_\Delta(\langle M_i \rangle_i, \langle N_i \rangle_i)$ to mean the sequence of heterogeneous equalities:

$$\mathsf{heq}(T_1, M_1, T_1, N_1), \quad \mathsf{heq}((T_1\,[x_1 := M_1]), M_2, (T_1\,[x_1 := N_1]), N_2), \quad \dots$$
$$\mathsf{heq}((T_n\,[\langle x_i := M_i \rangle_{i=1..n-1}]), M_n, (T_n\,[\langle x_i := N_i \rangle_{i=1..n-1}]), N_n)$$

Although heterogeneous equality is useful to express equalities between terms of inductive families as shown above, the elimination rule obtained from the inductive definition is not very useful in practice. Namely, we can define the following elimination:

$$\mathsf{heq\_rect} : \Pi(A : \mathsf{Type})(x : A)(P : \Pi(B : \mathsf{Type}).B \to \mathsf{Type}).$$
$$P\,A\,x \to \Pi(B : \mathsf{Type})(b : B), \mathsf{heq}\,A\,x\,B\,b \to P\,B\,b$$

However, this elimination rule is difficult to use given the universal quantification over types in the predicate P. McBride [59] recognized the importance of heterogeneous equality when programming with inductive families, and introduced an elimination scheme for heq that works on homogeneous instances:

$$\mathsf{heq\_rect} : \Pi(A : \mathsf{Type})(P : A \to \mathsf{Type})(x : A).\ P\,x \to \Pi(y : A).\ \mathsf{heq}(A, x, A, y) \to P\,y$$

This means that we can perform a rewrite using an heterogeneous equality if the equality is actually homogeneous. For example, in a sequence of equalities $\mathsf{heq}_\Delta(\vec{M}, \vec{N})$, the first equality is homogeneous. Applying the elimination rule above we can transform the second equality in a homogeneous equality. And we can repeat the process to transform all equalities into homogeneous equalities.

McBride also proved that this elimination rule is equivalent to axiom K. That is, each of them can be derived assuming the other as an axiom. Since axiom K is not derivable in CIC [47], neither is the elimination rule given above.

We can introduce the elimination rule as an axiom. This is the approach used by some tools in Coq designed to program with dependent types, such as Program [77] and Equation [78]. However, it would lack a computational behavior similar to eq.

The desired computation rule for $\mathsf{heq\_rect}$, called κ-reduction, is given by the following rule:

$$\mathsf{heq\_rect}\,A\,P\,x\,p\,y\,(\mathsf{heq\_refl}\,A'\,x') \to_\kappa p$$

The case of CIC$\hat{\ }$. Similar to other developments (e.g. [42]) we consider an extension of our system with an axiom for $\mathsf{heq\_rect}$ and κ-reduction. We make use of this extension in the next chapter, without making a detailed metatheoretical study, which we leave for future work. Given the restrictions we make on size variables, our construction is slightly different, having some similarities with the case construction.

Concretely, we extend the syntax of CIC$\hat{\ }$ with the following constructions

$$\begin{aligned} \mathcal{T} \quad ::= \dots \;&|\; \mathsf{heq}(\mathcal{T}, \mathcal{T}, \mathcal{T}, \mathcal{T}) \\ &|\; \mathsf{heq\_refl}(\mathcal{T}^\circ, \mathcal{T}) \\ &|\; \mathsf{heq\_rect}_{\mathcal{T}^\circ}\ \mathcal{T}\ \mathsf{in}\ \mathsf{heq}_{\mathcal{T}^\circ}(\mathcal{T}^\circ, \mathcal{V})\ \mathsf{of}\ \mathcal{T} \end{aligned}$$

In a term of the form $(\mathsf{heq\_rect}_{|P|}\ H\ \mathsf{in}\ \mathsf{heq}_{|T|}(|N_1|, y)\ \mathsf{of}\ M)$, y is bound in P (similar to the case construction). H has type $\mathsf{heq}(T, N_1, T, N_2)$ for some term N_2 (a homogeneous equality), M has type $P\,[y := N_1]$, while the whole expression has type $P\,[y := N_2]$. Thus, this construction allows to rewrite on the type of M, using the equality H. Note the similarities with the case construction: H is the argument of a case analysis, and M corresponds to the only branch (since there is only one constructor in heq).

We also extend the reduction rule with the following κ-reduction:

$$\mathsf{heq\_rect}_{P^\circ}\ (\mathsf{heq\_refl}(T'^\circ, N'^\circ))\ \mathsf{in}\ \mathsf{heq}_{T^\circ}({N_1}^\circ, y)\ \mathsf{of}\ M \quad \to_\kappa \quad M$$

If the lhs is well typed, then T'° and T° are convertible, as well as ${N_1}^\circ$, ${N_2}^\circ$ and N'°. Thus, κ-reduction satisfies the Subject Reduction property.

The typing rules for heq and $\mathsf{heq\_refl}$ are the expected:

$$(\text{heq}) \quad \frac{\Gamma \vdash M_1 : T_1 : \mathsf{Type}_0 \qquad \Gamma \vdash M_2 : T_2 : \mathsf{Type}_0}{\Gamma \vdash \mathsf{heq}(T_1, M_1, T_2, M_2) : \mathsf{Type}_0} \quad \mathsf{simple}(T_1, T_2) \wedge \mathrm{SV}(M_1, M_2) = \emptyset$$

$$(\text{heq\_refl}) \quad \frac{\Gamma \vdash M : T : \mathsf{Type}_0}{\Gamma \vdash \mathsf{heq\_refl}(|T|, M) : \mathsf{heq}(T, M, T, M)} \quad \mathsf{simple}(T) \wedge \mathrm{SV}(M) = \emptyset$$

where $\Gamma \vdash M : T : \mathsf{Type}_0$ is an abbreviation for $\Gamma \vdash M : T$ and $\Gamma \vdash T : \mathsf{Type}_0$. Like a parameter in a constructor, the first argument of $\mathsf{heq\_refl}$ is a bare term. Note that both type arguments of heq can have size variables (respecting the simple predicate). Finally, the typing rule associated with $\mathsf{heq\_rect}$ is the following:

$$(\text{heq\_rect}) \quad \frac{\begin{array}{c}\Gamma \vdash H : \mathsf{heq}(T, N_1, T, N_2) \qquad \Gamma(y : T) \vdash P : \mathsf{Type}_i \\ \Gamma \vdash M : P\,[y := N_1]\end{array}}{\Gamma \vdash \mathsf{heq\_rect}_{|P|}\ H\ \mathsf{in}\ \mathsf{heq}_{|T|}(|N_1|, y)\ \mathsf{of}\ M : P\,[y := N_2]} \quad \mathrm{SV}(M) = \emptyset$$

We do not make a detailed study of the metatheory of this extension. We assume that the properties of Chapter 3 are preserved, as well as Strong Normalization. This is the same approach taken in [42] and we make these assumptions for the same reasons (cf. Chap. 6).

We briefly sketch how to adapt the Λ-set model to handle $\mathsf{heq\_rect}$ and κ-reduction. Note that the interpretation of the type heq (following the construction of Sect. 4.3.3) is the following:

$$[\Gamma \vdash \mathsf{heq}(T_1, M_1, T_2, M_2)]_\gamma = \begin{cases} (1, \emptyset) & \text{if } [\Gamma \vdash T_1]_\gamma = [\Gamma \vdash T_2]_\gamma \text{ and } [\Gamma \vdash M_1]_\gamma = [\Gamma \vdash M_2]_\gamma \\ \emptyset & \text{otherwise} \end{cases}$$

Then, $\mathsf{heq\_rect}$ can be interpreted as follows:

$$[\Gamma \vdash \mathsf{heq\_rect}_{|P|}\ H\ \mathsf{in}\ \mathsf{heq}_{|T|}(|N_1|, y)\ \mathsf{of}\ M]_\gamma = \begin{cases} [\Gamma \vdash M]_\gamma & \text{if } [\Gamma \vdash H]_\gamma = (1, \emptyset) \\ [\Gamma(x : T^\infty) \vdash P]_{\gamma, \bot_{[\Gamma \vdash T^\infty](\gamma)}} & \text{otherwise} \end{cases}$$

Note again the similarities with the interpretation of the case construction. The intepretation reduces to the interpretation of the branch if the argument of the case is interpreted as a constructor. Otherwise, the interpretation is just the atomic term of the corresponding type. Finally, note that κ-reduction is sound in this interpretation.

We leave for future work a formal treatment of this extension.

5.3 Coinductive Types

Coinductive types play an important rôle in the context of proof assistants. They are used to specify infinite systems, such as network protocols or servers. In the context of the Calculus of Constructions, they were introduced by Coquand [31] and later extended by Giménez [39].

As we mentioned in Chapter 1, we do not consider the general case of coinductive types, but focus on a particular case: *streams*. In this section, we define an extension of CIC$\hat{}$ with streams, following [39]. In particular, we describe the extensions to the syntax, reduction and typing rules. We also describe how to interpret streams in the model of Chap. 4. We begin by describing how streams are implemented in CIC and Coq, showing some of the theoretical and practical limitations of this approach.

Streams in CIC. Streams are introduced by a coinductive definition:

$$\mathsf{CoInductive\ stream}\,(A : \mathsf{Type}) : \mathsf{Type}\ = \mathsf{scons} : A \to \mathsf{stream}\,A \to \mathsf{stream}\,A$$

This defines stream as a coinductive type with only one constructor, scons. Note that all terms of type stream are infinite, since the only constructor is recursive. If we define stream as an inductive type it would be empty, since it is not possible to build finite streams.

Destructors of coinductive types are case analysis and corecursive functions. Case analysis for coinductive types is defined in the same way as for inductive types.

A corecursive function defining a stream has the form

$$\mathsf{cofix}\ f : \Pi\Delta.\mathsf{stream}(A) := M$$

Corecursive definitions are similar to recursive definitions in their form. However, while the latter kind receive an inductive type as argument, the former kind return a coinductive type.

Intuitively, corecursive definitions are evaluated using the reduction rule

$$\mathsf{cofix}\ f : \Pi\Delta.\mathsf{stream}(A) := M \quad \to \quad M\,[f := (\mathsf{cofix}\ f : \Pi\Delta.\mathsf{stream}(A) := M)]$$

Note that adding such a rule would immediately break Strong Normalization. To avoid this problem, cofixpoint unfolding is only allowed in the context of a case construction. The actual reduction rule used, called ν-reduction, is the following:

$$\begin{aligned}\mathsf{case}_{P^\circ}\ x := (\mathsf{cofix}\ f : T^\star := M)\,\vec{a}\ \mathsf{in\ stream}\,(U)\ \mathsf{of\ scons} \Rightarrow N \quad \to_\nu \\ \mathsf{case}_{P^\circ}\ x := M\,[f := (\mathsf{cofix}\ f : T^\star := M)]\ \vec{a}\ \mathsf{in\ stream}\,(U)\ \mathsf{of\ scons} \Rightarrow N\end{aligned}$$

Cofixpoint unfolding is only allowed when the cofixpoint is the argument of a case construction.

As defined above, ν-reduction is strongly normalizing for productive corecursive definitions. The intuition is that, after expanding the cofixpoint, evaluation of the body (M in the case above) would eventually reduce to a constructor, since the cofixpoint is productive. Then, the case construction can be reduced. Unfolding cannot occur infinitely, as each unfolding triggers the consumption of a case construction.

Let us see some examples of corecursive definitions. For readability, we omit the type argument of the scons constructor. The alternating stream of ones and zeros of Sect. 1.1 can be written in CIC as follows:

$$\mathsf{alt} \stackrel{\mathrm{def}}{=} \mathsf{cofix\ alt} : \mathsf{stream(nat)} := \mathsf{scons(O, scons(S\,O, alt))}$$

Besides some syntactical differences, this definition is the same as the Haskell definition of Sect. 1.1.

Some useful functions to work with streams are map and sum. The former is the natural equivalent for streams of the map function for lists, while the latter returns the component-wise addition of two streams of natural numbers. They are defined as follows (we omit some type annotations for readability):

$$\begin{aligned}
&\mathsf{map} : \Pi(A : \mathsf{Type})(B : \mathsf{Type}).(A \to B) \to \mathsf{stream}(A) \to \mathsf{stream}(B)\\
&\mathsf{map} \stackrel{\text{def}}{=} \lambda\, A\, B\, f.\mathsf{cofix}\ \mathsf{map} : \mathsf{stream}(A) \to \mathsf{stream}(B) :=\\
&\qquad \lambda\, t.\mathsf{case}\ t\ \mathsf{of}\ \mathsf{scons} \Rightarrow \lambda\, a\, t'.\mathsf{scons}(f(a), \mathsf{map}\, t')\\
&\mathsf{sum} : \mathsf{stream}(\mathsf{nat}) \to \mathsf{stream}(\mathsf{nat}) \to \mathsf{stream}(\mathsf{nat})\\
&\mathsf{sum} \stackrel{\text{def}}{=} \mathsf{cofix}\ \mathsf{sum} : \mathsf{stream}(\mathsf{nat}) \to \mathsf{stream}(\mathsf{nat}) \to \mathsf{stream}(\mathsf{nat}) :=\\
&\qquad \lambda\, t_1\, t_2.\ \mathsf{case}\ t_1\ \mathsf{of}\ \mathsf{scons} \Rightarrow \lambda\, a_1\, t_1'.\\
&\qquad\qquad \mathsf{case}\ t_2\ \mathsf{of}\ \mathsf{scons} \Rightarrow \lambda\, a_2\, t_2'.\ \mathsf{scons}(a_1 + a_2, \mathsf{sum}\, t_1'\, t_2')
\end{aligned}$$

Both definitions are productive, since the corecursive calls are performed under a constructor.

Finally, let us see the typing rule associated with cofixpoints. Intuitively, cofixpoints can be typed with the following rule:

$$\frac{\Gamma(f : \Pi\Delta.\mathsf{stream}(A)) \vdash M : \Pi\Delta.\mathsf{stream}(A)}{\Gamma \vdash (\mathsf{cofix}\ f : \Pi\Delta.\mathsf{stream}(A) := M) : \Pi\Delta.\mathsf{stream}(A)}$$

However, such a liberal rule would allow non-productive cofixpoints.

Like non-termination, non-productivity could introduce inconsistencies. Therefore, in CIC and Coq, some restrictions are imposed to ensure productivity of corecursive definitions. These restrictions take the form of a guard predicate, similar to the guard condition of recursive definitions (cf. Sect. 1.2.1). We do not explain the guard condition in detail but only give some intuition.

Basically, a corecursive definition satisfies the guard condition if all corecursive calls are performed under a constructor. This ensures that when evaluating a corecursive definition lazily, at least one constructor will be exposed before a corecursive call needs to be reduced (such a call would stop lazy reduction). As we explained above, this ensures that the reduction is strongly normalizing, and that LC is satisfied. All the examples given above satisfy the guard condition.

However, the guard condition for corecursive definitions has some limitations. For example, the following definition is productive (cf. Sect. 1.1), but does not satisfy the guard condition:

$$\mathsf{cofix}\ \mathsf{nats} : \mathsf{stream}(\mathsf{nat}) := \mathsf{scons}(\mathsf{O}, \mathsf{map}\, (\lambda\, x.\ \mathsf{S}(x))\, \mathsf{nats})$$

The guard condition is not satisfied as the corecursive call is an argument of a function. The following definition of the Fibonacci sequence is also productive but not guarded:

$$\mathsf{cofix}\ \mathsf{fib} : \mathsf{stream}(\mathsf{nat}) := \mathsf{scons}(\mathsf{O}, \mathsf{sum}\, \mathsf{fib}\, (\mathsf{scons}(\mathsf{S}\,\mathsf{O}, \mathsf{fib}))$$

It is not difficult to see that this definition produces the Fibonacci sequence: 0,1,1,2,3,5,.... The definition is productive, by a similar reasoning as in the case of nats above. However, it does not satisfy the guard condition, as corecursive call are performed as argument of a function (sum in this case).

Our extension of CIC$\hat{}_{\sqsubset}$ with streams that overcomes some of the limitations of the guard condition. Productivity is checked using size annotations. This mechanism is more powerful than guard predicates due to the possibility of expressing size-preserving corecursive functions.

Metatheory of CIC with streams. The metatheoretical study of coinductive types in CIC presents some problems that are not found in the case of inductive types. Gimémez obverved that, in the presence of dependent types, ν-reduction does not satisfy SR [39]. A simple example given in op.cit. illustrates the situation. Consider the context

$$\Gamma_0 \stackrel{\text{def}}{=} (A : \mathsf{Type})(P : \mathsf{stream}^{\infty}(A) \to \mathsf{Type}) \\ (H : \Pi(a : A)(t : \mathsf{stream}^{\infty}(A)).P\,(\mathsf{scons}(A, a, t)))(a : A)$$

and the terms

$$R \equiv \mathsf{cofix}\ f : \mathsf{stream}^{\star}(A) := \mathsf{scons}(A, a, f)$$
$$M \equiv\ \mathsf{case}_{P\,x}\ x := R\ \mathsf{in}\ \mathsf{stream}(A)\ \mathsf{of}\ \mathsf{scons} \Rightarrow H$$

Then $\Gamma_0 \vdash M : P\,M$. Note that M admits the following reduction:

$$M \to \mathsf{case}_{P\,x}\ x := \mathsf{scons}(A, a, R)\ \mathsf{in}\ \mathsf{stream}(A)\ \mathsf{of}\ \mathsf{scons} \Rightarrow H \equiv N$$

We have $\Gamma_0 \vdash N : P(\mathsf{scons}(A, a, R))$, but $\Gamma_0 \not\vdash N : P\,M$. The problem is that $P\,M$ does not reduce to $P(\mathsf{scons}(A, a, R))$, since the cofixpoint does not occur as argument of a case construction.

Giménez considers an extended system where cofixpoint unfolding is not restricted in the conversion rule. The extended system satisfies SR. Since cofixpoint unfolding is not restricted, the reduction used in the conversion rule is not strongly normalizing. Hence, conversion is not decidable. However, Gimémez proves that the extended system satisfy SN for ν-reduction.

Therefore, one is faced with the decision of using a system where Subject Reduction is satisfied, or a system where SN is satisfied. The latter is a much more important property, as it ensures LC. In practice, the lack of SR is not important in practice. The current implementation of Coq is based on the work of Giménez and, therefore, does not satisfy SR. However, this is rarely a problem in practice.

A more satisfying definition of coinduction is an important topic of current research in Type Theory.

Streams in CIC$\hat{}_{\sqsubset}$. Coinductive types in CIC$\hat{}_{\sqsubset}$ are also decorated with stages, in the same way as inductive types. However, the semantic meaning of these size annotations is different. In the case of streams, a type of the form

$$\mathsf{stream}^{s}\,(\mathcal{T})$$

denotes the type of streams where *at least* s elements can be computed. That is, evaluation of a term of this type is guaranteed to produce at least s elements of the stream.

Dually to the case of inductive types, size annotations are contravariant in coinductive types. Thus, the subtyping rule associated with streams is the following:

$$\text{(st-stream)}\ \frac{s \sqsubseteq r \qquad T \leq U}{\mathsf{stream}^{r}\,(T) \leq \mathsf{stream}^{s}\,(U)}$$

The intuition is that if we can produce r elements of a stream, then we can produce any smaller number of elements. Similarly, this duality is shown also in the positivity condition for size variables in streams:

$$\frac{\imath \notin \mathrm{SV}(T)}{\imath \ \mathsf{neg}\ \mathsf{stream}^s(T)}$$

Then, for example, $\imath\ \mathsf{neg}\ \mathsf{stream}^\imath(T)$.

The syntax of CIC$^{\widehat{\ }}_{-}$ is extended with streams and cofixpoints as expected:

$$\mathcal{T} ::= \ldots \mid \mathsf{stream}^a\,(\mathcal{T}) \mid \mathsf{scons}(\mathcal{T}^\circ, \mathcal{T}, \mathcal{T}) \mid \mathsf{cofix}\ f : \mathcal{T}^\star := \mathcal{T}$$

Case analysis of streams is done in the same way as for inductive definitions; there is no need of a separate construction. Reduction of cofixpoints is defined as in the case of CIC.

The typing rules are extended for the new type and introduction rule (constructor scons) as expected:

$$(\text{stream})\ \frac{\Gamma \vdash T : \mathsf{Type}_0}{\Gamma \vdash \mathsf{stream}^s\,(T) : \mathsf{Type}_0} \qquad (\text{scons})\ \frac{\Gamma \vdash M : T \qquad \Gamma \vdash N : \mathsf{stream}^s\,(T)}{\Gamma \vdash \mathsf{scons}(|T|, M, N) : \mathsf{stream}^{\widehat{s}}\,(T)}$$

The typing rule for scons reflects the intuitive meaning of the size annotations on streams: if we can compute s elements of a stream N, then we can compute one more element of the stream $\mathsf{scons}(M, N)$.

The destructors associated with streams are case analysis and corecursive definitions. The typing rule for case analysis is the same as if streams where defined by an inductive definition:

$$\frac{\begin{array}{c}\Gamma \vdash M : \mathsf{stream}^{\widehat{s}}\,(T) \qquad \Gamma(x : \mathsf{stream}^{\widehat{s}}\,(T)) \vdash P : u \\ \Gamma \vdash N : \Pi(y_1 : T)(y_2 : \mathsf{stream}^s\,(T)).P\,[x := \mathsf{scons}(|T|, y_1, y_2)]\end{array}}{\Gamma \vdash \mathsf{case}_{|P|}\ x := M\ \mathsf{in}\ \mathsf{stream}\left(|\vec{T}|\right)\ \mathsf{of}\ \mathsf{scons} \Rightarrow N : P\,[x := M]}$$

Note that, as in the case of inductive definitions, the argument of the case (M above) must have a successor stage in the annotation of its type. In the case of inductive types, any term of an inductive type I can be used as argument of a case construction. For example, a term of type $I^\imath$ can be given the type $I^{\widehat{\imath}}$ by the subtyping rule and, therefore, it can be used as argument of a case construction. However, with streams this is not possible. A term of type $\mathsf{stream}^\imath$ cannot be used as argument of a case construction, since size annotations are contravariant for coinductive types. We show some examples below where this situation appears as a limitation.

Finally, the typing rule for corecursive definitions

$$(\text{cofix})\ \frac{\begin{array}{c}T \equiv \Pi\Delta.\mathsf{stream}^\imath\,(U) \\ \imath\ \mathsf{neg}\ \Delta \qquad \imath \notin \mathrm{SV}(\Gamma, U, M) \qquad \Gamma \vdash T : u \qquad \Gamma(f : T) \vdash M : T\,[\imath := \widehat{\imath}]\end{array}}{\Gamma \vdash \mathsf{cofix}\ f : T^\star := M : T\,[\imath := s]}$$

The typing rule is similar to that of fixpoints, but in this case a coinductive type is returned. The typing rule for the body says that M returns a stream where at least $\widehat{\imath}$ elements can be computed, assuming that f return a stream where at least $\imath$ elements can be computed. This condition is enough to ensure that the cofixpoint is productive.

Note that the fresh variable $\imath$ can appear negatively in Δ. Examples of valid types for corecursive functions are $\mathsf{nat}^\infty \to \mathsf{stream}^\imath(A)$, $\mathsf{stream}^\imath(\mathsf{nat}^\infty) \to \mathsf{stream}^\imath(\mathsf{nat}^\infty) \to \mathsf{stream}^\imath(\mathsf{nat}^\infty)$.

The examples given above, namely alt, map and sum can be typed using the above rule. Furthermore, map and sum can be given the following precise types:

$$\mathsf{map} : \Pi(A : \mathsf{Type})(B : \mathsf{Type}).(A \to B) \to \mathsf{stream}^s(A) \to \mathsf{stream}^s(B)$$
$$\mathsf{sum} : \mathsf{stream}^\imath(\mathsf{nat}^\infty) \to \mathsf{stream}^\imath(\mathsf{nat}^\infty) \to \mathsf{stream}^\imath(\mathsf{nat}^\infty)$$

These precise types allows us to accept the definitions of nats and fib given above that are not accepted under guard predicates.

However, this approach also has some limitations. For example, the following definition of the Fibonacci sequence (in Haskell) is productive:

```
fib = 0 : 1 : sum fib (tail fib)
```

In CIC$\hat{}_\sqsubset$ we can define the function tail that removes the first element of a stream, with type $\mathsf{stream}^{\widehat{s}}(A) \to \mathsf{stream}^s(A)$.

However, the direct translation of the above definition to CIC$\hat{}_\sqsubset$ is not accepted, since tail cannot be applied to the recursive call of fib whose size annotation in the type is not a successor stage. Nevertheless, we believe that the approach using sized types provides a more flexible and intuitive framework for checking productivity than guard predicates.

MiniAgda [3] includes coinductive definitions. A main difference with our approach is that MiniAgda includes a feature that allows to perform case analysis on term of type $\mathsf{stream}^\imath(A)$. This allows, for example, to apply the tail function to any stream. In turn, this allows to accept the second definition of the Fibonacci sequence given in this section, which is not accepted in CIC$\hat{}_\sqsubset$.

We leave for future work the development of a similar feature for CIC$\hat{}_\sqsubset$.

Metatheory. The basic metatheory of this extension can be developed in the same way as in [39]. However, Subject Reduction is not valid for ν-reduction.

Gimenéz considers a variant of the system with a modified conversion rule where cofixpoint unfolding is not restricted. This variant includes the original system, since the conversion rule is more liberal. The variant satisfies both SR and SN, and therefore the original system also satisfies SN. We do not go into details of the basic metatheory, and just assume that the development of [39] can be adapted to our case.

We focus instead on the extension of the model of Chap. 4. In particular, we show that the relational interpretation can be naturally extended to streams.

Streams are interpreted as set-theoretical infinite sequences. Concretely, the interpretation of the type $\mathsf{stream}(T)$ is a Λ-set $(X, \models, \bot)$ given by

- $X \stackrel{\text{def}}{=} \{(\alpha_1, \alpha_2, \ldots) : \alpha_i \sqsubset [\Gamma \vdash T]_\gamma\}$
- $M \models \langle \alpha_i \rangle_{i=0\ldots}$, for $M \in \mathrm{SN}$, iff $M \to^*_{\mathrm{wh}} \mathsf{scons}(U^\circ, N, P)$, $N \models_{[\Gamma \vdash T](\gamma)} \alpha_0$, and $P \models \langle \alpha_i \rangle_{i=1\ldots}$
- $\bot \stackrel{\text{def}}{=} (\bot_{[\Gamma \vdash T^\infty](\gamma)}, \bot_{[\Gamma \vdash T^\infty](\gamma)}, \ldots)$

This Λ-set is the *greatest fixed point* of the monotone operator defined in Sect. 4.3.3 applied to the definition of streams as an inductive type.

As in the case of recursive definitions, a cofixpoint is interpreted using Hilbert's choice operator:

$$[\Gamma \vdash \mathsf{cofix}\ f : T^\star := M]_\gamma = \epsilon(F, P)$$

where $T^\star = \Pi\Delta^\star.\mathsf{stream}^\star(T)$, $F \sqsubset [\Gamma \vdash T^\infty]_\gamma$, P stands for the property

$$\uparrow(F)(\delta) = \uparrow([\Gamma \vdash M]_{\gamma,F})(\delta),$$

for all $\delta \sqsubset [\Gamma \vdash \Delta^\infty]_\gamma$, and $\uparrow$ stands for $\uparrow^u_{\Pi([\Delta](\gamma),[\mathsf{stream}(T)](\gamma,\_))}$.

The invariance of this interpretation under ν-reductions is trivial. In the main soundness theorem, we prove that the typing rules ensure the existence of a unique function satisfying the cofixpoint equation of property P.

The relational interpretation can be naturally extended to streams. Basically, the type $\mathsf{stream}^s(T)$ is interpreted as a Λ-set whose carrier set is composed by pair of streams such that the first s elements are related in the interpretation of T. Concretely, $[\![\Gamma \vdash \mathsf{stream}^s(T)]\!]^\pi_{\gamma_1\sim\gamma_2}$ is a Λ-set $(X, \models, \bot)$, where

- $X \stackrel{\text{def}}{=} \{(\langle\alpha_i\rangle_i, \langle\beta_i\rangle_i) : \forall i = 0, \ldots, (\!|s|\!)_\pi.\ \alpha_i \sim \beta_i \sqsubset [\![\Gamma \vdash T]\!]^\pi_{\gamma_1\sim\gamma_2}\}$;
- $M \models (\langle\alpha_i\rangle_{i=0\ldots}, \langle\beta_i\rangle_{i=1\ldots})$, for $M \in \mathrm{SN}$, iff $M \to^*_{\mathrm{wh}} \mathsf{scons}(U^\circ, N, P)$, $N \models_{[\![\Gamma\vdash T]\!](\gamma_1\sim\gamma_2,\pi)} (\alpha_0, \beta_0)$, and $P \models (\langle\alpha_i\rangle_{i=1\ldots}, \langle\beta_i\rangle_{i=1\ldots})$;
- $\bot = ((\bot_1, \bot_1, \ldots), (\bot_2, \bot_2, \ldots))$ where $(\bot_1, \bot_2) = \bot_{[\![\Gamma\vdash T]\!](\gamma_1\sim\gamma_2,\pi)}$.

Note that the subtyping rule is sound in this interpretation: if $r \sqsubseteq s$ and $T \leq U$, then $[\![\Gamma \vdash \mathsf{stream}^s(T)]\!]^\pi_{\gamma_1\sim\gamma_2} \subseteq [\![\Gamma \vdash \mathsf{stream}^r(U)]\!]^\pi_{\gamma_1\sim\gamma_2}$.

The construction at the beginning of Sect. 4.4 can be adapted to streams to show the existence of a unique function satisfying the cofixpoint equation. Consider the term $\mathsf{cofix}\ f : |T|^\imath := M$, such that $\Gamma \vdash \mathsf{cofix}\ f : |T|^\imath := M : T$ is a valid judgment. Assume that $\gamma \sim \gamma \sqsubset [\![\Gamma]\!]^\pi$ and that if $\phi_1 \sim \phi_2 \sqsubset [\![\Gamma \vdash T]\!]^\pi_{\gamma\sim\gamma}$, then $[\Gamma(f : T) \vdash M]_{\gamma,\phi_1\sim\gamma,\phi_2} \sqsubset [\![\Gamma \vdash T\,[\imath := \widehat{i}]]\!]^\pi_{\gamma\sim\gamma}$. Under these assumptions, we can show the existence of a unique stream in $[\Gamma \vdash T^\infty]_\gamma$ satisfying the cofixpoint equation.

Following the development of Sect. 4.4, we define a sequence of sets $\Phi^{\mathfrak{a}}$, for $\mathfrak{a} \leq \omega$, such that $\Phi^{\mathfrak{a}} \subseteq [\Gamma \vdash T^\infty]_\gamma$. We proceed by transfinite induction. For $\mathfrak{a} = 0$, we simply define $\Phi^0 \stackrel{\text{def}}{=} [\Gamma \vdash T^\infty]_\gamma$. For a successor ordinal $\mathfrak{a}+1$, we define $\Phi^{\mathfrak{a}+1} \stackrel{\text{def}}{=} \{[\Gamma(f : T) \vdash M]_{\gamma,\phi} : \phi \in \Phi^{\mathfrak{a}}\}$. Finally, $\Phi^\omega \stackrel{\text{def}}{=} \bigcap_{\mathfrak{a}<\omega} \Phi^{\mathfrak{a}}$.

Intuitively, all the streams in $\Phi^{\mathfrak{a}}$ coincide in the first $\mathfrak{a}$ elements. Then, Φ^ω is a singleton containing the desired stream.

Chapter 6

A New Elimination Rule

In this chapter, we present a modification of the case analysis construction of Chapter 2. The new rule automatically discards impossible cases and propagates inversion constraints. We prove that the new rule preserves desired metatheoretical properties such as SR and LC relative to the extension with equality of Sect. 5.2.

6.1 Introduction

As we mentioned in the first chapters, inductive families can be used to define more precise data structures. The typical example is vectors, i.e., list indexed by their length. Let us recall the definition:

$$\begin{array}{rl}\mathsf{Ind}(\mathsf{vec}[A:\mathsf{Type}_0]^{\oplus}:\mathsf{nat}\to\mathsf{Type}_0:=& \mathsf{vnil}:\mathcal{X}\,\mathsf{O},\\ & \mathsf{vcons}:\Pi(n:\mathsf{nat}).A\to\mathcal{X}\,n\to\mathcal{X}\,(\mathsf{S}\,n))\end{array}$$

The usual tail function, that removes the first element of a non-empty vector can be given the type

$$\Pi(A:\mathsf{Type}_0)(n:\mathsf{nat}).\mathsf{vec}\,(A,\mathsf{S}\,n)\to\mathsf{vec}\,(A,n)\,,$$

thus ensuring that it cannot be applied to an empty vector. To write this function in CIC it is necessary to explicitly reason on the index in order to remove the impossible case, vnil, and propagate the inversion constraint in the case vcons:

$$\begin{array}{l}\mathsf{vtail}\stackrel{\mathrm{def}}{=}\lambda\,A\,n\,(v:\mathsf{vec}(A,\mathsf{S}\,n)).\\ \quad(\mathsf{case}_{\,n_0=\mathsf{S}\,n\to\mathsf{vec}(A,n)}\;v_0:=v\;\mathsf{in}\;\mathsf{vec}(A,n_0)\;\mathsf{of}\\ \quad\mid\mathsf{vnil}\Rightarrow\lambda(H:\mathsf{O}=\mathsf{S}\,n).\\ \qquad\mathsf{case}_{\,\mathsf{False}}\;x:=M_0\,H\;\mathsf{of}\\ \quad\mid\mathsf{vcons}\,n'\,a\,v'\Rightarrow\lambda(H:\mathsf{S}\,n'=\mathsf{S}\,n).\\ \qquad\mathsf{case}_{\,\mathsf{vec}(A,n_1)}\;x:=M_1\,H\;\mathsf{in}\;\mathsf{eq}(\mathsf{nat},n',n)\;\mathsf{of}\\ \qquad\mid\mathsf{refl}\Rightarrow v')\;(\mathsf{refl}(A,\mathsf{S}\,n))\end{array}$$

where $\mathsf{M_0}:\mathsf{O}=\mathsf{S}\,n\to\mathsf{False}$, and $\mathsf{M_1}:(\mathsf{S}\,n'=\mathsf{S}\,n)\to n'=n$.

Following the approach of Epigram [60] and Agda [69], we propose a different case analysis construction that automatically performs the index reasoning of the above example. The intuition behind the new rule is that it is possible to restrict case analysis on inductive families to a particular subfamily [62]. Constructors that do not belong to the subfamily

can be ignored, while constructors that partly belong to the family can be imposed further constraints (inversion constraints).

Let us illustrate with the example of vtail. The vector argument, v, has type $\mathsf{vec}\,(A, \mathsf{S}\,n)$. In the usual case analysis construction, we assume that v can be any vector, of type $\mathsf{vec}\,(A, n_0)$. In particular, we need to consider the case $v = \mathsf{vnil}$, even when it is not possible to for vnil to have type $\mathsf{vec}\,(A, \mathsf{S}\,n)$.

Our solution is to allow case analysis in a type subfamily of vectors. In this case we can take the subfamily

$$[n_0 : \mathsf{nat}]\ \mathsf{vec}\,(A, \mathsf{S}\,n_0)$$

I.e., the subfamily of types $\mathsf{vec}\,(A, \mathsf{S}\,n_0)$, for $n_0 : \mathsf{nat}$. Note that $v : \mathsf{vec}\,(A, \mathsf{S}\,n)$ belongs to this subfamily, with the constraint $n_0 = n$. Furthermore, it is clear that $\mathsf{vnil} : \mathsf{vec}\,(A, \mathsf{O})$ does not belong to this subfamily, while $\mathsf{vcons}(A, m, x, v) : \mathsf{vec}\,(A, \mathsf{S}\,m)$ belongs, with the constraint $n_0 = m$.

Before introducing the full version of the new case analysis construction, we exemplify it using vtail. Omitting the initial abstractions, we can define vtail as:

$$\begin{array}{l}\mathsf{case}\ v\ \mathsf{in}\ [n_0 : \mathsf{nat}]\,\mathsf{vec}(A, \mathsf{S}\,n_0)\ \mathsf{where}\ n_0 := n\ \mathsf{of}\\ |\ \mathsf{vnil} \Rightarrow \bot\\ |\ \mathsf{vcons}\,m\,x\,v' \Rightarrow v'\ \mathsf{where}\ n_0 := m\end{array}$$

We define the subfamily of vec that we analyze: $[n_0 : \mathsf{nat}]\,\mathsf{vec}\,(A, \mathsf{S}\,n_0)$. The clause $\mathsf{where}\ n_0 := n$ is a constraint on the index n_0 that shows that v is in the subfamily.

Let us look at the branches. In the vnil case, it is clear that this constructor does not belong to the subfamily $[n_0 : \mathsf{nat}]\,\mathsf{vec}\,(A, \mathsf{S}\,n_0)$, since its type is $\mathsf{vec}\,(A, \mathsf{O})$. Hence, the branch is simply $\bot$. We call this type of branch *impossible*. To check that a branch is impossible, we try to *unify* the indices of the subfamily under consideration ($\mathsf{S}\,n_0$ in this case) with the indices of the constructor (O in this case), for the variable n_0. That is, we try to find a term M such that $\mathsf{S}\,M \approx \mathsf{O}$. No such term exists, since constructors are disjoint. Hence, $\mathsf{S}\,n_0$ and O are not unifiable, and therefore, the branch is effectively impossible.

In the vcons case, the term $\mathsf{vcons}\,m\,x\,v'$ belongs to the subfamily $[n_0 : \mathsf{nat}]\,\mathsf{vec}\,(A, \mathsf{S}\,n_0)$ when the constraint $n_0 = m$ is satisfied. To check that constraints are valid we proceed as for impossible branches: we unify the indices of the subfamily with the indices of the constructor. In this case, we unify $\mathsf{S}\,n_0$ with $\mathsf{S}\,m$. Since constructors are injective, a unification substitution exists, namely $n_0 := m$, that validates the constraint.

Another way to define vtail is to perform case analysis on a more precise subfamily, namely

$$[]\ \mathsf{vec}\,(A, \mathsf{S}\,n)$$

That is, since the vector argument has type $\mathsf{vec}\,(A, \mathsf{S}\,n)$ we can consider just this type as a subfamily. The definition of vtail is the following:

$$\begin{array}{l}\mathsf{case}\ v\ \mathsf{in}\ []\,\mathsf{vec}(A, \mathsf{S}\,n)\ \mathsf{of}\\ |\ \mathsf{vnil} \Rightarrow \bot\\ |\ \mathsf{vcons}\,m\,x\,v' \Rightarrow v'\ \mathsf{where}\ m := n\end{array}$$

In this case, the branch vnil is still impossible, while for vcons the constraint $m := n$ is satisfied if v is of the form $\mathsf{vcons}\,m\,x\,v'$.

The rest of the chapter is organized as follows. In Sect. 6.2, we present the syntax of the new elimination rule and the reduction associated. We call the system in this chapter $\mathrm{CIC}^{\widehat{\ }}_{\mathrm{PM}}$.

The typing rule is presented in Sect. 6.3. In Sect. 6.4, we present some examples of the use of the new elimination rule. In Sect. 6.5, we prove that $\mathrm{CIC}^{\widehat{}}_{\mathrm{PM}}$ satisfies several desirable metatheoretical properties such as Subject Reduction. Finally, in Sect. 6.6, we sketch a proof of Strong Normalization by a type-preserving translation to $\mathrm{CIC}^{\widehat{}}_{-}$.

6.2 Syntax

In this section we introduce the syntax of $\mathrm{CIC}^{\widehat{}}_{\mathrm{PM}}$. The constructions of $\mathrm{CIC}^{\widehat{}}_{\mathrm{PM}}$ are the same as in $\mathrm{CIC}^{\widehat{}}_{-}$, except for the case analysis construction, and the addition of local definitions. We only present the differences of $\mathrm{CIC}^{\widehat{}}_{\mathrm{PM}}$ with respect to $\mathrm{CIC}^{\widehat{}}_{-}$.

Definition 6.1 (Terms)**.** *The generic set of terms over the set a is defined by the grammar:*

$$\begin{aligned}
\mathcal{T}[a] ::= {} & \ldots \\
& \mid \mathsf{case}_{\mathcal{T}^\circ}\ \mathcal{V} := \mathcal{T}[a]\ \mathsf{in}\ [\mathcal{G}[a]]\ \mathcal{I}\left(\vec{\mathcal{T}^\circ}, \vec{\mathcal{T}^\circ}\right)\ \mathsf{where}\ \vec{\mathcal{V}} := \mathcal{T}^\circ\ \mathsf{of}\ \langle \mathcal{C} \Rightarrow \mathcal{B}[a] \rangle && \textit{(case)} \\
& \mid \mathsf{let}\ \mathcal{V} := \mathcal{T}[a] : \mathcal{T}^\circ\ \mathsf{in}\ \mathcal{T}[a] && \textit{(definition)}
\end{aligned}$$

$$\mathcal{B}[a] ::= \bot \mid \mathcal{T}[a]\ \mathsf{where}\ \mathcal{D}[a]$$

$$\mathcal{D}[a] ::= \epsilon \mid \mathcal{D}(\mathcal{V} := \mathcal{T}[a])$$

Local definitions are needed to handle constraints in the branches of the case construction. Let us explain in detail the new case construction. A case expression has the form

$$\mathsf{case}_{P^\circ}\ x := M\ \mathsf{in}\ [\Delta]\ I\left(\vec{p^\circ}, \vec{t^\circ}\right)\ \mathsf{where}\ \mathrm{dom}\,(\Delta) := \vec{q^\circ}\ \mathsf{of}\ \langle C_i\, \vec{x_i} \Rightarrow b_i \rangle_i,$$

where the body of a branch (b_i above) can be either the symbol $\bot$ or a term of the form $N\ \mathsf{where}\ \sigma$, with σ being a sequence of variable definitions, i.e., a substitution. We refer to this substitution as the *constraints* of the branch.

The rôle of $[\Delta]\ I\left(\vec{p^\circ}, \vec{t^\circ}\right)$ is to characterize the subfamily of I, with parameters in Δ, over which the pattern matching is done. Some constructors may not belong to that subfamily, so the body of the corresponding branches is simply $\bot$ (impossible branches). On the other hand, some constructors may (partially) belong to the subfamily, so the bodies of the corresponding branches are of the form $N\ \mathsf{where}\ \vec{d}$, where N is the body proper and $\vec{d}$ defines some constraints on the arguments of the constructor that need to be satisfied in order to belong to the subfamily.

The case construction of $\mathrm{CIC}^{\widehat{}}_{-}$ is a special case of the new construction: we just set Δ to be the context of indices of the inductive type, i.e. Δ_a, and $\vec{t}$ to be $\mathrm{dom}\,(\Delta)$. As we see from the typing rules of next section, in this case all branches are possible.

The definition of contexts is extended to include local definitions.

Definition 6.2 (Contexts)**.** *The set of contexts over a set a is defined by the grammar:*

$$\mathcal{G}[a] ::= [] \mid \mathcal{G}(\mathcal{V} : \mathcal{T}[a]) \mid \mathcal{G}(\mathcal{V} := \mathcal{T}[a] : \mathcal{T}[a])$$

We adapt the subcontext relation $\leq$ on contexts to the case of definitions. Given two context Γ and Δ, $\Delta \leq \Gamma$ if the two following conditions hold:

- for every $(x : T) \in \Gamma$, either $(x : T') \in \Delta$ for some $T' \leq T$, or $(x := M : T') \in \Delta$ for some M and $T' \leq T$;
- for every $(x := M : T) \in \Gamma$, $(x := M : T') \in \Delta$, for some $T' \leq T$.

Reduction

The reduction rules for $\mathrm{CIC}^{\widehat{}}_{\mathrm{PM}}$ are similar to the reduction rules for $\mathrm{CIC}^{\widehat{}}_{-}$ (Def. 2.6). Since we consider local definitions, the reduction relation is parameterized by a context, having the form

$$\Gamma \vdash M \to N \ .$$

The reduction rules for applications (β) and fixpoints (μ) remain the same but we include them in the definition below for completeness.

Definition 6.3 (Reduction)**.** *Reduction $\to$ is defined as the compatible closure of β-reduction (for function application), ι-reduction (for case expression), μ-reduction (for fixpoint expressions), and δ-reduction (for local definitions):*

$$\begin{array}{rcl}
\Gamma \vdash (\lambda x : T^{\circ}.M)N & \beta & M\,[x := N] \\
\Gamma \vdash (\mathsf{fix}_n\ f : T^{\star} := M)\,\vec{N}\,C(\vec{p^{\circ}}, \vec{a}) & \mu & M\,[f := \mathsf{fix}_n\ f : T^{\star} := M]\ \vec{N}\,C(\vec{p^{\circ}}, \vec{a}) \\
\Gamma \vdash \mathsf{let}\ x := N : T^{\circ}\ \mathsf{in}\ M & \delta & M\,[x := N] \\
\Gamma_0(x := N : T)\Gamma_1 \vdash M & \delta & M\,[x := N] \\
\Gamma \vdash (\mathsf{case}_{P^{\circ}}\ x := C_j(\vec{q^{\circ}}, \vec{u})\ \mathsf{in} \ldots \mathsf{of}\ \{C_i\,\vec{x_i} \Rightarrow b_i\}_i) & \iota & t_j\sigma_j\,[\vec{x_j} := \vec{u}]
\end{array}$$

where $b_j = (t_j\ \mathsf{where}\ \sigma_j)$.

Observe that if the argument of a case expression is a constructor whose branch is impossible (i.e., $\bot$), then the term does not ι-reduce.

In the compatible closure of the reduction, we use the substitution of the branch. We have the rule

$$\frac{\Gamma(\vec{z_j} : \Delta_j\sigma_j) \vdash t_j \to t'_j}{\Gamma \vdash \mathsf{case} \ldots C\,\vec{z_i} \Rightarrow t_j\ \mathsf{where}\ \vec{\sigma_j} \to \mathsf{case} \ldots C\,\vec{z_i} \Rightarrow t'_j\ \mathsf{where}\ \vec{\sigma_j}}$$

Note that the context $\Gamma(\vec{z_j} : \Delta_j\sigma_j)$ may not be well typed, since it lacks the reorder that the unification may introduce.

The reduction relation of $\mathrm{CIC}^{\widehat{}}_{\mathrm{PM}}$ is confluent. We omit the proof, that follows the same pattern as in the case of $\mathrm{CIC}^{\widehat{}}_{-}$.

Lemma 6.4. *The reduction relation $\to$ is confluent.*

6.3 Typing Rules

We now proceed to explain formally the typing rule of the new constructions: case analysis and local definitions. The typing rules for local definitions given in Fig. 6.1. It is necessary to extend the judgment for well-formed contexts to account for local definitions.

In the rest of the section we present the typing rule of the new case analysis construction. Before presenting the actual rule, we describe the unification judgment that is used to check whether a branch is impossible or possible.

$$\text{(empty)}\quad \frac{}{\mathsf{WF}([])}$$

$$\text{(cons)}\quad \frac{\mathsf{WF}(\Gamma) \qquad \Gamma \vdash T : u}{\mathsf{WF}(\Gamma(x : T))}$$

$$\text{(localdef-cons)}\quad \frac{\mathsf{WF}(\Gamma) \qquad \Gamma \vdash M : T}{\mathsf{WF}(\Gamma(x := M : T))}\quad \mathrm{SV}(M) = \emptyset$$

$$\text{(localdef)}\quad \frac{\Gamma(x := M : T) \vdash N : U}{\Gamma \vdash (\mathsf{let}\ x := M : |T|\ \mathsf{in}\ N) : U}\quad \mathrm{SV}(M, N) = \emptyset$$

Figure 6.1: Typing rules for well-formed contexts and local definitions

Unification. A unification problem has the form

$$\Gamma; \Delta, \zeta \vdash [\vec{u} = \vec{u'} : \Theta],$$

where $\vec{u}$ and $\vec{u'}$ have type Θ under context $\Gamma\Delta$, and $\zeta \subseteq \mathrm{dom}\,(\Delta)$ is the set of variables that are open to unification. A solution for this problem is a substitution σ such that $\vec{u}\sigma \approx \vec{u'}\sigma$, or $\bot$, indicating that no unifier exists. Context Γ is intended to be the "outer context", i.e. the context where we want to type a case construction, while context Δ is defined inside the case. We only allow to unify variables in Δ, so that the unification is invariant under substitutions and reductions that happen outside the case. This is important in the proofs of the Substitution Lemma and Subject Reduction.

The unification algorithm is given by a judgment of the form

$$\Gamma; \Delta, \zeta \vdash [\vec{u} = \vec{v} : \Theta] \mapsto \mathcal{S} \tag{6.1}$$

where Γ, Δ, ζ, $\vec{u}$, $\vec{v}$, Θ are inputs, and $\mathcal{S}$ is the output of the unification. The unification judgment is defined by the rules of Fig. 6.2. These rules are based on the unification given in [62,69], with a notation close to that in [69]. A detailed explanation of the rules is given below. We assume that the rules are applied in the order given in Fig. 6.2. Thus, we can view these rules as a deterministic algorithm. Trying to solve a unification problem may have one of three possible outcomes:

positive success A derivation $\Gamma; \Delta, \zeta \vdash [\vec{u} = \vec{v} : \Theta] \mapsto \Delta', \zeta' \vdash \sigma$ is obtained, where σ is the unifier of $\vec{u}$ and $\vec{v}$, ζ' is the subset of variables of ζ that have not been unified (this means $\mathrm{dom}\,(\sigma) \subseteq \zeta \setminus \zeta'$), and Δ' is a reordering of Δ with the substitution σ applied as definitions.

negative success A derivation $\Gamma; \Delta, \zeta \vdash [\vec{u} = \vec{v} : \Theta] \mapsto \bot$ is obtained, meaning that $\vec{u}$ and $\vec{u'}$ are not unifiable;

failure No rule is applicable, hence no derivation is obtained (the unification problem is too difficult).

A precondition to the unification algorithm is that the input should be well-typed: $\Gamma\Delta \vdash \vec{u} : \Theta$ and $\Gamma\Delta \vdash \vec{v} : \Theta$ should be valid judgments, and also we should have $\zeta \subseteq \mathrm{dom}\,(\Delta)$ so that only variables in Δ are unified. In such case, the following invariants hold.

– For positive success, where $\mathcal{S}$ is $\Delta', \zeta' \vdash \sigma$, we have $\Delta' \leq \Delta$, $\mathsf{WF}(\Gamma\Delta')$, $\Gamma\Delta' \vdash \vec{u} \approx \vec{v}$, and $\Gamma\Delta \vdash \vec{u}\sigma \approx \vec{v}\sigma$ (cf. Lemma 6.13).

– For negative success, where $\mathcal{S}$ is $\bot$ there is no unifier. That is, there is no substitution ρ such that $\Gamma\Delta \vdash \vec{u}\sigma \approx \vec{v}\sigma$ (cf. Lemma 6.14).

We explain the unification rules. Rules (u-varl) and (u-varr) are the basic rules, concerning the unification of a variable with a term. As a precondition, the variable must be a variable open to unification (i.e., it must belong to ζ) and the equation must not be circular (i.e., x does not belong to the set $\mathrm{FV}(v)$), although this last condition is also ensured by the operation $\Delta_{\Gamma|x:=v}$. A reordering of the context Δ may be required in order to obtain a well-typed substitution. This is achieved by the (partial) operation $\Delta_{\Gamma|x:=t}$ defined as

$$\begin{aligned}(\Delta_0(x:T)\Delta_1)_{\Gamma|x:=t} = \Delta_0\Delta^t(x := t : T)\Delta_t \\ \text{where } (\Delta^t, \Delta_t) = \mathrm{strengthen}(\Delta_1, t) \\ \Gamma\Delta_0\Delta^t \vdash t : T \\ \Gamma\Delta_0\Delta^t(x:T) \vdash \Delta_t\end{aligned}$$

The strengthen operation [62] is defined as

$$\begin{aligned}\mathrm{strengthen}([], t) &= ([], []) \\ \mathrm{strengthen}((x:U)\Delta, t) &= \begin{cases}((x:U)\Delta_0, \Delta_1) & \text{if } x \in \mathrm{FV}(\Delta_0) \cup \mathrm{FV}(t) \\ (\Delta_0, (x:U)\Delta_1) & \text{if } x \notin \mathrm{FV}(\Delta_0) \cup \mathrm{FV}(t)\end{cases} \\ &\text{where } (\Delta_0, \Delta_1) = \mathrm{strengthen}(\Delta, t)\end{aligned}$$

Rules (u-discr) and (u-inj) codify the no-confusion property of inductive types: rule (u-discr) states that constructors are disjoint (negative success), while rule (u-inj) states that constructors are injective.

If the first four rules are not applicable, then the unification can succeed only if the terms are convertible. This is shown in rule (u-conv). Rules (u-empty) and (u-tel) concern the unification of sequence of terms. Finally, rules (u-inj-$\bot$), (u-tel-$\bot_1$), and (u-tel-$\bot_2$) correspond to the propagation of negative failure.

We could also add a rule for transforming a unification problem, such as

$$\frac{\Gamma; \Delta, \zeta \vdash [\vec{u'} = \vec{v'} : \Theta] \mapsto \Delta', \zeta' \vdash \sigma \qquad \vec{u} \to^* \vec{u'} \qquad \vec{v} \to^* \vec{v'}}{\Gamma; \Delta, \zeta \vdash [\vec{u} = \vec{v} : \Theta] \mapsto \Delta', \zeta' \vdash \sigma}$$

In an implementation, the above rule could be used by reducing $\vec{v}$ and $\vec{u}$ into a head-normal form that would expose the head constructor or head variable. For simplicity, we do not consider such rules.

We use $\mathcal{U}$ to denote unification problems. If $\mathcal{U}$ denotes the unification problem $\Gamma; \Delta, \zeta \vdash [\vec{u} = \vec{v} : \Theta]$, we write $\mathcal{U} \mapsto \bot$ and $\mathcal{U} \mapsto \Delta', \zeta' \vdash \sigma$ to denote the corresponding unification judgments.

The typing rule. In Fig. 6.3 we show the typing rule for the new elimination rule, and introduce a new judgment for typechecking branches. This new judgment has the form

$$\Gamma; \Delta_i; \Delta; [\vec{u} = \vec{v} : \Theta] \vdash b : T$$

The intuition is that we try to solve the unification problem $\Gamma; \Delta_i\Delta, \zeta \vdash [\vec{u} = \vec{v} : \Theta]$ (where ζ depends on the kind of branch considered), and use the result of the unification to check

$$\text{(u-varl)}\quad \frac{x \in \zeta \qquad x \notin \mathrm{FV}(v)}{\Gamma; \Delta, \zeta \vdash [x = v : T] \mapsto \Delta_{\Gamma|x:=v}, \zeta \setminus \{x\} \vdash \{x \mapsto v\}}$$

$$\text{(u-varr)}\quad \frac{x \in \zeta \qquad x \notin \mathrm{FV}(v)}{\Gamma; \Delta, \zeta \vdash [v = x : T] \mapsto \Delta_{\Gamma|x:=v}, \zeta \setminus \{x\} \vdash \{x \mapsto v\}}$$

$$\text{(u-discr)}\quad \frac{C_i \neq C_j}{\Gamma; \Delta, \zeta \vdash [C_i(\_, \vec{u}) = C_j(\_, \vec{v}) : T] \mapsto \bot}$$

$$\text{(u-inj)}\quad \frac{\Gamma\Delta \vdash T \approx I^{\widehat{s}}\,(\vec{p}, \vec{a}) \qquad \Gamma; \Delta, \zeta \vdash [\vec{u} = \vec{v} : \mathsf{argsConstr}^s_C(\vec{p})] \mapsto \Delta', \zeta' \vdash \sigma}{\Gamma; \Delta, \zeta \vdash [C(\_, \vec{u}) = C(\_, \vec{v}) : T] \mapsto \Delta', \zeta' \vdash \sigma}$$

$$\text{(u-inj-}\bot\text{)}\quad \frac{\Gamma\Delta \vdash T \approx I^{\widehat{s}}\,(\vec{p}, \vec{a}) \qquad \Gamma; \Delta, \zeta \vdash [\vec{u} = \vec{v} : \mathsf{argsConstr}^s_C(\vec{p})] \mapsto \bot}{\Gamma; \Delta, \zeta \vdash [C(\_, \vec{u}) = C(\_, \vec{v}) : T] \mapsto \bot}$$

$$\text{(u-conv)}\quad \frac{\Gamma\Delta \vdash u \approx v}{\Gamma; \Delta, \zeta \vdash [u = v : T] \mapsto \Delta, \zeta \vdash \varepsilon}$$

$$\text{(u-empty)}\quad \frac{}{\Gamma; \Delta, \zeta \vdash [\varepsilon = \varepsilon : []] \mapsto \Delta, \zeta \vdash \varepsilon}$$

$$\text{(u-tel)}\quad \frac{\begin{array}{c}\Gamma; \Delta, \zeta \vdash [u = v : T] \mapsto \Delta_1, \zeta_1 \vdash \sigma_1 \\ \Gamma; \Delta_1, \zeta_1 \vdash [\vec{u}\sigma_1 = \vec{v}\sigma_1 : \Theta\,[x := u]\,\sigma_1] \mapsto \Delta_2, \zeta_2 \vdash \sigma_2\end{array}}{\Gamma; \Delta, \zeta \vdash [u, \vec{u} = v, \vec{v} : (x : T)\Theta] \mapsto \Delta_2, \zeta_2 \vdash \sigma_1\sigma_2}$$

$$\text{(u-tel-}\bot_1\text{)}\quad \frac{\Gamma; \Delta, \zeta \vdash [u = v : T] \mapsto \bot}{\Gamma; \Delta, \zeta \vdash [u, \vec{u} = v, \vec{v} : (x : T)\Theta] \mapsto \bot}$$

$$\text{(u-tel-}\bot_2\text{)}\quad \frac{\begin{array}{c}\Gamma; \Delta, \zeta \vdash [u = v : T] \mapsto \Delta_1, \zeta_1 \vdash \sigma_1 \\ \Gamma; \Delta_1, \zeta_1 \vdash [\vec{u}\sigma_1 = \vec{v}\sigma_1 : \Theta\,[x := u]\,\sigma_1] \mapsto \bot\end{array}}{\Gamma; \Delta, \zeta \vdash [u, \vec{u} = v, \vec{v} : (x : T)\Theta] \mapsto \bot}$$

Figure 6.2: Unification rules

$$\text{(b-}\bot\text{)}\quad \frac{\Gamma; \Delta_i\Theta, \mathrm{dom}\,(\Delta_i) \cup \mathrm{dom}\,(\Theta) \vdash [\vec{u} = \vec{v} : \Delta_a] \mapsto \bot}{\Gamma; \Delta_i; \Theta; [\vec{u} = \vec{v} : \Delta_a] \vdash \bot : P}$$

$$\text{(b-sub)}\quad \frac{\begin{array}{c}\Gamma; \Delta_i\Theta, \mathrm{dom}\left(\vec{d}\right) \cup \mathrm{dom}\,(\Theta) \vdash [\vec{u} = \vec{v} : \Delta_a] \mapsto \Theta', \emptyset \vdash \sigma \\ \Gamma\Theta' \vdash t : P \qquad \Gamma\Theta' \vdash \vec{d} \approx \sigma\end{array}}{\Gamma; \Delta_i; \Theta; [\vec{u} = \vec{v} : \Delta_a] \vdash t \;\mathsf{where}\; \vec{d} : P}$$

$$\text{(case')}\ \frac{\begin{array}{c} I \in \Sigma \qquad \Delta_a \equiv \mathsf{argsInd}_I(\vec{p}) \qquad \Delta_i \equiv \mathsf{argsConstr}^s_{C_i}(\vec{p}) \\ \Gamma \vdash M : \widehat{I^s}\,(\vec{p}, \vec{a}\,[\mathrm{dom}\,(\Theta) := \vec{q}]) \qquad \Gamma \vdash \vec{q} : \Theta \\ \Gamma\Theta \vdash \vec{a} : \Delta_a \qquad \Gamma\Theta(x : \widehat{I^s}\,(\vec{p}, \vec{a})) \vdash P : u \\ \Gamma; (\vec{z_i} : \Delta_i); \Theta; [\mathsf{indices}_{C_i}(\vec{p}) = \vec{a} : \Delta_a] \vdash b_i : P\,[x := C_i(|\vec{p}|, \vec{z_i})] \end{array}}{\Gamma \vdash \left(\begin{array}{l}\mathsf{case}_{|P|}\ x := M\ \mathsf{in}\ [|\Theta|]\ I\,(|\vec{p}|, |\vec{a}|) \\ \mathsf{where}\ \mathrm{dom}\,(\Theta) := |\vec{q}|\ \mathsf{of}\ \langle C_i\ \vec{z_i} \Rightarrow b_i\rangle_i\end{array}\right) : P\,[\mathrm{dom}\,(\Theta) := \vec{q}]\,[x := M]} \quad \mathrm{SV}(\vec{q}) = \emptyset$$

Figure 6.3: Typing rules for the new elimination rule

the type of the branch. This judgment is defined by the rules (b-$\bot$) and (b-sub) in Fig. 6.3. In rule (b-$\bot$), that corresponds to impossible branches, we take ζ to be $\mathrm{dom}\,(\Delta_i) \cup \mathrm{dom}\,(\Delta)$, and we check that the unification succeeds negatively.

In rule (b-sub), that corresponds to possible branches, we take ζ to be $\mathrm{dom}\,(\Delta)$ together with the domain of the definitions of the branch ($\vec{d}$ in this case). Context Δ corresponds to the variables that define the subfamily under analysis. We check that the unification succeeds positively, leaving no variables open. We also check that the definitions $\vec{d}$ are valid using the judgment $\Gamma\Delta' \vdash \vec{d} \leq \sigma$; this means that, for every variable definition $(x := N)$ of $\vec{d}$, we have $\Gamma\Delta' \vdash N \approx x\sigma$. Then, we typecheck the body proper of the branch using the context given by the unification (Θ').

Finally, in the rule (t-match) we put everything together. The subfamily under analysis is defined by $[\Theta]\ I\,(\vec{p}, \vec{a})$, hence, we check that M belongs to it by checking that it has type $I\,(\vec{p}, \vec{a}\,[\mathrm{dom}\,(\Theta) := \vec{q}])$. We also check that $\vec{q}$ has the correct type; and also that P is a type. The return type P depends on x and $\mathrm{dom}\,(\Theta)$, similarly to the old rule, where P depended on the argument and the indices of the inductive type. In the branches, as in the old rule, x is replaced by the corresponding constructor applied to the arguments. Here, in contrast with the old rule where it was clear how to instantiate the indices, there are no obvious values we can give to the variables in $\mathrm{dom}\,(\Theta)$. Therefore, we try the unification between $\vec{a}$ (that defines the subfamily under analysis) and $\mathsf{indices}_{C_i}(\vec{p})$. Since, for possible branches, the unification does not leave open variables, we effectively find a value for each variable in $\mathrm{dom}\,(\Theta)$.

Remark: In a branch of the form N where σ, only $\mathrm{dom}\,(\sigma)$ is needed to compute the unification. There is no need to explicitly assign values to the variables in $\mathrm{dom}\,(\sigma)$ since they can be computed by the unification. If the values are present, they are checked to be valid with respect to the result of the unification. This is similar to the situation of *inaccessible patterns* in [42,69].

6.4 Examples

We illustrate the new elimination rule with some examples. To simplify the syntax, we do not explicitly write impossible branches. Missing constructors are treated as impossible branches.

Streicher's K axiom and heterogeneous equality. Axiom K, also known as *uniqueness of reflexivity proofs*, states that there is only one proof of $x = x$, namely refl. It is expressed by the formula:

$$\Pi(A : \mathsf{Type})(x : A)(P : \mathsf{eq}(A, x, x) \to \mathsf{Prop}).P(\mathsf{refl}(A, x)) \to \Pi(p : \mathsf{eq}(A, x, x)).P\, p \ .$$

This theorem is provable in Coquand's original proposal, Agda, Epigram, and, as we show below, our approach. However, it is not derivable in CIC, as shown by Hofmann and Streicher [47]; that is why it is called axiom. A naive attempt to prove axiom K in CIC would proceed by case analysis on the proof of equality:

$$\begin{array}{l} K \stackrel{\text{def}}{=} \lambda(A : \mathsf{Type})\,(x : A)\,(P : \mathsf{eq}(A, x, x) \to \mathsf{Prop})\,(H : P\,(\mathsf{refl}(A, x)))(p : \mathsf{eq}(A, x, x)). \\ \quad \mathsf{case}_{\,P\,p_0}\ p_0 := p\ \mathsf{in}\ \mathsf{eq}(A, x, y)\ \mathsf{of} \\ \quad \mid \mathsf{refl} \Rightarrow \ldots \end{array}$$

The problem is $P\,p_0$ is not well-typed, since p_0 has type $\mathsf{eq}(A, x, y)$, while it is expected to have $\mathsf{eq}(A, x, x)$.

McBride [59] shows that axiom K is equivalent to heterogeneous equality. It is thus no surprise that we can derive both axiom K and the elimination of homogeneous equality. For axiom K, the idea of the proof is to restrict the analysis to the subfamily of reflexivity proofs: $[]\ \mathsf{eq}(A, x, x)$. It can be defined as:

$$\begin{array}{l} K \stackrel{\text{def}}{=} \lambda\,(A : \mathsf{Type})\,(x : A)\,(P : \mathsf{eq}(A, x, x) \to \mathsf{Prop})\,(H : P\,(\mathsf{refl}(A, x)))\,(p : \mathsf{eq}(A, x, x)). \\ \quad \mathsf{case}_{\,P\,p_0}\ p_0 := p\ \mathsf{in}\ []\,\mathsf{eq}(A, x, x)\ \mathsf{of} \\ \quad \mid \mathsf{refl} \Rightarrow H \end{array}$$

Note the pattern $[]\ \mathsf{eq}(A, x, x)$ in the elimination. We fix the index of eq to be x, therefore p_0 has type $\mathsf{eq}(A, x, x)$, i.e. a reflexivity proof, and $P\,p_0$ is well typed.

For the heterogeneous equality defined by an inductive type, the elimination rule for homogeneous equations can be easily proved using the new rule:

$$\begin{array}{l} \mathsf{heq\_rect} : \Pi(A : \mathsf{Type}_0)\,(P : A \to \mathsf{Type}_0)\,(x : A).P\,x \to \Pi(y : A).\mathsf{heq}(A, x, A, y) \to P\,y \\ \mathsf{heq\_rect} \stackrel{\text{def}}{=} \lambda(A : \mathsf{Type}_0)(P : A \to \mathsf{Type}_0)(x : A)(p : P\,x)(y : A)(H : \mathsf{heq}(A, x, A, y)). \\ \quad \mathsf{case}_{\,P\,y_0}\ h_0 := H\ \mathsf{in}\ [y_0 : A]\,\mathsf{heq}(A, x, A, y_0)\ \mathsf{of} \\ \quad \mid \mathsf{heq\_refl} \Rightarrow p \end{array}$$

Similarly to axiom K, we use the new rule to restrict the subfamily under analysis; in this case, we restrict to homogeneous equalities, expressed by the pattern $[y_0 : A]\ \mathsf{heq}(A, x, A, y_0)$. Note that the first index of heq is fixed to be A.

Finite numbers. Consider the inductive type of finite numbers:

$$\begin{aligned}\mathsf{Ind}(\mathsf{fin} : \mathsf{nat} \to \mathsf{Type} := \mathsf{fO} &: \Pi(n : \mathsf{nat}).\mathcal{X}\,(\mathsf{S}\,n),\\ \mathsf{fS} &: \Pi(n : \mathsf{nat}).\mathcal{X}\,n \to \mathcal{X}\,(\mathsf{S}\,n))\ .\end{aligned}$$

Note that $\mathsf{fin}\,n$ has n elements and, intuitively, they represent the numbers $\{0, \ldots, n-1\}$. In particular, $\mathsf{fin}\,\mathsf{O}$ has no elements. This is proved by the following term, of type $\mathsf{fin}\,\mathsf{O} \to \mathsf{False}$:

$$\mathsf{fin\_zero\_False} \equiv \lambda(f : \mathsf{fin}\,\mathsf{O}).\ \mathsf{case}_{\mathsf{False}}\ f\ \mathsf{in}\ []\,\mathsf{fin}\,\mathsf{O}\ \mathsf{of}$$

It is easy to see that the indices of fO and fS, of the form $\mathsf{S}\ \_$, do not unify with O. Hence, both branches are impossible.

Finite numbers are useful to define a total lookup function for vectors:

$$\mathsf{lookup} : \Pi(A : \mathsf{Type})(n : \mathsf{nat}).\mathsf{vector}\,A\,n \to \mathsf{fin}\,n \to A\ .$$

We define it using a fixpoint and case analysis on the vector argument:

$$\begin{aligned}&\mathsf{lookup} \equiv \lambda(A : \mathsf{Type}).\ \mathsf{fix}\ \mathsf{lookup} : \Pi(n : \mathsf{nat}).\ \mathsf{vector}\,A\,n \to \mathsf{fin}\,n \to A := \lambda(n : \mathsf{nat})(v : \mathsf{vector}\,A\,n).\\ &\quad \mathsf{case}_{\mathsf{fin}\,n_0 \to A}\,v_0 := v\ \mathsf{in}\ [(n_0 : \mathsf{nat})]\,\mathsf{vector}\,A\,n_0\ \mathsf{where}\ n_0 := n\ \mathsf{of}\\ &\quad \mid \mathsf{vnil} \Rightarrow \lambda(f : \mathsf{fin}\,\mathsf{O}).\ \mathsf{elimFalse}\,(\mathsf{fin\_zero\_False}\,f)\,A\\ &\quad \mid \mathsf{vcons}\,n_1\,a_1\,v_1 \Rightarrow \lambda(f : \mathsf{fin}\,(\mathsf{S}\,n_1)).\ \mathsf{case}_A\ f_0 := f\ \mathsf{in}\ []\,\mathsf{fin}\,(\mathsf{S}\,n_1)\ \mathsf{of}\\ &\qquad\qquad\qquad\qquad \mid \mathsf{fO}\,(n_3 := n_1) \Rightarrow a_1\\ &\qquad\qquad\qquad\qquad \mid \mathsf{fS}\,(n_3 := n_1)\,f_3 \Rightarrow \mathsf{lookup}\,n_1\,v_1\,f_3\end{aligned}$$

Note that no restriction is imposed in the subfamily of vectors to analyze. The branch vnil is easily eliminated using $\mathsf{fin\_zero\_False}$. In the branch vcons we perform a case analysis on the index argument of type $\mathsf{fin}\,(\mathsf{S}\,n_1)$. By restricting to the subfamily $[]\,\mathsf{fin}\,(\mathsf{S}\,n_1)$, we can unify the first argument of the constructors fO and fS with the first argument of vcons. This ensures that the recursive call is well typed.

Another way of defining lookup is by case analysis on the index argument of type fin followed by a case analysis on the vector argument, restricted to non-empty vectors. Of the four combinations of constructors, only $\langle \mathsf{fO}, \mathsf{vnil} \rangle$ and $\langle \mathsf{fS}, \mathsf{vcons} \rangle$ need to be considered. The other two are impossible.

Less-or-equal relation on natural numbers. We show two examples concerning the relation less-or-equal for natural numbers defined inductively by

$$\begin{aligned}\mathsf{Ind}(\mathsf{leq} : \mathsf{nat} \to \mathsf{nat} \to \mathsf{Type}_0 := \mathsf{leq0} &: \Pi(n : \mathsf{nat}).\mathcal{X}\,\mathsf{O}\,n,\\ \mathsf{leqS} &: \Pi(m\,n : \mathsf{nat}).\mathcal{X}\,m\,n \to \mathcal{X}\,(\mathsf{S}\,m)\,(\mathsf{S}\,n))\ .\end{aligned}$$

First, we show that the successor of a number is not less-or-equal than the number itself. That is, we want to find a term of type

$$\Pi(n : \mathsf{nat}).\mathsf{leq}\,(\mathsf{S}\,n)\,n \to \mathsf{False}\ .$$

One possible solution is to take

$$\begin{array}{l}\mathsf{fix}\ f : \Pi(n : \mathsf{nat}).\mathsf{leq}(\mathsf{S}\,n)\,n \to \mathsf{False} := \\ \quad \lambda(n : \mathsf{nat})(H : \mathsf{leq}(\mathsf{S}\,n)\,n). \\ \quad\ \mathsf{case}_{\mathsf{False}}\ H\ \mathsf{in}\ [(n_0 : \mathsf{nat})]\,\mathsf{leq}(\mathsf{S}\,n_0)\,n_0\ \mathsf{where}\ n_0 := n\ \mathsf{of} \\ \quad\ |\ \mathsf{leqS}\,x\,y\,H \Rightarrow f\,y\,H\ \mathsf{where}\ (x := \mathsf{S}\,y)(n_0 := \mathsf{S}\,y)\end{array}$$

In the leq0 branch, the unification problem considered is

$$\{x, n_0\} \vdash [\mathsf{O}, x = \mathsf{S}\,n_0, n_0],$$

where x is a fresh variable that stands for the argument of leq0. Clearly, unification succeeds negatively because of the first equation. On the leqS branch, the unification problem is

$$\{x, n_0\} \vdash [\mathsf{S}\,x, \mathsf{S}\,y = \mathsf{S}\,n_0, n_0],$$

which succeeds positively with the substitution $\{x \mapsto \mathsf{S}\,y, n_0 \mapsto \mathsf{S}\,y\}$. Note that the unification gives us the value for n_0 that is necessary for the branch to have the required type, but also finds a relation between the arguments of the constructor x and y. Therefore, the body of the branch is typed in a context containing the declarations

$$(y : \mathsf{nat})(x := \mathsf{S}\,y : \mathsf{nat})(H : \mathsf{leq}\,x\,y)\ .$$

Note the reordering of x and y. In this context, the recursive call to f is well typed.

The second example shows that the relation leq is transitive. That is, we want to find a term of type $\Pi(x\,y\,z : \mathsf{nat}).\mathsf{leq}\,x\,y \to \mathsf{leq}\,y\,z \to \mathsf{leq}\,x\,z$. One possible solution is to define it by

$$\begin{array}{l}\mathsf{fix}\ \mathsf{trans} : \Pi(m\,n\,k : \mathsf{nat}).\mathsf{leq}\,m\,n \to \mathsf{leq}\,n\,k \to \mathsf{leq}\,m\,k := \\ \quad \lambda(m\,n\,k : \mathsf{nat})(H_1 : \mathsf{leq}\,m\,n)(H_2 : \mathsf{leq}\,n\,k). \\ \quad\ (\mathsf{case}_{\mathsf{leq}\,n_1\,k \to \mathsf{leq}\,m_1\,k}\ H_1\ \mathsf{in}\ [(m_1\,n_1 : \mathsf{nat})]\,\mathsf{leq}\,m_1\,n_1\ \mathsf{of} \\ \quad\ |\ \boxed{\mathsf{leq0}\,x} \Rightarrow \lambda(h_2 : \mathsf{leq}\,x\,k).\ \mathsf{leq0}\,k \\ \quad\ |\ \boxed{\mathsf{leqS}\,x\,y\,H} \Rightarrow \lambda(h_2 : \mathsf{leq}\,(\mathsf{S}\,y)\,k). \\ \qquad\ \mathsf{case}_{\mathsf{leq}\,(\mathsf{S}\,x)\,k_2}\ h_2\ \mathsf{in}\ [(k_2 : \mathsf{nat})]\,\mathsf{leq}\,(\mathsf{S}\,y)\,k_2\ \mathsf{of} \\ \qquad\ |\ \boxed{\mathsf{leqS}\,(x' := y)\,y'\,H'} \Rightarrow\ \mathsf{leqS}\,x\,y'\,(\mathsf{trans}\,H\,H')\ )\,H_2\end{array}$$

For the sake of readability, we have used implicit arguments (e.g., in the recursive call to trans), and omitted definitions that can be inferred by unification.

Nevertheless, this definition looks complicated. It consists of a nested case analysis on $\langle H_1, H_2\rangle$. However, to make the definition go through, we need to generalize the type of the hypothesis H_2 in the return type of the case analysis of H_1, so that we can match the common value n in the types of H_1 and H_2.

The case $\langle \mathsf{leq0}, \_\rangle$ is easy; the case $\langle \mathsf{leqS}, \mathsf{leq0}\rangle$ is impossible; finally, the case $\langle \mathsf{leqS}, \mathsf{leqS}\rangle$ is the most complicated. Note however, that things are simplified by stating that variable x' should be unified. The unification then finds a value for x' and checks that is convertible with y, thus unifying the arguments of both constructors. The body of the branch is typed in a context containing the declarations

$$(x\,y : \mathsf{nat})(H : \mathsf{leq}\,x\,y)(x' := y : \mathsf{nat})(y' : \mathsf{nat})(H' : \mathsf{leq}\,x'\,y')\ .$$

It is easy to see that, in this context, the recursive call to trans is well typed.

Let us compare this definition with its counterpart in Agda. Transitivity of leq can be defined in Agda as

$$\begin{aligned}
&\mathsf{trans} : (m\, n\, k : \mathsf{nat}) \to \mathsf{leq}\, m\, n \to \mathsf{leq}\, n\, k \to \mathsf{leq}\, m\, k\\
&\mathsf{trans}\ \lfloor 0 \rfloor\ \lfloor x \rfloor\ k\ \boxed{(\mathsf{leq0}\, x)}\ _ = \mathsf{leq0}\, k\\
&\mathsf{trans}\ \lfloor \mathsf{S}\, x \rfloor\ \lfloor \mathsf{S}\, y \rfloor\ \lfloor \mathsf{S}\, y' \rfloor\ \boxed{(\mathsf{leqS}\, x\, y\, H)}\ \boxed{(\mathsf{leqS}\ \lfloor y \rfloor\ y'\, H')} = \mathsf{leqS}\, x\, y'\, (\mathsf{trans}\, H\, H')
\end{aligned}$$

Besides writing the return types in both cases, and the fact that we generalize the type of the second argument, our definition looks very much like a direct translation of the Agda version to a nested case definition (compare the highlighted parts).

Trees and forests. We show some functions defined on trees and forests (i.e., lists of trees). Instead of the encoding used in Chap. 2, we use the following definition, taken from [72]:

$$\begin{aligned}
\mathsf{Ind}(\mathsf{TL}[A : \mathsf{Type}_0]^{\oplus} : \mathsf{bool} \to \mathsf{Type}_0 := \mathsf{node} &: A \to \mathcal{X}\, \mathsf{false} \to \mathcal{X}\, \mathsf{true},\\
\mathsf{nil} &: \mathcal{X}\, \mathsf{false},\\
\mathsf{cons} &: \mathcal{X}\, \mathsf{true} \to \mathcal{X}\, \mathsf{false} \to \mathcal{X}\, \mathsf{false})
\end{aligned}$$

We define then $\mathsf{tree}(A) \stackrel{\text{def}}{=} \mathsf{TL}\,(A)\ \mathsf{true}$ and $\mathsf{forest}(A) \stackrel{\text{def}}{=} \mathsf{TL}\,(A)\ \mathsf{false}$. We can easily restrict functions that operate on trees or forests, by restricting case analysis to the subfamilies TL A true and TL A false. Only the relevant constructors need to be considered.

Consider a function that adds a tree as a child of another. It can be defined as follows:

$$\begin{aligned}
&\mathsf{addTree} : \Pi(A : \mathsf{Type}_0).\mathsf{tree}(A) \to \mathsf{tree}(A) \to \mathsf{tree}(A)\\
&\mathsf{addTree} \stackrel{\text{def}}{=} \lambda\, (A : \mathsf{Type}_0)\, (t_1 : \mathsf{tree}(A))\, (t_2 : \mathsf{tree}(A)).\\
&\qquad \mathsf{case}_{\,\mathsf{tree}(A)}\ t_2\ \mathsf{in}\ []\,\mathsf{tree}(A)\ \mathsf{of}\\
&\qquad \mid \mathsf{node}\, a\, l \Rightarrow \mathsf{node}\, a\, (\mathsf{cons}\, t_1\, l)
\end{aligned}$$

Constructors nil and cons are not considered since they are trivially impossible. Similarly, we can write functions on forest that only consider the relevant constructors. The following function builds a tree from a forest using a default element for the nodes.

$$\begin{aligned}
&\mathsf{buildTree} : \Pi(A : \mathsf{Type}_0).A \to \mathsf{forest}(A) \to \mathsf{tree}(A)\\
&\mathsf{buildTree} \stackrel{\text{def}}{=} \lambda\, (A : \mathsf{Type}_0)\, (a : A)\, (l : \mathsf{forest}(A)).\\
&\qquad \mathsf{case}_{\,\mathsf{tree}(A)}\ l\ \mathsf{in}\ []\,\mathsf{forest}(A)\ \mathsf{of}\\
&\qquad \mid \mathsf{nil} \Rightarrow \mathsf{node}\, a\, \mathsf{nil}\\
&\qquad \mid \mathsf{cons}\, t\, l' \Rightarrow \mathsf{node}\, a\, (\mathsf{cons}\, t\, (\mathsf{cons}\, (\mathsf{buildTree}\, A\, a\, l')\, \mathsf{nil}))
\end{aligned}$$

6.5 Metatheory

In this section we show that $\mathrm{CIC}^{\widehat{}}_{\mathrm{PM}}$ satisfies some simple metatheoretical properties such as weakening, substitution and subject reduction. Most of the proofs proceed by induction on the type derivation. Hence adapting the proofs from $\mathrm{CIC}^{\widehat{}}_{-}$ is simply a matter of checking that the typing rules for the new constructions (case and let expressions) still satisfy the property.

We restate the definition of well-typed substitution to cope with local definitions.

Definition 6.5 (Well-typed substitution)**.** *Given a substitution σ and contexts Γ, Δ, we say that σ is* well-typed from Γ to Δ*, written $\sigma : \Gamma \to \Delta$, if* $\mathsf{WF}(\Gamma)$*,* $\mathsf{WF}(\Delta)$*,* $\mathrm{dom}\,(\sigma) \subseteq \mathrm{dom}\,(\Gamma)$*,*
- *for every $(x : T) \in \Gamma$, $\Delta \vdash x\sigma : T\sigma$, and*
- *for every $(x := t : T) \in \Gamma$, $\Delta \vdash x\sigma \approx t\sigma : T\sigma$.*

We state a generation lemma for the new constructions. For the constructions that are shared between $\mathrm{CIC}^{\widehat{}}_{\mathrm{PM}}$ and $\mathrm{CIC}^{\widehat{}}_{-}$, Lemma 3.10 still holds.

Lemma 6.6 (Generation lemma for case and let)**.**

1. *If $\Gamma \vdash \mathsf{let}\ x := N : T^\circ\ \mathsf{in}\ M : U$, then there exists T' and W such that*
   - $\Gamma \vdash N : T'$,
   - $|T'| \equiv T^\circ$,
   - $\Gamma(x := N : T') \vdash M : W$, *and*
   - $W \leq U$.
2. *If $\Gamma \vdash (\mathsf{case}_{P^\circ}\ x := M\ \mathsf{in}\ [\Theta^\circ]\, I\left(\vec{p^\circ}, \vec{t^\circ}\right)\ \mathsf{where}\ \mathrm{dom}\,(\Theta^\circ) := \vec{q^\circ}\ \mathsf{of}\ \langle C_i\, \vec{x_i} \Rightarrow b_i\rangle_i) : U$, then there exists $\vec{p'}$, $\vec{t'}$, P', Θ', and s such that*
   - $\Gamma \vdash M : I^{\widehat{s}}\left(\vec{p'}, \vec{t'}\left[\mathrm{dom}\,(\Theta^\circ) := \vec{q'}\right]\right)$,
   - $\Gamma\Theta' \vdash \vec{t'} : \Delta_a$,
   - $\Gamma \vdash \vec{q'} : \Theta'$,
   - $\Gamma\Theta\left(x : I^{\widehat{s}}\left(\vec{p'}, \vec{t'}\right)\right) \vdash P' : u$,
   - *for all i,* $\Gamma; \vec{z_i} : \Delta_i; \Theta'; [\mathsf{indices}_{C_i}(\vec{p}) = \vec{t'} : {\Delta_a}^*] \vdash b_i : P'\left[x := C_i(\vec{p^\circ}, \vec{z_i})\right]$
   - $|\vec{p'}| \equiv \vec{p^\circ}$, $|\vec{t'}| \equiv \vec{t^\circ}$, $|P'| \equiv P^\circ$, $|\Theta'| \equiv \Theta^\circ$,
   - $P'\left[\mathrm{dom}\,(\Delta^\circ) := \vec{q}\right][x := M] \leq U$

   where $\Delta_a \equiv \mathsf{argsInd}_I(\vec{p})$ and $\Delta_i \equiv \mathsf{argsConstr}^s_{C_i}(\vec{p})$.

Weakening and substitution for $\mathrm{CIC}^{\widehat{}}_{\mathrm{PM}}$ can be easily proved.

Lemma 6.7 (Weakening)**.** *If $\Gamma \vdash M : T$, $\mathsf{WF}(\Delta)$, and $\Delta \leq \Gamma$, then $\Delta \vdash M : T$.*

Proof. We prove simultaneously by induction on derivations the following statements:
- $\Gamma \vdash M : T \wedge \mathsf{WF}(\Gamma') \wedge \Gamma' \leq \Gamma \Rightarrow \Gamma' \vdash M : T$,
- $\Gamma; \Delta, \zeta \vdash [\vec{u} = \vec{v} : \Theta] \mapsto \Delta', \zeta' \vdash \sigma \wedge \mathsf{WF}(\Gamma') \wedge \Gamma' \leq \Gamma \Rightarrow \Gamma'; \Delta, \zeta \vdash [\vec{u} = \vec{v} : \Theta] \mapsto \Delta', \zeta' \vdash \sigma$,
- $\Gamma; \Delta, \zeta \vdash [\vec{u} = \vec{v} : \Theta] \mapsto \bot \wedge \mathsf{WF}(\Gamma') \wedge \Gamma' \leq \Gamma \Rightarrow \Gamma'; \Delta, \zeta \vdash [\vec{u} = \vec{v} : \Theta] \mapsto \bot$,
- if $\Delta_{\Gamma|x:=M}$ is defined, $\mathsf{WF}(\Gamma')$, $\Gamma' \leq \Gamma$, then $\Delta_{\Gamma'|x:=M}$ is defined and $\Delta_{\Gamma|x:=M} = \Delta_{\Gamma'|x:=M}$.

Rules (let) and (case') follow easily using the IH. □

Lemma 6.8 (Substitution)**.** *If $\Gamma \vdash M : T$ and $\sigma : \Gamma \to \Delta$, then $\Delta \vdash M\sigma : T\sigma$.*

Proof. We prove simultaneously by induction on derivations the following statements:
- $\Gamma \vdash M : T \wedge \sigma : \Gamma \to \Gamma' \Rightarrow \Gamma' \vdash M\sigma : T\sigma$,
- $\Gamma; \Delta, \zeta \vdash [\vec{u} = \vec{v} : \Theta] \mapsto \Delta', \zeta' \vdash \rho \wedge \sigma : \Gamma \to \Gamma' \Rightarrow \Gamma'; \Delta\sigma, \zeta \vdash [\vec{u}\sigma = \vec{v}\sigma : \Theta\sigma] \mapsto \Delta'\sigma, \zeta' \vdash \rho\sigma$,
- $\Gamma; \Delta, \zeta \vdash [\vec{u} = \vec{v} : \Theta] \mapsto \bot \wedge \wedge\sigma : \Gamma \to \Gamma' \Rightarrow \Gamma'; \Delta\sigma, \zeta \vdash [\vec{u}\sigma = \vec{v}\sigma : \Theta\sigma] \mapsto \bot$,

– if $\Delta_{\Gamma|x:=M}$ is defined and $\sigma : \Gamma \to \Gamma'$, then $\Delta\sigma_{\Gamma'|x:=M\sigma}$ is defined and $\Delta_{\Gamma|x:=M} = \Delta\sigma_{\Gamma'\sigma|x:=M\sigma}$.

Rules (let) and (case') follow easily using the IH. □

In the following we prove that successful unification is sound, in the sense that positive success returns a unifier, while negative success implies that there is no unifier. First, we need some simple lemmas about substitutions.

Lemma 6.9. *If $\sigma : \Delta \to \Gamma$, and $\Gamma \vdash M \approx M'$, then $\Delta \vdash M\sigma \approx M'\sigma$.*

Proof. We prove that if $\Gamma \vdash M \to M'$, then $\Delta \vdash M\sigma \approx M'\sigma$. The only interesting case is $\Gamma_0(x := u : U)\Gamma_1 \vdash M\ \delta\ M\,[x := u]$. Since σ is a substitution, we know that $\Delta \vdash x\sigma \approx u\sigma$, therefore $\Delta \vdash M\,[x := x\sigma] \approx M\,[x := u\sigma]$. This implies that $\Delta \vdash M\,[x := x\sigma]\,\sigma \approx M\,[x := u\sigma]\,\sigma$. But the lhs is equal to $M\sigma$ and the rhs is equal to $M\,[x := u]\,\sigma$, which is the desired result. □

Lemma 6.10. *If $\sigma : \Delta \to \Gamma$ and $\delta : \Theta \to \Delta$, then $\sigma\delta : \Theta \to \Gamma$.*

Proof. We know that for every $(x : T) \in \Gamma$, $\Delta \vdash x\sigma : T\sigma$. Applying the previous lemma, we obtain $\Theta \vdash x\sigma\delta : T\sigma\delta$, which is the desired result. □

In the following, we prove some lemmas about unification. First, we define the preconditions of a unification problem.

Definition 6.11 (Well-typed unification problem)**.** *Let $\mathcal{U}$ denote the unification problem $\Gamma; \Delta, \zeta \vdash [\vec{u} = \vec{v} : \Theta]$. We say that $\mathcal{U}$ is well-typed if the following conditions hold:*

– *$\zeta \subseteq \mathrm{dom}\,(\Delta)$, and*

– *$\Gamma\Delta \vdash \vec{u}, \vec{v} : \Theta$.*

The following two lemmas formalize the unification invariants mentioned in Sect. 6.3. In the case of positive success, the result is a unifier substitution, and a context that contains the substitution as definitions.

Lemma 6.12. *Let $\Gamma; \Delta, \zeta \vdash [\vec{u_1} = \vec{u_2} : \Theta]$ be a well-typed unification problem. If $\Gamma; \Delta, \zeta \vdash [\vec{u_1} = \vec{u_2} : \Theta] \mapsto \Delta', \zeta' \vdash \sigma$, then the following holds:*

i. $\Delta' \leq \Delta$,

ii. $\mathsf{WF}(\Gamma\Delta')$,

iii. $\mathrm{dom}\,(\Delta') = \mathrm{dom}\,(\Delta)$,

iv. $\Gamma \vdash \sigma : \Delta \to \Delta'$,

v. $\Gamma\Delta' \vdash \vec{u_1} \approx \vec{u_2}$,

vi. $\Gamma\Delta \vdash \vec{u_1}\sigma \approx \vec{u_2}\sigma$

Proof. We proceed by induction on the derivation of the unification judgment. The case for rules (u-varl), (u-varr), and (u-conv) are easy. □

Note that conditions (i.) and (ii.), together with Lemma 6.7, imply that $\Gamma\Delta' \vdash \vec{u_1} : \Theta$ and $\Gamma\Delta' \vdash \vec{u_2} : \Theta$.

The following lemma states that the result of the unification is a most general unifier.

Lemma 6.13. *Let* $\Gamma; \Delta, \zeta \vdash [\vec{u_1} = \vec{u_2} : \Theta] \mapsto \Delta', \zeta' \vdash \sigma$*, and* $\delta : \Gamma\Delta \to \Gamma$*, such that* $\Gamma \vdash \vec{u_1}\delta \approx \vec{u_2}\delta$*. Then there exists* $\delta' : \Gamma\Delta' \to \Gamma$*, such that* $\Gamma \vdash \delta \approx \sigma\delta'$*.*

If unification succeeds negatively, then the terms are not convertible.

Lemma 6.14. *If* $\Gamma; \Delta, \zeta \vdash [\vec{u_1} = \vec{u_2} : \Theta] \mapsto \bot$*, then there exists no* $\Gamma \vdash \delta : \Delta$ *such that* $\Gamma \vdash \vec{u_1}\delta \approx \vec{u_2}\delta$*.*

Using the previous lemmas, we can now prove subject reduction.

Lemma 6.15 (Subject Reduction)**.** *If* $\Gamma \vdash M : T$ *and* $\Gamma \vdash M \to M'$*, then* $\Gamma \vdash M' : T$*.*

Proof. We proceed by induction on the typing derivation, and case analysis in the reduction rule. We consider only the case rule, when reducing at the head. We have

$$\frac{\begin{array}{c}\mathsf{Ind}(I[\Delta_p]^{\vec{\nu}} : \Pi\Delta_a.u := \langle C_i : \Pi\Delta_i.\mathcal{X}\,\vec{t_i}\rangle_i) \in \Sigma \\ \Gamma \vdash C_i(\vec{r^\circ}, \vec{v}) : I\,\vec{p}\,\vec{u} \qquad \Gamma \vdash \vec{u} \approx \vec{a}\,[\mathrm{dom}\,(\Theta) := \vec{q}] \qquad \Gamma\Theta(x : I\,(\vec{p}, \vec{a})) \vdash P : s \\ \Gamma \vdash \vec{q} : \Theta \qquad \Gamma; (\vec{z_i} : {\Delta_i}^*); \Theta; [\vec{t_i}^* = \vec{a} : {\Delta_a}^*] \vdash b_i : P\left[x := C_i(\vec{p^\circ}, \vec{z_i})\right]\end{array}}{\Gamma \vdash \left(\begin{array}{l}\mathsf{case}_P\ x := C_i(\vec{r^\circ}, \vec{v})\ \mathsf{in}\ [|\Theta|]\,I\,(|\vec{p}|, |\vec{a}|) \\ \mathsf{where}\ \Theta := \vec{q}\ \mathsf{of}\ \{C_i\ \vec{z_i} \Rightarrow b_i\}_i\end{array}\right) : P\,[\mathrm{dom}\,(\Theta) := \vec{q}]\,\left[x := C_i(\vec{r^\circ}, \vec{v})\right]}$$

and the reduction

$$\Gamma \vdash \left(\begin{array}{l}\mathsf{case}_P\ x := C_i(\vec{r^\circ}, \vec{v})\ \mathsf{in}\ [|\Theta|]\,I\,(|\vec{p}|, |\vec{a}|) \\ \mathsf{where}\ \Theta := \vec{q}\ \mathsf{of}\ \{C_i\ \vec{z_i} \Rightarrow b_i\}_i\end{array}\right) \to N_i\,[\vec{z_i} := \vec{v}]$$

where $b_i \equiv N_i\ \mathsf{where}\ \vec{d_i}$.

For checking the ith-branch, we have the following judgment:

$$\frac{\begin{array}{c}\Gamma; (\vec{z_i} : {\Delta_i}^*)\Theta, \mathrm{dom}\left(\vec{d_i}\right) \cup \mathrm{dom}\,(\Theta) \vdash [\vec{t_i}^* = \vec{a} : {\Delta_a}^*] \mapsto \Theta', \zeta \vdash \sigma_i \\ \Gamma\Theta' \vdash N_i : P\,[x := C_i(\vec{p}, \vec{z_i})] \qquad \Gamma\Theta' \vdash \vec{d_i} \leq \sigma_i\end{array}}{\Gamma; (\vec{z_i} : {\Delta_i}^*); \Theta; [\vec{t_i}^* = \vec{a} : {\Delta_a}^*] \vdash N_i\ \mathsf{where}\ \vec{d_i} : P\,[x := C_i(\vec{p}, \vec{z_i})]}$$

By inverting the typing derivation of $C_i(\vec{r^\circ}, \vec{v})$, we obtain $\Gamma \vdash \vec{r^\circ} \approx |\vec{p}|$ and $\Gamma \vdash \vec{u} \approx \vec{t_i}^*\,[\vec{z_i} := \vec{v}]$. Hence, $\Gamma \vdash \vec{t_i}^*\,[\vec{z_i} := \vec{v}] \approx \vec{a}\,[\mathrm{dom}\,(\Theta) := \vec{q}]$. Consider the substitution ρ with domain $\mathrm{dom}\,({\Delta_i}^*) \cup \mathrm{dom}\,(\Theta)$ defined by:

$$\rho(x) = \begin{cases} v_j & \text{if } x = {\Delta_i}^*(j) \\ q_j & \text{if } x = \Theta(j) \end{cases}$$

It is not difficult to see that $\Gamma \vdash \rho : {\Delta_i}^*\Theta$. We have then, $\Gamma \vdash \vec{t_i}^*\rho \approx \vec{a}\rho$. By Lemma 6.13, there exists $\Gamma \vdash \rho' : \Theta'$ such that $\rho \approx \sigma_i\rho'$.

We have the valid judgment

$$\Gamma\Theta' \vdash N_i : P\,[x := C_i(\vec{p}, \vec{z_i})]$$

and a substitution $\rho' : \Gamma\Theta' \to \Gamma$, therefore

$$\Gamma \vdash N_i\rho' : P\,[x := C_i(\vec{p}, \vec{z_i})]\,\rho'$$

But ρ' contains all the definitions in σ_i and ρ. So we can prove that $P\,[x := C_i(\vec{p}, \vec{z_i})]\,\rho' \approx P\,[x := C_i(\vec{p}, \vec{z_i})]\,\rho \equiv P\,[\Theta := \vec{q}]\,\left[x := C_i(\vec{r^\circ}, \vec{v})\right]$. And that $N_i\rho' \approx N_i\rho \equiv N_i\,[\vec{z_i} := \vec{v}]$.

Put in another way. There is a context Θ'', with $\mathrm{dom}\,(\Theta'') = \mathrm{dom}\,({\Delta_i}^*) \cup \mathrm{dom}\,(\Theta)$, such that all definitions of ρ are in Θ''. It satisfies $\Gamma\Theta' \subseteq \Gamma\Theta''$, therefore

$$\Gamma\Theta'' \vdash N_i : P\,[x := C_i(\vec{p}, \vec{z_i})]$$

And then

$$\Gamma \vdash N_i\rho : P\,[x := C_i(\vec{p}, \vec{z_i})]\,\rho$$

which is the desired result.

□

6.6 From $\mathrm{CIC}^{\widehat{}}_{\mathrm{PM}}$ to $\mathrm{CIC}^{\widehat{}}_{=}$ extended with heterogeneous equality

In this section we show a type-preserving translation of $\mathrm{CIC}^{\widehat{}}_{\mathrm{PM}}$ to $\mathrm{CIC}^{\widehat{}}_{=}$ extended with heterogeneous equality as presented in Sect. 5.2. As a consequence, we can prove LC of $\mathrm{CIC}^{\widehat{}}_{\mathrm{PM}}$, relative to LC of the extension of $\mathrm{CIC}^{\widehat{}}_{=}$. The objective is to define a translation function

$$[\![.]\!] : \mathcal{T}_{\mathrm{CIC}^{\widehat{}}_{\mathrm{PM}}} \to \mathcal{T}_{\mathrm{CIC}^{\widehat{}}_{=}}$$

that takes a well-typed term in $\mathrm{CIC}^{\widehat{}}_{\mathrm{PM}}$ and returns a well-typed term in $\mathrm{CIC}^{\widehat{}}_{=}$.

Miscellaneous constructions.

For the translation we will make use of the following functions.

Injection. Let M be a term such that $\Gamma \vdash M : \mathsf{heq}(T, (C(\vec{p^\circ}, \vec{u})), T, (C(\vec{q^\circ}, \vec{v})))$, where C is the j-th constructor of the inductive type $\mathsf{Ind}(I[\Delta_p]^{\vec{r}} : A := \{C_i : \Pi\Delta_i.\mathcal{X}\,\vec{t_i}\})$. Then $\textsc{Injection}_C(M)$ is a sequence of terms, such that $\Gamma \vdash \textsc{Injection}_C(M) : \mathsf{heq}_{\Delta_j\left[\mathrm{dom}(\Delta_p) := \vec{p^\circ}\right]}(\vec{u}, \vec{v})$.

Furthermore, if $M \equiv \mathsf{heq\_refl}(T^\circ, N)$, for some T° and N, then $\textsc{Injection}_C(M) \to^* \vec{P}$, where each term in $\vec{P}$ is of the form $\mathsf{heq\_refl}(T'^\circ, N')$.

Discriminate. Let H be a term such that $\Gamma \vdash H : \mathsf{heq}(T, (C(\vec{p^\circ}, \vec{u})), T, (C'(\vec{q^\circ}, \vec{v})))$, where C_i and C_j are different constructors of an inductive type I. Then $\textsc{Discriminate}_{C,C'}(H)$ is a term such that $\Gamma \vdash \textsc{Discriminate}_{C,C'}(H) : \mathsf{False}$.

Rewrite. Let $\vec{M}$ be a sequence of terms such that $\Gamma \vdash \vec{M} : \Delta$, and $\vec{H}$ be a sequence of terms such that $\Gamma \vdash \vec{H} : \mathsf{heq}_\Theta(\vec{x}, \vec{P})$. (Note that the lhs of the equalities is composed of variables.) Then $\textsc{Rewrite}_{\vec{H}}(\vec{M})$ is a sequence of terms such that $\Gamma \vdash \textsc{Rewrite}_{\vec{H}}(\vec{M}) : \Delta\,[\vec{x} := \vec{p}]$.

Let $\vec{N}$ be a sequence of terms such that $\Gamma \vdash \vec{N} : \Delta\,[\vec{x} := \vec{p}]$, with $\Gamma \vdash \Delta$, and $\vec{H}$ a sequence of terms such that $\Gamma \vdash \vec{H} : \mathsf{heq}_\Theta(\vec{x}, \vec{p})$. Then $\textsc{Rewrite}^{\leftarrow}_{\vec{H}}(\vec{N})$ is a sequence of terms such that $\Gamma \vdash \textsc{Rewrite}^{\leftarrow}_{\vec{H}}(\vec{N}) : \Delta$.

Furthermore, if each term in $\vec{H}$ is of the form $\mathsf{heq\_refl}(T^\circ, P)$ for some T° and P, then $\textsc{Rewrite}_{\vec{H}}(\vec{M}) \to^* \vec{M}$ and $\textsc{Rewrite}^{\leftarrow}_{\vec{H}}(\vec{N}) \to^* \vec{N}$.

The definition of these constructions can be found in [59,61]. Although in a different framework, the definitions can be adapted to CIC.

6.6.1 Translation Function

The translation is described by a partial function on terms and contexts. It is partial in the sense that the translation of a unification problem is not defined if the problem is too difficult. The translation function is denoted by $[\![\ ]\!]$, and is defined by induction on the structure of terms and contexts.

The translation satisfies the following properties:

I. Stability under substitutions: for all M, N, x, $[\![M\,[x := N]]\!] \to^* [\![M]\!]\,[x := [\![N]\!]]$ (if both $[\![M]\!]$ and $[\![N]\!]$ are defined).

II. Stability under reductions: if $M \to N$, then $[\![M]\!] \to^+ [\![N]\!]$ (if both $[\![M]\!]$ and $[\![N]\!]$ are defined).

III. Preservation of well-typed terms: if $\Gamma \vdash M : T$, then $[\![\Gamma]\!]$, $[\![M]\!]$, and $[\![N]\!]$ are defined, and there exists T' such that $[\![\Gamma]\!] \vdash [\![M]\!] : T'$, with $[\![T]\!] \to^* T'$.

The translation of unification problems is not invariant under substitution, but only invariant modulo reduction. For this reason, the statements above show stability and preservation modulo reduction.

We define the translation of unification (needed to define the translation of the case construction). An *untyped unification problem* consists in a set of variables and a sequence of equations. It is denoted by $\zeta, [\vec{u} = \vec{v} : \Theta]$. We use $\mathcal{U}$ to denote untyped unification problems. The translation of an untyped unification problem $\mathcal{U}$ is a partial function that takes a sequence of terms $\vec{H}$ and returns one of two possible answers: either a pair formed by a sequence of terms and a pre-substitution (corresponding to positive sucess), or a single term (corresponding to negative success). It is denoted by $[\![\mathcal{U}]\!](\vec{H})$.

The meaning of the translation is given by the following statement of preservation of well-typed unification. Given a well-typed unification problem $\mathcal{U} :: \Gamma; \Delta, \zeta \vdash [\vec{u} = \vec{v} : \Theta]$ and a sequence of terms $\vec{H}$ such that $[\![\Gamma\Delta]\!] \vdash \vec{H} : \mathsf{heq}_{[\![\Theta]\!]}([\![\vec{u}]\!], [\![\vec{v}]\!])$, the following holds:

- if $\mathcal{U} \mapsto \Delta', \zeta \vdash \sigma$, then $[\![\zeta, [\vec{u} = \vec{v} : \Theta]]\!](\vec{H})$ is defined and returns a pair $(\vec{H'}, \sigma')$ satisfying $\sigma' \equiv \sigma$ and

$$[\![\Gamma\Delta]\!] \vdash \vec{H'} : (\mathsf{heq}(x_1, T_1)) \ldots (\mathsf{heq}(x_n, T_n))$$

  where $\sigma' = \{x_1 \mapsto T_1, \ldots, x_n \mapsto T_n\}$. Furthermore, if all the terms $\vec{M}$ are of the form $\mathsf{heq\_refl}(T^\circ, N)$, for some T° and N, then all the terms in $\vec{H'}$ reduce to terms of the form $\mathsf{heq\_refl}(T'^\circ, N')$, for some T'° and N'.
- If $\mathcal{U} \mapsto \bot$, then $[\![\zeta, [\vec{u} = \vec{v} : \Theta]]\!](\vec{H})$ is defined and returns a term H' satisfying

$$[\![\Gamma\Delta]\!] \vdash H' : \mathsf{False}$$

Translation of terms and contexts. The definition of the translation function proceeds by induction on the structure of terms and contexts. For constructions other than case

analysis, it is defined by the following rules:

$$\begin{aligned}
[\![[]]\!] &= [] \\
[\![(x:T)\Delta]\!] &= (x:[\![T]\!])[\![\Delta]\!] \\
[\![x]\!] &= x \\
[\![u]\!] &= u \\
[\![\lambda x:T^\circ.M]\!] &= \lambda x:[\![T^\circ]\!].[\![M]\!] \\
[\![M\ N]\!] &= [\![M]\!]\ [\![N]\!] \\
[\![\Pi x:T.U]\!] &= \Pi x:[\![T]\!].[\![U]\!] \\
[\![C(\vec{p^\circ},\vec{a})]\!] &= C([\![\vec{p^\circ}]\!],[\![\vec{a}]\!]) \\
[\![I^s\ (\vec{p},\vec{a})]\!] &= I^s\ ([\![\vec{p}]\!],[\![\vec{a}]\!]) \\
[\![\mathsf{fix}_n\ f:T^\star := M]\!] &= \mathsf{fix}_n\ f:[\![T^\star]\!] := [\![M]\!]
\end{aligned}$$

We concentrate now on the most interesting construction, case analysis. Consider a term of the form (we assume that is well typed in context Γ)

$$\begin{array}{l}
\mathsf{case}_P\ x := v\ \mathsf{in}\ [\Delta]\, I\left(\vec{p},\vec{t}\right)\ \mathsf{where}\ \Delta := \vec{q}\ \mathsf{of} \\
\ldots\ C_i\ \vec{z_i} \Rightarrow b_i\ \ldots \\
\mathsf{end}\ .
\end{array}$$

This term has type $P\,[\mathrm{dom}\,(\Delta) := \vec{q}]\ [x := v]$. The translation is the term

$$\begin{array}{l}
\mathsf{case}_{\Pi[\![\Delta]\!].\mathsf{heq}_{[\![\Delta_a{}^*]\!]}(\vec{y},[\![\vec{t}]\!])\to\Pi(x:I\left([\![\vec{p}]\!],[\![\vec{t}]\!]\right)).\mathsf{heq}(x,z)\to[\![P]\!]}z := [\![v]\!]\ \mathsf{in}\ I\ ([\![\vec{p}]\!],\vec{y})\ \mathsf{of} \\
\ldots\ C_i \Rightarrow [\![b_i]\!]\ \ldots \\
\mathsf{end}\ [\![\vec{q}]\!]\ [\![v]\!]\ (\mathsf{heq_refl}_{[\![\vec{\Delta_a{}^*}]\!]}\ [\![\vec{t}[\mathrm{dom}\,(\Delta) := \vec{q}]]\!])\ (\mathsf{heq_refl}\ [\![v]\!])
\end{array}$$

where $\mathsf{Ind}(I[\Delta_p] : \Pi\Delta_a.\mathsf{Type}_k := \langle C_i : T_i\rangle_i) \in \Sigma$, and $\Delta_a^* = \Delta_a\,[\mathrm{dom}\,(\Delta_p) := \vec{p}]$. (In the following, we write X^* to mean $X\,[\mathrm{dom}\,(\Delta_p) := \vec{p}]$, where X is either a term or a context.)

Let us explain the intuition behind this translation. Recall that for each branch of a case construction, we consider a unification problem between the indices of the argument and the indices of the constructor. The unification equations are translated to equalities. The translation of the unification algorithm transforms those equalities into equalities that represent the substitution (positive success) or a term showing that the original equalities are inconsistent (negative success).

The return type in the translation of the case introduces the equalities that correspond to the equations in unification:

$$\Pi[\![\Delta]\!].\mathsf{heq}_{[\![\Delta_a{}^*]\!]}(\vec{y},[\![\vec{t}]\!]) \to \Pi(x:I\left([\![\vec{p}]\!],[\![\vec{t}]\!]\right)).\mathsf{heq}(x,z) \to [\![P]\!]$$

We introduce the equalities between indices ($\mathsf{heq}(\vec{y},[\![\vec{t}]\!])$): the sequence $\vec{y}$ is replaced by the indices for each constructor. We also generalize the argument of the case and introduce an equality ($\mathsf{heq}(x,z)$). Note that this equality is heterogeneous: x has type $I\left([\![\vec{p}]\!],[\![\vec{t}]\!]\right)$, while z has type $I\,([\![\vec{p}]\!],\vec{y})$. Finally, the case is applied to the relevant arguments. Note that all the introduced equalities become reflexive. The resulting term has type $[\![P]\!]\,[\mathrm{dom}\,(\Delta) := [\![\vec{q}]\!]]\ [x := [\![v]\!]]$. By stability of substitution, it is obtained by reducing $[\![P\,[\mathrm{dom}\,(\Delta) := \vec{q}]\ [x := v]]\!]$, as desired.

The translation of the branches depends on the type of branch. Consider an impossible branch b_i (i.e., $b_i = \bot$). The unification problem associated with this branch is

$$\Gamma; \Delta(\vec{z_i} : {\Delta_i}^*), \zeta_i \vdash [\vec{u_i}^* = \vec{t} : {\Delta_a}^*]$$

where $\zeta_i = \mathrm{dom}\,(\vec{z_i}) \cup \mathrm{dom}\,(\Theta)$. Since it is an impossible branch, the unification returns negative success. Let $\mathcal{U}_i$ be the corresponding untyped unification problem $\zeta_i \vdash [\vec{u_i}^* = \vec{t} : {\Delta_a}^*]$. The translation of the branch is given by

$$\begin{array}{rcl} [\![b_i]\!] & \equiv & \lambda(\vec{w_i} : [\![{\Delta_i}^*]\!])[\![\Delta]\!](\vec{H} : \mathsf{heq}_{[\![{\Delta_a}^*]\!]}([\![\vec{u_I^i}^*]\!], [\![\vec{t}]\!]))(x : I\left([\![\vec{p}]\!], [\![\vec{t}]\!]\right))(\_ : \mathsf{heq}(x, C_i(|[\![\vec{p}]\!]|, \vec{w_i}))). \\ & & \quad \mathsf{elimFalse}\ ([\![\mathcal{U}_i]\!](\vec{H}))\ [\![P]\!] \end{array}$$

We introduce the necessary abstractions (including the arguments of the constructor). The translation of the unification (if defined) is a term that has type False (since the unification returns negative success). We use elimination on this term to obtain a term of the desired type ($[\![P]\!]$ in this case).

Finally we consider branches of the form t_i where σ_i. This is the most complicated part of the translation, so we proceed in a goal-directed way using a notation similar to the proof mode of Coq.

For this kind of branch, the unification problem associated is

$$\Gamma; \Delta(\vec{z_i} : {\Delta_i}^*), \mathrm{dom}\,(\sigma_i) \vdash [\vec{u_i}^* = \vec{t} : {\Delta_a}^*]$$

Let $\mathcal{U}_i$ be the untyped unification problem $\zeta_i \vdash [\vec{u_i}^* = \vec{t} : {\Delta_a}^*]$. The translation of the branch needs to satisfy the judgment

$$[\![\Gamma]\!] \vdash [\![b_i]\!] : \Pi(\vec{w_i} : {\Delta_i}^*)[\![\Delta]\!].\mathsf{heq}_{[\![{\Delta_a}^*]\!]}([\![\overrightarrow{u_i[\vec{w_i}]}^*]\!], [\![\vec{t}]\!]) \to \Pi(x : I\left([\![\vec{p}]\!], [\![\vec{t}]\!]\right)).\mathsf{heq}(x, (C_i(\_, \vec{w_i}))) \to [\![P]\!]$$

Written in a different way, we want to find a term $?_1$ satisfying the following judgment:

$$\frac{[\![\Gamma]\!]}{?_1 : \Pi(\vec{w_i} : {\Delta_i}^*)[\![\Delta]\!].\mathsf{heq}_{[\![{\Delta_a}^*]\!]}([\![\overrightarrow{u_i[\vec{w_i}]}^*]\!], [\![\vec{t}]\!]) \to \Pi(x : I\left([\![\vec{p}]\!], [\![\vec{t}]\!]\right)).\mathsf{heq}(x, (C_i(|[\![\vec{p}]\!]|, \vec{w_i}))) \to [\![P]\!]}$$

We take $?_1 \equiv \lambda(\vec{w_i} : {\Delta_i}^*)[\![\Delta]\!](\vec{H} : \mathsf{heq}_{[\![{\Delta_a}^*]\!]}([\![\overrightarrow{u_i[\vec{w_i}]}^*]\!], [\![\vec{t}]\!])).?_2$. Then $?_2$ needs to satisfy

$$\frac{\begin{array}{l}[\![\Gamma]\!] \\ (\vec{w_i} : {\Delta_i}^*) \\ [\![\Delta]\!] \\ (\vec{H} : \mathsf{heq}_{[\![{\Delta_a}^*]\!]}([\![\overrightarrow{u_i[\vec{w_i}]}^*]\!], [\![\vec{t}]\!]))\end{array}}{?_2 : \Pi(x : I\left([\![\vec{p}]\!], [\![\vec{t}]\!]\right)).\mathsf{heq}(x, (C_i(|[\![\vec{p}]\!]|, \vec{w_i}))) \to [\![P]\!]}$$

In this case, for the unification problem $\mathcal{U}_i$, the translation is a sequence of terms whose types are the equalities contained in σ_i. We can rewrite using the translation of $\mathcal{U}_i$ applied to

the equalities given in $\vec{H}$. Take $?_2 \equiv \textsc{Rewrite}^{\leftarrow}_{[\![\mathcal{U}_i]\!](\vec{H})}(?_3)$. Then $?_3$ needs to satisfy

$$\begin{array}{l}
[\![\Gamma]\!] \\
(\vec{w_i} : {\Delta_i}^*) \\
[\![\Delta]\!] \\
(\vec{H} : \mathsf{heq}_{[\![{\Delta_a}^*]\!]}(\overrightarrow{[\![u_i[\vec{w_i}]^*]\!]}, [\![\vec{t}]\!]))
\end{array}$$

$$?_3 : \Pi(x : I\left([\![\vec{p}]\!], [\![\vec{t}\sigma_i]\!]\right)).\mathsf{heq}(x, (C_i(|[\![\vec{p}]\!]|, \vec{w_i}\sigma_i))) \to [\![P\left[\mathrm{dom}\left(\Delta\right) := \mathrm{dom}\left(\Delta\right)\sigma_i\right]]\!]$$

Now we abstract over the first two hypotheses. Take

$$?_3 \equiv \lambda(x : I\left([\![\vec{p}]\!], [\![\vec{t}\sigma_i]\!]\right))(H_x : \mathsf{heq}(x, (C_i(|[\![\vec{p}]\!]|, \vec{w_i}\sigma_i)))).?_4$$

Then $?_4$ needs to satisfy

$$\begin{array}{l}
[\![\Gamma]\!] \\
(\vec{w_i} : {\Delta_i}^*) \\
[\![\Delta]\!] \\
(\vec{H} : \mathsf{heq}_{[\![{\Delta_a}^*]\!]}(\overrightarrow{[\![u_i[\vec{w_i}]^*]\!]}, [\![\vec{t}]\!])) \\
(x : I\left([\![\vec{p}]\!], [\![\vec{t}\sigma_i]\!]\right)) \\
(H_x : \mathsf{heq}(x, (C_i(|[\![\vec{p}]\!]|, \vec{w_i}\sigma_i))))
\end{array}$$

$$?_4 : [\![P\left[\mathrm{dom}\left(\Delta\right) := \mathrm{dom}\left(\Delta\right)\sigma_i\right]]\!]$$

We rewrite using H_x. Take $?_4 \equiv \textsc{Rewrite}^{\leftarrow}_{H_x}(?_5)$. Then $?_5$ needs to satisfy

$$\begin{array}{l}
[\![\Gamma]\!] \\
(\vec{w_i} : {\Delta_i}^*) \\
[\![\Delta]\!] \\
(\vec{H} : \mathsf{heq}_{[\![{\Delta_a}^*]\!]}(\overrightarrow{[\![u_i[\vec{w_i}]^*]\!]}, [\![\vec{t}]\!])) \\
(x : I\left([\![\vec{p}]\!], [\![\vec{t}\sigma_i]\!]\right)) \\
(H_x : \mathsf{heq}(x, (C_i(|[\![\vec{p}]\!]|, \vec{w_i}\sigma_i))))
\end{array}$$

$$?_5 : [\![P\left[\mathrm{dom}\left(\Delta\right) := \mathrm{dom}\left(\Delta\right)\sigma_i\right]\left[x := C_i(|[\![\vec{p}]\!]|, \vec{w_i}\sigma_i)\right]]\!]$$

From the typing rule of a possible branch (rule (b-sub)), we have that the body t_i has type $P\left[x := C_i(|\vec{p_i}|, \vec{z_i})\right]$ in a context containing the definitions given in σ_i. Hence, $t_i\sigma_i$ has type $P\sigma_i\left[x := C_i(|\vec{p_i}|, \vec{z_i}\sigma_i)\right]$. Since $\mathrm{FV}(P) \cap \mathrm{dom}\left(\sigma_i\right) \subseteq \mathrm{dom}\left(\Delta\right)$, this type is equal to $P\left[\mathrm{dom}\left(\Delta\right) := \mathrm{dom}\left(\Delta\right)\sigma_i\right]\left[x := C_i(|\vec{p_i}|, \vec{z_i}\sigma_i)\right]$. Therefore, we can take $?_5 \equiv [\![t_i\sigma_i\left[\vec{z_i} := \vec{w_i}\right]]\!]$, which has the correct type.

Translating unification. To complete the definition of the translation, we define the translation of an untyped unification problem. It is a partial function that takes an untyped unification problem and a sequence of terms and returns either a pair formed by a sequence of

terms and a pre-substitution, or a term. The definition of the translation follows the same pattern as the rules given in Fig. 6.2.

Let $\mathcal{U}$ be the unification problem $\zeta \vdash [\vec{u} = \vec{v} : \Theta]$, and $\vec{H}$ a sequence of terms such that $\#\vec{H} = \#\vec{u} = \#\vec{v}$. We define $[\![\mathcal{U}]\!]$ by analysis on the pair $\langle \vec{u}, \vec{v} \rangle$ (rules are tried in order):

1. If $\langle \vec{u}, \vec{v} \rangle = \langle x, N \rangle$ with $x \in \zeta$, then $[\![\mathcal{U}]\!](H) = \langle H, \{x \mapsto N\} \rangle$.
2. If $\langle \vec{u}, \vec{v} \rangle = \langle N, x \rangle$ with $x \in \zeta$, then $[\![\mathcal{U}]\!](H) = \langle H, \{x \mapsto N\} \rangle$.
3. If $\langle \vec{u}, \vec{v} \rangle = \langle C_1(\_, \vec{u_1}), C_2(\_, \vec{v_1}) \rangle$ with $C_1 \neq C_2$, then $[\![\mathcal{U}]\!](H) = \textsc{Discriminate}_{C_1,C_2}(H)$ (negative success).
4. If $\langle \vec{u}, \vec{v} \rangle = \langle C(\vec{p^\diamond}, \vec{u_1}), C(\_, \vec{v_1}) \rangle$, let $\mathcal{U}' = \zeta \vdash [\vec{u_1} = \vec{v_1} : \Theta_1]$, where Θ_1 is the context of arguments of C applied to $\vec{p^\diamond}$, and $\vec{H'} = \textsc{Injection}_C(H)$. If $[\![\mathcal{U}']\!](\vec{H'}) = H_0$, then $[\![\mathcal{U}]\!](H) = H_0$. If $[\![\mathcal{U}']\!](\vec{H'}) = (\vec{H_0}, \emptyset)$, then $[\![\mathcal{U}]\!](H) = (\varepsilon, \emptyset)$. Otherwise, if $[\![\mathcal{U}']\!](\vec{H'}) = (\vec{H_0}, \sigma)$, with $\sigma \neq \emptyset$, then $[\![\mathcal{U}]\!](H) = (\vec{H_0}, \sigma)$.
5. If $\langle \vec{u}, \vec{v} \rangle = \langle u_1, u_2 \rangle$ and $u_1 \approx u_2$, then $[\![U]\!](H) = \langle \varepsilon, \mathrm{id} \rangle$ (in this case, the above rules do not apply).
6. Otherwise $\langle \vec{u}, \vec{v} \rangle = \langle (u, \vec{u}), (v, \vec{v}) \rangle$. Then $\vec{H} = H, \vec{H'}$, and $\theta = (x : T)\Theta'$ Let $\langle \vec{H_1}, \sigma_1 \rangle = [\![\mathcal{U}_1]\!](H)$, where $\mathcal{U}_1 = \zeta \vdash [u = v : T]$. Let $\vec{H'} = \textsc{Rewrite}_{\vec{H_1}}(\textsc{Rewrite}_H(\vec{H}))$, and $\langle \vec{H_2}, \sigma_2 \rangle = [\![\mathcal{U}_2]\!](\vec{H'})$, where $\mathcal{U}_2 = \zeta_1 \setminus \mathrm{dom}\,(\sigma_1) \vdash [\vec{u}\sigma_1 = \vec{v}\sigma_2 : \theta'\,[x := u]]$. Finally, $[\![\mathcal{U}]\!](\vec{H}) = \langle \textsc{Rewrite}_{\vec{H_2}}(\vec{H_1})\,\vec{H_2}, \sigma_1\sigma_2 \rangle$.

Note that in the final case, we assume that both recursive translations return a pair. If one of the recursive calls returns a term, the result of the whole translation is the same term. In the case 4. (corresponding to rule (u-inj)), we make a distinction between the result of the recursive call to ensure that the translation is invariant under substitution.

Logical Consistency of CIC$^{\widehat{}}_{\mathrm{PM}}$. We prove that the translation preserves well-typed terms. Logical Consistency follows as corollary.

First we prove stability under substitution and stability under reduction of the translation of unification. Substitution of free variables for an untyped unification problem is defined as expected. Let $\mathcal{U} = \zeta \vdash [\vec{u} = \vec{v} : \Theta]$, $x \notin \zeta$ a variable, and N a term. Then $\mathcal{U}\,[x := N]$ is defined as $\zeta \vdash [\vec{u}\,[x := N] = \vec{v}\,[x := N] : \Theta\,[x := N]]$.

Lemma 6.16. *Let $\mathcal{U} = \zeta \vdash [\vec{u} = \vec{v} : \Theta]$ and $x \notin \zeta$. If $[\![\mathcal{U}]\!]$, $[\![N]\!]$, and $[\![\mathcal{U}\,[x := N]]\!]$ are defined, then $[\![\mathcal{U}\,[x := N]]\!] \equiv [\![\mathcal{U}]\!]\,[x := [\![N]\!]]$.*

Proof. We proceed by induction on the size of $\mathcal{U}$ (defined as the sum of the sizes of the terms in the equations). We use the fact that the miscellaneous constructions Injection, Discriminate, and Rewrite are stable under substitution. □

The translation of unification is stable under reductions. Reduction of an untyped unification problem is defined as reduction in any of the terms of the equations.

Lemma 6.17. *Let $\mathcal{U}$ and $\mathcal{U}'$ be untyped unification problems. If $[\![\mathcal{U}]\!]$ and $[\![\mathcal{U}']\!]$ are defined, then $[\![\mathcal{U}]\!] \to^* [\![U']\!]$.*

The translation of the unification judgment satisfies the properties stated in the following. If a unification succeeds positively, then the result of the translation is a sequence of terms whose types are the equalities given in the substitution.

Lemma 6.18. *Let* $\Gamma; \Delta, \zeta \vdash [\vec{u} = \vec{v} : \Theta] \mapsto \Delta', \zeta' \vdash \sigma$ *be a valid judgment. Assume that* $\Gamma\Delta \vdash \vec{u}, \vec{v} : \Theta$*. Let* Γ_0 *be a context and* $\vec{H}$ *be a sequence of terms (of CIC$\widehat{\ }_{-}$) such that* $\Gamma_0 \vdash_{-} \vec{H} : [\![\mathsf{heq}_\Theta(\vec{u}, \vec{v})]\!]$*.*

Then $[\![\zeta \vdash \vec{u} = \vec{v}]\!](\vec{H})$ *succeeds positively, returning a sequence of terms* $\vec{H'}$ *and* σ*. Furthermore, for each* $(x := M : T) \in \Delta'$ *with* $x \in \mathrm{dom}\,(\zeta)$*, there exists a term* H_0 *in* $\vec{H'}$ *such that* $\Gamma_0 \vdash_{-} H_0 : [\![\mathsf{heq}(T, x, T, M)]\!]$*.*

If a unification succeeds negatively, then no unifier exists, and the translation returns a proof of False.

Lemma 6.19. *Let* $\Gamma; \Delta, \zeta \vdash [\vec{u} = \vec{v} : \Theta] \mapsto \Delta', \zeta' \vdash \sigma$ *be a valid judgment. Assume that* $\Gamma\Delta \vdash \vec{u}, \vec{v} : \Theta$*. Let* Γ_0 *be a context and* $\vec{H}$ *be a sequence of terms (of CIC$\widehat{\ }_{-}$) such that* $\Gamma_0 \vdash_{-} \vec{H} : [\![\mathsf{heq}_\Theta(\vec{u}, \vec{v})]\!]$*.*

Then $[\![\zeta \vdash \vec{u} = \vec{v}]\!](\vec{H})$ *succeeds negatively, returning a term* H' *such that* $\Gamma_0 \vdash_{-} H' : \mathsf{False}$*.*

We prove similar results for the translation of terms. The following lemma states stability under substitution.

Lemma 6.20. *If* $[\![M]\!]$*,* $[\![N]\!]$*, and* $[\![M\,[x := N]]\!]$ *are defined, then* $[\![M\,[x := N]]\!] \equiv [\![M]\!]\,[x := [\![N]\!]]$*.*

Proof. By induction on the structure of terms, using Lemma 6.16. □

The following lemma state stability under reduction of the translation of terms.

Lemma 6.21. *If* $[\![M]\!]$ *and* $[\![N]\!]$ *are defined, and* $M \to N$*, then* $[\![M]\!] \to^+ N$*.*

Proof. By induction on the reduction relation, using Lemma 6.17, and the properties of the miscellaneous constructions Injection, Discriminate, and Rewrite. □

Finally, we prove that the translations towards CIC$\widehat{\ }_{-}$ extended with heterogeneous equality preserves well-typed terms.

Lemma 6.22. *If* $\Gamma \vdash M : T$*, then* $[\![\Gamma]\!]$*,* $[\![M]\!]$*, and* $[\![T]\!]$ *are defined, and* $[\![\Gamma]\!] \vdash [\![M]\!] : [\![T]\!]$*.*

Proof. By induction on the type derivation. □

A corollary of the last result is Logical Consistency of CIC$\widehat{\ }_{\mathrm{PM}}$ relative to the Logical Consistency of CIC$\widehat{\ }_{-}$ extended with heterogeneous equality.

Corollary 6.23. *There is no term* M *such that* $\vdash M : \mathsf{False}$*.*

6.7 Related Work

Pattern matching and axiom K. Coquand [30] was the first to consider the problem of pattern matching with dependent types. He already observed that the axiom K is derivable in his setting. Hofmann and Streicher [47] later proved that pattern matching with dependent types is not a conservative extension of Type Theory, by showing that K is not derivable in Type Theory. Finally, Goguen *et al.* [42] proved that pattern matching can be translated into a Type Theory with K as an axiom, showing that K is sufficient to support pattern matching — this result was already discovered by McBride [59]. Given this series of results, it is not surprising that axiom K is derivable with the rule we propose.

Epigram and Agda. Two modern presentations of Coquand's work, which are important inspirations for this work, are the programming languages Epigram [60] and Agda [69].

The pattern matching mechanism of Epigram, described by McBride and McKinna in [62], provides a way to reason by case analysis, not only on constructors, but using elimination principles. In that sense, it is more general than our approach. They also define a mechanism to perform case analysis on intermediate expressions (called the *with* construction). This is not necessary in our case, where we have a more primitive notion of pattern matching (we can simply do a case analysis on any expression). Finally, they define a simplification method based on first-order unification, that we have reformulated here.

Agda's pattern matching mechanism, described in [69], allows definitions by a sequence of (possibly overlapping) equations, and uses the *with* construct to analyze intermediate expressions, in a similar way to [62]. The first-order unification algorithm used in Agda served as basis of our own presentation. Internally, pattern matching definitions are translated in Agda to nested case definitions, which is what we directly write in our approach.

The *with* construct developed in Epigram and Agda does not increase the expressive power of those systems — internally, it is translated into more primitive expressions. However, it does provide a concise and elegant way of writing functions. In comparison, definitions written using our proposed rule are more verbose and difficult to write by hand (cf. the example on transitivity of less-or-equal in Sect. 6.4). On the other hand, since our rule handles much of the work necessary to typecheck an Agda-style definition (e.g., unification of inversion constraints, elimination of impossible cases), it should not be difficult to translate from an Agda-style definition to a nested case definition using the new rule.

Coq. The current implementation of Coq [80] provides mechanisms to define functions by pattern matching. The basic pattern-matching algorithm, initially written by Cristina Cornes and extended by Hugo Herbelin, supports omission of impossible cases by encoding the proofs of negative success of the first-order unification process within the return predicate of a case expression.

Another approach, provided by the Program construction of Matthieu Sozeau [77], allows to exploit inversion constraints using heterogeneous equality for typing dependent pattern-matching in a way similar to what is done in Epigram. The distinguishing feature of Program is that it separates the computational definition of the function, from the proof that the definition is valid. Let us illustrate this point with the vtail function. Using the Program construction we can write vtail as:

```
Program vtail A n (v : vec A (S n)) : vec n :=
  match v with
  | vcons n' x v' => v'
  end.
```

Then, the proof that the case vnil is not needed, and that v' has the correct type is done separately. Program helps the user by automatically introducing the index equalities (like in CIC$\hat{}_{-}$) and even automatically proving some cases. Then, it combines both parts and generates a term that looks like the term we presented for CIC$\hat{}_{-}$. However, because Coq lacks the reduction rule of axiom K, not all definitions built by this algorithm are computable.

The advantage of the rule we present in this chapter, is that the reasoning steps are done automatically by the type-checking algorithm. Furthermore, they are not part of the term,

which means that reduction is more efficient, since it does not have to handle all the reasoning terms, which have no computational value, but are necessary to convince the type-checker.

On the other hand, the approach of Program is more general than ours. It is particularly useful to use with *subset types* which are not covered in our approach.

More recently, Sozeau [78] introduced another mechanism for defining functions by pattern matching called Equations. In this approach, the user writes a sequence of pattern-matching equations in a similar way as it would do in Agda. Then, the definition is translated to a case tree. As with Program, index equalities are automatically introduced in order to prove impossible branches and propagate inversion constraints.

However, as in the case of Program, the term generated contains all the reasoning steps necessary to convince the type-checker of Coq that the function is total and terminating.

Both approaches, the extensions we presented in this work and Equations, pursue similar objectives, which is to provide tools that help the user in the definitions of functions in Coq. They are not conflicting, but complimentary. More precisely, the Equations approach could benefit from a more powerful kernel in Coq, since it would be possible to generate more efficient terms.

Other approaches. Oury [70] proposed a different approach to remove impossible cases based on set approximations. His approach allows the removal of cases in situations where unification is not sufficient. As mentioned in [70], it remains to be seen if the combination of both techniques can be used to remove more cases.

Chapter 7

Conclusions

In this thesis, we have presented two extensions of CIC aimed at improving the elimination rules. In the first extension we considered a type-based framework for ensuring termination of recursive definitions. We have proved several desired metatheoretical properties, including Strong Normalization and Logical Consistency. The second extension is a new case-analysis construction that eases the task of writing pattern-matching definitions on inductive families by automatically eliminating impossible cases and propagating inversion constraints.

Our motivation for this work comes from the Coq proof assistant. The guard condition currently used in Coq to check termination has several drawbacks that often appear in practice. Type-based termination mechanisms are a good candidate for replacing guard predicates in proof assistants. They are more powerful and relatively easier to implement, while more intuitive and predictable for the user.

Coq includes several tools aimed at easing the task of programming with dependent types, built on top of the core theory. We believe that improving the core theory will make it easier to develop more efficient and usable tools on the upper layers of the Coq system. Our proposed extension of the case-analysis construction build directly on previous research in the area of pattern-matching with dependent types. In our opinion, it provides a reasonable balance between the benefits for the user and the added complexity of the implementation.

Our long-term objective is to have a sound theoretical basis for a future implementation of Coq that overcomes (at least partially) the limitations we address in this work. We do not claim that the extensions we propose provide a definite answer, but we believe they form a reasonable basis for future developments, in terms of added complexities for the implementation, as well as for users.

With this objective in mind, several lines of future work can be considered. Size inference is important to hide termination checking from the user. It should not be difficult to adapt the size-inference algorithm of CIC$\hat{}$ [19] to our extension, given the similarities between both systems.

Another important aspect to consider is coinduction. We sketched an extension of CIC$\hat{}_{_}$ with streams, showing that it is possible to adapt our proof of Strong Normalization to handle coinductive types. However, a more formal development is needed to ensure that coinductive types as implemented in Coq can be adapted to our type-based termination approach. On the other hand, a more satisfiable setting for coinduction in dependent types is currently a major research topic in Type Theory.

On a more basic line of research, it could be interesting to develop the Λ-set model we

propose in a categorical setting. Also, the treatment of fixpoints and valid types for recursion is rather ad-hoc. It could be useful to study them in a more uniform way as in e.g. [2].

Bibliography

[1] Andreas Abel. *A Polymorphic Lambda-Calculus with Sized Higher-Order Types*. PhD thesis, Ludwig-Maximilians-Universität München, 2006.

[2] Andreas Abel. Semi-continuous sized types and termination. *Logical Methods in Computer Science*, 4(2), 2008.

[3] Andreas Abel. MiniAgda: Integrating sized and dependent types. In Ana Bove, Ekaterina Komendantskaya, and Milad Niqui, editors, *Workshop on Partiality And Recursion in Interative Theorem Provers (PAR 2010), Satellite Workshop of ITP'10 at FLoC 2010*, 2010.

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

[5] Peter Aczel. On relating type theories and set theories. In Thorsten Altenkirch, Wolfgang Naraschewski, and Bernhard Reus, editors, *Types for Proofs and Programs, International Workshop TYPES '98, Kloster Irsee, Germany, March 27-31, 1998, Selected Papers*, volume 1657 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 1998.

[6] Robin Adams. Pure type systems with judgemental equality. *J. Funct. Program.*, 16(2):219–246, 2006.

[7] Thorsten Altenkirch. *Constructions, Inductive Types and Strong Normalization*. PhD thesis, University of Edinburgh, November 1993.

[8] Thorsten Altenkirch, Conor McBride, and James McKinna. Why dependent types matter. Manuscript, available online, April 2005.

[9] Roberto M. Amadio, editor. *Foundations of Software Science and Computational Structures, 11th International Conference, FOSSACS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29 - April 6, 2008. Proceedings*, volume 4962 of *Lecture Notes in Computer Science*. Springer, 2008.

[10] Lennart Augustsson. Cayenne - a language with dependent types. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming, ICFP '98, Baltimore, Maryland, USA, September 27-29, 1998*, pages 239–250. PUB-ACM, September 1998.

[11] Henk 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.

[12] Henk Barendregt and Tobias Nipkow, editors. *Types for Proofs and Programs, International Workshop TYPES'93, Nijmegen, The Netherlands, May 24-28, 1993, Selected Papers*, volume 806 of *Lecture Notes in Computer Science*. Springer, 1994.

[13] Bruno Barras. *Auto-validation d'un système de preuves avec familles inductives*. Thèse de doctorat, Université Paris 7, 1999.

[14] Bruno Barras. Sets in Coq, Coq in sets. 1st Coq Workshop, August 2009.

[15] Bruno Barras and Bruno Bernardo. The implicit calculus of constructions as a programming language with dependent types. In Amadio [9], pages 365–379.

[16] Bruno Barras, Pierre Corbineau, Benjamin Grégoire, Hugo Herbelin, and Jorge Luis Sacchini. A new elimination rule for the Calculus of Inductive Constructions. In Berardi et al. [22], pages 32–48.

[17] Gilles Barthe, Maria João Frade, E. Giménez, Luis Pinto, and Tarmo Uustalu. Type-based termination of recursive definitions. *Mathematical Structures in Computer Science*, 14(1):97–141, 2004.

[18] Gilles Barthe, Benjamin Grégoire, and Fernando Pastawski. Practical inference for type-based termination in a polymorphic setting. In Pawel Urzyczyn, editor, *Typed Lambda Calculi and Applications, 7th International Conference, TLCA 2005, Nara, Japan, April 21-23, 2005, Proceedings*, volume 3461 of *Lecture Notes in Computer Science*, pages 71–85. Springer, 2005.

[19] Gilles Barthe, Benjamin Grégoire, and Fernando Pastawski. CIC$\widehat{}$: Type-based termination of recursive definitions in the Calculus of Inductive Constructions. In Hermann and Voronkov [46], pages 257–271.

[20] Gilles Barthe, Benjamin Grégoire, and Colin Riba. A tutorial on type-based termination. In Ana Bove, Luís Soares Barbosa, Alberto Pardo, and Jorge Sousa Pinto, editors, *Language Engineering and Rigorous Software Development, International LerNet ALFA Summer School 2008, Piriapolis, Uruguay, February 24 - March 1, 2008, Revised Tutorial Lectures*, volume 5520 of *Lecture Notes in Computer Science*, pages 100–152. Springer, 2008.

[21] Gilles Barthe, Benjamin Grégoire, and Colin Riba. Type-based termination with sized products. In Michael Kaminski and Simone Martini, editors, *Computer Science Logic, 22nd International Workshop, CSL 2008, 17th Annual Conference of the EACSL, Bertinoro, Italy, September 16-19, 2008. Proceedings*, volume 5213 of *Lecture Notes in Computer Science*, pages 493–507. Springer, 2008.

[22] Stefano Berardi, Ferruccio Damiani, and Ugo de'Liguoro, editors. *Types for Proofs and Programs, International Conference, TYPES 2008, Torino, Italy, March 26-29, 2008, Revised Selected Papers*, volume 5497 of *Lecture Notes in Computer Science*. Springer, 2009.

[23] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004.

[24] Yves Bertot and Ekaterina Komendantskaya. Using structural recursion for corecursion. In Berardi et al. [22], pages 220–236.

[25] Frédéric Blanqui. A type-based termination criterion for dependently-typed higher-order rewrite systems. In Vincent van Oostrom, editor, *Rewriting Techniques and Applications,*

15th International Conference, RTA 2004, Aachen, Germany, June 3-5, 2004, Proceedings, volume 3091 of *Lecture Notes in Computer Science*, pages 24–39. Springer, 2004.

[26] Frédéric Blanqui and Colin Riba. Combining typing and size constraints for checking the termination of higher-order conditional rewrite systems. In Hermann and Voronkov [46], pages 105–119.

[27] Ana Bove. General recursion in type theory. In Geuvers and Wiedijk [37], pages 39–58.

[28] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing mathematics with the Nuprl proof development system.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1986.

[29] Thierry Coquand. Metamathematical investigations of a calculus of constructions. In Piergiorgio Odifreddi, editor, *Logic and computer science*, pages 91–122. Academic Press, London, 1990.

[30] Thierry Coquand. Pattern matching with dependent types. In B. Nordström, K. Pettersson, and G. Plotkin, editors, *Informal Proceedings Workshop on Types for Proofs and Programs (Båstad, Sweden)*, 1992.

[31] Thierry Coquand. Infinite objects in type theory. In Barendregt and Nipkow [12], pages 62–78.

[32] Thierry Coquand and Gerard Huet. The calculus of constructions. *Inf. Comput.*, 76:95–120, February 1988.

[33] Thierry Coquand and Christine Paulin. Inductively defined types. In Per Martin-Löf and Grigori Mints, editors, *COLOG-88, International Conference on Computer Logic, Tallinn, USSR, December 1988, Proceedings*, volume 417 of *Lecture Notes in Computer Science*, pages 50–66. Springer, 1988.

[34] Randy Pollack *et.al.* The LEGO Proof Assistant, 2001. `http://www.dcs.ed.ac.uk/home/lego/`.

[35] Maria João Frade. *Type-Based Termination of Recursive Definitions and Constructor Subtyping in Typed Lambda Calculi.* PhD thesis, Universidade do Minho, 2003.

[36] Herman Geuvers and Mark-Jan Nederhof. Modular proof of strong normalization for the calculus of constructions. *J. Funct. Program.*, 1(2):155–189, 1991.

[37] Herman Geuvers and Freek Wiedijk, editors. *Types for Proofs and Programs, Second International Workshop, TYPES 2002, Berg en Dal, The Netherlands, April 24-28, 2002, Selected Papers*, volume 2646 of *Lecture Notes in Computer Science*. Springer, 2003.

[38] Eduardo Giménez. Codifying guarded definitions with recursive schemes. In Peter Dybjer, Bengt Nordström, and Jan M. Smith, editors, *Types for Proofs and Programs, International Workshop TYPES'94, Båstad, Sweden, June 6-10, 1994, Selected Papers*, volume 996 of *Lecture Notes in Computer Science*, pages 39–59. Springer, 1994.

[39] Eduardo Giménez. *A Calculus of Infinite Constructions and its application to the verification of communicating systems.* PhD thesis, Ecole Normale Supérieure de Lyon, 1996.

[40] Eduardo Giménez. Structural recursive definitions in type theory. In Kim Guldstrand Larsen, Sven Skyum, and Glynn Winskel, editors, *Automata, Languages and Programming, 25th International Colloquium, ICALP'98, Aalborg, Denmark, July 13-17, 1998,*

Proceedings, volume 1443 of *Lecture Notes in Computer Science*, pages 397–408. Springer, 1998.

[41] Healfdene Goguen. *A Typed Operational Semantics for Type Theory*. PhD thesis, Laboratory for Foundations of Computer Science, University of Edinburgh, 1994.

[42] Healfdene Goguen, Conor McBride, and James McKinna. Eliminating dependent pattern matching. In K. Futatsugi, J. P. Jouannaud, and J. Meseguer, editors, *Algebra, Meaning, and Computation, Essays Dedicated to Joseph A. Goguen on the Occasion of His 65th Birthday*, volume 4060 of *LNCS*. Springer, 2006.

[43] Georges Gonthier. The four colour theorem: Engineering of a formal proof. In Deepak Kapur, editor, *Computer Mathematics, 8th Asian Symposium, ASCM 2007, Singapore, December 15-17, 2007. Revised and Invited Papers*, volume 5081 of *Lecture Notes in Computer Science*, page 333. Springer, 2007.

[44] Benjamin Grégoire and Jorge Luis Sacchini. On strong normalization of the calculus of constructions with type-based termination. In Christian G. Fermüller and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 17th International Conference, LPAR-17, Yogyakarta, Indonesia, October 10-15, 2010. Proceedings*, volume 6397 of *Lecture Notes in Computer Science*, pages 333–347. Springer, 2010.

[45] Robert Harper, Furio Honsell, and Gordon D. Plotkin. A framework for defining logics. *J. ACM*, 40(1):143–184, 1993.

[46] Miki Hermann and Andrei Voronkov, editors. *Logic for Programming, Artificial Intelligence, and Reasoning, 13th International Conference, LPAR 2006, Phnom Penh, Cambodia, November 13-17, 2006, Proceedings*, volume 4246 of *Lecture Notes in Computer Science*. Springer, 2006.

[47] Martin Hofmann and Thomas Streicher. The groupoid model refutes uniqueness of identity proofs. In *Proceedings, Ninth Annual IEEE Symposium on Logic in Computer Science, 4-7 July 1994, Paris, France*, pages 208–212. IEEE Computer Society, 1994.

[48] John Hughes, Lars Pareto, and Amr Sabry. Proving the correctness of reactive systems using sized types. In *POPL*, pages 410–423, 1996.

[49] Jan Willem Klop. *Combinatory Reduction Systems*. PhD thesis, University of Utrecht, 1980.

[50] Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. In *POPL*, pages 81–92, 2001.

[51] Gyesik Lee and Benjamin Werner. A proof-irrelevant model of CIC with predicative induction and judgemental equality. Unpublished, 2010.

[52] Xavier Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

[53] Pierre Letouzey. *Programmation fonctionnelle certifiée – L'extraction de programmes dans l'assistant Coq*. PhD thesis, Université Paris-Sud, July 2004.

[54] Azriel Levy. *Basic Set Theory*. Springer-Verlag, 1979.

[55] Zhaohui Luo. A higher-order calculus and theory abstraction. *Inf. Comput.*, 90(1):107–137, 1991.

[56] Zhaohui Luo. *Computation and reasoning: a type theory for computer science*. Oxford University Press, Inc., New York, NY, USA, 1994.

[57] Lena Magnusson and Bengt Nordström. The ALF proof editor and its proof engine. In Barendregt and Nipkow [12], pages 213–237.

[58] Per Martin-Löf. An intuitionistic theory of types: Predicative part. In H. E. Rose and J. C. Shepherdson, editors, *Logic Colloquium '73, Proceedings of the Logic Colloquium*, volume 80 of *Studies in Logic and the Foundations of Mathematics*, pages 73–118. Elsevier, 1975.

[59] Conor McBride. *Dependently Typed Functional Programs and their Proofs*. PhD thesis, University of Edinburgh, 1999.

[60] Conor McBride. Epigram: Practical programming with dependent types. In V. Vene and T. Uustalu, editors, *AFP 2004, Estonia, 2004, Revised Lectures*, volume 3622 of *LNCS*. Springer, 2004.

[61] Conor McBride, Healfdene Goguen, and James McKinna. A few constructions on constructors. In Jean-Christophe Filliâtre, Christine Paulin-Mohring, and Benjamin Werner, editors, *Types for Proofs and Programs, International Workshop, TYPES 2004, Jouy-en-Josas, France, December 15-18, 2004, Revised Selected Papers*, volume 3839 of *Lecture Notes in Computer Science*, pages 186–200. Springer, 2004.

[62] Conor McBride and James McKinna. The view from the left. *J. Funct. Program.*, 14(1):69–111, 2004.

[63] Paul-André Melliès and Benjamin Werner. A generic normalisation proof for pure type systems. In Eduardo Giménez and Christine Paulin-Mohring, editors, *Types for Proofs and Programs, International Workshop TYPES'96, Aussois, France, December 15-19, 1996, Selected Papers*, volume 1512 of *LNCS*, pages 254–276. Springer, 1996.

[64] Nax Paul Mendler. Inductive types and type constraints in the second-order lambda calculus. *Ann. Pure Appl. Logic*, 51(1-2):159–172, 1991.

[65] Alexandre Miquel. *Le calcul des constructions implicite: syntaxe et sémantique*. PhD thesis, Université Paris 7, 2001.

[66] Alexandre Miquel and Benjamin Werner. The not so simple proof-irrelevant model of CC. In Geuvers and Wiedijk [37], pages 240–258.

[67] Nathan Mishra-Linger and Tim Sheard. Erasure and polymorphism in pure type systems. In Amadio [9], pages 350–364.

[68] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory. An Introduction*. Oxford University Press, 1990.

[69] Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.

[70] Nicolas Oury. Pattern matching coverage checking with dependent types using set approximations. In A. Stump and H. Xi, editors, *Proceedings of the ACM Workshop Programming Languages meets Program Verification, PLPV 2007, Freiburg, Germany, October 5, 2007*, pages 47–56. ACM, 2007.

[71] Lars Pareto. *Types for Crash Prevention*. PhD thesis, Chalmers University of Technology, 2000.

[72] Christine Paulin-Mohring. *Définitions Inductives en Théorie des Types d'Ordre Supérieur*. Habilitation à diriger les recherches, Université Claude Bernard Lyon I, December 1996.

[73] John C. Reynolds. Polymorphism is not set-theoretic. In Gilles Kahn, David B. MacQueen, and Gordon D. Plotkin, editors, *Semantics of Data Types, International Symposium, Sophia-Antipolis, France, June 27-29, 1984, Proceedings*, volume 173 of *Lecture Notes in Computer Science*, pages 145–156. Springer, 1984.

[74] Tim Sheard and Nathan Linger. Programming in Omega. In Zoltán Horváth, Rinus Plasmeijer, Anna Soós, and Viktória Zsók, editors, *Central European Functional Programming School, Second Summer School, CEFP 2007, Cluj-Napoca, Romania, June 23-30, 2007, Revised Selected Lectures*, volume 5161 of *Lecture Notes in Computer Science*, pages 158–227. Springer, 2007.

[75] Vincent Siles and Hugo Herbelin. Equality is typable in semi-full Pure Type Systems. In *Proceedings of the 25th Annual IEEE Symposium on Logic in Computer Science, LICS 2010, 11-14 July 2010, Edinburgh, United Kingdom*, pages 21–30. IEEE Computer Society, 2010.

[76] Vincent Siles and Hugo Herbelin. Pure Type Systems conversion is always typable. Submitted, 2010.

[77] Matthieu Sozeau. Subset coercions in Coq. In Thorsten Altenkirch and Conor McBride, editors, *Types for Proofs and Programs, International Workshop, TYPES 2006, Nottingham, UK, April 18-21, 2006, Revised Selected Papers*, volume 4502 of *LNCS*, pages 237–252. Springer, 2006.

[78] Matthieu Sozeau. Equations: A dependent pattern-matching compiler. In Matt Kaufmann and Lawrence C. Paulson, editors, *Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, volume 6172 of *Lecture Notes in Computer Science*, pages 419–434. Springer, 2010.

[79] Aaron Stump, Morgan Deters, Adam Petcher, Todd Schiller, and Timothy W. Simpson. Verified programming in Guru. In Thorsten Altenkirch and Todd D. Millstein, editors, *Proceedings of the 3rd ACM Workshop Programming Languages meets Program Verification, PLPV 2009, Savannah, GA, USA, January 20, 2009*, pages 49–58. ACM, 2009.

[80] The Coq Development Team. *The Coq Reference Manual, version 8.1*, February 2007. Distributed electronically at `http://coq.inria.fr/doc`.

[81] L. S. van Benthem Jutting. Typing in pure type systems. *Inf. Comput.*, 105(1):30–41, 1993.

[82] David Wahlstedt. *Dependent Type Theory with Parameterized First-Order Data Types and Well-Founded Recursion*. PhD thesis, Chalmers University of Technology, 2007. ISBN 978-91-7291-979-2.

[83] Benjamin Werner. Sets in types, types in sets. In Martín Abadi and Takayasu Ito, editors, *Theoretical Aspects of Computer Software, Third International Symposium, TACS '97, Sendai, Japan, September 23-26, 1997, Proceedings*, volume 1281 of *Lecture Notes in Computer Science*, pages 530–346. Springer, 1997.

[84] Benjamin Werner. On the strength of proof-irrelevant type theories. *Logical Methods in Computer Science*, 4(3), 2008.

[85] Hongwei Xi. Dependent Types for Program Termination Verification. *Journal of Higher-Order and Symbolic Computation*, 15:91–131, October 2002.

[86] Hongwei Xi. Applied Type System (extended abstract). In *Post-Workshop Proceedings of TYPES 2003*, pages 394–408. Springer-Verlag LNCS 3085, 2004.

Terminaison basée sur les types et filtrage dépendant pour le calcul des constructions inductives

Résumé :

Les assistants de preuve basés sur des théories des types dépendants sont de plus en plus utilisé comme un outil pour développer programmes certifiés. Un exemple réussi est l'assistant de preuves Coq, fondé sur le Calcul des Constructions Inductives (CCI). Coq est un langage de programmation fonctionnel dont un expressif système de type qui permet de préciser et de démontrer des propriétés des programmes dans une logique d'ordre supérieur.

Motivé par le succès de Coq et le désir d'améliorer sa facilité d'utilisation, dans cette thèse nous étudions certaines limitations des implémentations actuelles de Coq et sa théorie sous-jacente, CCI. Nous proposons deux extension de CCI que partiellement resourdre ces limitations et que on peut utiliser pour des futures implémentations de Coq.

Nous étudions le problème de la terminaison des fonctions récursives. En Coq, la terminaison des fonctions récursives assure la cohérence de la logique sous-jacente. Les techniques actuelles assurant la terminaison de fonctions récursives sont fondées sur des critères syntaxiques et leurs limitations apparaissent souvent dans la pratique. Nous proposons une extension de CCI en utilisant un mécanisme basé sur les type pour assurer la terminaison des fonctions récursives. Notre principale contribution est une preuve de la normalisation forte et la cohérence logique de cette extension.

Nous étudions les définitions par filtrage dans le CCI. Avec des types dépendants, il est possible d'écrire des définitions par filtrage plus précises, par rapport à des langages de programmation fonctionnels Haskell et ML. Basé sur le succès des langages de programmation avec types dépendants, comme Epigram et Agda, nous développons une extension du CCI avec des fonctions similaires.

Mots clés : Terminaison basé sur les types, types dependants, filtrage par motifs, Calcul des Constructions Inductives

On Type-Based Termination and Dependent Pattern Matching in the Calculus of Inductive Constructions

Abstract:

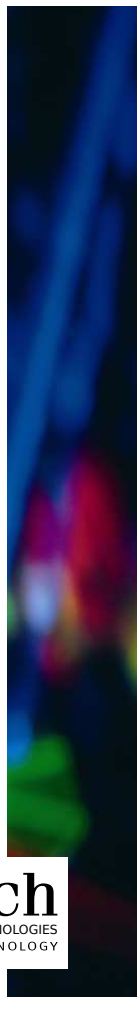
Proof assistants based on dependent type theory are progressively used as a tool to develop certified programs. A successful example is the Coq proof assistant, an implementation of a dependent type theory called the Calculus of Inductive Constructions (CIC). Coq is a functional programming language with an expressive type system that allows to specify and prove properties of programs in a higher-order predicate logic.

Motivated by the success of Coq and the desire of improving its usability, in this thesis we study some limitations of current implementations of Coq and its underlying theory, CIC. We propose two extension of CIC that partially overcome these limitations and serve as a theoretical basis for future implementations of Coq.

First, we study the problem of termination of recursive functions. In Coq, all recursive functions must be terminating, in order to ensure the consistency of the underlying logic. Current techniques for checking termination are based on syntactical criteria and their limitations appear often in practice. We propose an extension of CIC using a type-based mechanism for ensuring termination of recursive functions. Our main contribution is a proof of Strong Normalization and Logical Consistency for this extension.

Second, we study pattern-matching definitions in CIC. With dependent types it is possible to write more precise and safer definitions by pattern matching than with traditional functional programming languages such as Haskell and ML. Based on the success of dependently-typed programming languages such as Epigram and Agda, we develop an extension of CIC with similar features.

Keywords: Type-based termination, dependent types, pattern matching, Calculus of Inductive Constructions



MINES ParisTech

ParisTech
INSTITUT DES SCIENCES ET TECHNOLOGIES
PARIS INSTITUTE OF TECHNOLOGY