Constructor Subtyping in the Calculus of Inductive Constructions

Gilles Barthe[1,2] and Femke van Raamsdonk[3,4]

[1] INRIA Sophia-Antipolis, France,
Gilles.Barthe@inria.fr
[2] Departamento de Informática, Universidade do Minho, Portugal
[3] Division of Mathematics and Computer Science, Faculty of Sciences,
Vrije Universiteit, Amsterdam, The Netherlands,
femke@cs.vu.nl
[4] CWI, Amsterdam, The Netherlands

Abstract. The Calculus of Inductive Constructions (CIC) is a powerful type system, featuring dependent types and inductive definitions, that forms the basis of proof-assistant systems such as Coq and Lego. We extend CIC with constructor subtyping, a basic form of subtyping in which an inductive type σ is viewed as a subtype of another inductive type τ if τ has more elements than σ. It is shown that the calculus is well-behaved and provides a suitable basis for formalizing natural semantics in proof-development systems.

1 Introduction

Proof-development systems like Coq [4], Hol [21], Isabelle [28] and PVS [32] rely on powerful type systems featuring (co-)inductive types. The latter, which capture in a type-theoretical framework the notions of initial algebra or final coalgebra, are extensively used in the formalization of programming languages, reactive and embedded systems, communication and cryptographic protocols... While such works witness that formal verification has reached a certain maturity, users' efforts are often hindered by the rigidity of the existing tools. Thus providing type-theoretical tools for increasing the usability of proof-development systems remains an important objective.

Subtyping is a relation on types that expresses that one type is at least as general as another one and is embedded in the type system via the subsumption rule, stating that a term of type a is also of type b whenever a is a subtype of b. While subtyping has long been perceived as a tool which could significantly improve the usability of proof-development systems, many of the existing approaches to subtyping are inappropriate for the (co-)inductive approach to formalization (see Section 5).

Constructor subtyping [7,8] is a basic form of subtyping in which an inductive type σ is viewed as a subtype of another inductive type τ if τ has more inhabitants than σ. It is fully compatible with the (co-)inductive approach to

J. Tiuryn (Ed.): FOSSACS 2000, LNCS 1784, pp. 17–34, 2000.

formalization and may be used to specify most of the examples arising in natural semantics [23]. For example, constructor subtyping may be used to formalize the expressions of the call-by-value λ-calculus and of the ς-calculus, or the set of Harrop formulae [7,8]. It may also be used to formalize the datatype of lists/non-empty lists:

Parameters:	X
Sorts:	list, nelist
Subsort relation:	nelist $\leq$ list
Declarations:	nil : $X \to$ list
	cons : $X \to$ list $\to$ nelist

The salient feature of constructor subtyping is to impose suitable *coherence* conditions on constructor overloading: roughly speaking, constructor declarations are supposed to be monotonic, i.e. if $c : A \to \sigma$ and $c : B \to \tau$ are constructor declarations with $\sigma \sqsubseteq \tau$, then one must have $A \sqsubseteq B$. This is trivially satisfied in the case of the lists/non-empty lists, as constructors are declared only once, and in most datatypes with overloaded constructors such as the one of odd/even/natural numbers:

Parameters:	
Sorts:	even, odd, nat
Subsort relation:	even, odd $\leq$ nat
Declarations:	0 : even
	S : even $\to$ odd
	S : odd $\to$ even
	S : nat $\to$ nat

The immediate benefit of coherence is that the above definitions may be viewed as deterministic rule sets in the sense of [1]—see Section 2 for the difficulties with datatypes that do not yield deterministic rule sets. Therefore they support recursive definitions and may be integrated safely to typed λ-calculi [7,8].

In the present paper, we study constructor subtyping in the context of the Calculus of Inductive Constructions (CIC) [35], a dependently typed λ-calculus that forms the basis of several proof-assistants, including Coq [4] and Lego [25]. In particular, we show that adding constructor subtyping preserves some fundamental properties of CIC, including confluence, subject reduction, strong normalization and decidability of type-checking. These results scale up to dependent types those of previous papers [7,8], and open the road for an integration of constructor subtyping in proof-development systems such as Coq and Lego.

The remaining of the paper is organized as follows: in Section 2, we present an extension of the Calculus of Inductive Constructions with Constructor Subtyping. As we shall explain, our presentation is slightly different from the one of [7,8] so as to scale up to dependent types. In Section 3, we prove that the main meta-theoretical properties of the Calculus of Inductive Constructions are preserved. While the confluence and strong normalization proofs rely on standard techniques, both Subject Reduction and decidability of type-checking do

pose some interesting difficulties that are pervasive in all calculi which combine parametric inductive types and subtyping. In Section 4, we extend the type system with unbounded recursion, leaving aside the somewhat intrinsic problem of guarded recursive definitions. We show the calculus remains confluent and enjoys the subject reduction property—strong normalization obviously fails and hence so does decidability of type-checking. Finally, Section 5 concludes with related work and directions for further research. Most proofs are only sketched.

2 Syntax

A countably infinite set $\mathcal{V}$ of *variables*, written as $x, y, z, \ldots$ is assumed. We assume further a set $\mathcal{D}$ of *datatypes* and a set $\mathcal{C}$ of *constructors*. Datatypes are written as $\sigma, \ldots$, and constructors are written as $f, \ldots$. Every datatype σ and every constructor f comes equipped with a fixed *arity*, which is a natural number indicating the number of parameters it is supposed to have. The arity of a symbol s is denoted by $\mathsf{ar}(s)$. In addition, every datatype σ comes equipped with a set of constructors, denoted by $\mathsf{C}(\sigma)$. Finally, two *sorts* are assumed: the sort of *types*, written as $\star$, and the sort of *kinds*, written as $\Box$. The set $\{\star, \Box\}$ is denoted by $\mathcal{S}$.

In the remainder of the paper, we will make use the following two examples. The first one is the datatype nat of natural numbers, with $\mathsf{ar}(\mathsf{nat}) = 0$ and $\mathsf{C}(\mathsf{nat}) = \{\mathsf{0}, \mathsf{S}\}$. We further assume $\mathsf{ar}(\mathsf{0}) = 0$ and $\mathsf{ar}(\mathsf{S}) = 1$. The second one is the datatype list of polymorphic lists, with $\mathsf{ar}(\mathsf{list}) = 1$ and $\mathsf{C}(\mathsf{list}) = \{\mathsf{nil}, \mathsf{cons}\}$. The argument of list is meant to specify the type of the elements of the list; for instance $\mathsf{list}\ \mathsf{nat}$ represents the type of lists of natural numbers. We further assume $\mathsf{ar}(\mathsf{nil}) = 1$ and $\mathsf{ar}(\mathsf{cons}) = 3$.

Pseudo-Terms. Pseudo-terms are built from the standard constructions for dependent types, datatypes and constructors, and case-expressions. The latter are annotated by their type (superscript) and by the type of the expression being matched (subscript). The purpose of both annotations is to guide type-inference; see Section 3. Finally, note that at this point no constructor for fixpoints is present. The introduction of such a constructor is postponed until Section 4.

Definition 1. The set $\mathcal{T}$ of *pseudo-terms* is defined inductively as follows:

1. $\mathcal{S} \cup \mathcal{V} \subseteq \mathcal{T}$;
2. if $x \in \mathcal{V}$ and $A, B \in \mathcal{T}$, then $\Pi x : A.\, B,\ \lambda x : A.\, B,\ A\ B \in \mathcal{T}$;
3. if $\sigma \in \mathcal{D}$ with $\mathsf{ar}(\sigma) = n$, and $M_1, \ldots, M_n \in \mathcal{T}$, then $\sigma\ M_1\ \ldots\ M_n \in \mathcal{T}$;
4. if $f \in \mathcal{C}$ with $\mathsf{ar}(f) = n$, and $M_1, \ldots, M_n \in \mathcal{T}$, then $f M_1 \ldots M_n \in \mathcal{T}$;
5. if $\sigma \in \mathcal{D}$ with $\mathsf{ar}(\sigma) = n$, and $\mathsf{C}(\sigma) = \{f_1, \ldots, f_k\}$, and further $M, M_1, \ldots, M_k$, $P_1, \ldots, P_n, Q \in \mathcal{T}$, then $\mathsf{case}^{Q}_{\sigma P_1 \ldots P_n}\ M\ \mathsf{of}\ \{f_1 \to M_1, \ldots, f_k \to M_k\} \in \mathcal{T}$.

The notions of free and bound variable, α-conversion and substitution are defined as usual. We write $M[x := N]$ for the result of substituting all free occurrences of x in M by N. Further, we write $A \to B$ for $\Pi x : A.\, B$ if x does not occur free

in B. The set of closed pseudo-terms (i.e. without free variables) is denoted by $\mathcal{T}_0$. Finally, $P_1\ \ldots\ P_n$ is sometimes abbreviated as $\boldsymbol{P}$. We then use the notation $\#\boldsymbol{P}$ to denote the length of such a vector.

Rewriting. We consider three reduction rules on pseudo-terms, for β-reduction, ι-reduction, and κ-reduction respectively:

1. $(\lambda x : A.\, M)\, N \to_\beta M[x := N]$;
2. $\mathsf{case}^Q_{\sigma \boldsymbol{P}}\ (f_i\ \boldsymbol{P'}\ N_1\ \ldots\ N_m)\ \mathsf{of}\ \{f_1 \to M_1, \ldots, f_k \to M_k\} \to_\iota M_i\ N_1\ \ldots\ N_m$ with $\#\boldsymbol{P} = \#\boldsymbol{P'} = \mathsf{ar}(\sigma)$ and $\mathsf{ar}(f_i) = \mathsf{ar}(\sigma) + m$;
3. $f\ \boldsymbol{P}\ \boldsymbol{N} \to_\kappa f\ \overbrace{\Box\ \ldots\ \Box}^{\#\boldsymbol{P}\ \text{times}}\ \boldsymbol{N}$ where $f \in \mathsf{C}(\sigma)$ with $\mathsf{ar}(\sigma) = \#\boldsymbol{P}$, and $\mathsf{ar}(f) = \#\boldsymbol{P} + \#\boldsymbol{N}$.

As usual, the reduction relation $\to_R$ is defined as the smallest compatible closure of $\to_R$. The definitions of β- and ι-reduction are standard. The κ-reduction relation is less standard; it is there to enforce subject reduction to hold, and is only used in the type conversion rule (see below). We write $\to_{\beta\iota}$ for the reduction relation $\to_\beta \cup \to_\iota$, and $\to_{\beta\iota\kappa}$ for the reduction relation $\to_\beta \cup \to_\iota \cup \to_\kappa$.

Note that in the ι-reduction rule, $\boldsymbol{P}$ and $\boldsymbol{P'}$ are not required to be identical, the idea being that $\boldsymbol{P'}$ may be a subtype of $\boldsymbol{P}$ (pointwise). Note further that κ-reduction does not affect the number of arguments of f.

Subtyping. We assume given a binary subtyping relation $\sqsubseteq_d$ over $\mathcal{D}$ that is reflexive and transitive, and that in addition satisfies the following two requirements for every $\sigma, \tau \in \mathcal{D}$:

1. if $\sigma \sqsubseteq_d \tau$, then $\mathsf{C}(\sigma) \subseteq \mathsf{C}(\tau)$;
2. if $\mathsf{C}(\sigma) \cap \mathsf{C}(\tau) \neq \emptyset$, then $\mathsf{ar}(\sigma) = \mathsf{ar}(\tau)$.

An example also used in the remainder of the paper is that of odd and even natural numbers. Those are represented by datatypes odd and even with $\mathsf{ar}(\mathsf{odd}) = \mathsf{ar}(\mathsf{even}) = 0$, and $\mathsf{C}(\mathsf{odd}) = \{\mathsf{S}\}$, and $\mathsf{C}(\mathsf{even}) = \{\mathsf{0}, \mathsf{S}\}$. We assume given the subtyping relation $\mathsf{odd} \sqsubseteq \mathsf{nat}$ and $\mathsf{even} \sqsubseteq \mathsf{nat}$.

The relation $\sqsubseteq_d$ is used in the definition of the subtyping relation $\sqsubseteq$. In the presence of dependent types, the subtyping relation needs to encompass the convertibility relation, and is therefore undecidable. For some purposes, in particular for the notion of strict overloading to be decidable, it is convenient to consider in addition a restricted notion of subtyping $\sqsubseteq_s$ which does not account for convertibility. The two subtyping relations are defined as follows.

Definition 2.

1. The *subtyping relation* $\sqsubseteq$ is defined by the following rules:

$$\text{(refl)}\ \frac{}{A \sqsubseteq A} \qquad \text{(trans)}\ \frac{A \sqsubseteq A' \quad A' \sqsubseteq A''}{A \sqsubseteq A''} \qquad \text{(prod)}\ \frac{A' \sqsubseteq A \qquad B \sqsubseteq B'}{\Pi x : A.B \sqsubseteq \Pi x : A'.B'}$$

$$\text{(conv)}\ \frac{A =_{\beta\iota\kappa} B}{A \sqsubseteq B} \qquad \text{(data)}\ \frac{\sigma \sqsubseteq_d \tau \quad A_1 \sqsubseteq B_1 \quad \ldots \quad A_{\mathsf{ar}(\sigma)} \sqsubseteq B_{\mathsf{ar}(\sigma)}}{\sigma\ A_1\ \ldots\ A_{\mathsf{ar}(\sigma)} \sqsubseteq \tau\ B_1\ \ldots\ B_{\mathsf{ar}(\sigma)}}$$

2. The *conversion-free subtyping relation* $\sqsubseteq_s$ is defined by all the rules above except (conv).

A major design decision is that subtyping is defined independently of typing by defining it on pesudo-terms as in [10,36]. This allows to break the circularity between typing and subtyping found in [2], where subtyping is only defined on legal terms and thereby depends on typing.

The unusual rule (data) requires that inductive types are monotonic in their parameters. It is used for instance to derive $\mathsf{list\ odd} \sqsubseteq \mathsf{list\ nat}$. An alternative is to consider a polarized calculus as e.g. in [33] but most datatypes of interest are monotonic in their parameters, so we feel the complications are not justified here.

Finally, note that the example of odd and even natural numbers illustrates that it is not possible to use set-theoretic inclusion on the set of constructors to define subtyping on datatypes; this would yield $\mathsf{odd} \sqsubseteq \mathsf{even}$ which is undesirable.

Typing constructors. A next question is how to provide types for the datatypes and constructors. In order to do this, we assume given two mappings $\mathsf{K} : \mathcal{D} \to \mathcal{T}_0$ and $\mathsf{D} : \prod \sigma \in \mathcal{D}.\mathsf{C}(\sigma) \to \mathcal{T}_0$ such that for every datatype $\sigma \in \mathcal{D}$ and for every constructor $c \in \mathsf{C}(\sigma)$ we have the following:

1. $\mathsf{K}(\sigma)$ is of the form $\Pi \boldsymbol{x} : \boldsymbol{A}. \star$ where $\#\boldsymbol{x} = \mathsf{ar}(\sigma)$,
2. $\mathsf{D}(\sigma, f)$ is of the form $\Pi \boldsymbol{x} : \boldsymbol{A}. \Pi \boldsymbol{y} : \boldsymbol{B}. \sigma\ \boldsymbol{x}$ where $\mathsf{K}(\sigma) = \Pi \boldsymbol{x} : \boldsymbol{A}. \star$ and $\#\boldsymbol{x} + \#\boldsymbol{y} = \mathsf{ar}(f)$.

If $\mathsf{D}(\sigma, f) = \Pi \boldsymbol{x} : \boldsymbol{A}. \Pi \boldsymbol{y} : \boldsymbol{B}. \sigma\ \boldsymbol{x}$, we write $\mathsf{D}^{\Box}(\sigma, f)$ for $\Pi \boldsymbol{x} : \boldsymbol{A}. \Pi \boldsymbol{y} : \boldsymbol{B}. \Box$.

Before proceeding with an example, let us emphasize that we do not consider inductive families since the codomain of $\mathsf{D}(\sigma, f)$ is of the form $\sigma\ \boldsymbol{x}$. It is straightforward to add inductive families to our calculus, but more difficult to adapt the notion of constructor subtyping to inductive families, see Section 5 for a discussion.

To illustrate the intended use of K and D, we consider the datatype list of polymorphic lists. Natural mappings K and D are defined by the following:

$$\begin{array}{ll} \mathsf{K}\,(\mathsf{list}) & = \star \to \star \\ \mathsf{D}\,(\mathsf{list}, \mathsf{nil}) & = \Pi x : \star.\, \mathsf{list}\ x \\ \mathsf{D}\,(\mathsf{list}, \mathsf{cons}) & = \Pi x : \star.\, x \to \mathsf{list}\ x \to \mathsf{list}\ x \end{array}$$

Using the typing system defined below, we have, with $n : \mathsf{nat}$ and $l \in \mathsf{list\ nat}$:

$$\begin{array}{ll} \mathsf{list\ nat} & : \star \\ \mathsf{nil\ nat} & : \mathsf{list\ nat} \\ \mathsf{cons\ nat}\ n\ l & : \mathsf{list\ nat} \end{array}$$

Overloading. In contrast with the Calculus of Inductive Constructions, constructors may be overloaded. This is crucial to the applicability of constructor subtyping, as most examples require constructors to be overloaded. However, in presence of subsumption, overloading leads to difficulties with subject reduction.

To illustrate the problem, we consider two sorts: $\bot$ and $\top$, with $\bot \sqsubseteq_d \top$. We assume the following constructors for these sorts: the constructor of $\bot$ is $f : \top \to \bot$, and the constructors of $\top$ are $f : \bot \to \top$ and $t : \top$. Note that the function f is overloaded. Further, some type A is assumed, and two terms $g : \bot \to A$ and $a : A$. We have (we omit sub- and superscripts in the case-expression)

$$\mathsf{case}^A_\top\ (f\ t)\ \mathsf{of}\ \{f \to g, t \to a\} : A$$

This term reduces to $g\,t$, which is not typable. So subject reduction fails.

Hence overloading must be constrained in some way. The solution advocated in [7,8] is to require the following:

1. $\mathsf{D}^\Box$ is anti-monotonic in its first argument: if f is a constructor for σ and τ with $\sigma \sqsubseteq_d \tau$, then the domain of f w.r.t. τ must be a subtype of the domain of f w.r.t. σ;
2. if f is a constructor for σ and τ, then σ and τ must be parameterized over the same types.

Here we follow a similar approach but, in order to enforce decidability, we rely on $\sqsubseteq_s$ rather than on $\sqsubseteq$ to compare domains.

Definition 3 (Strict overloading). A constructor f is *strictly overloaded* if for every $\sigma,\ \tau \in \mathcal{D}$ and $f \in \mathsf{C}(\sigma) \cap \mathsf{C}(\tau)$ we have the following:

1. if $\sigma \sqsubseteq_d \tau$ then $\mathsf{D}^\Box(\tau, f) \sqsubseteq_s \mathsf{D}^\Box(\sigma, f)$,
2. $\mathsf{K}(\sigma) = \mathsf{K}(\tau)$.

The constructor S for successor of the datatype for odd and even and natural numbers is strictly overloaded. Indeed, we have:

$$\mathsf{K}(\mathsf{even}) = \mathsf{K}(\mathsf{odd}) = \mathsf{K}(\mathsf{nat}) = \star$$

and:

$$\begin{array}{lll} \mathsf{D}(\mathsf{even},\mathsf{S}) = \mathsf{odd} \to \mathsf{even} & & \mathsf{D}^\Box(\mathsf{even},\mathsf{S}) = \mathsf{odd} \to \Box \\ \mathsf{D}(\mathsf{odd},\mathsf{S}) = \mathsf{even} \to \mathsf{odd} & & \mathsf{D}^\Box(\mathsf{odd},\mathsf{S}) = \mathsf{even} \to \Box \\ \mathsf{D}(\mathsf{nat},\mathsf{S}) = \mathsf{nat} \to \mathsf{nat} & & \mathsf{D}^\Box(\mathsf{nat},\mathsf{S}) = \mathsf{nat} \to \Box \end{array}$$

So:

$$\begin{array}{l} \mathsf{D}^\Box(\mathsf{nat},\mathsf{S}) \sqsubseteq_s \mathsf{D}^\Box(\mathsf{even},\mathsf{S}) \\ \mathsf{D}^\Box(\mathsf{nat},\mathsf{S}) \sqsubseteq_s \mathsf{D}^\Box(\mathsf{odd},\mathsf{S}) \end{array}$$

From now on, we assume all constructors to be strictly overloaded.

Typing System. The typing system features the standard rules for the Calculus of Inductive Constructions, with the exception of the conversion rule which is replaced by the more general subsumption rule. Note that, in order for datatypes and constructors to be fully applied, the typing relation $\vdash$ is defined via some auxiliary relations $\vdash_n$ where $n \in \mathbb{N}$. We adopt the conventions $\vdash_0 = \vdash$ and $0 - 1 = 0$.

Definition 4. The typing relation $\Gamma \vdash M : A$ is defined by the rules of figure 1, where in the (case) rule it is assumed that:

1. $\mathsf{C}(\sigma) = \{f_1, \ \ldots \ , f_k\}$;
2. $\mathsf{K}(\sigma) = \Pi \boldsymbol{x} : \boldsymbol{A}. \star$;
3. $\mathsf{D}(\sigma, f_i) = \Pi \boldsymbol{x} : \boldsymbol{A}. \Pi \boldsymbol{y_i} : \boldsymbol{B_i}. \sigma \ \boldsymbol{x}$ for all i with $1 \leq i \leq k$;
4. $C_i = \Pi \boldsymbol{y_i} : \boldsymbol{B_i}[\boldsymbol{x} := \boldsymbol{E}]. Q \ (f_i \ \boldsymbol{E} \ \boldsymbol{y_i})$ for all i with $1 \leq i \leq k$;
5. $Q \ M \twoheadrightarrow_{\beta\iota} M'$.

The premises in the (datatype) and (constructor) rules are meant to ensure that the types of datatypes and constructors are well-formed. Indeed, not every datatype will be legal: e.g. the datatype σ with $\mathsf{K}(\sigma) = \Pi x : \Box.\Box$ is not legal.

$$\text{(axiom)} \quad \frac{}{\vdash \star : \Box}$$

$$\text{(start)} \quad \frac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A}$$

$$\text{(weakening)} \quad \frac{\Gamma \vdash A : s \qquad \Gamma \vdash M : B}{\Gamma, x : A \vdash M : B}$$

$$\text{(product)} \quad \frac{\Gamma \vdash A : s \qquad \Gamma, x : A \vdash B : s'}{\Gamma \vdash \Pi x : A.B : s'}$$

$$\text{(abstraction)} \quad \frac{\Gamma, x : A \vdash M : B \qquad \Gamma \vdash \Pi x : A.B : s}{\Gamma \vdash \lambda x : A.M : \Pi x : A.B}$$

$$\text{(application)} \quad \frac{\Gamma \vdash_n M : \Pi x : A.B \qquad \Gamma \vdash N : A}{\Gamma \vdash_{(n-1)} MN : B[x := N]}$$

$$\text{(datatype)} \quad \frac{\vdash \mathsf{K}(\sigma) : K}{\vdash_{\mathsf{ar}(\sigma)} \sigma : \mathsf{K}(\sigma)}$$

$$\text{(constructor)} \quad \frac{\vdash \mathsf{D}(\sigma, c) : C}{\vdash_{\mathsf{ar}(c)} c : \mathsf{D}(\sigma, c)}$$

$$\text{(case)} \quad \frac{\Gamma \vdash M : \sigma \ \boldsymbol{E} \quad \Gamma \vdash Q : \sigma \ \boldsymbol{E} \to \star \quad \Gamma \vdash N_i : C_i \quad (1 \leq i \leq k)}{\Gamma \vdash \mathsf{case}^{M'}_{\sigma \ \boldsymbol{E}} \ M \ \mathsf{of} \ \{f_1 \to N_1, \ldots, f_k \to N_k\} : Q \ M}$$

$$\text{(subsumption)} \quad \frac{\Gamma \vdash M : A \qquad \Gamma \vdash B : s \qquad A \sqsubseteq B}{\Gamma \vdash M : B}$$

Fig. 1. Typing rules

To illiustrate the use of case-expressions, we consider a definition of the predecessor function on natural numbers. We have:

$$\begin{array}{ll} \mathsf{C}(\mathsf{nat}) & = \{0, s\} \\ \mathsf{K}(\mathsf{nat}) & = \star \\ \mathsf{D}(\mathsf{nat}, 0) & = \mathsf{nat} \\ \mathsf{D}(\mathsf{nat}, \mathsf{S}) & = \mathsf{nat} \to \mathsf{nat} \end{array}$$

Further, we use:

$$\begin{array}{ll} \lambda x : \mathsf{nat}.\, \mathsf{nat} & : \mathsf{nat} \to \star \\ 0 & : \mathsf{nat} \\ \lambda x : \mathsf{nat}.\, x & : \mathsf{nat} \to \mathsf{nat} \end{array}$$

as Q, N_1, and N_2 in the (case) typing rule. Note that indeed $(\lambda x : \mathsf{nat}.\, \mathsf{nat})\, 0 \to_\beta \mathsf{nat}$ and $\Pi y : \mathsf{nat}.\, (\lambda x : \mathsf{nat}.\, \mathsf{nat})\, (\mathsf{S}\, y) \to_\beta (\mathsf{nat} \to \mathsf{nat})$. Then we have:

$$\mathsf{case}^{\mathsf{nat}}_{\mathsf{nat}}\, M \text{ of } \{0 \to 0, \mathsf{S} \to \lambda x : \mathsf{nat}.\, x\} : \mathsf{nat}$$

with $M : \mathsf{nat}$. The rewrite rules yield that we have

$$\begin{array}{ll} \mathsf{case}^{\mathsf{nat}}_{\mathsf{nat}}\, 0 \text{ of } \{0 \to 0, \mathsf{S} \to \lambda x : \mathsf{nat}.\, x\} & \to 0 \\ \mathsf{case}^{\mathsf{nat}}_{\mathsf{nat}}\, (\mathsf{S}\, n) \text{ of } \{0 \to 0, \mathsf{S} \to \lambda x : \mathsf{nat}.\, x\} & \to (\lambda x : \mathsf{nat}.\, x)\, n \to n \end{array}$$

3 Metatheory

Confluence. The first part of the following proposition follows because $\to_{\beta\iota}$ is orthogonal.

Proposition 1.

1. *$\to_{\beta\iota}$ is confluent on the set of pseudo-terms.*
2. *$\to_{\beta\iota\kappa}$ is confluent on the set of pseudo-terms.*

Subtyping. We present an alternative definition of subtyping, denoted by $\sqsubseteq_{int}$, that is shown to be equivalent to the original one. The subtyping relation $\sqsubseteq_{int}$ is used to prove subject reduction.

Definition 5. *The relation $\sqsubseteq_{int}$ is defined by the following rules:*

$$(prod)\ \frac{C =_{\beta\iota\kappa} \Pi x : A.\, B \quad C' =_{\beta\iota\kappa} \Pi x : A'.\, B' \quad A' \sqsubseteq_{int} A \qquad B \sqsubseteq_{int} B'}{C \sqsubseteq_{int} C'}$$

$$(data)\ \frac{C =_{\beta\iota\kappa} \sigma\ \boldsymbol{A} \quad C' =_{\beta\iota\kappa} \tau\ \boldsymbol{B} \quad \sigma \sqsubseteq_d \tau \quad A_i \sqsubseteq_{int} B_i \quad (1 \le i \le \mathsf{ar}(\sigma))}{C \sqsubseteq_{int} C'}$$

$$(conv)\ \frac{C =_{\beta\iota\kappa} C'}{C \sqsubseteq_{int} C'}$$

Here the reflexivity and transitivity rules are eliminated, and the conversion rule is distributed over the remaining ones. Note the system is not syntax-directed because of the (conv) rule.

Proposition 2. *$A \sqsubseteq_{int} B$ if and only if $A \sqsubseteq B$.*

Proof. It follows by induction on the definition of $\sqsubseteq_{int}$ that $A \sqsubseteq_{int} B$ implies $A \sqsubseteq B$.

Suppose that $A \sqsubseteq B$. We proceed by induction on the definition of $\sqsubseteq$. The problematic case is if $A \sqsubseteq B$ is the conclusion of the rule (trans). First it can be shown by induction on the derivation of $A \sqsubseteq_{int} B$ that if $A =_{\beta\iota\kappa} A'$, $B =_{\beta\iota\kappa} B'$ and $A \sqsubseteq_{int} B$, then $A' \sqsubseteq_{int} B'$ and moreover the derivation of both judgments have the same length. Then it can be shown that if $A \sqsubseteq_{int} B$ and $B \sqsubseteq_{int} C$,then $A \sqsubseteq_{int} C$ by induction on the derivation of $A \sqsubseteq_{int} B$.

Subject Reduction. The intermediate presentation of subtyping is used to prove the following lemma, which is crucial to prove subject reduction.

Lemma 1. *If $\Pi x : A.\, B \sqsubseteq \Pi x : A'.\, B'$ then $A' \sqsubseteq A$ and $B \sqsubseteq B'$.*

Using this lemma, subject reduction can be proved by adapting the standard proof for pure type systems (see for example [15]) to the case of pure type systems with subtying, as also done in [36].

Note however that the use of κ-reduction in the (conv) rule is crucial. Indeed, consider the term $M = \mathsf{cons\ even\ 0\ (nil\ even)}$ which has type $\mathsf{list\ even}$. We have (using notation as in the definition of the typing rules) $N_1 : C_1$ and $N_2 : C_2$ with $N_2 = \lambda n : \mathsf{nat}.\, \lambda l : \mathsf{list\ nat}.\, N_2'$, $C_1 = Q\ (\mathsf{nil\ nat})$, $C_2 = \Pi h : \mathsf{nat}.\, \Pi t : \mathsf{list\ nat}.\, Q\ (\mathsf{cons\ nat}\ h\ t)$, for some suitably typed N_1 and N_2', and some $Q : \mathsf{list\ nat} \to \star$. Now we have:

$$\mathsf{case}\ M\ \mathsf{of}\ \{\mathsf{nil} \to N_1, \mathsf{cons} \to N_2\} : Q\ (\mathsf{cons\ even\ 0\ (nil\ even)})$$

This term reduces to:

$$N_2\ 0\ (\mathsf{nil\ even}) : Q\ (\mathsf{cons\ nat\ 0\ (nil\ even)})$$

We have the conversion: $Q\ (\mathsf{cons\ even\ 0\ (nil\ even)}) =_\kappa Q\ (\mathsf{cons\ nat\ 0\ (nil\ even)})$ and hence by the conversion rule we have

$$N_2\ 0\ (\mathsf{nil\ even}) : Q\ (\mathsf{cons\ even\ 0\ (nil\ even)}).$$

Proposition 3 (Subject Reduction). *If $\Gamma \vdash M : A$ and $M \to_{\beta\iota} M'$ then $\Gamma \vdash M' : A$.*

Proof. The proof proceeds as in [15] by induction on the structure of derivations, proving simultaneously the following two implications: if $\Gamma \vdash M : A$ and $M \to_{\beta\iota} M'$, then $\Gamma \vdash M' : A$, and if $\Gamma \vdash M : A$ and $\Gamma \to_{\beta\iota} \Gamma'$ then $\Gamma' \vdash M : A$.

Here we only consider the case where the last typing rule is (case) rule. Let $M = \mathsf{case}^{M''}_{\sigma\ E}\ M'$ of $\{f_1 \to N_1, \ldots, f_n \to N_k\} : Q\ M'$ with $M' = f_i\ E'\ P$ and $M \twoheadrightarrow_{\beta\iota} M''$. For simplicity we restrict our attention to the case where the constructor f_i has just one argument, and the type σ has just one parameter. Let the last rule of the derivation be the (case) rule, as follows:

$$\frac{\Gamma \vdash M' : \sigma\ E \quad \Gamma \vdash Q : \sigma\ E \to \star \quad \Gamma \vdash N_i : C_i \quad (1 \leq i \leq k)}{\Gamma \vdash \mathsf{case}^{M''}_{\sigma\ E}\ M' \text{ of } \{f_1 \to N_1, \ldots, f_n \to N_k\} : Q\ M'}$$

We have $M \to_{\beta\iota} N_i\ P$. By generation, we have: $E \sqsubseteq E'$, $\Gamma \vdash N_l : \Pi y : B[x := E].\, Q\ (f_l\ E\ y)$, and $\Gamma \vdash E' : A$.

Moreover, there exists $\tau \sqsubseteq_d \sigma$ such that $\mathsf{D}_\tau(f_l) = \Pi x : A.\, \Pi y : B'.\, \tau\ H$, and $\Gamma \vdash P : B'[x := E']$. By strict overloading we have $B' \sqsubseteq B$. Because parameters occur positively in constructor declarations, we have $B'[x := E'] \sqsubseteq B[x := E]$. Subsumption yields that $\Gamma \vdash P : B[x := E]$, and the rule for application that $\Gamma \vdash N_l P : Q\ (f_l\ E\ P)$. Finally, by convertibility we have $\Gamma \vdash N_l P : Q\ (f_l\ E'\ P)$.

Termination. Thus far, we have not imposed any restriction on D and as a consequence the calculus is not terminating; in fact, it is possible to encode Girard's system U into our calculus, see [12, page 113].

In order to ensure termination, we must impose some conditions on D: constructors must be monotonic w.r.t. parameters and datatypes. In order to handle mutual recursion, we introduce a *precedence relation* $\blacktriangleleft$ on $\mathcal{D}$, which is supposed to be a pre-order. Below we let $\blacktriangleright$ be defined by $\tau \blacktriangleright \sigma$ iff $\sigma \blacktriangleleft \tau$, $\blacktriangleleft\!\blacktriangleright$ be defined as $\blacktriangleleft \cap \blacktriangleright$ and $\blacktriangleleft\!\blacktriangleleft\!\blacktriangleright$ be defined as $\blacktriangleleft \setminus \blacktriangleright$. Moreover we require:

1. $\blacktriangleleft\!\blacktriangleleft\!\blacktriangleright$ is well-founded;
2. the precedence relation is respected, i.e. if $\tau \boldsymbol{z}$ occurs in $\mathsf{D}(\sigma, c)$ then $\tau \blacktriangleleft \sigma$;
3. parameters must occur positively in the body of the declarations, i.e. if $\mathsf{D}(\sigma, c)$ is of the form $\Pi \boldsymbol{x} : \boldsymbol{A}.\, \Pi \boldsymbol{y} : \boldsymbol{B}.\, \sigma\ \boldsymbol{x}$, then every $x \in \boldsymbol{x}$ occurs positively in $\Pi \boldsymbol{y} : \boldsymbol{B}.\, \sigma\ \boldsymbol{x}$ (the precise definition of positivity may be found e.g. in [19]);
4. datatypes must occur positively in the body of their declarations, i.e. if $\mathsf{D}(\sigma, c)$ is of the form $\Pi \boldsymbol{x} : \boldsymbol{A}.\, \Pi \boldsymbol{y} : \boldsymbol{B}.\, \sigma\ \boldsymbol{x}$, then for every $\tau\ \blacktriangleleft\!\blacktriangleright\ \sigma$ every instance of $\tau\ \boldsymbol{z}$ occurs positively in $\Pi \boldsymbol{y} : \boldsymbol{B}.\, \sigma\ \boldsymbol{x}$.

Under these hypotheses, it is possible to show termination of our calculus by the well-known technique due to Tait and Girard, see e.g. [14,34] for an application to the Calculus of Constructions.

Theorem 1 (Termination). *If $\Gamma \vdash M : A$ then M is $\beta\iota$-terminating.*

It is then easy to conclude that legal terms are $\beta\iota\kappa$-normalizing and hence that convertibility between legal terms is decidable.

Decidability. As usual, the type-checking algorithm is decomposed into:

1. a type-inference algorithm which computes, if it exists, the minimal type of a term in a given context;
2. a subtype-checking algorithm (SCA) based on $\sqsubseteq_{int}$.

There are two problems with the existence of minimal types. We briefly explain what they are and how to solve them.

1. The first problem has to do with the existence of least upper bounds and is independent of constructor subtyping. Consider a $\mathsf{if} \ldots \mathsf{then} \ldots \mathsf{else}$ statement: if b is a boolean, $a : A$ and $a' : A'$ then for every B such that $A, A' \sqsubseteq B$, one has $\mathsf{if}\ b\ \mathsf{then}\ a\ \mathsf{else}\ a' : B$. In order for the above expression (which is of course a case-expression in disguise) to have a minimal type, A and A' must have a least upper bound. This needs not be the case in general, since we do not require $\sqsubseteq_d$ to be an upper semi-lattice. To solve this problem, we require case-expressions to be tagged by their type.
2. The second problem with minimal types is caused by constructor overloading. Suppose $\clubsuit, \spadesuit \in \mathcal{D}$ with arity 0 and $f \in \mathsf{C}(\clubsuit), \mathsf{C}(\spadesuit)$ with $\mathsf{D}(\clubsuit, f) = \mathsf{nat} \to \clubsuit$ and $\mathsf{D}(\spadesuit, f) = \mathsf{nat} \to \spadesuit$. Then we can derive $x : \mathsf{nat} \vdash f\ x : \clubsuit$ and $x : \mathsf{nat} \vdash f\ x : \spadesuit$. If $\clubsuit$ and $\spadesuit$ are unrelated by subtyping, then $f\ x$ does not have a minimal type. To solve this problem, we require constructors to be *regular*—a notion derived from order-sorted algebra, see e.g. [20].

First we introduce a notation. Let $f \in \mathsf{C}(\sigma)$ for some datatype σ, and suppose that $\mathsf{D}(\sigma, f) = \Pi \boldsymbol{x} : \boldsymbol{A}. \Pi \boldsymbol{y} : \boldsymbol{B}. \sigma\ \boldsymbol{x}$ with $\#\boldsymbol{A} = \mathsf{ar}(\sigma)$. Recall that $\mathsf{D}^{\Box}(\sigma, f)$ denotes $\Pi \boldsymbol{x} : \boldsymbol{A}. \Pi \boldsymbol{y} : \boldsymbol{B}. \Box$. We will use $\mathsf{D}^{\Box}_{[\boldsymbol{x} := \boldsymbol{E}]}(\sigma, c)$ to denote $\Pi \boldsymbol{y} : \boldsymbol{B}[\boldsymbol{x} := \boldsymbol{E}]. \Box$. Regularity is then defined as follows.

Definition 6. *A constructor f is said to be* regular *if for every datatype σ, and for all terms $P, \boldsymbol{E}$ such that $P \sqsubseteq \mathsf{D}^{\Box}_{[\boldsymbol{x} := \boldsymbol{E}]}(\sigma, c)$, we have that the set*

$$\{\tau \in \mathcal{D} \mid F \sqsubseteq \mathsf{D}^{\Box}_{[\boldsymbol{x} := \boldsymbol{E}]}(\tau, c)\}$$

contains a minimum element.

Under the assumption of regularity, minimal types exist.

Proposition 4. *If all constructors are regular, then the calculus has minimal types, i.e. if $\Gamma \vdash M : A$, then there exists $A' \in \mathcal{T}$ such that $\Gamma \vdash M : A'$ and $A' \sqsubseteq A''$ for every $A'' \in \mathcal{T}$ such that $\Gamma \vdash M : A''$.*

Proof. By induction on the structure of terms. We only consider the case of a constructor term with one parameter and one argument, so suppose $M = f\ E\ P$ and $\Gamma \vdash M : \sigma\ E$. By the induction hypothesis, the term P has a minimal type, say C, with $\Gamma \vdash P : C$. By generation, we have $C \sqsubseteq \mathsf{D}^{\Box}_{[x := E]}(\sigma, f)$. By regularity there exists a minimal ρ such that $C \sqsubseteq \mathsf{D}^{\Box}_{[x := E]}(\rho, f)$. It is easy to verify that the minimal type of M is $\rho\ E$.

Note that the notion of regularity is based on $\sqsubseteq$ and is therefore undecidable. However, we cannot rely on $\sqsubseteq_s$ instead since conversion may lead to new conflicts. Consider for instance a slight modification of the example above, where $\mathsf{D}(\spadesuit, f)$ is now defined by the equation $\mathsf{D}(\spadesuit, f) = ((\lambda \alpha : \star. \alpha)\ \mathsf{nat}) \to \spadesuit$. As before, $f\ x$ does not have a minimal type.

Also note that we need to consider *instances* of constructors, as instantiating constructors may lead to new conflicts. Consider for example $\clubsuit, \spadesuit \in \mathcal{D}$ with arity 1 and $f \in \mathsf{C}(\clubsuit), \mathsf{C}(\spadesuit)$ with $\mathsf{D}(\clubsuit, f) = \Pi\alpha : \star.\, \mathsf{nat} \to \clubsuit\ \alpha$ and $\mathsf{D}(\spadesuit, f) = \Pi\alpha : \star.\, \alpha \to \spadesuit\alpha$. Then we can derive $x : \mathsf{nat} \vdash i\mathsf{nat}x : \clubsuit\mathsf{nat}$ and $x : \mathsf{nat} \vdash i\mathsf{nat}x : \spadesuit\mathsf{nat}$. If $\clubsuit$ and $\spadesuit$ are unrelated by subtyping, then $f\,x$ does not have a minimal type.

The notion of regularity being undecidable, it is of some interest to provide some decidable sufficient condition for constructors to be regular. We present such a criterion. It is not the most general one possible, but it is relatively simple.

The idea is to distinguish for each constructor a set of inductive positions and require overloaded declarations only to vary in these inductive positions. So for each constructor $f \in \mathcal{C}$, we assume given a set $\mathsf{ip}(f) \subseteq \{1, \ldots, \mathsf{ar}(f)\}$ and only allow constructor declarations to vary on these positions.

Definition 7.

1. A constructor f is *safe* if for every $\sigma, \tau \in \mathcal{D}$ such that $f \in \mathsf{C}(\sigma) \cap \mathsf{C}(\tau)$ with $\mathsf{ar}(\sigma) = \mathsf{ar}(\tau) = n$ and
$$\mathsf{D}(\sigma, f) = \Pi\boldsymbol{x} : \boldsymbol{A}.\, \Pi\boldsymbol{y} : \boldsymbol{B}.\, \sigma\boldsymbol{x}$$
$$\mathsf{D}(\tau, c) = \Pi\boldsymbol{x} : \boldsymbol{A}.\, \Pi\boldsymbol{y} : \boldsymbol{B'}.\, \tau\boldsymbol{x}$$
one has the following:
   (a) for every $i \in \mathsf{ip}(f)$, B_i is of the form $\sigma_i\ \boldsymbol{x}$;
   (b) for every $i \notin \mathsf{ip}(f)$, $B_i = B'_i$.
2. Let $i_1, \ldots, i_k$ is an increasing enumeration of $\mathsf{ip}(f)$. We let f_σ denote $\Pi y_{i_1} : B_{i_1}.\ \ldots\ \Pi y_{i_k} : B_{i_k}.\, \Box$, and for $\rho_1, \ldots, \rho_k \in \mathcal{D}$ we let $f\{\rho_1, \ldots, \rho_k\}$ denote $\Pi y_{i_1} : \rho_1\ \boldsymbol{x}.\ \ldots\ \Pi y_{i_k} : \rho_k\ \boldsymbol{x}.\, \Box$.

The following proposition gives a sufficient condition

Proposition 5. *If f is a safe constructor and for every $\sigma \in \mathcal{D}$ such that $f \in \mathsf{C}(\sigma)$ and $\boldsymbol{\rho} \in \mathcal{D}$ such that $f\{\boldsymbol{\rho}\} \sqsubseteq_s c_\sigma$ the set*
$$\{\tau \mid f\{\boldsymbol{\rho}\} \sqsubseteq_s f_\tau\}$$
has a minimal element, then f is regular.

Proof. Assume $\mathsf{D}(c, \sigma) = \Pi\boldsymbol{x} : \boldsymbol{A}.\, \Pi\boldsymbol{y} : \boldsymbol{B}.\, \sigma\boldsymbol{x}$ and let $F, \boldsymbol{E} \in \mathcal{T}$ such that $F \sqsubseteq \mathsf{D}^{\Box}_{[\boldsymbol{x}:=\boldsymbol{E}]}(\sigma, c)$. Without loss of generality, one can assume $F = \Pi\boldsymbol{y} : \boldsymbol{C}.\, \Box$ with $C_i = B_i[\boldsymbol{x} := \boldsymbol{E}]$ if $i \notin \mathsf{ip}(c)$ and $C_i = \rho_i\ \boldsymbol{E}$. Let $c\{F\} = \Pi y_{i_1} : \rho_1\ \boldsymbol{x}.\ \ldots\ \Pi y_{i_k} : \rho_k\ \boldsymbol{x}.\, \Box$. It is then easy to check that for every $\tau \in \mathcal{D}$ such that $c \in \mathsf{C}(\tau)$, $c\{F\} \sqsubseteq_s c_\tau$ iff $F \sqsubseteq \mathsf{D}^{\Box}_{[\boldsymbol{x}:=\boldsymbol{E}]}(\tau, c)$.

We now turn to the SCA algorithm. It is obtained by specifying a reduction strategy for convertibility and by eliminating redundancies caused by the (conv) rule. Here we use the fact that legal types are either syntactically equal to a sort or weak-head reduce to a product $\Pi x : A.\, B$, a base term $x\,\boldsymbol{P}$ or a datatype $\sigma\,\boldsymbol{A}$.

Definition 8.

1. *Weak-head reduction* $\rightarrow_{wh}$ is the smallest relation such that for every $x \in \mathcal{V}$ and $A, P, Q, \boldsymbol{R} \in \mathcal{T}$ we have:

$$(\lambda x : A.\, P)\, Q\, \boldsymbol{R} \quad \rightarrow_{wh} \quad P[x := Q]\, \boldsymbol{R}$$

   (Weak-head reduction differs from β-reduction by applying only at the top-level.) The reflexive-transitive closure of $\rightarrow_{wh}$ is denoted by $\twoheadrightarrow_{wh}$.
2. The SCA is given by the following rules:

$$\text{(prod)}\ \frac{C \twoheadrightarrow_{wh} \Pi x : A.B \quad C' \twoheadrightarrow_{wh} \Pi x : A'.B' \quad A' \sqsubseteq_{alg} A \qquad B \sqsubseteq_{alg} B'}{C \sqsubseteq_{alg} C'}$$

$$\text{(data)}\ \frac{C \twoheadrightarrow_{wh} \sigma\ \boldsymbol{A} \quad C' \twoheadrightarrow_{wh} \tau\ \boldsymbol{B} \quad \sigma \sqsubseteq_{d} \tau \quad A_i \sqsubseteq_{alg} B_i \quad (1 \leq i \leq \mathsf{ar}(\sigma))}{C \sqsubseteq_{alg} C'}$$

$$\text{(var)}\ \frac{C \twoheadrightarrow_{wh} x\ \boldsymbol{A} \quad C' \twoheadrightarrow_{wh} x\ \boldsymbol{B} \quad A_i =_{\beta\iota} B_i \quad (1 \leq i \leq n)}{C \sqsubseteq_{alg} C'}$$

$$\text{(sort)}\ \frac{}{\star \sqsubseteq_{alg} \star}$$

In order to complete the description of our algorithm, one needs to specify how to test convertibility between expressions. This may be done in exactly the same way as in [13], although one has to take care not to compare the types of arguments in constructors (so as to handle κ-conversion). The SCA algorithm is sound and complete w.r.t. $\sqsubseteq$ on legal types.

Proposition 6. *Assume* $\Gamma \vdash A : s$ *and* $\Gamma' \vdash B : s'$. *Then* $A \sqsubseteq_{alg} B$ *iff* $A \sqsubseteq B$.

Proof. We use the fact that $A \sqsubseteq_{int} B$ iff $A \sqsubseteq B$. Soundness is trivial. Completeness is proved by induction over the derivation of $A \sqsubseteq_{int} B$.

Now decidability of type-checking follows from the existence of minimal types and the decidability of subtyping on legal terms. Hence we have the following result.

Theorem 2 (Decidability of type-checking). *If all constructors are regular, then* $\Gamma \vdash M : A$ *is decidable.*

4 Fixpoints

In this section we indicate how the somewhat limited computational power of the calculus can be increased by adding mutually dependent fixpoints. This is done as follows:

1. The set $\mathcal{T}$ of *pseudo-terms* is extended with the clause: if $x_1, \ldots, x_n \in \mathcal{V}$ are distinct variables and $\tau_1, \ldots, \tau_n, a_1, \ldots, a_n \in \mathcal{T}$, then $\mathsf{letrec}_i(x_1 : \tau_1 = a_1, \ldots, x_n : \tau_n = a_n) \in \mathcal{T}$;
2. The typing relation is extended with the rule

$$\frac{\Gamma\, f_1 : \tau_1, \ldots,\, f_n : \tau_n \;\vdash\; a_i : \tau_i \qquad (1 \leq i \leq n)}{\Gamma \;\vdash\; \mathsf{letrec}_j(f_1 : \tau_1 = a_1, \ldots, f_n : \tau_n = a_n) : \tau_j}$$

   with $1 \leq j \leq n$.
3. Fixpoint reduction $\rightarrow_\mu$ is defined as the compatible closure of the rule:

$$\mathsf{letrec}_i(\boldsymbol{x} : \boldsymbol{\tau} =_{\boldsymbol{k}} \boldsymbol{a}) \quad \rightarrow_\mu$$
$$a_i[x_1 := \mathsf{letrec}_1(\boldsymbol{x} : \boldsymbol{\tau} =_{\boldsymbol{k}} \boldsymbol{a}), \ldots, x_n := \mathsf{letrec}_n(\boldsymbol{x} : \boldsymbol{\tau} =_{\boldsymbol{k}} \boldsymbol{a})]$$

While fixpoints are crucial for the expressivity of the calculus, their introduction leads to non-termination and undecidable type-checking. However, confluence, subject reduction and minimal types are preserved, as stated in the following proposition. The proof is omitted.

Proposition 7.

1. *$\rightarrow_{\beta\iota\mu}$ is confluent;*
2. *If $\Gamma \vdash M : A$ and $M \rightarrow_{\beta\iota\mu} N$ then $\Gamma \vdash N : A$;*
3. *If all constructors are regular and $\Gamma \vdash M : A$ then there exists $A' \in \mathcal{T}$ such that $\Gamma \vdash M : A'$ and $A' \sqsubseteq A''$ for every $A'' \in \mathcal{T}$ such that $\Gamma \vdash M : A''$.*

5 Concluding Remarks

We have defined constructor subtyping for the Calculus Inductive of Constructions, a powerful type system that forms the basis of proof-development systems as Coq and Lego, and shown the resulting calculus to be well-behaved. A side-effect of our work is to provide a general approach to enforce subject reduction in calculi which combine parametric inductive types, dependent types and subtyping [3,18].

Related work. Subtyping in dependent type systems is an active research area, with some main trends to be distinguished:

- *name inequivalence based subtyping* assumes a subtyping relation on ground types; this relation is then extended to all types. Nordlander [26,27] has been developing such a theory of subtyping for Haskell; his approach allows to capture those instances of constructor subtyping which do not involve overloading, such as the datatype of lists/non-empty lists, but fails to capture those instances of constructor subtyping involving overloading, as even/odd/natural numbers.
  On a more theoretical level, Poll has been investigating subtyping between (co-)inductive types [30,31]. His approach, which is framed in a categorical setting, captures both constructor subtyping and its dual, destructor subtyping, whose prime example is record subtyping. However, Poll does not focus on the syntactic aspects of this form of subtyping;

– *declarative subtyping* allows to declare $X \sqsubseteq A : \star$ in contexts and was originally used in conjunction with related ideas, most notably bounded quantification [9], in order to provide a type-theoretical semantics of object-oriented languages, see e.g. [22]. However, declarative subtyping may also be used to represent formal languages in logical frameworks, see [29] for motivations, examples and a dependent type system based on refinement types. The interaction between dependent types and declarative subtyping has been studied by Aspinall and Compagnoni [2] for the logical frameworks, by Chen [11] for the Calculus of Constructions and by Zwanenburg [36] for Pure Type Systems. One major difference between [2] and [11,36] is that the former lets subtyping depend on typing, which leads to substantial complications in the theoretical study of the system. In order to avoid those, we have followed [11,36] and defined subtyping independently of typing. More recently, Castagna and Chen [10] have extended Chen's variant of Aspinall and Compagnoni's $\lambda P_{\leq}$ with late-binding. Their calculus is a significant improvement over $\lambda P_{\leq}$ and allows to formalize the examples of [29]. However, declarative subtyping, even combined with late-binding, is not appropriate for the inductive approach to formalization;
– *implicit coercions* [5,24] allow to view a term a of type A as a term of type B whenever there is a previously agreed upon function, called coercion, from A to B. This approach, which leads to extremely powerful type systems, is implemented in several proof-development systems, including Coq and Lego, and has proved useful in several efforts to formalize mathematics in type theory. However, implicit coercions also yield intricate coherence problems: one would like to make sure that every two coercions from A to B are extensionally equal, a property which is undecidable in presence of parametric coercions. Moreover, implicit coercions do not capture constructor subtyping.

Further work Much work remains to be done. We indicate some topics that deserve further investigation:

– *inductive families*: it is straightforward to extend our calculus with inductive families. It is however more difficult to define constructor subtyping for inductive families. Such a form of subtyping is useful for formalizing type systems with subtyping. For example, consider a type system with a set of types $\mathsf{object\text{-}types}$ and a subtyping relation $\prec$ on $\mathsf{object\text{-}types}$; one would like to be able to define inductive families of the form $el : \mathsf{Object-Types} \to \star$ such that $el\ \alpha \sqsubseteq el\ \alpha'$ whenever $\alpha \prec \alpha'$, where $\mathsf{Object\text{-}Types}$ is the datatype describing $\mathsf{object\text{-}types}$. A possible approach to integrate constructor subtyping to inductive families is to replace the (data) rule by

$$\frac{\sigma \sqsubseteq_d \tau \quad (A_1, \ldots, A_{\mathsf{ar}(\sigma)}) \sqsubseteq_\sigma (B_1, \ldots, B_{\mathsf{ar}(\sigma)})}{\sigma\ A_1\ \ldots\ A_{\mathsf{ar}(\sigma)} \sqsubseteq \tau\ B_1\ \ldots\ B_{\mathsf{ar}(\sigma)}}$$

where $\sqsubseteq_\sigma$ is a relation on tuples of pseudo-terms defined for each datatype σ. This approach is currently under investigation;

- *guarded recursive definitions*: while fixpoints are crucial for the expressivity of the calculus, their introduction leads to non-termination and undecidable type-checking. In order to recover both properties, one needs to restrict the typing rule for fixpoints, for example by using a suitable notion of *guard*, see e.g. [17], or a suitable type system based on subtyping, see e.g. [3,18]. We feel the latter approach provides an appealing alternative for our purpose but the technical details remain to be unveiled;
- *canonical inhabitants*: as pointed out in [7], the system is not well-behaved with respect to canonical inhabitants: e.g. both nil even and nil nat are closed normal inhabitants of list nat. This example illustrates how the equational theory is too weak. In [8], we show that an η-expansion rule for datatypes solves the problem in the simply typed case.[1] It should be possible to adopt the same solution for the Calculus of Constructions, although the combination of η-expansion with dependent types is somewhat intricate [6,16].

Addressing these issues should bring us closer to our overall objective, namely to integrate constructor subtyping as a primitive in proof-development systems.

Acknowledgments The authors are grateful to M.J. Frade and to the anonymous referees for their constructive comments on the paper. The first author is partially supported by the Portuguese Science Foundation (FCT) under the project FACS (PRAXIS P/EEI/14172/1998).

References

1. P. Aczel. An introduction to inductive definitions. In J. Barwise, editor, *Handbook of mathematical logic*, volume 90 of *Studies in Logic and the Foundations of Mathematics*, pages 739–782. North-Holland, 1977.
2. D. Aspinall and A. Compagnoni. Subtyping dependent types. In *Proceedings of LICS'96*, pages 86–97. IEEE Computer Society Press, 1996.
3. B. Barras. *Auto-validation d'un système de preuves avec familles inductives*. PhD thesis, Université Paris 7, 1999.
4. B. Barras *et. al. The Coq Proof Assistant User's Guide. Version 6.2*, May 1998.
5. G. Barthe. Implicit coercions in type systems. In S. Berardi and M. Coppo, editors, *Proceedings of TYPES'95*, volume 1158 of *Lecture Notes in Computer Science*, pages 16–35. Springer-Verlag, 1996.
6. G. Barthe. Expanding the cube. In W. Thomas, editor, *Proceedings of FOSSACS'99*, volume 1578 of *Lecture Notes in Computer Science*, pages 90–103. Springer-Verlag, 1999.
7. G. Barthe. Order-sorted inductive types. *Information and Computation*, 149(1):42–76, February 1999.
8. G. Barthe and M.J. Frade. Constructor subtyping. In D. Swiestra, editor, *Proceedings of ESOP'99*, volume 1576 of *Lecture Notes in Computer Science*, pages 109–127. Springer-Verlag, 1999.
9. L. Cardelli and P. Wegner. On understanding types, data abstraction and polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[1] It also eliminates the need for κ-conversion.

10. G. Castagna and G. Chen. Dependent types with subtyping and late-bound overloading. *Information and Computation*, 1999. To appear.
11. G. Chen. Subtyping calculus of constructions. In I. Prívara and P. Ruzicka, editors, *Proceedings of MFCS'97*, volume 1295 of *Lecture Notes in Computer Science*, pages 189–198. Springer-Verlag, 1997.
12. T. Coquand. Metamathematical investigations of a calculus of constructions. In P. Odifreddi, editor, *Logic and Computer Science*, pages 91–122. Academic Press, 1990.
13. T. Coquand. An algorithm for type-checking dependent types. *Science of Computer Programming*, 26(1–3):167–177, May 1996.
14. H. Geuvers. A short and flexible proof of strong normalisation for the Calculus of Constructions. In P. Dybjer, B. Nordström, and J. Smith, editors, *Proceedings of TYPES'94*, volume 996 of *Lecture Notes in Computer Science*, pages 14–38. Springer-Verlag, 1995.
15. H. Geuvers and M.J. Nederhof. A modular proof of strong normalisation for the Calculus of Constructions. *Journal of Functional Programming*, 1(2):155–189, April 1991.
16. N. Ghani. Eta-expansions in dependent type theory—the calculus of constructions. In P. de Groote and J. Hindley, editors, *Proceedings of TLCA'97*, volume 1210 of *Lecture Notes in Computer Science*, pages 164–180. Springer-Verlag, 1997.
17. E. Giménez. *Un calcul de constructions infinies et son application à la vérification de systèmes communicants*. PhD thesis, Ecole Normale Superieure de Lyon, 1996.
18. E. Giménez. Structural recursive definitions in Type Theory. In K.G. Larsen, S. Skyum, and G. Winskel, editors, *Proceedings of ICALP'98*, volume 1443 of *Lecture Notes in Computer Science*, pages 397–408. Springer-Verlag, 1998.
19. E. Giménez. A tutorial on recursive types in coq. Technical Report RT-0221, INRIA, 1998.
20. J. Goguen and J. Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105(2):216–273, 1992.
21. M.J.C. Gordon and T.F. Melham, editors. *Introduction to HOL: A theorem proving environment for higher-order logic*. Cambridge University Press, 1993.
22. C.A. Gunter and J.C. Mitchell. *Theoretical Aspects of Object-Oriented Programming: Types, Semantics and Language Design*. The MIT Press, 1994.
23. G. Kahn. Natural semantics. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer-Verlag, 1987.
24. Z. Luo. Coercive subtyping. *Journal of Logic and Computation*, 9:105–130, February 1999.
25. Z. Luo and R. Pollack. LEGO proof development system: User's manual. Technical Report ECS-LFCS-92-211, LFCS, University of Edinburgh, May 1992.
26. J. Nordlander. Pragmatic subtyping in polymorphic languages. In *Proceedings of ICFP'98*. ACM Press, 1998.
27. J. Norlander. *Reactive Objects and Functional Programming*. PhD thesis, Chalmers Tekniska Högskola, 1999.
28. L. Paulson. *Isabelle: a generic theorem prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.
29. F. Pfenning. Refinement types for logical frameworks. In H. Geuvers, editor, *Informal Proceedings of TYPES'93*, pages 285–299, 1993.

30. E. Poll. Subtyping and Inheritance for Inductive Types. In *Proceedings of TYPES'97 Workshop on Subtyping, inheritance and modular development of proofs, Durham, UK*, 1997.
31. E. Poll. Subtyping and Inheritance for Categorical Datatypes. In *Proceedings of Theories of Types and Proofs (TTP) - Kyoto*, RIMS Lecture Notes 1023, pages 112–125. Kyoto University Research Insitute for Mathematical Sciences, 1998.
32. N. Shankar, S. Owre, and J.M. Rushby. *The PVS Proof Checker: A Reference Manual*. Computer Science Laboratory, SRI International, February 1993. Supplemented with the PVS2 Quick Reference Manual, 1997.
33. M. Steffen. *Polarized Higher-order Subtyping*. PhD thesis, Department of Computer Science, University of Erlangen, 1997.
34. J. Terlouw. Strong normalization in type systems: a model theoretic approach. *Annals of Pure and Applied Logic*, 73(1):53–78, May 1995.
35. B. Werner. *Méta-théorie du Calcul des Constructions Inductives*. PhD thesis, Université Paris 7, 1994.
36. J. Zwanenburg. Pure type systems with subtyping. In J.-Y. Girard, editor, *Proceedings of TLCA'99*, volume 1581 of *Lecture Notes in Computer Science*, pages 381–396. Springer-Verlag, 1999.