A New Elimination Rule for the Calculus of Inductive Constructions

Bruno Barras[1], Pierre Corbineau[2], Benjamin Grégoire[3], Hugo Herbelin[1], and Jorge Luis Sacchini[3]

[1] INRIA Saclay – Île-de-France
{Bruno.Barras,Hugo.Herbelin}@inria.fr
[2] Université Joseph Fourier, INPG, CNRS
Pierre.Corbineau@imag.fr
[3] INRIA Sophia-Antipolis
{Benjamin.Gregoire,Jorge-Luis.Sacchini}@inria.fr

Abstract. In Type Theory, definition by dependently-typed case analysis can be expressed by means of a set of equations — the semantic approach — or by an explicit pattern-matching construction — the syntactic approach. We aim at putting together the best of both approaches by extending the pattern-matching construction found in the Coq proof assistant in order to obtain the expressivity and flexibility of equation-based case analysis while remaining in a syntax-based setting, thus making dependently-typed programming more tractable in the Coq system. We provide a new rule that permits the omission of impossible cases, handles the propagation of inversion constraints, and allows to derive Streicher's K axiom. We show that subject reduction holds, and sketch a proof of relative consistency.

1 Introduction

The Calculus of Inductive Constructions (CIC) [13, 10] is an extension of the Calculus of Constructions with inductive types and universes. Inductive types can be added to the system by specifying their constructors (introduction rules). To reason about inductive types, CIC includes a mechanism for performing pattern matching. It allows to define a function on an inductive type by giving computation rules for its constructors, in a similar way as in functional programming languages, such as Haskell or ML.

It is well known that dependent types add a new dimension to the pattern matching mechanism. This was first observed by Coquand [2], and later studied by other authors [5, 7, 3, 8]. A simple example is provided by the definition of lists indexed with their length, which we call here vectors. In CIC, given a type X, vectors are introduced by a constant vector of type $\mathsf{nat} \to \mathtt{Type}$, where $\mathsf{vector}\, n$ represents lists of n elements of type X. The constructors are $\mathsf{nil} : \mathsf{vector}\, 0$ for the empty vector, and $\mathsf{cons} : \Pi(n : \mathsf{nat}).X \to \mathsf{vector}\, n \to \mathsf{vector}\, (\mathsf{S}\, n)$ for adding an element to a vector. One of the slogans of using inductive families and dependently typed languages is the fact that functions can be given a more

precise typing. The usual tail function, that removes the first element of a non-empty vector can be given the type $\Pi(n : \mathsf{nat}).\mathsf{vector}\,(\mathsf{S}\,n) \to \mathsf{vector}\,n$, thus ensuring that it cannot be applied to an empty vector. In Coquand's setting, we could write the tail function as

$$\mathsf{tail}\,n\,(\mathsf{cons}\,k\,x\,t) = t$$

Note the missing case for nil. This definition is accepted because the type system can ensure that the vector argument, being a term of type $\mathsf{vector}\,(\mathsf{S}\,n)$, cannot reduce to nil.

In CIC, the direct translation of the above definition is rejected, because of the missing case. Instead, we are forced to make a explicit proof that the nil case is not necessary. This makes the function more difficult to write by hand, and the reasoning necessary to rule out impossible cases hinders the intended computational rules. As a consequence, CIC is not well suited to be the basis for a programming language with dependent types.

Our objective is to adapt the work that has been done in dependent pattern matching to the CIC framework, thus reducing the gap between current implementations of CIC, such as Coq [1], and programming languages such as Epigram [6, 7] and Agda [8] — at least, in terms of programming facilities. In particular, we propose a new rule for pattern matching that automatically handles the reasoning steps mentioned above (Sect. 4). The new rule, which allows the user to write more direct and more efficient functions, combines explicit restriction of pattern-matching to inductive subfamilies, (as independently investigated by the second author for deriving axiom K and by the third and fifth authors for simulating Epigram in Coq without computational penalty) and translation of unification constraints into local definitions of the typing context (as investigated by the first and fourth authors). At the end, we prove that the type system satisfies subject reduction and outline a proof of relative consistency (Sect. 5).

2 A Primer on Pattern Matching in CIC

In this section, we study in detail how to write functions by pattern matching in CIC. The presentation is intentionally informal because we want to give some intuition on the problem at hand, and our proposed solution.

Let us consider the definition of tail. The naive solution is to write $\mathsf{tail}\,n\,v$ as

$$\texttt{match}\ v\ \texttt{with} \mid \mathsf{nil} \Rightarrow\ ? \mid \mathsf{cons}\ k\ x\ t \Rightarrow t\ .$$

There are two problems with this definition. The first is that we need to complete the nil branch with a term explicitly ruling out this case. The second is that the body of the cons branch is not well-typed, since we are supposed to return a term of type $\mathsf{vector}\,n$, while t has type $\mathsf{vector}\,k$. 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 nil. 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 $\mathsf{vector}\,(\mathsf{S}\,n)$, while nil has type $\mathsf{vector}\,0$. Since distinct constructors build distinct objects (the "no confusion" property), we can prove that $0 \neq \mathsf{S}\,n$, and, as a consequence, v cannot reduce to nil. This is translated to the definition of tail by generalizing the type of each branch to include a proof of equality between the indices. The definition of tail looks something like this:

$$\begin{aligned}\mathtt{match}\ v\ \mathtt{with}\ &\mid \mathsf{nil} \Rightarrow \lambda(H : 0 = \mathsf{S}\,n).\ \textit{here a proof of contradiction from } H\\ &\mid \mathsf{cons}\,k\,x\,t \Rightarrow \lambda(H : \mathsf{S}\,k = \mathsf{S}\,n).t\end{aligned}$$

where, in the nil 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 nil branch provides a way out. Note that, in the cons 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 $\mathsf{vector}\,n$. In the end, the body of this branch is a term built from H and t that *changes* the type of t from $\mathsf{vector}\,k$ to $\mathsf{vector}\,n$.

This solves both problems, but the type of the function obtained is $\mathsf{S}\,n = \mathsf{S}\,n \to \mathsf{vector}\,n$, 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 $v = \mathsf{cons}\,n\,h\,t$, we have $\mathsf{tail}\,n\,v \to^+ t$. In particular, in the body of the cons 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.

Our proposition is a new rule for pattern matching that allows to write dependent pattern matching in a direct way, avoiding pollution of the underlying program with proofs of equality statements and confining the justifications of the correctness of the dependencies to the typing rules. We would then be able to write the tail function as follows:

$$\mathsf{tail} := \lambda(n : \mathsf{nat})(v : \mathsf{vector}\,(\mathsf{S}\,n)).\mathtt{match}\ v\ \mathtt{with}\ \mid \mathsf{cons}\,k\,x\,t \Rightarrow t\ \mathtt{where}\ k := n$$

where some constructors are omitted (like nil above), and for the present constructors, some additional information is given (like $k := n$ above). The typing rules justify that the nil case is not necessary, and that the definition $k := n$ is valid to use in the typing of the cons branch.

In the general case, checking whether a pattern-matching branch is useless is undecidable [2, 11, 7, 9]. To remain in a decidable framework, we propose to only address the detection of clauses whose inaccessibility is provable using a simple evidence based on first-order unification of the inductive structure of the indices. The idea is to generate, for each constructor in a pattern matching definition,

a set of equations between the indices of the inductive type in question, in the same way as shown at the beginning of this section. The goal is then to find a unification substitution for these equations. In the case of tail, the unification for nil fails, while the unification for cons succeeds. This approach is based on McBride and McKinna [7] and Norell [8] and is described in detail in Sect. 4.

3 The Calculus of Inductive Constructions

In this section, we give a (necessarily) short description of CIC, specially focusing on inductive types and pattern matching.

The sorts of CIC are $\texttt{Set}$, $\texttt{Prop}$ and $\texttt{Type}_i$, for $i \in \mathbb{N}$. The terms are variables, λ-abstractions $\lambda x : T.M$, applications $M\,N$, products $\Pi x : T.U$ (we write $T \to U$ if x is not used in U), local definitions $[x := N : T]\,M$ [4], and constructions related to inductive types that are described below. We use $\mathrm{FV}(M)$ to denote the set of free variables of M, and $M[x := N]$ to denote the term obtained by substituting every free occurrence of x in M with N.

A context is a sequence of *declarations*, i.e., *assumptions* of the form $(x : T)$ or *definitions* of the form $(x := M : T)$; the empty context is denoted by $[]$. We use $\mathcal{D}om\,(\Gamma)$ to denote the ordered sequence of variables declared in Γ.

We use $m, n, k, p, q, t, u, M, N, P, T, U, \ldots$ to denote terms, $x, y, z, \ldots$ to denote variables, $\Gamma, \Delta, \Theta, \ldots$ to denote contexts and the letter s and its variants to denote sorts. We use $\boldsymbol{X}$ to denote a sequence of X, ε to denote the empty sequence, and $\#\,(\boldsymbol{X})$ to denote the length of the sequence $\boldsymbol{X}$. We use de Bruijn telescopes: the notation $\Pi\Delta.T$ (resp. $\lambda\Delta.T$) abbreviates the iterated expansion of the declarations in Δ into products (resp. abstractions) or local definitions.

CIC comes equipped with a notion of convertibility between terms, written $\Gamma \vdash T \approx U$ and a notion of subtyping, written $\Gamma \vdash T \leq U$.

We consider two typing judgments:

- $\Gamma \vdash t : T$ means that, under context Γ, the term t has type T;
- $\Gamma \vdash \boldsymbol{t} : \Delta$ means that, under context Γ, the terms $\boldsymbol{t}$ form an instance of Δ.[5]

Inductive Types. Terms of CIC also include names of inductive types, names of constructors, fixpoint declarations $\mathsf{fix}_n f : T := M$, and pattern matching $\texttt{match}\ M\ \texttt{as}\ x\ \texttt{in}\ I\,\boldsymbol{p}\,\boldsymbol{y}\ \texttt{return}\ P\ \texttt{with}\ \{C_i\,\boldsymbol{z_i} \Rightarrow t_i\}_i$. We use the letter I and its variants to denote inductive types, and C to denote constructors.

Inductive definitions are declared in a *signature*. A signature Σ is a sequence of declarations of the form

$$\mathsf{Ind}(I[\Delta_p] : \Pi\Delta_a.s := \{C_i : \Pi\Delta_i.I\,\mathcal{D}om\,(\Delta_p)\ \boldsymbol{u_i}\}_i)$$

where I is the name of the inductive type, Δ_p is the context of its parameters, Δ_a is the context of its indices, s is a sort denoting the universe where the type is

[4] Local definitions are not part of the usual definition of CIC. We have include them here because they play an important part in the typing of pattern matching.

[5] In particular, if $f : \Pi\Delta.T$ then $f\ \boldsymbol{t}$ is well-typed.

defined. In the general case, due to the dependency over parameters and indices, I is an inductive family. The type of I is then $\Pi\Delta_p\Delta_a.s$. To the right of the $:=$ symbol are names and types of the constructor. We assume in the sequel that all typing rules are parameterized by a fixed signature Σ.

We describe in detail the pattern matching mechanism. In a term of the form $(\texttt{match}\ M\ \texttt{as}\ x\ \texttt{in}\ I\,\boldsymbol{p}\,\boldsymbol{y}\ \texttt{return}\ P\ \texttt{with}\ \{C_i\,\boldsymbol{x_i} \Rightarrow t_i\}_i)$, M is the term to destruct, P is the return type (which depends on $\boldsymbol{y}$ and x), and t_i represent the body of the i-th branch, with $\boldsymbol{x_i}$ the arguments of the i-th constructor, bound in t_i. The reduction rule associated, denoted ι, is the following:

$$(\texttt{match}\ C_j\,\boldsymbol{t}\,\boldsymbol{a}\ \texttt{as}\ x\ \texttt{in}\ I\,\boldsymbol{p}\,\boldsymbol{y}\ \texttt{return}\ P\ \texttt{with}\ \{C_i\,\boldsymbol{x_i} \Rightarrow t_i\}_i) \ \rightarrow_\iota\ t_j[\boldsymbol{x_j} := \boldsymbol{a}]$$

where, by typing invariants, $\boldsymbol{t} \approx \boldsymbol{p}$ and $\#(\boldsymbol{x_j}) = \#(\boldsymbol{a})$.

Figure 1 shows the typing rules of pattern matching, where we have $\Delta_i^* = \Delta_i[\mathcal{D}om\,(\Delta_p) := \boldsymbol{p}]$, $\boldsymbol{u_i^*} = \boldsymbol{u_i}[\mathcal{D}om\,(\Delta_p) := \boldsymbol{p}]$, and $\Delta_a^* = \Delta_a^*[\mathcal{D}om\,(\Delta_p) := \boldsymbol{p}]$. To be accepted, a `match` construction has to satisfy a predicate ELIM that takes as argument the inductive type and the sort of the return type [10] (the exact definition of ELIM is not important in our context).

The typing rule for pattern matching is complicated by the fact that the return type can depend on the term being destructed. If P does not depend on x nor $\boldsymbol{y}$, i.e. they are not used in P, then the typing reduces to something close to non-dependent languages like Haskell or ML. But P can depend on x (of type an instance of I), and, therefore, it should also depend on the indices of the type of x (i.e. $\boldsymbol{y}$). In each branch, we instantiate both x with the corresponding constructor applied to the arguments of the branch, and $\boldsymbol{y}$ with the indices of the inductive type corresponding to that constructor. Finally, the type of the whole `match` is obtained by replacing x with M in P, and $\boldsymbol{y}$ accordingly.

$$\frac{\begin{array}{c}\mathsf{Ind}(I[\Delta_p] : \Pi\Delta_a.s := \{C_i : \Pi\Delta_i.I\,\mathcal{D}om\,(\Delta_p)\,\boldsymbol{u_i}\}_i) \in \Sigma \\ \Gamma \vdash M : I\,\boldsymbol{p}\,\boldsymbol{u} \qquad \Gamma(\boldsymbol{y} : \Delta_a^*)(x : I\,\boldsymbol{p}\,\boldsymbol{y}) \vdash P : s \\ \textsc{Elim}(I, s) \qquad \Gamma(\boldsymbol{x_i} : \Delta_i^*) \vdash t_i : P[\boldsymbol{y} := \boldsymbol{u_i^*}][x := C_i\,\boldsymbol{p}\,\boldsymbol{x_i}]\end{array}}{\Gamma \vdash \texttt{match}\ M\ \texttt{as}\ x\ \texttt{in}\ I\,\boldsymbol{p}\,\boldsymbol{y}\ \texttt{return}\ P\ \texttt{with}\ \{C_i\,\boldsymbol{x_i} \Rightarrow t_i\}_i : P[\boldsymbol{y} := \boldsymbol{u}][x := M]}$$

Fig. 1. Typing rules for pattern matching in CIC

4 A New Elimination Rule

In this section we present the new rule for pattern matching. We modify the syntax of terms with the construction

$$\texttt{match}\ M\ \texttt{as}\ x\ \texttt{in}\ [\Delta]\,I\,\boldsymbol{p}\,\boldsymbol{t}\ \texttt{where}\ \Delta := \boldsymbol{q}\ \texttt{return}\ P\ \texttt{with}\ \{C_i\,\boldsymbol{x_i} \Rightarrow b_i\}_i$$

where the body of a branch (b_i above) can be either the symbol $\bot$ or a term of the form N $\texttt{where}$ $\boldsymbol{z_i} := \boldsymbol{u_i}$. The rôle of $[\Delta]\, I\, \boldsymbol{p}\, \boldsymbol{t}$ 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 $\bot$. On the other hand, some constructors may (partially) belong to the subfamily, so the bodies of the corresponding branches are of the form N $\texttt{where}$ $\boldsymbol{z_i} := \boldsymbol{u_i}$, where N is the body proper and $\boldsymbol{z_i} := \boldsymbol{u_i}$ defines some restrictions on the arguments of the constructor that need to be satisfied in order to belong to the subfamily.

Before showing the typing rule for this new construction, that is more complex than the one presented in the previous section, we show how to write our running example, the tail function, as follows:

$$
\begin{array}{l}
\mathsf{tail} := \lambda(n : \mathsf{nat})(v : \mathsf{vector}\,(\mathsf{S}\,n)).\\
\quad \texttt{match}\ v\ \texttt{as}\ x\ \texttt{in}\ [(n_0 : \mathsf{nat})]\,\mathsf{vector}\,(\mathsf{S}\,n_0)\ \texttt{where}\ n_0 := n\ \texttt{return}\ \mathsf{vector}\,n_0\\
\quad \texttt{with} \mid \mathsf{nil} \Rightarrow \bot \mid \mathsf{cons}\,k\,x\,v' \Rightarrow v'\ \texttt{where}\ n_0 := k
\end{array}
$$

Comparing with the generic term above, M is v, Δ is $(n_0 : \mathsf{nat})$, and $\boldsymbol{q}$ is n. We explain how to check that this definition is accepted. Note that we are targeting a particular subfamily of the inductive type, $[(n_0 : \mathsf{nat})]\,\mathsf{vector}\,(\mathsf{S}\,n_0)$, parametrized by n_0, which is the subfamily of non-empty vectors. We need to make sure that v belongs to that family. In the general case, this means instantiating Δ with $\boldsymbol{q}$ and checking that M has type $I\,\boldsymbol{p}\,(\boldsymbol{t}[\mathcal{D}om\,(\Delta) := \boldsymbol{q}])$. In the particular case of tail, we check that v has type $\mathsf{vector}\,(\mathsf{S}\,n_0)[n_0 := n]$.

Let us look at the return type. In the general case, the return type P depends on x and Δ. Hence, we need to check that

$$\Gamma\Delta(x : I\,\boldsymbol{p}\,\boldsymbol{t}) \vdash P : s$$

where Γ is the context where we are typing the whole $\texttt{match}$. In the particular case of tail, the return type depends on n_0 but not on x.

Finally, we look at the branches. In the nil case, it is clear that this constructor does not belong to the subfamily $[(n_0 : \mathsf{nat})]\,\mathsf{vector}\,(\mathsf{S}\,n_0)$, since its type is $\mathsf{vector}\,0$. Hence, the branch is simply $\bot$. We call this type of branch *impossible*. How do we 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 (0 in this case), for the variables in $\mathcal{D}om\,(\Delta)$. As we said, this problem is undecidable in general, so we proceed by first-order unification with constructor theory. In this case, $\mathsf{S}\,n_0$ and 0 are not unifiable (constructors are disjoint), and therefore, the branch is effectively impossible.

In the cons case, we have the body proper v' and the substitution $n_0 := k$. The return type of this branch should be $\mathsf{vector}\,n_0$, where x is replaced by $\mathsf{cons}\,k\,x\,v'$, and n_0 is replaced by k as dictated by the substitution. To check this kind of branch, we need to check that the given substitution is correct. This is done, as for impossible branches, by unification. In this case, unifying $\mathsf{S}\,n_0$ with $\mathsf{S}\,k$ (this last value corresponds to the index of $\mathsf{cons}\,k\,x\,v'$), for n_0. Since S is injective,

the result is the substitution $\{n_0 \mapsto k\}$. If the unification succeeds, we apply the substitution obtained in the return type. In the case of the cons branch, its type should be $\mathsf{vector}\,(\mathsf{S}\,n_0)[n_0 := k]$.

Note that the procedure to check branches is similar for both kinds. First, we unify the indices of the subfamily with the indices of the constructor. If they are not unifiable, the branch is impossible. If the unification succeeds, we obtain a substitution σ that is applied to the return type in order to check the body proper of the branch. There is one third possibility, though. Since the unification problem is undecidable, it is possible that the procedure gets stuck. In that case, we simply give up, and the typechecking fails.

We now proceed to explain formally the typing rule for this `match`. The presentation is divided into three parts: first, we describe substitutions, then the unification judgment, and finally, we show the typechecking of branches and put everything together. Then, we discuss the associated reduction rule.

Substitutions. 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 $\mathcal{D}om\,(\sigma)$. Given a term t and a substitution σ, we write $t\sigma$ to mean the term obtained by substituting every free variable x of t with $\sigma(x)$. The set of free variables of a pre-substitution σ is defined as $\mathrm{FV}(\sigma) = \cup_{x \in \mathcal{D}om(\sigma)} \mathrm{FV}(x\sigma)$.

A *substitution* σ from Γ to Δ, denoted $\sigma : \Gamma \to \Delta$ is a pre-substitution with $\mathcal{D}om\,(\sigma) \subseteq \Gamma$, idempotent (i.e., $\mathrm{FV}(\sigma) \cap \mathcal{D}om\,(\sigma) = \emptyset$), such that for every $(x : T) \in \Gamma$, $\Delta \vdash x\sigma : T\sigma$, and for every $(x := t : T) \in \Gamma$, $\Delta \vdash x\sigma \approx t\sigma : T\sigma$. [6] We use $\sigma, \rho, \delta, \ldots$ to denote (pre-)substitutions. We sometimes write $\Gamma \vdash \sigma : \Delta \to \Theta$ to denote a substitution $\sigma : \Gamma\Delta \to \Gamma\Theta$, with $\mathcal{D}om\,(\sigma) \subseteq \mathcal{D}om\,(\Delta)$.

Unification. We now describe in detail the unification judgment. A unification problem is written:

$$\Gamma; \Delta, \zeta \vdash [\boldsymbol{u} = \boldsymbol{u'} : \Theta]$$

meaning that $\boldsymbol{u}$ and $\boldsymbol{u'}$ have type Θ under context $\Gamma\Delta$, and $\zeta \subseteq \mathcal{D}om\,(\Delta)$ is the set of variables that are open to unification. Context Γ is intended to be the "outer context", i.e. the context where we want to type a `match` construction, while context Δ is defined inside the `match`. We only allow to unify variables in Δ, so that the unification is invariant under substitutions and reductions that happen outside the `match`. This is important in the proofs of the Substitution Lemma and Subject Reduction.

The unification judgment is defined by the rules of Fig. 2. These rules are based on the unification given in [7, 8], with a notation close to that in [8]. Trying to unify $\boldsymbol{u}$ and $\boldsymbol{u'}$ may have one of three possible outcomes:

positive success A derivation $\Gamma; \Delta, \zeta \vdash [\boldsymbol{u} = \boldsymbol{u'} : \Theta] \mapsto \Delta', \zeta' \vdash \sigma$ is obtained, meaning that σ is a substitution ($\Gamma \vdash \sigma : \Delta \to \Delta'$) that unifies $\boldsymbol{u}$ and $\boldsymbol{u'}$ with domain $\zeta \setminus \zeta'$ and ζ' is the set of variables that are still open to unification;

[6] The judgment $\Gamma \vdash t \approx u : T$ is shorthand for $\Gamma \vdash t : T$, $\Gamma \vdash u : T$ and $\Gamma \vdash t \approx u$.

negative success A derivation $\Gamma; \Delta, \zeta \vdash [\boldsymbol{u} = \boldsymbol{u'} : \Theta] \mapsto \bot$ is obtained, meaning that $\boldsymbol{u}$ and $\boldsymbol{u'}$ are not unifiable;

failure No rule is applicable, hence no derivation is obtained (the unification problem is too difficult).

In the rules (U-VarL) and (U-VarR), 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

$$\begin{aligned}(\Delta^0(x:T)\Delta^1)_{\Gamma|x:=t} &= \Delta^0\Delta^t(x := t : T)\Delta_t \\ &\quad \text{where } (\Delta^t, \Delta_t) = \textsc{Strengthen}(\Delta_1, t, T) \\ &\qquad \Gamma\Delta^0\Delta^t \vdash t : T \\ &\qquad \Gamma\Delta^0\Delta^t(x:T) \vdash \Delta_t\end{aligned}$$

The Strengthen operation [7] is defined as

$$\textsc{Strengthen}([], t, T) = ([], [])$$

$$\textsc{Strengthen}((x:U)\Delta, t, T) = \begin{cases}((x:U)\Delta_0, \Delta_1) & \text{if } x \in \mathrm{FV}(\Delta_0) \cup \mathrm{FV}(t) \cup \mathrm{FV}(T) \\ (\Delta_0, (x:U)\Delta_1) & \text{if } x \notin \mathrm{FV}(\Delta_0) \cup \mathrm{FV}(t) \cup \mathrm{FV}(T)\end{cases}$$

$$\text{where } (\Delta_0, \Delta_1) = \textsc{Strengthen}(\Delta, t, T)$$

We give some informal explanations of 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 $\Delta_{\Gamma|x:=v}$.

Rules (U-Discr) and (U-Inj) codify the no-confusion property: 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). Finally, rules (U-Empty) and (U-Tel) concern the unification of sequence of terms. Missing from Fig. 2 are the corresponding rules to (U-Inj) and (U-Tel) that propagate a negative unification (i.e., $\bot$).

The typing rule. In Fig. 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

$$\Gamma; \Delta_i; \Delta; [\boldsymbol{u} = \boldsymbol{v} : \Theta] \vdash b : T$$

The intuition is that we take the unification problem $\Gamma; \Delta_i\Delta, \zeta \vdash [\boldsymbol{u} = \boldsymbol{v} : \Theta]$ (where ζ depends on the kind of branch considered), and take the result of the unification into account while checking the body of the branch. This judgment is defined by the rules (B-$\bot$) and (B-Sub) in Fig. 3. In rule (B-$\bot$),

$$\text{(U-VarL)}\quad \frac{x \in \zeta \qquad x \notin \mathrm{FV}(v)}{\Gamma; \Delta, \zeta \vdash [x = v : T] \mapsto \Delta_{\Gamma|x:=v}, \zeta \setminus \{x\} \vdash \{x \mapsto v\}}$$

$$\text{(U-VarR)}\quad \frac{x \in \zeta \qquad x \notin \mathrm{FV}(v)}{\Gamma; \Delta, \zeta \vdash [v = x : T] \mapsto \Delta_{\Gamma|x:=v}, \zeta \setminus \{x\} \vdash \{x \mapsto v\}}$$

$$\text{(U-Discr)}\quad \frac{C_1 \neq C_2}{\Gamma; \Delta, \zeta \vdash [C_1\, \boldsymbol{u} = C_2\, \boldsymbol{v} : T] \mapsto \bot}$$

$$\text{(U-Inj)}\quad \frac{\Gamma; \Delta, \zeta \vdash [\boldsymbol{u} = \boldsymbol{v} : \Theta] \mapsto \Delta', \zeta' \vdash \sigma \qquad \mathsf{Type}(C) = \Pi\Theta.I\, \boldsymbol{p}\, \boldsymbol{a}}{\Gamma; \Delta, \zeta \vdash [C\, \boldsymbol{u} = C\, \boldsymbol{v} : T] \mapsto \Delta', \zeta' \vdash \sigma}$$

$$\text{(U-Conv)}\quad \frac{\Gamma \Delta \vdash u \approx v}{\Gamma; \Delta, \zeta \vdash [u = v : T] \mapsto \Delta, \zeta \vdash id}$$

$$\text{(U-Empty)}\quad \frac{}{\Gamma; \Delta, \zeta \vdash [\varepsilon = \varepsilon : []] \mapsto \Delta, \zeta \vdash id}$$

$$\text{(U-Tel)}\quad \frac{\begin{array}{c}\Gamma; \Delta, \zeta \vdash [u = v : T] \mapsto \Delta_1, \zeta_1 \vdash \sigma_1 \\ \Gamma; \Delta_1, \zeta_1 \vdash [\boldsymbol{u}\sigma_1 = \boldsymbol{v}\sigma_1 : \Theta[x := u]\sigma_1] \mapsto \Delta_2, \zeta_2 \vdash \sigma_2\end{array}}{\Gamma; \Delta, \zeta \vdash [u\, \boldsymbol{u} = v\, \boldsymbol{v} : (x : T)\Theta] \mapsto \Delta_2, \zeta_2 \vdash \sigma_1\sigma_2}$$

Fig. 2. Unification rules

that corresponds to impossible branches, we take ζ to be $\mathcal{D}om\,(\Delta_i) \cup \mathcal{D}om\,(\Delta)$, and we check that the unification succeeds negatively.

In rule (B-Sub), that corresponds to possible branches, we take ζ to be $\mathcal{D}om\,(\Delta)$ together with the domain of the substitution given in the branch (ρ in this case). Δ corresponds to the variables that define the subfamily under analysis. We check that the unification succeed positively, leaving no variables open, and that the resulting substitution (σ) is convertible with the one given in the branch (ρ). Then, we typecheck the body proper of the branch using the context given by the unification (Δ').

Finally, in the rule (T-Match) we put everything together. We have $\Delta_i^* = \Delta_i[\mathcal{D}om\,(\Delta_p) := \boldsymbol{p}]$, $\boldsymbol{u_i^*} = \boldsymbol{u_i}[\mathcal{D}om\,(\Delta_p) := \boldsymbol{p}]$, and $\Delta_a^* = \Delta_a^*[\mathcal{D}om\,(\Delta_p) := \boldsymbol{p}]$.

The subfamily under analysis is defined by $[\Delta]\, I\, \boldsymbol{p}\, \boldsymbol{t}$, hence, we check that M belongs to it by checking that $\Gamma \vdash \boldsymbol{u} \approx \boldsymbol{t}[\Delta := \boldsymbol{q}]$. We also check that $\boldsymbol{q}$ has the correct type; and also that P is a type. The predicate P depends on x and $\mathcal{D}om\,(\Delta)$, similarly to the old rule, where P depended on x and $\boldsymbol{y}$ (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 replace $\boldsymbol{y}$, there are no obvious values we can give to the variables in $\mathcal{D}om\,(\Delta)$. Therefore, we try the unification between $\boldsymbol{t}$ (that defines the subfamily under analysis) and $\boldsymbol{u_i}$ (the indices in the type of x). Since, for possible branches, the unification does not leave open variables, we effectively find a value for each variable in $\mathcal{D}om\,(\Delta)$.

$$\text{(B-}\bot\text{)}\quad \frac{\Gamma; \Delta_i \Delta, \mathcal{D}om\,(\Delta_i) \cup \mathcal{D}om\,(\Delta) \vdash [\boldsymbol{u} = \boldsymbol{v} : \Theta] \mapsto \bot}{\Gamma; \Delta_i; \Delta; [\boldsymbol{u} = \boldsymbol{v} : \Theta] \vdash \bot : P}$$

$$\text{(B-Sub)}\quad \frac{\begin{array}{c}\Gamma; \Delta_i \Delta, \mathcal{D}om\,(\rho) \cup \mathcal{D}om\,(\Delta) \vdash [\boldsymbol{u} = \boldsymbol{v} : \Theta] \mapsto \Delta', \emptyset \vdash \sigma \\ \Gamma\Delta' \vdash t : P \qquad \Gamma\Delta' \vdash \sigma \approx \rho\end{array}}{\Gamma; \Delta_i; \Delta; [\boldsymbol{u} = \boldsymbol{v} : \Theta] \vdash t\ \texttt{where}\ \rho : P}$$

$$\text{(T-Match)}\ \frac{\begin{array}{c}\mathsf{Ind}(I[\Delta_p] : \Pi \Delta_a.s := \{C_i : \Pi \Delta_i.I\,\mathcal{D}om\,(\Delta_p)\ \boldsymbol{u_i}\}_i) \in \Sigma \\ \Gamma \vdash M : I\,\boldsymbol{p}\,\boldsymbol{u} \qquad \Gamma \vdash \boldsymbol{u} \approx \boldsymbol{t}[\Delta := \boldsymbol{q}] \qquad \Gamma\Delta(x : I\,\boldsymbol{p}\,\boldsymbol{t}) \vdash P : s \\ \Gamma \vdash \boldsymbol{q} : \Delta \qquad \Gamma; (\boldsymbol{z_i} : \Delta_i^*); \Delta; [\boldsymbol{u_i^*} = \boldsymbol{t} : \Delta_a^*] \vdash b_i : P[x := C_i\,\boldsymbol{p}\,\boldsymbol{z_i}]\end{array}}{\Gamma \vdash \begin{pmatrix}\texttt{match}\ M\ \texttt{as}\ x\ \texttt{in}\ [\Delta]\,I\ \boldsymbol{p}\ \boldsymbol{t}\ \texttt{where}\ \Delta := \boldsymbol{q} \\ \texttt{return}\ P\ \texttt{with}\ \{C_i\ \boldsymbol{z_i} \Rightarrow b_i\}_i\end{pmatrix} : P[\Delta := \boldsymbol{q}][x := M]}$$

Fig. 3. Typing rules for the new elimination rule

Reduction. The reduction rule is the same as the original elimination rule of CIC, except that it is only applicable to possible constructors.

$$(\texttt{match}\ C_j\,\boldsymbol{t}\,\boldsymbol{a}\ \texttt{as}\ x\ \texttt{in}\ I\,\boldsymbol{p}\,\boldsymbol{y}\ldots\texttt{with}\ \{C_i\,\boldsymbol{x_i} \Rightarrow b_i\}_i) \ \rightarrow_\iota\ t_j[\boldsymbol{x_j} := \boldsymbol{a}]$$

where $b_j = (t_j\ \texttt{where}\ \sigma_j)$, $\#\,(\boldsymbol{t}) = \#\,(\boldsymbol{p})$ and $\#\,(\boldsymbol{x_i}) = \#\,(\boldsymbol{a})$.

In the compatible closure of the reduction, we do not use the substitutions of each branch. Hence, we have the following rule

$$\frac{\Gamma(\boldsymbol{z_i} : \Delta_i) \vdash t_j \rightarrow t'_j}{\Gamma \vdash \texttt{match}\ldots C\,\boldsymbol{z_i} \Rightarrow t_j\ \texttt{where}\ \sigma_j \rightarrow \texttt{match}\ldots C\,\boldsymbol{z_i} \Rightarrow t'_j\ \texttt{where}\ \sigma_j}$$

Allowing the substitution as part of Γ when reducing the body of the branch t_j would mean to break confluence on pseudoterms (although, in that case, confluence remains valid for well-typed terms).

Remarks. The unification algorithm needs the *domain* of the substitutions to compute, but not its values (similar to the situation of *inaccessible patterns* in [3, 8]). In the examples below, we sometimes omit, for readability, parts of the substitutions that can be automatically inferred by the unification; in some other cases, the substitutions are "inlined" in the arguments of a constructor.

Note that the usual rule for pattern matching in CIC is a special case of the new rule: we just set Δ to be the context of indices of the inductive type, and $\boldsymbol{t}$ to be $\mathcal{D}om\,(\Delta)$. It is not difficult to see that the unification succeeds positively for each branch.

4.1 Examples

We illustrate the new elimination rule with some examples. We have already seen how to type the tail function. We show two sets of examples, one about Streicher's K axiom and heterogeneous equality, and the other about the less-or-equal relation on natural numbers. To simplify the syntax, we assume that missing constructors are impossible.

Streicher's K axiom. This axiom, also known as *uniqueness of reflexivity proofs*, has the following type:

$$\Pi(X : Set)(x : X)(P : \mathsf{eq}\, X\, x\, x \to \mathtt{Prop}).P(\mathsf{refl}\, X\, x)) \to \Pi p : \mathsf{eq}\, X\, x\, x.P\, p$$

In [5], McBride proved that axiom K is equivalent to heterogeneous equality (together with its elimination rule), defined as follows:

$$\begin{array}{l}\mathsf{Ind}(\mathsf{Heq}(A : \mathtt{Set})(x : A) : \Pi(B : \mathtt{Set}).B \to \mathtt{Prop} := \mathsf{Hrefl} : \mathsf{Heq}\, A\, x\, A\, x) \\ \mathsf{Subst} : \Pi(A : \mathtt{Set})(x\, y : A).P\, x \to \mathsf{Heq}\, A\, x\, A\, y \to P\, y\end{array}$$

Note that the derived induction principle for this equality is not very useful. Therefore, McBride proposed the more conservative elimination rule given above: only homogeneous equations can be eliminated.

Axiom K and the elimination rule for Heq are not derivable in CIC [4], but it is no surprise that they are derivable with the new rule:

$$\begin{array}{rl} K := & \lambda(X : \mathtt{Set})(x : X)(P : \mathsf{eq}\, X\, x\, x \to \mathtt{Prop})(H : P(\mathsf{refl}\, X\, x))(p : \mathsf{eq}\, X\, x\, x). \\ & \mathtt{match}\ p\ \mathtt{as}\ p_0\ \mathtt{in}\ []\, \mathsf{eq}\, X\, x\, x\ \mathtt{return}\ P\, p_0\ \mathtt{with}\ \mathsf{refl} \Rightarrow H \\ \mathsf{Subst} := & \lambda(X : \mathtt{Set})(x\, y : X)(P : X \to \mathtt{Set})(M : P\, x)(H : \mathsf{Heq}\, A\, x\, A\, y). \\ & \mathtt{match}\ H\ \mathtt{as}\ h_0\ \mathtt{in}\ [(y_0 : X)]\, \mathsf{Heq}\, A\, x\, A\, y_0\ \mathtt{where}\ y_0 := y\ \mathtt{return}\ P\, y_0 \\ & \mathtt{with}\ \mathsf{Hrefl} \Rightarrow M\ \mathtt{where}\ y_0 := x \end{array}$$

Less-or-equal relation on natural numbers. We show two examples concerning the relation less-or-equal for natural numbers defined inductively as follows:

$$\begin{array}{l}\mathsf{Ind}(\mathsf{leq} : \mathsf{nat} \to \mathsf{nat} \to \mathtt{Prop} := \mathsf{leq0} : \Pi(n : \mathsf{nat})\mathsf{leq}\, 0\, n, \\ \qquad\qquad\qquad\qquad\qquad\qquad \mathsf{leqS} : \Pi(m\, n : \mathsf{nat}).\mathsf{leq}\, n\, m \to \mathsf{leq}\, (\mathsf{S}\, m)\, (\mathsf{S}\, n))\end{array}$$

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 $\Pi(n : \mathsf{nat})(H : \mathsf{leq}\, (\mathsf{S}\, n)\, n).\mathsf{False}$. One possible solution is to take

$$\begin{array}{l}\mathtt{fix}\, f : \Pi(n : \mathsf{nat}).\mathsf{leq}(\mathsf{S}\, n)n \to \mathsf{False} := \\ \mathtt{match}\ H\ \mathtt{as}\ h_0\ \mathtt{in}\ [(n_0 : \mathsf{nat})]\, \mathsf{leq}(\mathsf{S}\, n_0)n_0\ \mathtt{where}\ n_0 := n\ \mathtt{return}\ \mathsf{False}\ \mathtt{with} \\ \mid \mathsf{leqS}\, x\, y\, H \Rightarrow f\, y\, H\ \mathtt{where}\ (x := \mathsf{S}\, y)(n_0 := \mathsf{S}\, y)\end{array}$$

In the leq0 branch, the unification problem considered is $\{x, n_0\} \vdash [0, x = \mathsf{S}\, n_0, n_0]$, which clearly 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 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 the context

$$(y : \mathsf{nat})(x := \mathsf{S}\, y : \mathsf{nat})(H : \mathsf{leq}\, x\, y)\ .$$

Note the reordering of x and y.

The second example shows that the relation leq is transitive. That is, we want to find a term of type $\Pi(x\,y\,z : \mathsf{nat}).\mathsf{leq}\,x\,y \to \mathsf{leq}\,y\,z \to \mathsf{leq}\,x\,z$. Using the implicit syntax mentioned above, one possible solution is

$$
\begin{array}{l}
\mathtt{fix}\ \mathit{trans} : \Pi(m\,n\,k : \mathsf{nat}).\mathsf{leq}\,m\,n \to \mathsf{leq}\,n\,k \to \mathsf{leq}\,m\,k := \\
\quad \lambda(m\,n\,k : \mathsf{nat})(H_1 : \mathsf{leq}\,m\,n)(H_2 : \mathsf{leq}\,n\,k). \\
\quad (\mathtt{match}\ H_1\ \mathtt{in}\ [(m_1\,n_1 : \mathsf{nat})]\,\mathsf{leq}\,m_1\,n_1\ \mathtt{return}\ \mathsf{leq}\,n_1\,k \to \mathsf{leq}\,m_1\,k\ \mathtt{with} \\
\quad | \boxed{\mathsf{leq0}\,x} \Rightarrow \lambda(h_2 : \mathsf{leq}\,x\,k).\ \mathsf{leq0}\,k \\
\quad | \boxed{\mathsf{leqS}\,x\,y\,H} \Rightarrow \lambda(h_2 : \mathsf{leq}\,(\mathsf{S}\,y)\,k). \\
\qquad \mathtt{match}\ h_2\ \mathtt{in}\ [(k_2 : \mathsf{nat})]\,\mathsf{leq}\,(\mathsf{S}\,y)\,k_2\ \mathtt{return}\ \mathsf{leq}\,(\mathsf{S}\,x)\,k_2\ \mathtt{with} \\
\qquad | \boxed{\mathsf{leqS}\,(\_ := y)\,y'\,H'} \Rightarrow\ \mathsf{leqS}\,x\,y'\,(\mathit{trans}\,H\,H')\ )\,H_2
\end{array}
$$

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 simple; 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 using the dotted pattern in the second case analysis, which guarantees that the recursive call is well-typed. In Agda, transitivity of leq can be defined as

$$
\begin{array}{l}
\mathit{trans} : (m\,n\,k : \mathsf{nat}) \to \mathsf{leq}\,m\,n \to \mathsf{leq}\,n\,k \to \mathsf{leq}\,m\,k \\
\mathit{trans}\ .0\ .x\ k\ \boxed{(\mathsf{leq0}\,x)}\ \_ = \mathsf{leq0}\,k \\
\mathit{trans}\ .(\mathsf{S}\,x)\ .(\mathsf{S}\,y)\ .(\mathsf{S}\,y')\ \boxed{(\mathsf{leqS}\,x\,y\,H)}\ \boxed{(\mathsf{leqS}\ .y\,y'\,H')} = \mathsf{leqS}\,x\,y'\,(\mathit{trans}\,H\,H')
\end{array}
$$

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).

5 Metatheory

In this section we state some metatheoretical properties about the system. The most important result of this section is consistency which follows as a corollary to Lemma 4. The substitution lemma is still valid with the new rule.

Lemma 1. *If $\Gamma \vdash t : T$ and $\sigma : \Gamma \to \Delta$, then $\Delta \vdash t\sigma : T\sigma$.*

The following lemma formally states the intuitive meaning of a successful unification judgment.

Lemma 2. *Let $\Gamma; \Delta, \zeta \vdash [\boldsymbol{u} = \boldsymbol{v} : \Theta] \mapsto \Delta', \zeta' \vdash \sigma$ be a unification judgment, with $\Gamma\Delta \vdash \boldsymbol{u} : \Theta$, and $\Gamma\Delta \vdash \boldsymbol{v} : \Theta$. Then $\Gamma \vdash \sigma : \Delta \to \Delta'$, and $\Gamma\Delta' \vdash \boldsymbol{u}\sigma \approx \boldsymbol{v}\sigma$.*

The proof of Subject Reduction proceeds by induction on the typing derivation. The only difficult case is, of course, the new elimination rule.

Lemma 3 (Subject Reduction). *If $\Gamma \vdash M : T$, and $\Gamma \vdash M \to M'$, then $\Gamma \vdash M' : T$.*

To prove consistency, we define a type-preserving translation of our system, to the system CIC+Heq, which is CIC together with the elimination rule of heterogeneous equality (remember that this rule is not derivable in CIC, while it is in our system). Therefore, our consistency result is relative to the consistency of CIC+Heq. The translation is similar to the translation described in [3].

The translation function is written $[\![\ ]\!]$ and defined by structural induction on the terms. The only interesting case is that of the new elimination rule. The intuitive idea is to generate the term we usually build in CIC by generating equalities between the indices of the inductive type, as described in Sect. 2. Given a term

$$\texttt{match}\ M\ \texttt{as}\ x\ \texttt{in}\ [\Delta]\, I\, \boldsymbol{p}\, \boldsymbol{t}\ \texttt{where}\ \Delta := \boldsymbol{q}\ \texttt{return}\ P\ \texttt{with}\ \{C_i\, \boldsymbol{x_i} \Rightarrow b_i\}_i$$

its translation along $[\![\ ]\!]$ is (we use $=$ to denote Heq omitting types for readability)

$$\begin{array}{l}(\texttt{match}\ [\![M]\!]\ \texttt{as}\ z\ \texttt{in}\ I\ [\![\boldsymbol{p}]\!]\ \boldsymbol{y} \\ \texttt{return}\ \Pi [\![\Delta]\!](x : I\ [\![\boldsymbol{p}]\!]\ [\![\boldsymbol{t}]\!]).\boldsymbol{y} = [\![\boldsymbol{t}]\!] \to x = z \to [\![P]\!]\ \texttt{with} \\ \{C_i\ \boldsymbol{z_i} \Rightarrow B_i\}_i)\ [\![\boldsymbol{q}]\!]\ [\![M]\!]\ (\mathsf{Hrefl}\ [\![\boldsymbol{t}[\mathcal{D}om\,(\Delta) := q]]\!])\ (\mathsf{Hrefl}\ [\![M]\!])\end{array}$$

Note that we generalize equalities between the indices, in the same way as shown in Sect. 2. This is where we need heterogeneous equality (for instance, observe that x and z have different types in $x = z$). We also generalize over Δ and x, and then apply the resulting term to $[\![\boldsymbol{q}]\!]$ (for Δ), $[\![M]\!]$ (for x), and trivial proofs of equality (for the equalities between indices). Each branch B_i takes the form

$$\lambda [\![\Delta]\!](x : I\ [\![\boldsymbol{p}]\!]\ [\![\boldsymbol{t}]\!])(\boldsymbol{H} : [\![\boldsymbol{u_i}]\!] = [\![\boldsymbol{t}]\!])(H : x = (C_i\ [\![\boldsymbol{p}]\!]\ \boldsymbol{z_i})). \ldots$$

We also define a translation of the unification judgment, that takes as input the sequence of equalities $\boldsymbol{H}$, and returns a sequence of terms whose type corresponds to the substitution of the branch (if the unification succeeds positively); or a proof of contradiction (if the unification succeeds negatively). In the latter case, we are done, while in the former, we use the returned sequence of terms to rewrite in the translation of the body proper, and obtain a term of the right return type. We can prove that typing is preserved by this translation:

Lemma 4. *If $\Gamma \vdash M : T$, then $[\![\Gamma]\!] \vdash_{CIC+Heq} [\![M]\!] : [\![T]\!]$.*

Since $[\![\forall P : \texttt{Prop}.P]\!] = \forall P : \texttt{Prop}.P$, consistency of our system with respect to consistency of CIC+Heq follows immediately.

Corollary 1. *If CIC+Heq is consistent, then the new rule is consistent. That is, there is no term M such that $[] \vdash M : (\forall P : Prop.P)$.*

6 Related Work

Coquand [2] 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 [4] later proved that pattern matching is not a conservative extension of Type Theory, by showing that K is not derivable in Type Theory. Finally, Goguen *et al.* [3] 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 [5]. Given this series of results, it is not surprising that axiom K is derivable with the rule we propose.

Two modern presentations of Coquand's work, and also important inspirations for this work, are the programming languages Epigram [6] and Agda [8].

The pattern matching mechanism of Epigram, described by McBride and McKinna in [7], provides a way to reason by case analysis, not only on constructors, but using more general elimination principles. In that sense, it is more general than our approach. They also define a mechanism to perform case analysis on intermediate expressions. 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 also define a simplification method based on first-order unification, that we have reformulated here.

Agda's pattern matching mechanism, described in [8], allows definitions by a sequence of (possibly overlapping) equations, and uses the "with construct" to analyze intermediate expressions, in a similar way to [7]. The first-order unification algorithm used in Agda served as basis of our own presentation. Internally, pattern matching definitions are translated in Agda to a case tree, which is what we directly write in our approach (see the examples of Sect. 4.1).

Oury [9] proposed a different approach to remove impossible cases based on set approximations. His approach allows to remove cases in situations where unification is not sufficient. As mentioned in [9], it remains to see if the combination of both techniques can be used to remove more cases.

Coq [1] provides mechanisms to define functions by pattern matching. The basic pattern-matching algorithm, initially written by Cristina Cornes and extended by Hugo Herbelin, supports for instance omission of impossible cases by encoding the proofs of negative success of the first-order unification process within the return predicate of the `match` expression (see Coq version 8.2beta). Another algorithm of Coq, provided by the `Program` construction of Matthieu Sozeau [12], allows to exploit inversion constraints using heterogeneous equality for typing dependent pattern-matching in a way similar to what is done in Epigram. Because Coq lacks the reduction rule of axiom K, not all definitions built by this algorithm are computable. Our rule not only simplifies the underlying computational structure of programs typed using explicit insertion of heterogeneous equalities but also removes the limitations in code execution that the absence of reduction rule for K induces.

7 Conclusions and Future Work

We have presented a new rule for performing pattern matching in CIC. Functions on inductive families are simpler to write and more efficient using the new rule. Also, the underlying theory is slightly increased by providing axiom K and its reduction rule, which means that the new system is more amenable to use as the basis for a programming language with dependent types.

For future work, the obvious first step is implementation. Since the new rule is not much different from the current elimination rule, adapting it to existent implementations, e.g. Coq, should not be difficult. However, taking full advantage of the new possibilities would mean to redesign many tactics. Also, it could be of interest to implement multi-patterns *à la* Agda, on top of the new rule.

On another direction, there is lots of room for improving the unification. We could add the treatment of circular equations, such as $n = \mathsf{S}\, n$, that are provably false in CIC. Also, it could be of interest to have a more general notion of injective and discriminative constants, so that we are able to write functions by pattern matching when the indices are not necessarily inductive objects.

References

1. The Coq Development Team. *The Coq Reference Manual, version 8.1*, February 2007. Distributed electronically at http://coq.inria.fr/doc.
2. 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.
3. Healfdene Goguen, Conor McBride, and James McKinna. Eliminating dependent pattern matching. In K. Futatsugi, J. P. Jouannaud, and J. Meseguer, editors, *Essays Dedicated to J. Goguen*, volume 4060 of *LNCS*. Springer, 2006.
4. Martin Hofmann and Thomas Streicher. The groupoid model refutes uniqueness of identity proofs. In *LICS*, pages 208–212. IEEE Computer Society, 1994.
5. Conor McBride. *Dependently Typed Functional Programs and their Proofs*. PhD thesis, University of Edinburgh, 1999.
6. 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. http://www.e-pig.org.
7. Conor McBride and James McKinna. The view from the left. *J. Funct. Program.*, 14(1):69–111, 2004.
8. Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.
9. Nicolas Oury. Pattern matching coverage checking with dependent types using set approximations. In A. Stump and H. Xi, editors, *PLPV*, pages 47–56. ACM, 2007.
10. Christine Paulin-Mohring. Inductive definitions in the system Coq - rules and properties. In *TLCA '93*, volume 664 of *LNCS*. Springer, 1993.
11. Carsten Schürmann and Frank Pfenning. A coverage checking algorithm for LF. In *TPHOLs '03*, volume 2758 of *LNCS*, pages 120–135. Springer, 2003.
12. Matthieu Sozeau. Subset coercions in coq. In Thorsten Altenkirch and Conor McBride, editors, *TYPES*, volume 4502 of *LNCS*, pages 237–252. Springer, 2006.
13. Benjamin Werner. *Une Théorie des Constructions Inductives*. PhD thesis, Université Paris 7, 1994.

A Proof of Subject Reduction

The objective of this section is to prove Subject Reduction (Lemma 3). First, we need some preliminary results. We use $\Gamma \vdash \sigma : \Delta$ to denote the substitution $\Gamma \vdash \sigma : \Delta \to []$.

Lemma 5. *Let σ, ρ be pre-substitutions with ρ idempotent, such that* $\mathrm{FV}(\rho) \cap \mathcal{D}om\,(\sigma) = \emptyset$*, and* $\mathcal{D}om\,(\rho) \cap \mathcal{D}om\,(\sigma) = \emptyset$*. Then $\rho\sigma\rho = \sigma\rho$.*

Proof. We want to prove that for all variables x, $\rho(\sigma(\rho(x))) = \rho(\sigma(x))$. If $x \in \mathcal{D}om\,(\rho)$, then $\sigma(\rho(x)) = \rho(x)$, since $\mathrm{FV}(\rho) \cap \mathcal{D}om\,(\sigma) = \emptyset$. The result follows because ρ is idempotent, and $x \notin \mathcal{D}om\,(\sigma)$. If $x \notin \mathcal{D}om\,(\rho)$, the result is trivial. □

Using the previous lemma, we can prove the Lemma 1 (Substitution Lemma).

Proof (Lemma 1). We prove by simultaneous induction on derivations that

- if $\Gamma \vdash t : T$, then $\Gamma' \vdash t\sigma : T\sigma$; and
- if $\Gamma; \Delta, \zeta \vdash [\boldsymbol{u} = \boldsymbol{v} : \Theta] \mapsto \Delta', \zeta' \vdash \sigma$, then $\Gamma'; \Delta\rho, \zeta \vdash [\boldsymbol{u}\rho = \boldsymbol{v}\rho : \Theta\rho] \mapsto \Delta'\rho, \zeta' \vdash \sigma\rho$.

The most difficult cases are (U-VarL), (U-VarR), and (U-Tel). In the first two cases, all we need to check is that the operation Strengthen is not affected by the substitution.

In the third case, from IH we have

$$\Gamma'; \Delta\rho, \zeta \vdash [u\rho = v\rho : T\rho] \mapsto \Delta_1\rho, \zeta_1 \vdash \sigma_1\rho \tag{1}$$

$$\Gamma'; \Delta_1\rho, \zeta_1 \vdash [\boldsymbol{u}\sigma_1\rho = \boldsymbol{v}\sigma_1\rho : \Theta[x := u]\sigma_1\rho] \mapsto \Delta_2\rho, \zeta_2 \vdash \sigma_2\rho \tag{2}$$

Now, σ_1 and ρ satisfy the hypotheses of the previous lemma. Therefore, $\sigma_1\rho = \rho\sigma_1\rho$. Using this equation in 2 IH we obtain

$$\Gamma'; \Delta_1\rho, \zeta_1 \vdash [\boldsymbol{u}\rho\sigma_1\rho = \boldsymbol{v}\rho\sigma_1\rho : \Theta\rho[x := u\rho]\sigma_1\rho] \mapsto \Delta_2\rho, \zeta_2 \vdash \sigma_2\rho \tag{3}$$

Applying rule (U-Tel) to 1 and 3 we obtain

$$\Gamma'; \Delta\rho, \zeta \vdash [u\boldsymbol{u}\rho = v\boldsymbol{v}\rho : \Theta\rho] \mapsto \Delta_2\rho, \zeta_2 \vdash \sigma_1\rho\sigma_2\rho \tag{4}$$

Since σ_2 and ρ satisfy the hypotheses of the previous lemma, we have $\sigma_2\rho = \rho\sigma_2\rho$, and the result follows. □

We next show some result on the unification. First, we give the proof of Lemma 2, that shows that if the unification succeeds positively, then we obtain the expected result. Second, we show that if the unification succeeds positively, the result is, in a sense, the most general unifier.

Proof (of Lemma 2). By induction on the derivation. We only consider the case (U-Tel) since it's the most interesting.

From IH, $\Gamma \vdash \sigma_1 : \Delta \to \Delta_1$, and $\Gamma\Delta_1 \vdash u\sigma_1 \approx v\sigma_1$. From the hypotheses, we know that $\Gamma\Delta \vdash \boldsymbol{u} : \Theta[x := u]$. Then, applying σ_1, we get $\Gamma\Delta_1 \vdash \boldsymbol{u}\sigma_1 : \Theta[x := u]\sigma_1$. Analogously, $\Gamma\Delta_1 \vdash \boldsymbol{v}\sigma_1 : \Theta[x := v]\sigma_1$. On the other hand, $\Gamma\Delta_1 \vdash \Theta[x := u]\sigma_1 \approx \Theta[x := v]\sigma_1$. Hence, from IH, we get $\Gamma\Delta_2 \vdash \sigma_2 : \Delta_1 \to \Delta_2$, and $\Gamma\Delta_2 \vdash \boldsymbol{u}\sigma_1\sigma_2 \approx \boldsymbol{v}\sigma_1\sigma_2$.

From here it's easy to conclude, since $\Gamma \vdash \sigma_1\sigma_2 : \Delta \to \Delta_2$ by composition of substitutions. □

Lemma 6. *If $\Gamma; \Delta, \zeta \vdash [\boldsymbol{u} = \boldsymbol{v} : \Theta] \mapsto \Delta', \zeta' \vdash \sigma$, and $\Gamma \vdash \rho : \Delta$, such that $\Gamma \vdash \boldsymbol{u}\rho \approx \boldsymbol{v}\rho$. Then $\Gamma \vdash \rho : \Delta'$. Moreover, for each $x \in \mathcal{D}om\,(\sigma)$, $\Gamma \vdash x\rho \approx x\sigma\rho$.*

Proof. By induction on the derivation. We want to prove that for each $(y : T) \in \Delta'$, $\Gamma \vdash y\rho : T\rho$, and for each $(y := t : T) \in \Delta'$, $\Gamma \vdash y\rho \approx t\rho : T\rho$.

Rule (U-VarL). If $y \in \mathcal{D}om\,(\Delta_0) \cup \mathcal{D}om\,(\Delta^t) \cup \mathcal{D}om\,(\Delta_t)$, then the result is obvious. If $y = x$, then $\Gamma \vdash x\rho \approx t\rho : T\rho$ by hypothesis, which also proves the second part.

Rule (U-VarR) is analogous to the previous case.

Rule (U-Inj) follows directly from IH.

Rules (U-Conv) and (U-Empty) are trivial.

Rule (U-Tel). From IH, we get $\Delta \vdash \rho : \Delta_1$. Also, $\Gamma \vdash x\sigma_1\rho \approx x\rho$, for each $x \in \mathcal{D}om\,(\sigma_1)$. From this last equation, $\Gamma \vdash \boldsymbol{u}\sigma_1\rho \approx \boldsymbol{u}\rho$, and $\Gamma \vdash \boldsymbol{v}\sigma_1\rho \approx \boldsymbol{v}\rho$. From IH, $\Gamma \vdash \rho : \Delta_2$, and $\Gamma \vdash x\sigma_2\rho \approx x\rho$ for each $x \in \mathcal{D}om\,(\sigma_2)$. Since $\mathcal{D}om\,(\sigma_1) \cap \mathcal{D}om\,(\sigma_2) = \emptyset$, we have $\Gamma \vdash x\sigma_1\sigma_2\rho \approx x\rho$ for each $x \in \mathcal{D}om\,(\sigma_1) \cup \mathcal{D}om\,(\sigma_2)$. □

Finally, we show the proof of Subject Reduction.

Proof (Lemma 3 — Subject Reduction). By induction on the type derivation. We consider the case of the new elimination rule, and ι reduction. We have

$$\frac{\begin{array}{c}\mathsf{Ind}(I[\Delta_p] : \Pi\Delta_a.s := \{C_i : \Pi\Delta_i.I\,\mathcal{D}om\,(\Delta_p)\ \boldsymbol{u_i}\}_i) \in \Sigma \\ \Gamma \vdash C_i\,\boldsymbol{r}\,\boldsymbol{a} : I\,\boldsymbol{p}\,\boldsymbol{u} \qquad \Gamma \vdash \boldsymbol{u} \approx \boldsymbol{t}[\Delta := \boldsymbol{q}] \qquad \Gamma\Delta(x : I\,\boldsymbol{p}\,\boldsymbol{t}) \vdash P : s \\ \Gamma \vdash \boldsymbol{q} : \Delta \qquad \Gamma; (\boldsymbol{z_i} : \Delta_i^*); \Delta; [\boldsymbol{u_i^*} = \boldsymbol{t} : \Delta_a^*] \vdash b_i : P[x := C_i\,\boldsymbol{p}\,\boldsymbol{z_i}]\end{array}}{\Gamma \vdash \begin{pmatrix}\texttt{match}\ C_i\,\boldsymbol{r}\,\boldsymbol{a}\ \texttt{as}\ x\ \texttt{in}\ [\Delta]\,I\ \boldsymbol{p}\ \boldsymbol{t}\ \texttt{where}\ \Delta := \boldsymbol{q} \\ \texttt{return}\ P\ \texttt{with}\ \{C_i\ \boldsymbol{z_i} \Rightarrow b_i\}_i\end{pmatrix} : P[\Delta := \boldsymbol{q}][x := C_i\,\boldsymbol{r}\,\boldsymbol{a}]}$$

and the reduction

$$\Gamma \vdash \begin{pmatrix}\texttt{match}\ C_i\,\boldsymbol{r}\,\boldsymbol{a}\ \texttt{as}\ x\ \texttt{in}\ [\Delta]\,I\ \boldsymbol{p}\ \boldsymbol{t}\ \texttt{where}\ \Delta := \boldsymbol{q}\ \texttt{return}\ P\ \texttt{with} \\ \ldots\ C_i\ \boldsymbol{z_i} \Rightarrow\ t_i\ \texttt{where}\ \sigma_i \ldots \\ \texttt{end}\end{pmatrix} \to t_i[\boldsymbol{z_i} := \boldsymbol{a}]$$

We want to prove that $\Gamma \vdash t_i[\boldsymbol{z_i} := \boldsymbol{a}] : P[\Delta := \boldsymbol{q}][x := C_i\,\boldsymbol{r}\,\boldsymbol{a}]$.

For checking the branch corresponding to the constructor C_i, we have the following judgment:

$$\frac{\begin{array}{c}\Gamma; (\boldsymbol{z_i} : \Delta_i^*)\Delta, \mathcal{D}om\,(\sigma_i) \cup \mathcal{D}om\,(\Delta) \vdash [\boldsymbol{u_i^*} = \boldsymbol{t} : \Delta_a^*] \mapsto \Delta', \emptyset \vdash \sigma \\ \Gamma\Delta' \vdash t_i : P[x := C_i\,\boldsymbol{p}\,\boldsymbol{z_i}] \qquad \Gamma\Delta' \vdash \sigma \approx \sigma_i\end{array}}{\Gamma; (\boldsymbol{z_i} : \Delta_i^*); \Delta; [\boldsymbol{u_i^*} = \boldsymbol{t} : \Delta_a^*] \vdash t_i\ \texttt{where}\ \sigma_i : P[x := C_i\,\boldsymbol{p}\,\boldsymbol{z_i}]}$$

We can assume that $\boldsymbol{z_i} = \mathcal{D}om\,(\Delta_i)$. By inverting the typing derivation of $C_i\,\boldsymbol{r}\,\boldsymbol{a}$, we obtain $\Gamma \vdash \boldsymbol{p} \approx \boldsymbol{r}$ and $\Gamma \vdash \boldsymbol{u} \approx \boldsymbol{u_i^*}[\boldsymbol{z_i} := \boldsymbol{a}]$. From this last equation, we obtain $\Gamma \vdash \boldsymbol{u_i^*}[\boldsymbol{z_i} := \boldsymbol{a}] \approx \boldsymbol{t}[\mathcal{D}om\,(\Delta) := \boldsymbol{q}]$.

Consider the substitution $\Gamma \vdash \rho : \Delta_i\Delta$ defined by

$$\rho = \{\boldsymbol{z_i} \mapsto \boldsymbol{a}\} \cup \{\mathcal{D}om\,(\Delta) \mapsto \boldsymbol{q}\}\ .$$

We have then, $\Gamma \vdash \boldsymbol{u_i^*}\rho \approx \boldsymbol{t}\rho$. By the previous lemma, $\Gamma \vdash \rho : \Delta'$. By the substitution lemma

$$\Gamma \vdash t_i\rho : P[x := C_i\,\boldsymbol{p}\,\boldsymbol{z_i}]\rho$$

But then, $t_i\rho \equiv t_i[\boldsymbol{z_i} := \boldsymbol{a}]$ and $P[x := C_i\,\boldsymbol{p}\,\boldsymbol{z_i}]\rho \equiv P[\mathcal{D}om\,(\Delta) := \boldsymbol{q}][x := C_i\,\boldsymbol{p}\,\boldsymbol{a}]$. □