Termination and Confluence of CC

Bachelor's Thesis

Jonas Oberhauser

Supervised by Prof. Dr. Gert Smolka *and* Chad E. Brown
Programming Systems Lab, Computer Science
Saarland University

Termination and Confluence of CC
Bachelor's Thesis

Termination and Confluence of CC

Bachelor's Thesis

Jonas Oberhauser

April 10, 2013

Saarland University
Natural and Technical Sciences, Faculty 1

ABSTRACT

Typed lambda calculi can be used as computational logical systems. Logical consistency of some type theories can be reduced to the properties of termination and confluence. We formally verify a proof of those properties for CC_{Σ} in Coq, building on the formalization done by Barras [4].

CONTENTS

1. INTRODUCTION

Typed lambda calculi are formal systems based on computation and typing: equality in these systems is defined by reduction rules, and the correspondence between propositions and their proofs is defined by typing rules. These typing rules justify *typing judgements*, which are statements of the form $\Gamma \vdash t : T$, where T is a type and t a term of that type in the context Γ. Following the Curry-Howard-de Bruijn isomorphism [8, p. 211, 19], it is possible to think of types as propositions and terms as their proofs. A proposition is thus provable in a context Γ if and only if it is *inhabited*, i.e., there is a term with that type, in Γ. Which types are admissible is defined by so called universes, which can be intuitively seen as the types of types.

Lambda calculus was first introduced in an untyped version by Alonzo Church in 1932 [10], but this version was proven to be unfit as a foundation for logic by Kleene and Rosser [20]. To deal with this, Church introduced a notion of typing and invented what is now called the Simple Thoery of Types in 1940 [9]. In this logical framework, types are used solely to restrict the terms a function can be applied to; a function with the type $\mathbb{N} \to \mathbb{N}$ could only be applied to natural numbers, instead of any term as in the untyped lambda calculus. While the Simple Theory of Types as presented by Church does not make use of the Curry-Howard-de Bruijn isomorphism, it can be seen as the grandfather of modern type theory.

According to Barendregt[6, p. 126], the Simple Theory of Types corresponds to the Simply Typed Lambda Calculus, a modern type theory that could in theory be expressed using a single universe. Adding a second universe allows for three orthogonal extensions: polymorphism, type operators and dependent types can be added independently to the Simply Typed Lambda Calculus, yielding a total of eight different type theories of which many have been studied thoroughly in the late eighties and early nineties[27, 11, 15, 18, 22, 21]. The richest of these type theories, where all of these extensions are added, is known as the Calculus of Constructions (short CC)[11], but can still be extended further. Two directions for extensions are conceivable: vertically, i.e., by adding further universes, or horizontally, i.e., by adding further categories of types. Examples are the Extended Calculus of Constructions, which is an extension of CC with an arbitrary number of universes and has been developed and studied extensively by Luo [23], or CC with small sigma types (short CC_Σ), which is a horizontal extension of CC. These sigma types correspond to existential quantification and dependent pairs in other formal systems.

Not all type theories that have been studied are logically consistent. As an

example, consider Martin-Lfs Theory of Types, which had only a single universe [24]. It was proven to be inconsistent by Girard in 1972 [14]. This leads to the question which ingredients or properties are required for a consistent type theory. For ECC (and thus any subsystem), Luo has given an answer to this question in his PhD thesis [23, Theorem 5.4, p. 105]: he shows that confluence and termination yield logical consistency of the system.

In this project we formalized proofs of these properties for CC_Σ. For this we built upon a formalization of these proofs for CC which was done by Barras [4]. To *formalize* in this sense means to bring the definitions and proofs into a form in which they can be algorithmically checked for correctness by so called *theorem provers* or *proof assistants*, which are programs that implement formal systems. This requires us to scrutinize the proof. It makes many details explicit which are hidden in the informal proof, if only behind phrases like "trivial" or "left as an exercise for the reader". Often these details are in fact trivial, but tedious. Many proof assistants come with automatization techniques that allow us to deal with these tedious parts automatically. In other cases, it turns out that things are not as trivial as they seemed, and the proof needs to be reconsidered. How hard it is to formalize a proof and whether these details go through depends often on how the definitions and propositions in the proof were formalized and which formal system is used. In the next sections we discuss our decisions on the more foundational issues and how the alternatives might have affected our project.

In addition to this, we give a mathematical description of the proofs, which serve to ease the digestion of the rather technical formal proof, and to explain many puzzling details. The mathematical proof of termination is strongly influenced by Geuvers' proof of termination[13], but has been adapted to be closer to the formalization. The mathematical proof of confluence has been taken from Barendregt[5] and is not original in any way.

1.1 The Proof Assistant Coq

The first decision we had to make was the logical system we would use to formalize the proofs. We chose to use the proof assistant Coq, mostly due to our previous experience with the system. Coq implements the Calculus of Inductive Constructions (CiC), which is a horizontal extension of ECC that adds inductive types. These inductive types serve well to formalize CC_Σ and many of the operations and properties thereof. However, CiC - without assuming further axioms - is an intuitionistic and intensional logic, which complicates the formalization of classical proofs. This thesis can hence also be seen as a proof that termination and confluence of CC_Σ can be proven in an intuitionistic and intensional setting.

For a short introduction to the Coq syntax, see Chapter A in the appendix. For a more complete description of Coq and the CiC, see the Coq reference manual `http://coq.inria.fr/distrib/V8.4/refman/` and "Coq'Art: the Calculus of Inductive Constructions" [7]. For a comparison of various proof assistants

on the task of proving that $\sqrt{2}$ is not a rational number, see "The Seventeen Provers of the World" [31].

1.2 De Bruijn Indices

An even more interesting decision is how we formalize the names of variables. There are generally three ways to formalize the names of variables. The first approach is to use strings to denote variables, like in mathematics. This is often called a named approach and has been used, among others, by Norrish [25]. While being very close to mathematics and hence more human readable than the other approaches, it comes with two disadvantages. The first disadvantage is that many properties and operations that depend on the name of the variables of a term are very hard to formalize with a named approach. The second disadvantage is that it requires alpha-equivalence as equality instead of Coq's Leibniz equality [25, Equation (3) on p. 172]. While possible, this appears to be clumsy and impractical to use. There are other options to deal with alpha-equivalence, like giving canonical forms by using nominal unification [30]. Approaches like this, however, seem out of reach for the scope of this project.

The second option, which we decided to take, is to abstract away from the name of bound variables. There are generally two ways to do this: using de Bruijn indices[12] for all variables, or using a hybrid approach which uses de Bruijn indices for bound variables and names for free variables [12, p. 392, 17, 2].

While we chose to use de Bruijn indices, this appears to have been some sort of a mistake. Many definitions (and hence, proofs) in the current formalization are complicated by the inability to distinguish between bound and free variables. Following the approach outlined in [2] would solve many of these complications while also bringing the formalization closer to the mathematical proof.

The third approach, which appears not to be formalizable in Coq for technical reasons, is using what is called the Higher Order Abstract Syntax (short HOAS)[1, 26]. This option abstracts away from variables altogether by embedding functions from the metalogic into the terms of the type theory. This has the advantage that the alpha-equivalence and function application of the metalogic carry over into the formalization, thus replacing reduction with function application and eliminating the difficulties of alpha-equivalence.

1.3 Barras' Coq in Coq

The next question we faced was whether to start anew or use an existing project and extend it. We began by formalizing the Simply Typed Lambda Calculus as well as formalizing termination and confluence for this system, following the proofs outlined in Barendregt [5] and Takahashi [29].

We then decided to build upon Barras' formal development of termination and confluence of CC. This project seems to be based on Geuver's proof technique [13], which (among other things) uses different syntactic representations

for term variables and type variables. Since Barras' goal was to work towards a type checker for Coq, which does not have such a distinction, there is only one type of variable in his development, and the "class" of the variable depends on the context Γ. In this thesis, we give a variant of Geuver's proof which does not depend on the syntactic distinction, but this comes at the cost of a greater complexity. As an example, some of the functions were previously defined only on some syntactic classes of terms. Now that the syntactic classification have been eliminated, the functions are partially defined on all terms, and we need to prove that they are defined in the situations where we use them.

While Barras' formalization took care of many of the pitfalls that arise when a mathematical proof that deals with partial functions and extensional set equality is to be formalized in Coq, these complications still affect the mathematical presentation of the proof.

1.4 Structure of the Document

In this thesis, we first present a mathematical formulation of the problem and proof steps. This mathematical formulation takes away many of the intricacies of the formal version, many of which deal with the intuitionistic and intensional nature of Coq. In Chapter 2 we define the underlying type theory and its semantics, and prove a few fundamental properties, including confluence. Chapter 3 extends these definitions by the definitions required to prove termination, and ends with the statement of termination. The mathematical formulation is then used in Chapter 4 to explain the formal version and some of the technicalities present therein.

1.5 Acknowledgements

I would like to thank Chad E. Brown for supervising this thesis. His tough questions and valuable criticism gave me an incentive to dive deeply into the research topic and understand it fully. His deep knowledge in the field and his countless suggestions helped shape this thesis into what it is now. I owe many thanks to Prof. Dr. Gert Smolka, who invited me to a great opportunity of research and who allowed me to write this thesis when I had the time to do it.

My gratitude goes as well to Bruno Barras, Ute Hornung, Naomi Nir-Bleimling, Susanne Oberhauser and Tobias Tebbi, and all of my friends and family.

2. PRELIMINARIES

Following Luo [23] and Barendregt [5], we define Λ, i.e. the set of terms, inductively. Usually when we write t, u, v, T, U, V, D, C, or K, we mean terms. We will also use t_1, u_2, T' etc. if we have multiple terms that need naming.

s : Every universe s is a term. The universes of CC_Σ are $\{\mathrm{Prop}, \mathrm{Type}_0\}$

x : Every variable x is a term. Other generic variables we will use are y and z. Note that there are infinitely many variables.

$t\,u$: If t and u are terms, so is the application $t\,u$, where t is applied to u.

$\lambda x : T.t$: If t and T are terms and x is a variable, the lambda term $\lambda x : T.t$ is a term.

$\forall x : T.U$: If T and U are terms and x is a variable, the product $\forall x : T.U$ with domain T and codomain U is a term.

$\Sigma x : T.U$: If T and U are terms and x is a variable, the sigma $\Sigma x : T.U$ is a term.

$(t, u)_{\Sigma x:T.U}$: If t, u, T, and U are terms and x is a variable, the pair $(t, u)_{\Sigma x:T.U}$ is a term.

$\pi_i(t)$: If $i \in \{1, 2\}$ and t is a term, the projection $\pi_i(t)$ is a term.

We will sometimes use D to denote the domain and C to denote the codomain of products and lambdas. We also distinguish between three levels of term names: the lower case names (t, u, etc.), the capital case names (T, U, D, etc.) and K. These levels will represent the place of the term in the type hierarchy, which will be introduced later.

We can also write $T \to U$ for $\forall x : T.U$ whenever $x \notin \mathcal{F}(U)$, where $\mathcal{F}(U)$ denotes the set of free variables of a term U:

Definition 1.

$$\begin{aligned}
\mathcal{F}(s) &:= \emptyset \\
\mathcal{F}(x) &:= \{x\} \\
\mathcal{F}(\lambda x : T.t) &:= \mathcal{F}(T) \cup (\mathcal{F}(t) \setminus \{x\}) \\
\mathcal{F}(u\, v) &:= \mathcal{F}(u) \cup \mathcal{F}(v) \\
\mathcal{F}(\forall x : T.U) &:= \mathcal{F}(T) \cup (\mathcal{F}(U) \setminus \{x\}) \\
\mathcal{F}(\Sigma x : T.U) &:= \mathcal{F}(T) \cup (\mathcal{F}(U) \setminus \{x\}) \\
\mathcal{F}((t,u)_{\Sigma x:T.U}) &:= \mathcal{F}(t) \cup \mathcal{F}(u) \cup \mathcal{F}(T) \cup (\mathcal{F}(U) \setminus \{x\}) \\
\mathcal{F}(\pi_i(t)) &:= \mathcal{F}(t)
\end{aligned}$$

Note the cases where a variable is removed from the set of free variables. We call all occurrences of this variable in the right subterm *bound*. As an example, there is a single bound occurrence of x in $\forall x : T.x$, but none in $\forall x : x.y$ or x. In any case, we call the $\forall$, λ, or Σ the binder of that variable.

The set of terms, by itself, does not possess any meaning. We will first define reduction, then we will restrict Λ to CC_Σ by adding environments and typing rules. Typing and reduction together form the semantics of terms. Before we can start this, we need to define what it means to substitute a term t for a variable x in some term u.

2.1 Equality

Usually, equality means syntactic equality. In our case, we will consider terms to be equal if they are syntactically equal up to the name of bound variables. As an example, $x \neq y$ but $(\forall x : X.x) = (\forall y : X.y)$. This is acceptable because in the formalization, we use a technique called "de Bruijn indices"[12] which abstracts away the names of bound (and free) variables in a meaning-preserving way.

As a consequence, we can rename bound variables. This technique is known as *alpha-renaming*. As an example, $(\forall x : X.\forall x : Y.x) = (\forall x : X.\forall y : Y.y)$.

2.2 Valuations

We will first talk about functions that map variables to elements of some set X. We call these *X-valuations*. We can update the value that some X-valuation f assigns to some variable x with the update operation, f^x_t. A similar definition can be found in Stoughton[28, p. 2].

Definition 2.

$$f^x_v := (y \mapsto \text{ if } x = y \text{ then } v \text{ else } f(y)\ )$$

Consecutive updates are written as follows: $f^{x_1,\ldots,x_k}_{t_1,\ldots,t_k}$

It is a trivial fact that non-conflicting updates can be reordered and that conflicting updates are overridden:

Lemma 1. *For every x, y, v_x, v_y and f,*

$$f^{x,y}_{v_x,v_y} = \begin{cases} f^{y,x}_{v_y,v_x} & x \neq y \\ f^{y}_{v_y} & \textit{otherwise} \end{cases}$$

2.3 Assignments

An *assignment* is a term-valuation. In other words, it is a function that maps variables to terms. Since variables are terms, we can also assign a variable to itself. One of the simplest assignments is the *identity assignment* where every variable is assigned to itself. We call this assignment ρ_I. We can lift application of an assignment ρ to a term t by going through the term and applying the assignment to all variables in that term. However, whenever a binding is introduced, we must make sure not to override that binding. For this we define the predicates of capturing and freshness. We say z is *captured* by ρ in (x, U) if:

$$\exists y \in \mathcal{F}(U) .y \neq x \wedge z \in \mathcal{F}(\rho(y))$$

Note that x will correspond to the binder, U to the term to which the binder applies, ρ will be the substitution that needs to avoid capturing, and z will be any variable. We say z is *ρ-fresh* in (x, U) if it is not captured by ρ in (x, U). This is equivalent to:

$$\forall y \in \mathcal{F}(U), z \in \mathcal{F}(\rho(y)) \implies y = x$$

We call the resulting substitution the parallel substitution of ρ in t and denote it as $\rho(t)$. The assignment ρ_I applied to a term t yields t.

Definition 3.

$$\begin{aligned}
\rho(s) &:= s \\
\rho(x) &:= \rho(x) \\
\rho(\lambda x : T.t) &:= \lambda z : \rho(T).\rho^x_z(t) && \text{where } z \text{ is } \rho\text{-fresh in } (x, t) \\
\rho(u\, v) &:= \rho(u)\, \rho(v) \\
\rho(\forall x : T.U) &:= \forall z : \rho(T).\rho^x_z(U) && \text{where } z \text{ is } \rho\text{-fresh in } (x, U) \\
\rho(\Sigma x : T.U) &:= \Sigma z : \rho(T).\rho^x_z(U) && \text{where } z \text{ is } \rho\text{-fresh in } (x, U) \\
\rho\left((t, u)_{\Sigma x:T.U}\right) &:= (\rho(t), \rho(u))_{\Sigma z:\rho(T).\rho^x_z(U)} && \text{where } z \text{ is } \rho\text{-fresh in } (x, U) \\
\rho(\pi_1(t)) &:= \pi_1(\rho(t)) \\
\rho(\pi_2(t)) &:= \pi_2(\rho(t))
\end{aligned}$$

We prove that such a ρ-fresh z always exists. There are infinitely many variables, but in every term there are only finitely many. Thus for some term t, $\mathcal{F}(t)$ and $\mathcal{F}(\rho(y))$ for $y \in \mathcal{F}(t)$ are finite. Since the union of finitely many finite sets is still finite, $\bigcup_{y \in \mathcal{F}(t)} (\mathcal{F}(\rho(y)))$ is finite. Hence even when we take those variables away, infinitely many remain. Since for us, terms are equal up to

the names of bound variables, we can actually choose such a z. Depending on the context, we will sometimes make use of this (when we care for the z being chosen) and sometimes pick any z (when the z does not matter for us).

Now we prove that the identity assignment applied to a term T yields T.

Lemma 2. *For all T, $\rho_I(T) = T$.*

Proof. Via straightforward induction over T. The base cases are trivial, and we will only show the case where T is a product.

$$\rho_I(\forall x : U.V) = \forall z : \rho_I(U).{\rho_I}_z^x(V)$$

where z is ρ_I-fresh in (x, V). Trivially, x is ρ_I-fresh in (x, V), so we chose z to be x. Then we have to prove the following:

$$\forall x : \rho_I(U).{\rho_I}_x^x(V) = \forall x : U.V$$

By the induction hypothesis for U and V and the definition of ρ_I, this is true. □

Now the substitution of a single term, u_t^x, is actually the application of ${\rho_I}_t^x$ to u.

Definition 4.

$$u_t^x := {\rho_I}_t^x(u)$$

The concatenation of two assignments ρ_1 and ρ_2, denoted by $\rho_1 \circ \rho_2$, is an assignment itself.

Definition 5.

$$\rho_1 \circ \rho_2 := (y \mapsto \rho_1(\rho_2(y)))$$

We now want to prove a couple of important properties of assignments. Many of the proofs use a property which is often called coincidence: whenever ρ_1 and ρ_2 agree on all the free variables of U, then $\rho_1(U) = \rho_2(U)$.

Lemma 3. *For any ρ_1 and ρ_2,*

$$(\forall x \in \mathcal{F}(U).\rho_1(x) = \rho_2(x)) \implies \rho_1(U) = \rho_2(U)$$

Proof. By straightforward induction on U. Since the individual cases are trivial we will not go into more detail. □

We can now prove that for any ρ_1 and ρ_2, $\rho_1(\rho_2(u)) = (\rho_1 \circ \rho_2)(u)$.

Lemma 4. *For any ρ_1 and ρ_2,*

$$\rho_1(\rho_2(u)) = (\rho_1 \circ \rho_2)(u)$$

Proof. By induction on u. In the variable case, this follows directly from definition; In the other cases, we use the induction hypothesis. However for binders, there is some small trick involved. In one case, two assignments will be modified whenever a binder is introduced, while in the other case only a single assignment is changed. We will consider the product case.

For the sake of brevity, we write ρ' for $\rho_1 \circ \rho_2$.

We have to prove the following:

$$\rho_1(\rho_2(\forall x : T.U)) = \rho'(\forall x : T.U)$$

By definition, this is equivalent to proving

$$\forall y : \rho_1(\rho_2(T)).{\rho_1}_y^z({\rho_2}_z^x(U)) = \forall y : \rho'(T).{\rho'}_y^x(U)$$

where z is ρ_2-fresh in (x, U) and y is both ρ_1-fresh in $(z, {\rho_2}_z^x(U))$ and ρ'-fresh in (x, U). This is equivalent to proving that the domains and the codomains are respectively equal. For the domains, the equality follows directly from the induction hypothesis. For the codomains, we need to prove that ${\rho_1}_y^z({\rho_2}_z^x(U)) = {\rho'}_y^x(U)$. The induction hypothesis gives us that ${\rho_1}_y^z({\rho_2}_z^x(U)) = \left({\rho_1}_y^z \circ {\rho_2}_z^x\right)(U)$. Thus and by Lemma 3, we only have to prove that $\left({\rho_1}_y^z \circ {\rho_2}_z^x\right)$ and ${\rho'}_y^x$ agree on all free variables x' of U. We perform a case distinction on whether $x' = x$ or not.

$x' = x$: We have $\left({\rho_1}_y^z \circ {\rho_2}_z^x\right)(x) = {\rho_1}_y^z(z) = y$. Similarly, ${\rho'}_y^x(x) = y$. Therefore

$$\left({\rho_1}_y^z \circ {\rho_2}_z^x\right)(x) = {\rho'}_y^x(x)$$

$x' \neq x$: In this case, $\left({\rho_1}_y^z \circ {\rho_2}_z^x\right)(x') = {\rho_1}_y^z(\rho_2(x'))$. As z is ρ_2-fresh in (x, U) and $x' \neq x$, $z \notin \mathcal{F}(\rho_2(x'))$. Thus ${\rho_1}_y^z(\rho_2(x')) = \rho_1(\rho_2(x'))$ by Lemma 3. This, by definition, equals $(\rho_1 \circ \rho_2)(x')$ and hence $\rho'(x')$. Since $x' \neq x$, this equals ${\rho'}_y^x(x')$.

□

From this, we can immediately conclude $\rho(u_t^x) = \rho(\rho_{I_t^x}(u)) = (\rho \circ \rho_{I_t^x})(u)$. We want to prove that this equals $\rho^x_{\rho(t)}(u)$. Coincidence does the trick.

Lemma 5. *For all ρ, x, t and all u,*

$$\rho(u_t^x) = \rho^x_{\rho(t)}(u)$$

Proof. We prove the intermediate equality

$$(\rho \circ \rho_{I_t^x})(u) = \rho^x_{\rho(t)}(u)$$

By Lemma 3, we only need to prove

$$(\rho \circ \rho_{I_t^x})(y) = \rho^x_{\rho(t)}(y)$$

where y is free in u. We perform a case distinction on whether $y = x$ or not.

$y = x$: We have $(\rho \circ \rho_{I}{}_{t}^{x})(x) = \rho(t) = \rho^{x}_{\rho(t)}(x)$ by definition.

$y \neq x$: In this case, $(\rho \circ \rho_{I}{}_{t}^{x})(y) = \rho(y) = \rho^{x}_{\rho(t)}(y)$ by definition.

□

Another fact is that if some variable z does not appear free in U, and we rename a free variable x to z in U, and then substitute all of the free occurrences of z in that U' by some term t, we could have just as well replaced all the free occurrences of x in U by t.

Lemma 6. *For any z which is ρ-fresh in (x, U) and not free in U,*

$$\rho^{x}_{t}(U) = \rho^{x}_{z}(U)^{z}_{t}$$

Proof. We want to use Lemma 3, but we currently have to argue the equality of two sequential assignments and a single assignment. We use Lemma 4 to transform the two assignments on the right hand side into a single one:

$$\begin{aligned} \rho^{x}_{z}(U)^{z}_{t} &= \rho_{I}{}_{t}^{z}(\rho^{x}_{z}(U)) \\ &= (\rho_{I}{}_{t}^{z} \circ \rho^{x}_{z})(U) \end{aligned}$$

By Lemma 3, we consider what happens with a free variable y in U. As y is free in U and z is not, $y \neq z$. We make a case distinction on $y = x \vee y \neq x$.

$y = x$: Then we can easily argue:

$$\begin{aligned} \rho^{x}_{t}(x) &= t \\ &= z^{z}_{t} \\ &= (\rho^{x}_{z}(x))^{z}_{t} \\ &= (\rho_{I}{}_{t}^{z} \circ \rho^{x}_{z})(x) \end{aligned}$$

This is exactly what we had to prove.

$y \neq x$: Then we can argue:

$$\begin{aligned} \rho^{x}_{t}(y) &= \rho(y) \\ &= \rho_{I}(\rho(y)) \\ &= \rho_{I}{}_{t}^{z}(\rho(y)) \end{aligned}$$

The last two steps follow from Lemma 2 and from Lemma 3 as z is ρ-fresh in (x, U) and hence not free in $\rho(y)$. By definition, $\rho_{I}{}_{t}^{z}(\rho(y))$ equals $(\rho_{I}{}_{t}^{z} \circ \rho^{x}_{z})(y)$.

□

2.4 One-Step Reduction

The reduction rules are the key element of the semantics of CC_Σ. The important rules are the beta rule, where a lambda term is applied to its arguments, and the projection rules, where the specified element of the pair is returned.

$$\textit{beta}\quad \frac{}{(\lambda x : T.t)\, u \to_\beta t^x_u} \qquad \textit{proj1}\quad \frac{}{\pi_1((t,u)_{\Sigma x:T.U}) \to_\beta t}$$

$$\textit{proj2}\quad \frac{}{\pi_2((t,u)_{\Sigma x:T.U}) \to_\beta u}$$

All the other rules just allow for one of these rules to be applied anywhere in the term. We call these the congruence rules.

$$\frac{t \to_\beta t'}{t\, u \to_\beta t'\, u} \qquad \frac{u \to_\beta u'}{t\, u \to_\beta t\, u'} \qquad \frac{T \to_\beta T'}{\lambda x : T.t \to_\beta \lambda x : T'.t} \qquad \frac{t \to_\beta t'}{\lambda x : T.t \to_\beta \lambda x : T.t'}$$

$$\frac{T \to_\beta T'}{\forall x : T.U \to_\beta \forall x : T'.U} \qquad \frac{U \to_\beta U'}{\forall x : T.U \to_\beta \forall x : T.U'} \qquad \frac{T \to_\beta T'}{\Sigma x : T.U \to_\beta \Sigma x : T'.U}$$

$$\frac{U \to_\beta U'}{\Sigma x : T.U \to_\beta \Sigma x : T.U'} \qquad \frac{t \to_\beta t'}{(t,u)_{\Sigma x:T.U} \to_\beta (t',u)_{\Sigma x:T.U}}$$

$$\frac{u \to_\beta u'}{(t,u)_{\Sigma x:T.U} \to_\beta (t,u')_{\Sigma x:T.U}} \qquad \frac{T \to_\beta T'}{(t,u)_{\Sigma x:T.U} \to_\beta (t,u)_{\Sigma x:T'.U}}$$

$$\frac{U \to_\beta U'}{(t,u)_{\Sigma x:T.U} \to_\beta (t,u)_{\Sigma x:T.U'}} \qquad \frac{t \to_\beta t'}{\pi_i(t) \to_\beta \pi_i(t')}$$

For example, we want to reduce $((\lambda x : \mathit{Prop}.\lambda y : \mathit{Prop}.x \to y)\,\mathrm{True})\,\mathrm{False}$. We can apply the beta rule to reduce the subterm $(\lambda x : \mathit{Prop}.\lambda y : \mathit{Prop}.x \to y)\,\mathrm{True}$ to $\lambda y : \mathit{Prop}.\mathrm{True} \to y$. Then we use the congruence rule to lift this reduction to the whole term:

$$((\lambda x : \mathit{Prop}.\lambda y : \mathit{Prop}.x \to y)\,\mathrm{True})\,\mathrm{False} \to_\beta (\lambda y : \mathit{Prop}.\mathrm{True} \to y)\,\mathrm{False}$$

When we have a term t, we define β_t to be the set of all terms that t can one-step reduce to:

Definition 6.

$$\beta_t = \{t' \in \Lambda \mid t \to_\beta t'\}$$

We call a term where no reduction rule applies *normal*:

Definition 7. A term t is normal if $\beta_t = \emptyset$.

We have another important definition:

Definition 8. A term t is *neutral* if it is a universe, a variable, an application, a product, a sigma, or a projection.

In other words, it is neutral if it is not a pair or a lambda term. Note that the beta and projection rules consider only pairs and lambda terms. Hence if t is neutral, only the congruence rules apply.

The reflexive, transitive closure of one-step reduction is called *reduction*. We denote it with $t \to_\beta^* t'$ and say t reduces to t'. The reflexive, transitive, symmetric closure of one-step reduction is called *conversion* and is denoted by $t \approx_\beta u$. We say t and u are convertible.

2.5 Confluence and the Diamond Property

An interesting and important property of these reduction rules is *confluence*: Whenever t reduces to two alternatives, u_1 and u_2, both of them reduce to a common term t'. Another way to read this is to say that whenever u_1 and u_2 have a common ancestor, they also have a common descendant.

Confluence can be formalized as a well understood predicate called the *diamond property*, or $\diamondsuit$ [5]:

Definition 9.

$$\diamondsuit R := \forall x, y_1, y_2 . x\, R\, y_1 \wedge x\, R\, y_2 \;\to\; \exists z . y_1\, R\, z \wedge y_2\, R\, z$$

We know about $\diamondsuit$ that it is propagated to the reflexive transitive closure: $\diamondsuit R \implies \diamondsuit R^*$. Also we know that for any R and Q such that $R \subseteq Q$ and $Q \subseteq R^*$, $R^* = Q^*$. If we put these two theorems together, we have for any Q and R such that $R \subseteq Q$ and $Q \subseteq R^*$, $\diamondsuit Q \implies \diamondsuit R^*$. Hence, when we want to prove $\diamondsuit \to_\beta^*$, we only have to find a reduction relation Q which lies between $\to_\beta$ and $\to_\beta^*$ and satisfies the diamond property.

Lemma 7. *For any binary relation R,*

$$\diamondsuit R \implies \diamondsuit R^*$$

Proof. The basic idea is to fill the big diamond from R^* with small diamonds from R. We will do this row by row. We will first prove that we can fill the individual row:

$$\diamondsuit R \implies \forall x, y_1, y_2 . x\, R^*\, y_1 \wedge x\, R\, y_2 \;\to\; \exists z . y_1\, R\, z \wedge y_2\, R^*\, z$$

By induction on the number of R-steps in $x\, R^*\, y_1$.

$x = y_1$: We choose y_2. $y_1\, R\, y_2$ as $y_1 = x$ and $x\, R\, y_2$ is an assumption. $y_2\, R^*\, y_2$ is trivial.

$x\,R^*\,x'$ *and* $x'\,R\,y_1$: By the induction hypothesis, we have z' such that $x'\,R\,z'$ and $y_2\,R^*\,z'$. Therefore by $\diamondsuit R$, we get a z such that $y_1\,R\,z$ and $z'\,R\,z$. By transitivity, we get $y_2\,R^*\,z$.

Now we can fill the whole diamond row by row. We prove the original statement via induction on the number of R-steps in $x\,R^*\,y_2$.

$x = y_2$: We choose y_1. $y_1\,R^*\,y_1$ is trivial. $y_2\,R^*\,y_1$ as $x = y_2$ and $x\,R^*\,y_1$ is an assumption.

$x\,R^*\,x'$ *and* $x'\,R\,y_2$: By the induction hypothesis, we have z' such that $y_1\,R^*\,z'$ and $x'\,R^*\,z'$. Therefore by $\diamondsuit R$ and the sublemma, we get a z such that $y_2\,R^*\,z$ and $z'\,R\,z$. By transitivity, we get $y_1\,R^*\,z$.

□

Lemma 8. *For binary relations Q and R such that $R \subseteq Q$ and $Q \subseteq R^*$, $Q^* = R^*$*

Proof. By $R \subseteq Q$, we know that $R^* \subseteq Q^*$. It remains to show that $Q^* \subseteq R^*$, i.e. for all x and y, $x\,Q^*\,y \implies x\,R^*\,y$. We prove this by induction on the number of Q-steps in $x\,Q^*\,y$:

$x = y$: Trivial since R^* is reflexive.

$x\,Q^*\,x'$ *and* $x'\,Q\,y$: By the induction hypothesis, $x\,R^*\,x'$, and since $Q \subseteq R^*$, $x'\,R^*\,y$. By transitivity, $x\,R^*\,y$.

□

2.6 Parallel Beta

Now we need to find such a reduction relation. The first reduction relation we could try is $\to_\beta$, but we can easily find out that $\to_\beta$ does not satisfy the diamond property: we assume we have some u of the form $\lambda x : T.t$ where x appears in many places in t, i.e. $t = x \ldots x \ldots x$, and we also have a v of which we know that it one-step-reduces to v'. Thus we have (at least) two ways to one-step reduce $u\,v$: we can either apply u to v and get t_1 of the which is $t_v^x = v \ldots v \ldots v$, or we can reduce v to v' and get t_2 which is $u\,v'$. We now want to get a common successor t', and the basic idea is to mirror the reduction step $u\,v \to_\beta t_1$ in $t_2 \to_\beta t'$ and the reduction step $u\,v \to_\beta t_2$ in $t_1 \to_\beta t'$. The first one is easy: we simply apply u to v' and get $t_{v'}^x = v' \ldots v' \ldots v'$. However, if we used the beta rule first, we would now have to reduce all the occurrences of v in t_v^x to v' at once, and one-step-reduction does not allow this.

This motivates *parallel reduction* [5], where we can one-step-reduce any number of subterms in one single step. We slightly change the beta and projection rules, add reflexive variable and universe rules to allow for cases where we one-step-reduce none of the subterms, and then extend the other rules to allow all of

the subterms to reduce in parallel. Note that again, we have some special rules where reduction happens or which are reflexive, as well as many congruence rules which allow to lift the reduction of subterms to the whole term.

$$\textit{beta}\quad \frac{t \Rightarrow_\beta t' \qquad u \Rightarrow_\beta u'}{(\lambda x : T.t)\, u \Rightarrow_\beta t'^{x}_{u'}} \qquad \textit{proj1}\quad \frac{t \Rightarrow_\beta t'}{\pi_1((t,u)_{\Sigma x:T.U}) \Rightarrow_\beta t'}$$

$$\textit{proj2}\quad \frac{u \Rightarrow_\beta u'}{\pi_2((t,u)_{\Sigma x:T.U}) \Rightarrow_\beta u'} \qquad \textit{variable}\quad \frac{}{x \Rightarrow_\beta x} \qquad \textit{universe}\quad \frac{}{s \Rightarrow_\beta s}$$

$$\frac{t \Rightarrow_\beta t' \qquad u \Rightarrow_\beta u'}{t\, u \Rightarrow_\beta t'\, u'} \qquad \frac{T \Rightarrow_\beta T' \qquad t \Rightarrow_\beta t'}{\lambda x : T.t \Rightarrow_\beta \lambda x : T'.t'} \qquad \frac{T \Rightarrow_\beta T' \qquad U \Rightarrow_\beta U'}{\forall x : T.U \Rightarrow_\beta \forall x : T'.U'}$$

$$\frac{T \Rightarrow_\beta T' \qquad U \Rightarrow_\beta U'}{\Sigma x : T.U \Rightarrow_\beta \Sigma x : T'.U'} \qquad \frac{t \Rightarrow_\beta t' \qquad u \Rightarrow_\beta u' \qquad T \Rightarrow_\beta T' \qquad U \Rightarrow_\beta U'}{(t,u)_{\Sigma x:T.U} \Rightarrow_\beta (t',u')_{\Sigma x':T'.U'}}$$

$$\frac{t \Rightarrow_\beta t'}{\pi_i(t) \Rightarrow_\beta \pi_i(t')}$$

We first acknowledge the fact that the two reflexive rules are enough for the whole reduction relation to be reflexive:

Lemma 9. *For all t, $t \Rightarrow_\beta t$.*

Proof. This is a trivial induction on t. The base cases, s and x, are handled by the reflexive rules; the other cases follow directly from the congruence rules and the induction hypotheses. □

Recall that we need to show that $\Rightarrow_\beta$ lies between $\rightarrow_\beta$ and $\rightarrow_\beta^*$. However before we can do this, we need to show that it does not matter if we substitute first and reduce then, or reduce first and then substitute. This requires the fact that parallel reduction does not introduce new free variables.

Lemma 10. *For any terms t and t' such that $t \Rightarrow_\beta t'$, $\mathcal{F}(t') \subseteq \mathcal{F}(t)$.*

Proof. By induction over the derivation of $t \Rightarrow_\beta t'$. As all other cases are trivial or analogous, we only consider the beta rule.

We have $(\lambda x : T.t)\, u \Rightarrow_\beta t'^{x}_{u'}$. By the induction hypothesis, $\mathcal{F}(t') \subseteq \mathcal{F}(t)$ and $\mathcal{F}(u') \subseteq \mathcal{F}(u)$. Clearly $(\mathcal{F}(t) \setminus \{x\}) \cup \mathcal{F}(u) \subseteq \mathcal{F}((\lambda x : T.t)\, u)$. Similarly, since all free occurrences of x have been replaced by u', $\mathcal{F}\left(t'^{x}_{u'}\right) \subseteq (\mathcal{F}(t') \setminus \{x\}) \cup \mathcal{F}(u')$. The induction hypothesis allows us to infer $(\mathcal{F}(t') \setminus \{x\}) \cup \mathcal{F}(u') \subseteq (\mathcal{F}(t) \setminus \{x\}) \cup \mathcal{F}(u)$. By the transitivity of the subset relation, we conclude $\mathcal{F}\left(t'^{x}_{u'}\right) \subseteq \mathcal{F}((\lambda x : T.t)\, u)$. □

Lemma 11. *For all t, t', x, u and u' such that $t \Rightarrow_\beta t'$ and $u \Rightarrow_\beta u'$, $t^x_u \Rightarrow_\beta t'^{x}_{u'}$.*

Proof. We prove the stronger statement that for all t, t', such that $t \Rightarrow_\beta t'$ and all ρ, ρ' such that for all x, $\rho(x) \Rightarrow_\beta \rho'(x)$, $\rho(t) \Rightarrow_\beta \rho'(t')$. The original lemma follows from the fact that $\Rightarrow_\beta$ is reflexive on variables and $u \Rightarrow_\beta u'$.

By induction over the rules of $t \Rightarrow_\beta t'$. We will only treat the variable, beta and product cases, as all the other cases are either trivial or analogous.

$y \Rightarrow_\beta y$: We have $\rho(y) \Rightarrow_\beta \rho'(y)$ by assumption.

$(\lambda x : T.t)\, u \Rightarrow_\beta {t'}^x_{u'}$: We have to prove that $\rho((\lambda x : T.t)\, u) \Rightarrow_\beta \rho'({t'}^x_{u'})$. We know $\rho((\lambda x : T.t)\, u) = (\lambda z : \rho(T).\rho^x_z(t))\rho(u)$, where z is ρ-fresh in (x, t), and $\rho'({t'}^x_{u'}) = {\rho'}^x_{\rho'(u')}(t')$. By the beta rule and the induction hypotheses, we also know:

$$(\lambda z : \rho(T).\rho^x_z(t))\rho(u) \Rightarrow_\beta {\rho'}^x_z(t')^z_{\rho'(u')}$$

By Lemma 10, z is also ρ'-fresh in (x, t') and thus by Lemmata 6 and 5, ${\rho'}^x_z(t')^z_{\rho'(u')} = {\rho'}^x_{\rho'(u')}(t') = \rho'({t'}^x_{u'})$. Therefore, we have what we needed.

$\forall x : T.U \Rightarrow_\beta \forall x : T'.U'$: We have to prove that $\rho(\forall x : T.U) \Rightarrow_\beta \rho'(\forall x : T'.U')$. By definition this means that for some z which is ρ-fresh in (x, U) and ρ'-fresh in (x, U')[1], $\forall z : \rho(T).\rho^x_z(U) \Rightarrow_\beta \forall z : \rho'(T').{\rho'}^x_z(U')$. By the induction hypothesis, $\rho(T) \Rightarrow_\beta \rho(T')$ and $\rho^x_z(U) \Rightarrow_\beta {\rho'}^x_z(U')$ as $z \Rightarrow_\beta z$.

□

Lemma 12. $\rightarrow_\beta \subseteq \Rightarrow_\beta$

Proof. The proof is very simple. We do an induction on $t \rightarrow_\beta t'$ and use the corresponding rule from $\Rightarrow_\beta$. However we will often experience a situation where exactly one of the subterms one-step-reduces, and we need to show that we can emulate this by not touching the other subterms at all. As an example, we consider one of the lambda rules: $\lambda x : T.t \rightarrow_\beta \lambda x : T'.t$ when $T \rightarrow_\beta T'$. By the induction hypothesis, we have $T \Rightarrow_\beta T'$. We need to prove that $\lambda x : T.t \Rightarrow_\beta \lambda x : T'.t$, but the only applicable rule requires $t \Rightarrow_\beta t$. This is proven in Lemma 9. □

Lemma 13. $\Rightarrow_\beta \subseteq \rightarrow^*_\beta$

Proof. This proof is simple as well, this time by induction on $t \Rightarrow_\beta t'$. We emulate parallel reduction in $\rightarrow^*_\beta$ by reducing the subterms one after another. By the transitivity of $\rightarrow^*_\beta$, this is enough. We consider the example from before: $\lambda x : T.t \Rightarrow_\beta \lambda x : T'.t'$ where $T \Rightarrow_\beta T'$ and $t \Rightarrow_\beta t'$. By the induction hypothesis, we know that $T \rightarrow^*_\beta T'$ and $t \rightarrow^*_\beta t'$. Therefore it is easy to show that $\lambda x : T.t \rightarrow^*_\beta \lambda x : T'.t \rightarrow^*_\beta \lambda x : T'.t'$.

This covers all except the five special rules. The first three are treated similarly, except that we use $\rightarrow_\beta$ in an intermediary step and argue that $\rightarrow_\beta \subseteq \rightarrow^*_\beta$, and hence by transitivity we get the result we need to show. As an example, we treat the beta and the proj1 cases.

[1] Recall that we can choose the z. Here we choose the same z for (x, U) and for (x, U').

$(\lambda x : T.t)u \Rightarrow_\beta t'^{x}_{u'}$ *where* $t \Rightarrow_\beta t'$ *and* $u \Rightarrow_\beta u'$: By the induction hypotheses, we have $t \rightarrow_\beta^* t'$ and $u \rightarrow_\beta^* u'$. But then can easily prove $(\lambda x : T.t)u \rightarrow_\beta^* (\lambda x : T.t')u \rightarrow_\beta^* (\lambda x : T.t')u' \rightarrow_\beta t'^{x}_{u'}$. As $\rightarrow_\beta \subseteq \rightarrow_\beta^*$ and $\rightarrow_\beta^*$ is transitive, this implies $(\lambda x : T.t)u \rightarrow_\beta^* t'^{x}_{u'}$.

$\pi_1((t,u)_{\Sigma c:T.U}) \Rightarrow_\beta t'$ *where* $t \Rightarrow_\beta t'$: By the induction hypothesis, we have $t \rightarrow_\beta^* t'$. We argue that $\pi_1((t,u)_{\Sigma c:T.U}) \rightarrow_\beta t \rightarrow_\beta^* t'$. As $\rightarrow_\beta \subseteq \rightarrow_\beta^*$ and $\rightarrow_\beta^*$ is transitive, this implies $\pi_1((t,u)_{\Sigma c:T.U}) \rightarrow_\beta^* t'$.

The only rules that remain are the reflexive variable and universe rules. As $\rightarrow_\beta^*$ is also reflexive, this is trivial. □

The only thing that remains to show is $\diamondsuit \Rightarrow_\beta$. Again, the idea is to mirror all the reductions that occurred in $t \Rightarrow_\beta t_1$ in $t_2 \Rightarrow_\beta t'$ and vice versa.

Lemma 14. *For all* t, t_1 *and* t_2,

$$t \Rightarrow_\beta t_1 \wedge t \Rightarrow_\beta t_2 \implies \exists t'.t_1 \Rightarrow_\beta t' \wedge t_2 \Rightarrow_\beta t'$$

Proof. By induction on $t \Rightarrow_\beta t_1$ and subsequent inversion of $t \Rightarrow_\beta t_2$. This yields many tedious cases, most of which are dealt with by the induction hypothesis. The only interesting cases are the ones where in one case, we applied either the beta or one of the projection rules, and in the other one we did not. We treat the cases where $t \Rightarrow_\beta t_1$ was inferred from such a rule and $t \Rightarrow_\beta t_2$ was not.

$(\lambda x : T.t)\, u \Rightarrow_\beta t_1{}^{x}_{u_1}$ *and* $(\lambda x : T.t)\, u \Rightarrow_\beta (\lambda x : T_2.t_2)\, u_2$: By the induction hypothesis, we have t' and u' such that $t_1 \Rightarrow_\beta t'$, $t_2 \Rightarrow_\beta t'$, $u_1 \Rightarrow_\beta u'$ and $u_2 \Rightarrow_\beta u'$. By the rules and our assumptions, we know $(\lambda x : T_2.t_2)\, u_2 \Rightarrow_\beta t'^{x}_{u'}$. By Lemma 11, we also have $t_1{}^{x}_{u_1} \Rightarrow_\beta t'^{x}_{u'}$.

$\pi_1((t,u)_{\Sigma x:T.U}) \Rightarrow_\beta t_1$ *and* $\pi_1((t,u)_{\Sigma x:T.U}) \Rightarrow_\beta \pi_1((t_2,u_2)_{\Sigma x:T_2.U_2})$: By the induction hypothesis, we have t' such that $t_1 \Rightarrow_\beta t'$, $t_2 \Rightarrow_\beta t'$. By the rules and our assumptions, we know $t_1 \Rightarrow_\beta t'$ and $\pi_1((t_2,u_2)_{\Sigma x:T_2.U_2}) \Rightarrow_\beta t'$.

$\pi_2((t,u)_{\Sigma x:T.U}) \Rightarrow_\beta u_1$ *and* $\pi_2((t,u)_{\Sigma x:T.U}) \Rightarrow_\beta \pi_2((t_2,u_2)_{\Sigma x:T_2.U_2})$: By the induction hypothesis, we have u' such that $u_1 \Rightarrow_\beta u'$, $u_2 \Rightarrow_\beta u'$. We choose u'.

□

It is a corollary that the diamond property holds for $\rightarrow_\beta^*$. We can use this result to prove the so-called Church-Rosser property: whenever u and v are convertible, there is a single term t that both reduce to.

Lemma 15.

$$\forall u, v.u \approx_\beta v \implies \exists t.u \rightarrow_\beta^* t \wedge v \rightarrow_\beta^* t$$

Proof. The proof is a straightforward induction over the number of $\rightarrow_\beta$-steps in $u \approx_\beta v$.

$u = v$: We need to prove that there exists a t such that u reduces to t. We choose u.

$u \approx_\beta v'$ *where* $v' \to_\beta v$: By the induction hypothesis, we have t' such that u and v' reduce to t'. We need to find a t such that u reduces to t and v reduces to t. By the transitivity of $\to_\beta^*$, we only have to prove that there is a t that both t' and v reduce to. As $v' \to_\beta^* t'$ and $v' \to_\beta^* v$, we get such a t by the diamond property.

$u \approx_\beta v'$ *where* $v \to_\beta v'$: By the induction hypothesis, we have t' such that u and v' reduce to t'. We need to find a t such that u reduces to t and v reduces to t. Since v reduces to v' and v' reduces to t', v reduces to t'.

□

It follows easily that if two normal terms are convertible, they must be equal. We will not go through the proof.

2.7 Termination

We say a term t *terminates* if all reduction paths starting from t are finite. We denote this by $t{\downarrow}$:

Definition 10.

$$t{\downarrow} := \forall t' \in \beta_t, t'{\downarrow}$$

We write SN for the set of all terminating terms.

We say a set of terms *terminates* if all elements of that set terminate:

Definition 11.

$$M{\downarrow} := \forall t \in M, t{\downarrow}$$

It is an interesting question to ask whether all the terms of CC_Σ terminate. It is easy to find terms which do not terminate. As the canonical example consider $(\lambda x : \mathrm{Prop}.xx)\,(\lambda x : \mathrm{Prop}.xx)$, which one-step-reduces to itself. The interesting task is to find some way to restrict the terms to a terminating subset, where we do not lose too many terms, and where we can easily decide whether a term is in that set or not. The way we do this is by introducing typing judgements of the form $\Gamma \vdash t : T$. Only if such a statement can be justified will we consider t and T to be terms of CC_Σ.

2.8 Typing Rules

First of all, we need to introduce environments. An *environment* Γ is a list of pairs of the form $x : T$. We write $\Gamma, x : T$ for Γ extended by the pair $x : T$. If a variable x does not appear in any of the judgements $y : \_$ in Γ, we say x is $\Gamma-$fresh. By mutual induction, we define $\vdash$ as well as what it means for Γ to be well-formed. We write $\Gamma \vdash t : T : K$ if we mean that $\Gamma \vdash t : T$ and $\Gamma \vdash T : K$. We write $\Gamma' \subseteq \Gamma$ if Γ' is a prefix of Γ, and $\Gamma' \subset \Gamma$ if it is a proper prefix.

Definition 12.

$$\textit{prop}\quad \frac{\Gamma \text{ is well-formed}}{\Gamma \vdash \text{Prop} : \text{Type}_0}$$

$$\textit{variable}\quad \frac{\Gamma \text{ is well-formed} \qquad \Gamma', x : T \subseteq \Gamma \text{ for some } \Gamma'}{\Gamma \vdash x : T}$$

$$\textit{application}\quad \frac{\Gamma \vdash t : (\forall x : T.U) \qquad \Gamma \vdash u : T}{\Gamma \vdash t\, u : U_u^x}$$

$$\textit{lambda}\quad \frac{\Gamma \vdash T : s_1 \qquad \Gamma, x : T \vdash u : U \qquad \Gamma, x : T \vdash U : s_2}{\Gamma \vdash (\lambda x : T.u) : (\forall x : T.U)}$$

$$\textit{product}\quad \frac{\Gamma \vdash T : s_1 \qquad \Gamma, x : T \vdash U : s_2}{\Gamma \vdash (\forall x : T.U) : s_2}$$

$$\textit{sigma}\quad \frac{\Gamma \vdash T : \text{Prop} \qquad \Gamma, x : T \vdash U : \text{Prop}}{\Gamma \vdash (\Sigma x : T.U) : \text{Prop}}$$

$$\textit{pair}\quad \frac{\Gamma \vdash t : T \qquad \Gamma \vdash u : U_t^x \qquad \Gamma \vdash T : \text{Prop} \qquad \Gamma, x : T \vdash U : \text{Prop}}{\Gamma \vdash (t, u)_{\Sigma x:T.U} : (\Sigma x : T.U)}$$

$$\textit{proj1}\quad \frac{\Gamma \vdash t : (\Sigma x : T.U)}{\Gamma \vdash \pi_1 t : T} \qquad\qquad \textit{proj2}\quad \frac{\Gamma \vdash t : (\Sigma x : T.U)}{\Gamma \vdash \pi_2 t : U_{\pi_1(t)}^x}$$

$$\textit{conversion}\quad \frac{\Gamma \vdash t : T \qquad \Gamma \vdash U : s \qquad T \approx_\beta U}{\Gamma \vdash t : U}$$

$$\textit{wf-ax}\quad \frac{}{\emptyset \text{ is well-formed}} \qquad\qquad \textit{wf-cons}\quad \frac{x \in \Gamma\text{−fresh} \qquad \Gamma \vdash T : s}{\Gamma, x : T \text{ is well-formed}}$$

Now that we have typing rules and reduction we have the semantics of CC_Σ. We can denote propositions as types, a correspondence known also as the Curry-Howard isomorphism [19].

The typing rules distinguish between four levels, where if $\Gamma \vdash t : T$, then t should be of a lower level than T. The highest level is the level of Type_0. We will soon see that Type_0 never appears on the left hand side of a typing judgement. So in a way, Type_0 is not really part of CC_Σ: it is there only as the roof of our building, to make sure everything below has a type. The second level is where the kinds live. The kind of propositions, Prop, and the kinds of type operators like $\text{Prop} \to \text{Prop}$ or $\text{Prop} \to (\text{Prop} \to \text{Prop})$, are at this level. The third level is the level of propositions and type operators. We call them constructors.

For example, we have propositions like truth (True $:= \forall x : \text{Prop}, x \to x$) and falsity (False $:= \forall x : \text{Prop}, x$), as well as type operators like negation ($\neg := \lambda x : \text{Prop}.x \to \text{False}$), conjunction ($\wedge := \lambda x : \text{Prop}.\lambda y : \text{Prop}.\Sigma z : x.y$) and disjunction ($\vee := \lambda x : \text{Prop}.\lambda y : \text{Prop}.\forall z : \text{Prop}.(x \to z) \to ((y \to z) \to z)$). We will use infix notation where appropriate. The lowest level is the level of proofs, or objects. We have the polymorphic identity I ($\lambda x : \text{Prop}.\lambda y : x.y$). Even complex proofs like this, which we shall call t for now : $\lambda x_1 : \text{Prop}.\lambda x_2 : \text{Prop}.\lambda z' : (x_1 \wedge x_2).\lambda z : \text{Prop}.\lambda y_1 : x_1 \to z.\lambda y_2 : x_2 \to z.y_1\, \pi_1(z')$. It is easy to check that $\emptyset \vdash I : \text{True}$ and $\emptyset \vdash t : \forall x_1 : \text{Prop}.\forall x_2 : \text{Prop}.(x_1 \wedge x_2) \to (x_1 \vee x_2)$.

Let us define these levels formally:

Definition 13.

$$\begin{aligned} \text{Kind}_\Gamma &:= \{t \mid \Gamma \vdash t : \text{Type}_0\} \\ \text{Constr}_\Gamma &:= \{t \mid \exists K, \Gamma \vdash K : \text{Type}_0 \wedge \Gamma \vdash t : K\} \\ \text{Obj}_\Gamma &:= \{t \mid \exists K, \Gamma \vdash K : \text{Prop} \wedge \Gamma \vdash t : K\} \end{aligned}$$

There are two important properties of well-formedness: when Γ gives a type to some term, then Γ is well-formed; and when Γ is well-formed, so is any prefix of Γ.

Lemma 16. *For all Γ, t and T, $\Gamma \vdash t : T$ implies Γ is well-formed.*

Proof. We will only sketch the proof. By straightforward induction on the typing derivation. The base cases have as an assumption that Γ is well-formed, and the inductive cases all have it as an induction hypothesis. □

Lemma 17. *If Γ is well-formed, then every Γ' which is a prefix of Γ is well-formed, too.*

Proof. We prove instead the equivalent statement that for all Γ and Γ' such that Γ', Γ is well-formed, Γ' is also well-formed. We prove this by induction on the structure of Γ.

$\emptyset$: Thus $\Gamma', \Gamma = \Gamma'$ and Γ' is well-formed.

$\Gamma, x : T$: We argue that Γ', Γ is well-formed because $\Gamma', \Gamma \vdash T : s$ and Lemma 16. The claim then follows from the induction hypothesis.

□

Now if we have a typing judgement $\Gamma \vdash t : T$, we can analyze the structure of t and by inversion retrieve information about the T. However, inversion usually yields two cases: one where the derivation ends with the application of the usual typing rule, and one where it ends with the application of the conversion rule. This motivates an inversion lemma that combines those two cases:

Lemma 18. *For all Γ and T,*

1. $\Gamma \vdash \text{Prop} : T$ *implies* $\text{Type}_0 \approx_\beta T$
2. $\Gamma \nvdash \text{Type}_0 : T$
3. $\Gamma \vdash x : T$ *implies that there is a* U *and* Γ' *such that* $\Gamma', x : U \subseteq \Gamma$ *and* $U \approx_\beta T$
4. $\Gamma \vdash u\, v : T$ *implies there are* U *and* V *such that* $\Gamma \vdash u : (\forall x : U.V), v : U$ *and* $V_v^x \approx_\beta T$
5. $\Gamma \vdash (\lambda x : U.u) : T$ *implies that there are* s_1, s_2 *and* V *such that* $\Gamma \vdash U : s_1$, $\Gamma, x : U \vdash u : V : s_2$ *and* $(\forall x : U.V) \approx_\beta T$
6. $\Gamma \vdash (\forall x : U.V) : T$ *implies that there are* s_1 *and* s_2 *such that:*
   - $\Gamma \vdash U : s_1$,
   - $\Gamma, x : U \vdash V : s_2$,
   - *and* $s_2 \approx_\beta T$
7. $\Gamma \vdash (\Sigma x : U.V) : T$ *implies*
   - $\Gamma \vdash U : \text{Prop}$,
   - $\Gamma, x : U \vdash V : \text{Prop}$,
   - *and* $\text{Prop} \approx_\beta T$
8. $\Gamma \vdash ((u, v)_{\Sigma x:U.V}) : T$ *implies*
   - $\Gamma \vdash u : U, v : V_u^x, U : \text{Prop}$,
   - $\Gamma, x : U \vdash V : \text{Prop}$,
   - *and* $(\Sigma x : U.V) \approx_\beta T$
9. $\Gamma \vdash (\pi_1(t)) : T$ *implies that there are* U *and* V *such that* $\Gamma \vdash t : (\Sigma x : U.V)$ *and* $U \approx_\beta T$
10. $\Gamma \vdash (\pi_2(t)) : T$ *implies that there are* U *and* V *such that* $\Gamma \vdash t : (\Sigma x : U.V)$ *and* $V_{\pi_1(t)}^x \approx_\beta T$

Proof. The proof is very elaborate, but also very simple. We will only sketch the proof here. We first prove that for any Γ, T_1 and T_2 such that $T_1 \approx_\beta T_2$, we can infer the conclusion for T_2 if we have it for T_1[2]. The result follows directly from the transitivity of conversion.

Now we prove the original result by mutual induction on the typing derivation. All of the cases follow directly from the induction hypotheses except the case where the last rule was the conversion rule. That case is proven by the sublemma from above. □

We will often require the fact that if Γ is enough to give t a type, then except for the variables in Γ, there are no free variables in t.

[2] E.g., in case 3, $\exists U.x : U \in \Gamma \wedge U \approx_\beta T_1$ implies $\exists U.x : U \in \Gamma \wedge U \approx_\beta T_2$

Lemma 19. *For any* Γ*,* t *and variable* x *which is* $\Gamma-$fresh*,*

$$\Gamma \vdash t : T \implies x \notin \mathcal{F}(t)$$

Proof. By induction on the typing derivation. We will consider only the variable and product cases, the other cases are similar.

variable : We have $\Gamma \vdash y : U$ where $\Gamma', y : U$ is a prefix of Γ, so y is not $\Gamma-$fresh. $\mathcal{F}(y) = \{y\}$. As x is $\Gamma-$fresh, $x \neq y$ and hence $x \notin \mathcal{F}(y)$.

product : We have $\Gamma \vdash (\forall y : T.U) : s_2$ where $\Gamma \vdash T : s_1$ and $\Gamma, y : T \vdash U : s_2$. By the induction hypotheses, $x \notin \mathcal{F}(T)$ and if $x \neq y$ then $x \notin \mathcal{F}(U)$. Hence $x \notin \mathcal{F}(U) \setminus \{y\}$. Therefore $x \notin \mathcal{F}(T) \cup (\mathcal{F}(U) \setminus \{y\})$.

□

We will sometimes use *weakening*: if we extend some Γ to $\Gamma, x : T$, where x is $\Gamma-$fresh and $\Gamma \vdash T : s$, then any judgement $u : U$ that could be derived in Γ can still be derived in the larger environment $\Gamma, x : T$.

Lemma 20. *For any* Γ*,* x *which is* $\Gamma-$fresh *and* $\Gamma \vdash T : s$*, then for all* u *and* U*,*

$$\Gamma \vdash u : U \implies \Gamma, x : T \vdash u : U$$

Proof. We prove the stronger statement that we can add $x : T$ anywhere in the context. So for $\Gamma, \Gamma' \vdash u : U$ we prove if $\Gamma, x : T, \Gamma'$ is well-formed, then $\Gamma, x : T, \Gamma' \vdash u : U$.

We do this by induction over the typing derivation of $\Gamma, \Gamma' \vdash u : U$. We will only consider the variable case and the product case, the other cases are similar.

variable : We have $\Gamma, \Gamma' \vdash y : U$, where $\Gamma_1, y : U \subseteq \Gamma, \Gamma'$ for some Γ_1. This is equivalent to $y : U \in \Gamma, \Gamma'$. Thus we know $y : U \in \Gamma, x : T, \Gamma'$. Therefore there is a prefix of $\Gamma, x : T, \Gamma'$ of the form $\Gamma_2, y : U$ for some Γ_2. Thus by the variable rule, $\Gamma, x : T, \Gamma' \vdash y : U$.

product : We have $\Gamma, \Gamma' \vdash (\forall y : D, C) : s_2$ where $\Gamma, \Gamma' \vdash D : s_1$ and $\Gamma, \Gamma', y : D \vdash C : s_2$. By the induction hypothesis for $\Gamma, \Gamma' \vdash D : s_1$, we have $\Gamma, x : T, \Gamma' \vdash D : s_1$. Hence $\Gamma, x : T, \Gamma', y : D$ is well-formed (if $x \neq y$, which we can assume without loss of generality by alpha-renaming). Thus by the induction hypothesis for $\Gamma, \Gamma', y : D \vdash C : s_2$ we also have $\Gamma, x : T, \Gamma', y : D \vdash C : s_2$. By the product rule, we conclude $\Gamma, x : T, \Gamma' \vdash (\forall y : D, C) : s_2$

□

We can also prove that whenever we substitute something for a variable that has the same type as the variable, the type of the whole term does not change. We first define what we mean by a term having the same type of a variable. We say an assignment ρ *translates* Γ to Γ' or write $\rho \vDash \Gamma \to \Gamma'$ when:

Definition 14.

$$\rho \vDash \Gamma \to \Gamma' := \forall x : T \in \Gamma, \Gamma' \vdash \rho(x) : \rho(T)$$

We can prove that if ρ translates the variables from Γ to Γ', it also translates all other terms:

Lemma 21. *For ρ, well-formed Γ and Γ' such that $\rho \vDash \Gamma \to \Gamma'$, for any t and T such that $\Gamma \vdash t : T$, $\Gamma' \vdash \rho(t) : \rho(T)$.*

Proof. Via induction on the typing derivation. We will only treat the prop, variable, application, and product cases, as the other cases are very similar or trivial.

prop : Suppose the typing derivation ends with

$$prop \quad \frac{\Gamma \text{ is well-formed}}{\Gamma \vdash \text{Prop} : \text{Type}_0}$$

Since $\rho(\text{Prop}) = \text{Prop}$ and $\rho(\text{Type}_0) = \text{Type}_0$ and Γ' is well-formed, the prop rule applies.

variable : Suppose the typing derivation ends with

$$variable \quad \frac{\Gamma \text{ is well-formed} \qquad \Gamma'', x : T \subseteq \Gamma \text{ for some } \Gamma''}{\Gamma \vdash x : T}$$

By $\Gamma'', x : T \subseteq \Gamma$, we conclude $x : T \in \Gamma$. Thus we can apply $\rho \vDash \Gamma \to \Gamma'$.

application : Suppose the typing derivation ends with

$$application \quad \frac{\Gamma \vdash t : (\forall x : T.U) \qquad \Gamma \vdash u : T}{\Gamma \vdash t\, u : U^x_u}$$

We have $\Gamma' \vdash \rho(t) : \rho(\forall x : T.U)$ and $\Gamma' \vdash \rho(u) : \rho(T)$ by the induction hypotheses. By the definition of ρ, this means:

$$\Gamma' \vdash \rho(t) : (\forall z : \rho(T).\rho^x_z(U))$$

where z is ρ-fresh in (x, U). We need to prove that $\Gamma' \vdash \rho(t\, u) : \rho(U^x_u)$. We know $\rho(U^x_u) = \rho^x_{\rho(u)}(U)$. Since z is ρ-fresh in (x, U) and using Lemma 6, this equals $\rho^x_z(U)^z_{\rho(u)}$. Note that we have to choose our z to not be free in U. Hence we need to prove $\Gamma' \vdash \rho(t)\, \rho(u) : (\rho^x_z(U))^z_{\rho(u)}$. This follows directly from the application rule and our assumptions.

product : Suppose the typing derivation ends with

$$product \quad \frac{\Gamma \vdash T : s_1 \qquad \Gamma, x : T \vdash U : s_2}{\Gamma \vdash (\forall x : T.U) : s_2}$$

By the induction hypotheses, $\Gamma' \vdash \rho(T) : s_1$ and $\Gamma', z : \rho_z^x(T) \vdash \rho_z^x(U) : s_2$ if the following holds

$$\rho_z^x \vDash (\Gamma, x : T) \to (\Gamma', z : \rho_z^x(T))$$

(where z is ρ-fresh in (x, U)).

We need to show for all pairs $x' : T'$ in $\Gamma, x : T$,

$$\Gamma', z : \rho_z^x(T) \vdash \rho_z^x(x') : \rho_z^x(T')$$

For the old pairs (where $x' \neq x$), we can use Lemma 20. The details are left as an exercise for the reader.

For the new pair $x : T$, we need to argue a little bit more. We need to show:

$$\Gamma', z : \rho_z^x(T) \vdash \rho_z^x(x) : \rho_z^x(T)$$

This follows from the variable rule if $\Gamma' \vdash \rho_z^x(T) : s_1$. By alpha-renaming, we know $x \notin \mathcal{F}(T)$ and hence $\rho_z^x(T) = \rho(T)$. However, we already have $\Gamma' \vdash \rho(T) : s_1$. We need to prove $\Gamma' \vdash \rho(\forall x : T.U) : \rho(s_2)$, i.e.:

$$\Gamma' \vdash (\forall z : \rho(T).\rho_z^x(U)) : s_2$$

This follows directly from the product rule and our assumptions.

□

It is an important fact (which we will not prove) that when $\Gamma \vdash t : T$, $\rho_{I_t^x}$ translates $\Gamma, x : T$ to Γ.

Lemma 22. *For any Γ, t, T, such that $\Gamma \vdash t : T$, then for all x which are* $\Gamma-$fresh*,*

$$\rho_{I_t^x} \vDash \Gamma, x : T \to \Gamma$$

Another result that we will often use is that typing is capped only at Type_0:

Lemma 23. *For any t, T and Γ, when $\Gamma \vdash t : T$, then $T \neq \text{Type}_0$ implies $\exists s.\Gamma \vdash T : s$.*

Proof. We do an induction along the typing derivation.

prop : Suppose the typing derivation ends with

$$\textit{prop}\ \frac{\Gamma \text{ is well-formed}}{\Gamma \vdash \text{Prop} : \text{Type}_0}$$

Since $\text{Type}_0 \neq \text{Type}_0$ is false, we do not have to prove anything.

variable : Suppose the typing derivation ends with

$$\textit{variable}\quad \frac{\Gamma \text{ is well-formed} \qquad \Gamma', x : T \subseteq \Gamma \text{ for some } \Gamma'}{\Gamma \vdash x : T}$$

Since $\Gamma', x : T \subseteq \Gamma$ and Γ is well-formed, $\Gamma', x : T$ is well-formed by Lemma 17. Thus $\Gamma' \vdash T : s$ for some s. By repeated application of Lemma 20, $\Gamma \vdash T : s$.

application : Suppose the typing derivation ends with

$$\textit{application}\quad \frac{\Gamma \vdash t : (\forall x : T.U) \qquad \Gamma \vdash u : T}{\Gamma \vdash t\, u : U^x_u}$$

By the induction hypothesis for $\Gamma \vdash t : (\forall x : T.U)$, we know that there is some s such that $\Gamma \vdash (\forall x : T.U) : s$. By Lemma 18.6 we can infer that $\Gamma, x : T \vdash U : s$. Therefore by Lemma 21 and $\Gamma \vdash u : T$, $\Gamma \vdash U^x_u : s$.

lambda : Suppose the typing derivation ends with

$$\textit{lambda}\quad \frac{\Gamma \vdash T : s_1 \qquad \Gamma, x : T \vdash u : U \qquad \Gamma, x : T \vdash U : s_2}{\Gamma \vdash (\lambda x : T.u) : (\forall x : T.U)}$$

We have $\Gamma \vdash T : s_1$ and $\Gamma, x : T \vdash U : s_2$. Therefore $\Gamma \vdash (\forall x : T.U) : s_2$ by the product rule.

product : Suppose the typing derivation ends with

$$\textit{product}\quad \frac{\Gamma \vdash T : s_1 \qquad \Gamma, x : T \vdash U : s_2}{\Gamma \vdash (\forall x : T.U) : s_2}$$

We have $\Gamma, x : T \vdash U : s_2$. As $s_2 \neq \text{Type}_0$, we have $s_2 = \text{Prop}$. Since Γ is well-formed, we have $\Gamma \vdash \text{Prop} : \text{Type}_0$ by the prop rule.

sigma : Suppose the typing derivation ends with

$$\textit{sigma}\quad \frac{\Gamma \vdash T : \text{Prop} \qquad \Gamma, x : T \vdash U : \text{Prop}}{\Gamma \vdash (\Sigma x : T.U) : \text{Prop}}$$

As Γ is well-formed, we have $\Gamma \vdash \text{Prop} : \text{Type}_0$ by the prop rule.

pair : Suppose the typing derivation ends with

$$\textit{pair}\quad \frac{\Gamma \vdash t : T \qquad \Gamma \vdash u : U^x_t \qquad \Gamma \vdash T : \text{Prop} \qquad \Gamma, x : T \vdash U : \text{Prop}}{\Gamma \vdash (t, u)_{\Sigma x : T.U} : (\Sigma x : T.U)}$$

By the sigma rule and the assumptions, $\Gamma \vdash (\Sigma x : T.U) : \text{Prop}$.

proj1 : Suppose the typing derivation ends with

$$\textit{proj1}\quad \frac{\Gamma \vdash t : (\Sigma x : T.U)}{\Gamma \vdash \pi_1 t : T}$$

By the induction hypothesis for $\Gamma \vdash t : (\Sigma x : T.U)$, we know $\Gamma \vdash (\Sigma x : T.U) : s$. By Lemma 18.7, we have $\Gamma \vdash T : \text{Prop}$.

proj2 : Suppose the typing derivation ends with

$$\textit{proj2}\quad \frac{\Gamma \vdash t : (\Sigma x : T.U)}{\Gamma \vdash \pi_2 t : U^x_{\pi_1(t)}}$$

By the induction hypothesis for $\Gamma \vdash t : (\Sigma x : T.U)$, we know $\Gamma \vdash (\Sigma x : T.U) : s$. By Lemma 18.7, we have $\Gamma, x : T \vdash U : \text{Prop}$. By Lemma 21, we have $\Gamma \vdash U^x_{\pi_1(t)} : \text{Prop}$ if $\Gamma \vdash \pi_1(t) : T$. This we have by the proj1 rule.

conversion : We have $\Gamma \vdash U : s$ as an assumption.

□

2.9 Type Uniqueness and Subject Reduction

Two other well known properties of CC_Σ are type uniqueness, i.e. that if a term has two types, those types are convertible, and subject reduction, i.e. that typing is preserved by reduction. We will not prove these results, you can find these proofs in the literature [23].

Lemma 24. *For any Γ, t, T_1 and T_2 such that $\Gamma \vdash t : T_1$ and $\Gamma \vdash t : T_2$, $T_1 \approx_\beta T_2$.*

Lemma 25. *For any Γ, t and T such that $\Gamma \vdash t : T$, and for any t' such that $t \to^*_\beta t'$, $\Gamma \vdash t' : T$.*

From these lemmas in conjunction with Lemma 15, we also get that whenever convertible terms have a type in some environment Γ, then those types are convertible.

Lemma 26. *For any Γ, t_1 and T_1 such that $\Gamma \vdash t_1 : T_1$, t_2 and T_2 such that $\Gamma \vdash t_2 : T_2$, $t_1 \approx_\beta t_2$ implies $T_1 \approx_\beta T_2$.*

Proof. By Lemma 15, we have a t' that both t_1 and t_2 reduce to. By Lemma 25, $\Gamma \vdash t' : T_1$ and $\Gamma \vdash t' : T_2$. Therefore by Lemma 24, $T_1 \approx_\beta T_2$. □

Another helpful corollary is that whenever $\Gamma \vdash U : K$, $\Gamma \vdash T : V$ and $U \approx_\beta V$, then $\Gamma \vdash V : K'$ for some K' where $K \approx_\beta K'$. Since in this case, K' can only be a universe, we can even prove a stricter version:

Lemma 27. *For any Γ, t, U, T, and s such that $\Gamma \vdash U : s$, $\Gamma \vdash t : T$ and $T \approx_\beta U$, then also $\Gamma \vdash T : s$.*

Proof. We will want to use Lemma 23. For this we first need to show that $T \neq \mathrm{Type}_0$. If $T = \mathrm{Type}_0$, then $U \to_\beta^* T$ as Type_0 is normal. But then by Lemma 25, $\Gamma \vdash \mathrm{Type}_0 : s$. By Lemma 18.2, this is not the case. Now we can use Lemma 23 and conclude that $\Gamma \vdash T : s'$ for some s'. From Lemma 26, we know that $s \approx_\beta s'$. Since s and s' are normal, this implies $s = s'$. □

2.10 Possible Forms

We will now prove that kinds and constructors have a certain form in CC_Σ: for all T such that $\Gamma \vdash T : \mathrm{Type}_0$, T is either a universe or T is a product $\forall x : D.C$ where $\Gamma, x : D \vdash C : \mathrm{Type}_0$. We will often make use of this fact.

Lemma 28. *For all Γ and $T \in \mathrm{Kind}_\Gamma$, $T = \mathrm{Prop}$ or $T = \forall x : D.C$ where $C \in \mathrm{Kind}_{\Gamma, x:D}$ (for some D and C).*

Proof. By induction over the structure of T. We will treat only the cases where T is a universe, a variable, an application, a lambda or a product, as the other cases are trivial.

Prop : This is exactly what we need to prove.

Type_0 : This case cannot happen by Lemma 18.2.

x : By Lemma 18.3, we get $\Gamma', x : U \subseteq \Gamma$, where $U \approx_\beta \mathrm{Type}_0$. Since Γ is well-formed and by Lemma 17, $\Gamma', x : U$ is well-formed. Thus $\Gamma' \vdash U : s$ for some s. Since Type_0 is normal and $U \approx_\beta \mathrm{Type}_0$, $U \to_\beta^* \mathrm{Type}_0$. By Lemma 25, this means $\Gamma \vdash \mathrm{Type}_0 : s$ However by Lemma 18.2, this is not possible.

$u\,v$: By Lemma 18.4, there is a U such that $\Gamma \vdash u : \forall x : U.V$, $\Gamma \vdash v : U$, and $V_v^x \approx_\beta \mathrm{Type}_0$. By a combination of Lemma 23 and Lemma 18.6, we find out that $\Gamma \vdash U : s_1$ and $\Gamma, x : U \vdash V : s_2$ where s_1 and s_2 are universes. By Lemma 21 we have $\Gamma \vdash V_v^x : s_2$ if we can prove that $\rho_{I_v^x} \vDash \Gamma, x : U \to \Gamma$ and $\Gamma, x : U$ is well-formed. For the translation, we only consider the hard case where $\Gamma \vdash x_v^x : U_v^x$. By alpha-renaming, we can assume without loss of generality that x is not free in U. Hence we only have to prove that $\Gamma \vdash v : U$, which we have. For $\Gamma, x : U$ being well-formed, it suffices to argue that $\Gamma \vdash U : s_1$, which we also have.

Since Type_0 is normal and $V_v^x \approx_\beta \mathrm{Type}_0$, we have $V_v^x \to_\beta^* \mathrm{Type}_0$. Thus by Lemma 25, $\Gamma \vdash V_v^x : s_2$ also implies that $\Gamma \vdash \mathrm{Type}_0 : s_2$. However by Lemma 18.2, this is not possible.

$\lambda x : T.u$: By Lemma 18.5, we have $\forall x : T.U \approx_\beta \mathrm{Type}_0$ for some U. This, however, is not possible.

$\forall x : T.U$: By Lemma 18.6, we have $\Gamma, x : T \vdash U : s$ where $s \approx_\beta \text{Type}_0$. As universes are normal, this means that $s = \text{Type}_0$. This however is exactly what we need to prove.

□

Similarly, we can prove that for all $T \in \text{Constr}_\Gamma$, T cannot be a universe, pair or projection.

Lemma 29. *For all Γ and $T \in \text{Constr}_\Gamma$ implies $T = x$ or $T = \forall x : U.V$ or $T = \lambda x : U.V$ or $T = U\,V$ or $T = \Sigma x : U.V$ (for some x, U and V).*

Proof. By induction over the structure of T. We will treat only the cases where T is a universe, a pair or a projection, as the other cases are trivial. Note that since $T \in \text{Constr}_\Gamma$, there is a K such that $\Gamma \vdash T : K : \text{Type}_0$.

s : By Lemma 18.2, T would have to be Prop. Thus by Lemma 18.1, $K \approx_\beta \text{Type}_0$. By Lemma 15 and the fact that Type_0 is normal, this means K reduces to Type_0. By Lemma 25, this implies that $\Gamma \vdash \text{Type}_0 : \text{Type}_0$, which by Lemma 18.2 cannot be.

$(u, v)_{\Sigma x:U.V}$: By Lemma 18.8, we have $K \approx_\beta \Sigma x : U.V$, $\Gamma \vdash U : \text{Prop}$ and $\Gamma, x : U \vdash V : \text{Prop}$. We know $\Gamma \vdash (\Sigma x : U.V) : \text{Prop}$ by the sigma rule. By Lemmata 15 and 25, this implies that $\text{Prop} = \text{Type}_0$, and that is not the case.

$\pi_1(t)$: By Lemma 18.9, $\Gamma \vdash t : (\Sigma x : U.V)$ and $U \approx_\beta K$. By Lemma 23, $\Gamma \vdash (\Sigma x : U.V) : s$. By Lemma 18.7, $\Gamma \vdash U : \text{Prop}$. Therefore by the same reasoning as before, $\text{Prop} = \text{Type}_0$, which is not the case.

$\pi_2(t)$: By Lemma 18.10, $\Gamma \vdash t : (\Sigma x : U.V)$ and $V^x_{\pi_1(t)} \approx_\beta K$. By Lemma 23, $\Gamma \vdash (\Sigma x : U.V) : s$. By Lemma 18.7, $\Gamma, x : U \vdash V : \text{Prop}$. By Lemma 21, $\Gamma \vdash V^x_{\pi_1(t)} : \text{Prop}$. Therefore by the same reasoning as before, $\text{Prop} = \text{Type}_0$, which is not the case.

□

Note that the restrictions in both cases are much more severe than stated. For example, the T in $\forall x : T.U \in \text{Kind}_\Gamma$ can not be an element of Obj_Γ, and the t in $\lambda x : T.t$ can not be an element of Kind_Γ. We will use these stronger conditions in the proof in order to restrict the number of necessary cases further. These individual contradictions can be derived using Lemmata 18 and 23, the details are left as an exercise for the reader.

3. MATHEMATICAL PROOF OF TERMINATION

Recall that at least for some of the propositions P we would expect to be provable, we can get a proof p such that $\emptyset \vdash p : P$. Can we get such a proof for some proposition which we expect not to be provable? One such proposition is False (Section 2.8, p. 19). It is obvious from the definition of False that if it has a proof, we can derive a p for any proposition P such that $\emptyset \vdash p : P$. In other words, if there exists a proof of False, CC_Σ is logically inconsistent. It seems desirable to rule out this possibility. As Luo points out [23, Theorem 5.4, p. 105], subject reduction and termination imply this result for ECC, of which CC_Σ is a strict subset. This gives us a strong motivation to prove those three properties. While we referred to the literature for the proof of subject reduction, we gave a proof of confluence and will prove termination, too. We will do so by interpreting every type under some valuation ξ with a set of terms. We choose sets that are, among other things, terminating. We also need to make sure that this key property holds:

$$\Gamma \vdash t : T \;\rightarrow\; t \in [\![T]\!]_\xi$$

We will call this property the soundness of the interpretation.

Since $[\![T]\!]_\xi$ has the termination property, t terminates. Similarly it follows that T terminates: because we have that $\Gamma \vdash t : T$, we also know that either $\exists s.T = s$ - in which case it terminates because all universes terminate - or $\exists K.\Gamma \vdash T : K$, in which case we can apply the soundness property to conclude that T must terminate.

We will not be able to prove the soundness property directly. We will prove a more general form, which includes parallelly substituting the variables in t. We can then get this specific version by substituting with ρ_I, i.e., substituting every variable with itself and not changing t at all.

Note that many of the following Lemmata come from [13]. However, Geuvers does not give proofs except for Lemma 41 (where he gives a proof sketch) and Theorem 3 (where he gives a proof only for some of the cases). We give proofs here.

3.1 Classification

In some fixed environment Γ, we can classify terms into three levels: kinds like Prop, Prop $\rightarrow$ Prop, etc., constructors like True, negation, conjunction etc. and

objects like I, the proof of $\forall x : \mathrm{Prop}, y : \mathrm{Prop}, (x \wedge y) \to (x \vee y)$, etc. (Section 2.8, p. 19).

Instead of using environments Γ, we will often use class-valuations η to determine what class a term is in. Hence we define sets $\mathcal{K}_\eta$, $\mathcal{C}_\eta$ and $\mathcal{O}_\eta$ to be the kinds, constructors and objects under η, respectively. The basic idea is that instead of extending the Γ with a judgement $x : T$, we will extend η with the class of T under η. Then we can reconstruct the class of variables and hence of whole terms. First we use three distinct symbols, $\mathcal{K}$, $\mathcal{C}$ and $\mathcal{O}$ to denote membership in $\mathcal{K}_\eta$, $\mathcal{C}_\eta$ and $\mathcal{O}_\eta$, respectively. η is a $\{\mathcal{K}, \mathcal{C}, \mathcal{O}\}$-valuation. We define $\mathrm{class}_\eta(t)$ to be the class of t under η:

Definition 15.

$$\begin{aligned}
\mathrm{class}_\eta(s) &:= \mathcal{K} \\
\mathrm{class}_\eta(x) &:= \begin{cases} \mathcal{C} & \text{if } \eta(x) = \mathcal{K} \\ \mathcal{O} & \text{otherwise} \end{cases} \\
\mathrm{class}_\eta(tu) &:= \mathrm{class}_\eta(t) \\
\mathrm{class}_\eta(\lambda x : T.u) &:= \mathrm{class}_{\eta^x_{\mathrm{class}_\eta(T)}}(u) \\
\mathrm{class}_\eta(\forall x : T.U) &:= \mathrm{class}_{\eta^x_{\mathrm{class}_\eta(T)}}(U) \\
\mathrm{class}_\eta((t,u)_{\Sigma x:T.U}) &:= \mathcal{O} \\
\mathrm{class}_\eta(\Sigma x : T.U) &:= \mathcal{C} \\
\mathrm{class}_\eta(\pi_i(t)) &:= \mathcal{O}
\end{aligned}$$

It is important to note that when $x \notin \mathcal{F}(t)$, the value we assign to x in η is irrelevant for $\mathrm{class}_\eta(t)$.

Lemma 30. *For all t, η and η' such that $\eta(x) = \eta'(x)$ for all x which are free in t, $\mathrm{class}_\eta(t) = \mathrm{class}_{\eta'}(t)$.*

Proof. By induction over t. We will only consider the cases where t is an application or a product. The other cases are either trivial or analogous.

$t\,u$: By definition, $\mathrm{class}_\eta(t\,u) = \mathrm{class}_\eta(t)$ and $\mathrm{class}_{\eta'}(t\,u) = \mathrm{class}_{\eta'}(t)$, so it suffices to prove $\mathrm{class}_\eta(t) = \mathrm{class}_{\eta'}(t)$. For this we use the inductive hypothesis, which requires us to prove that $\eta(x) = \eta(x')$ for all $x \in \mathcal{F}(t)$. We have this for all $x \in \mathcal{F}(t\,u)$, of which $\mathcal{F}(t)$ is a subset.

$\forall x : T.U$: By definition, $\mathrm{class}_\eta(\forall x : T.U) = \mathrm{class}_{\eta^x_{\mathrm{class}_\eta(T)}}(U)$ and

$$\mathrm{class}_{\eta'}(\forall x : T.U) = \mathrm{class}_{\eta'^x_{\mathrm{class}_{\eta'}(T)}}(U)$$

We first prove $\mathrm{class}_\eta(T) = \mathrm{class}_{\eta'}(T)$, which follows from the induction hypothesis for T and the fact that $\mathcal{F}(T) \subseteq \mathcal{F}(\forall x : T.U)$. Now we prove $\mathrm{class}_{\eta^x_{\mathrm{class}_\eta(T)}}(U) = \mathrm{class}_{\eta'^x_{\mathrm{class}_\eta(T)}}(U)$. This follows from the induction hypothesis for U if we can prove that for all $y \in \mathcal{F}(U)$, $\eta^x_{\mathrm{class}_\eta(T)}(y) =$

$\eta'^{x}_{\mathrm{class}_\eta(T)}(y)$. If $x = y$, this is trivial ; and $\mathcal{F}(U) \setminus \{x\}$ is a subset of $\mathcal{F}(\forall x : T.U)$, for which we have the property.

□

Now we define the individual classes by using the symbol we get from $\mathrm{class}_\eta(t)$:

Definition 16.

$$\mathcal{K}_\eta := \{t \in \Lambda \mid \mathrm{class}_\eta(t) = \mathcal{K}\}$$
$$\mathcal{C}_\eta := \{t \in \Lambda \mid \mathrm{class}_\eta(t) = \mathcal{C}\}$$
$$\mathcal{O}_\eta := \{t \in \Lambda \mid \mathrm{class}_\eta(t) = \mathcal{O}\}$$

It is obvious that these classes are disjoint and that every term has a class. Hence the class of a term is well defined and we can make case distinctions depending on the class of a term.

We want classes and levels to correspond. A term $T \in \mathrm{Kind}_\Gamma$ should also be in $\mathcal{K}_\eta$, etc. - at least if η and Γ satisfy a well-chosen property. Consider terms T and K such that $\Gamma \vdash T : K$, where $K \neq \mathrm{Type}_0$. By Lemma 23, we have $\Gamma \vdash K : s$ for some s. Let C_T and C_K be the classes of T and K under η, respectively. We want to enforce a relation between C_T and s and between C_K and s. We call these relations $\ltimes_2$ and $\ltimes_1$, respectively. It is possible to think of the index as the "distance" between the class and the universe.

For a class C and a universe s we define $C \ltimes_1 s$ in the following way:

Definition 17.

$$\mathcal{K} \ltimes_1 \mathrm{Type}_0$$
$$\mathcal{C} \ltimes_1 \mathrm{Prop}$$

Similarly we define $C \ltimes_2 s$ as follows:

Definition 18.

$$\mathcal{C} \ltimes_2 \mathrm{Type}_0$$
$$\mathcal{O} \ltimes_2 \mathrm{Prop}$$

We want to lift $\ltimes_1$ from classes and universes to class-valuations and environments. For this we define what it means for an η to *preserve* a Γ.

Definition 19. We define $\eta \ltimes \Gamma$ inductively:

$$\textit{preservation empty} \quad \frac{}{\eta \ltimes \emptyset}$$

$$\textit{preservation extend} \quad \frac{\eta \ltimes \Gamma \qquad \exists s, \Gamma \vdash T : s \wedge C \ltimes_1 s}{\eta^x_C \ltimes \Gamma, x : T}$$

Now we prove that when we have $\eta \ltimes \Gamma$, we can argue for all pairs $x : T \in \Gamma$ and all s such that T has type s that $\eta(x) \ltimes_1 s$. Note that we go from "exists s such that ..." in $\eta \ltimes \Gamma$ to "for all s such that ...". We will prove that this is acceptable by proving that there is only one such s, which can be concluded from Lemma 24.

Lemma 31. *For well-formed Γ and η such that $\eta \ltimes \Gamma$, for all prefixes $\Gamma', x : T$ of Γ and for all s such that $\Gamma' \vdash T : s$, $\eta(x) \ltimes_1 s$.*

Proof. We prove this by induction on the statement $\eta \ltimes \Gamma$.

$\eta \ltimes \emptyset$: Since there are no prefixes $\Gamma', x : T$ of $\emptyset$, the statement is vacuously true.

$\eta^x_C \ltimes \Gamma, x : T$: We need to prove that for all prefixes $\Gamma', y : U$ of $\Gamma, x : T$, and all s such that $\Gamma' \vdash U : s$, $\eta^x_C(y) \ltimes_1 s$. We distinguish between two cases: either $\Gamma', y : U = \Gamma, x : T$, or $\Gamma', y : U$ is a proper prefix of $\Gamma, x : T$. In this case, $\Gamma', y : U$ is a prefix of Γ.

- $\Gamma', y : U = \Gamma, x : T$: We have $\Gamma' = \Gamma$, $y = x$, and $U = T$. We have to prove $C \ltimes_1 s$ where $\Gamma \vdash T : s$. We have as an assumption that for some s' such that $\Gamma \vdash T : s'$, $C \ltimes_1 s'$. We will prove that $s = s'$. Since convertible normal terms are equal, it suffices to show that $s \approx_\beta s'$. This follows from Lemma 24.
- $\Gamma', y : U$ *is a prefix of* Γ : Since $\Gamma, x : T$ is well-formed and thus x is $\Gamma-$fresh, we know $y \neq x$. We have to prove $\eta(y) \ltimes_1 s$ where $\Gamma' \vdash T : s$. We use the induction hypothesis for $\eta \ltimes \Gamma$. We have to prove that Γ is well-formed, which we have by Lemma 17 since $\Gamma, x : T$ is well-formed.

□

Note that both Geuvers' and Barras' proofs of termination differ slightly from the proof presented here. In Geuvers, classification is done by syntactically separating variables into kind and constructor variables. Hence η becomes a part of the term, and preservation becomes trivially true. In fact, Geuvers makes no distinction between the class of a term and its level [13, p. 17]. Barras does not use the property of preservation. Instead, this correspondence is proven for each level individually, first for kinds, then for constructors, and finally for objects. The proof presented here is shorter and more direct.

If η preserves Γ, then it will also preserve the levels of terms in Γ.

Theorem 1. *For well-formed Γ and all η such that $\eta \ltimes \Gamma$, for all T and K such that $\Gamma \vdash T : K$, we have:*

$$K = \mathrm{Type}_0 \wedge \mathrm{class}_\eta(T) = \mathcal{K}$$
$$\vee \exists s, \Gamma \vdash K : s \wedge \mathrm{class}_\eta(T) \ltimes_2 s$$

Proof. We prove this by induction on the typing derivation of $\Gamma \vdash T : K$. We only show the variable, lambda, and application cases; the other cases are trivial or similar.

variable : Suppose the typing derivation ends with

$$\textit{variable}\quad \frac{\Gamma \text{ is well-formed} \qquad \Gamma', x : T \subseteq \Gamma \text{ for some } \Gamma'}{\Gamma \vdash x : T}$$

We will prove $\exists s, \Gamma \vdash T : s \wedge \text{class}_\eta\,(x) \ltimes_2 s$. Since $\Gamma', x : T$ is a prefix of Γ and Γ is well-formed, we know $\Gamma', x : T$ is well-formed by Lemma 17. Thus $\Gamma' \vdash T : s$ for some s.

We will prove $\Gamma \vdash T : s$ and $\text{class}_\eta\,(x) \ltimes_2 s$. The first follows from repeated application of Lemma 20. For $\text{class}_\eta\,(x) \ltimes_2 s$, we first argue $\eta(x) \ltimes_1 s$ by $\eta \ltimes \gamma$ and Lemma 31. Then by definition of $\eta(x) \ltimes_1 s$, we have either $\eta(x) = \mathcal{K}$ and $s = \text{Type}_0$ or $\eta(x) = \mathcal{C}$ and $s = \text{Prop}$. We only treat the first case, the second case is analogous. In this case, $\text{class}_\eta\,(x) = \mathcal{C}$ and we have to prove $\mathcal{C} \ltimes_2 \text{Type}_0$, which is trivial.

lambda : Suppose the typing derivation ends with

$$\textit{lambda}\quad \frac{\Gamma \vdash T : s_1 \qquad \Gamma, x : T \vdash u : U \qquad \Gamma, x : T \vdash U : s_2}{\Gamma \vdash (\lambda x : T.u) : (\forall x : T.U)}$$

We decide to prove $\exists s, \Gamma \vdash (\forall x : T.U) : s \wedge \text{class}_\eta\,(\lambda x : T.u) \ltimes_2 s$. We choose $s := s_2$. By our assumptions and the product rule, we conclude $\Gamma \vdash (\forall x : T.U) : s_2$, so it only remains to prove that $\text{class}_\eta\,(\lambda x : T.u) \ltimes_2 s_2$.

We will first prove that $\eta^x_{\text{class}_\eta(T)} \ltimes \Gamma, x : T$. It suffices to show that there is an s such that $\Gamma \vdash T : s$ and $\text{class}_\eta\,(T) \ltimes_1 s$. We choose s_1. Since $\Gamma \vdash T : s_1$ by assumption, we only need to prove $\text{class}_\eta\,(T) \ltimes_1 s_1$. For this we use the induction hypothesis for $\Gamma \vdash T : s_1$, which leaves us in one of two cases

$s_1 = \text{Type}_0 \wedge \text{class}_\eta\,(T) = \mathcal{K}$: We only have to prove $\mathcal{K} \ltimes_1 \text{Type}_0$, which is true by definition.

$\exists s', \Gamma \vdash s_1 : s' \wedge \text{class}_\eta\,(T) \ltimes_2 s'$. We can easily argue that $s_1 = \text{Prop}$ and $s' = \text{Type}_0$ in this case by case analysis of s_1 and Lemma 18. Consequently $\text{class}_\eta\,(T) \ltimes_2 \text{Type}_0$, which implies $\text{class}_\eta\,(T) = \mathcal{C}$. We thus have to prove $\mathcal{C} \ltimes_1 \text{Prop}$, which is true by definition.

We use the inductive hypothesis for $\Gamma, x : T \vdash u : U$ and $\eta^x_C \ltimes \Gamma, x : T$ where $C := \text{class}_\eta\,(T)$, which leaves us in one of two cases.

$U = \text{Type}_0 \wedge \text{class}_{\eta^x_C}\,(u) = \mathcal{K}$: This is a contradiction by Lemma 18.2 and $\Gamma, x : T \vdash U : s_2$.

$\exists s'', \Gamma, x : T \vdash U : s'' \wedge \mathrm{class}_{\eta_C^x}(u) \ltimes_2 s''$: We prove that $s'' = s_2$, which holds by Lemma 24 since convertible normal terms are equal. Consequently we have $\mathrm{class}_{\eta_C^x}(u) \ltimes_2 s_2$ and need to prove:

$$\mathrm{class}_\eta(\lambda x : T.u) \ltimes_2 s_2$$

which by definition of $\mathrm{class}_\eta(\lambda x : T.u)$ are equivalent.

application : Suppose the typing derivation ends with

$$\textit{application}\quad \frac{\Gamma \vdash t : (\forall x : T.U) \qquad \Gamma \vdash u : T}{\Gamma \vdash t\, u : U_u^x}$$

We decide to prove $\exists s, \Gamma \vdash U_u^x : s \wedge \mathrm{class}_\eta(t\, u) \ltimes_2 s$. We use the induction hypothesis for $\Gamma \vdash t : (\forall x : T.U)$, which leaves us in one of two cases.

$\forall x : T.U = \mathrm{Type}_0 \wedge \mathrm{class}_\eta(t) = \mathcal{K}$: This is obviously a contradiction since $\forall x : T.U \neq \mathrm{Type}_0$.

$\exists s, \Gamma \vdash (\forall x : T.U) : s \wedge \mathrm{class}_\eta(t) \ltimes_2 s$: We choose the same s for our goal. We now prove $\Gamma \vdash U_u^x : s$. By Lemma 21, it suffices to prove $\Gamma, x : T \vdash U : s$ and $\Gamma \vdash u : T$. The first follows from Lemma 18.6 and the fact that convertible normal terms are equal ; the second we have as an assumption.

It remains to be proven that $\mathrm{class}_\eta(t\, u) \ltimes_2 s$. Since $\mathrm{class}_\eta(t\, u) = \mathrm{class}_\eta(t)$ by definition, it suffices to prove $\mathrm{class}_\eta(t) \ltimes_2 s$, which we have as an assumption.

□

For all Γ which are well-formed, we have an η such that $\eta \ltimes \Gamma$. We define one by induction on Γ:

Definition 20.

$$\eta_\emptyset := (x \mapsto \mathcal{K})$$
$$\eta_{\Gamma, x:T} := \eta_\Gamma{}^x_{\mathrm{class}_{\eta_\Gamma}(T)}$$

When in the future we write $\mathcal{K}_\Gamma$, $\mathcal{C}_\Gamma$ or $\mathcal{O}_\Gamma$, we mean $\mathcal{K}_{\eta_\Gamma}$, $\mathcal{C}_{\eta_\Gamma}$ or $\mathcal{O}_{\eta_\Gamma}$, respectively. We also write $\mathrm{class}_\Gamma(t)$ for $\mathrm{class}_{\eta_\Gamma}(t)$.

We prove that $\eta_\Gamma \ltimes \Gamma$:

Lemma 32. *For well-formed* Γ, $\eta_\Gamma \ltimes \Gamma$.

Proof. By induction on the structure of Γ.

$\emptyset$: This is true by definition.

$\Gamma, x : T$: We need to prove that there is an s such that $\Gamma \vdash T : s$ and $\mathrm{class}_{\eta_\Gamma}(T) \ltimes_1 s$. We choose the s we get from $\Gamma, x : T$ being well-formed, and it suffices to prove $\mathrm{class}_{\eta_\Gamma}(T) \ltimes_1 s$. We use Theorem 1, which leaves us in one of two cases.

$s = \mathrm{Type}_0 \wedge \mathrm{class}_{\eta_\Gamma}(T) = \mathcal{K}$: We only have to prove $\mathcal{K} \ltimes_1 \mathrm{Type}_0$, which is true by definition.

$\exists s', \Gamma \vdash s : s' \wedge \mathrm{class}_{\eta_\Gamma}(T) \ltimes_2 s'$. We can easily argue that $s = \mathrm{Prop}$ and $s' = \mathrm{Type}_0$ in this case by case analysis of s and Lemma 18. Consequently $\mathrm{class}_{\eta_\Gamma}(T) \ltimes_2 \mathrm{Type}_0$, which implies $\mathrm{class}_{\eta_\Gamma}(T) = \mathcal{C}$. We thus have to prove $\mathcal{C} \ltimes_1 \mathrm{Prop}$, which is true by definition.

□

Note that for terms T which have a type in Γ, Theorem 1 allows us to go back and forth between the level and the class of a term. Since classes are disjoint, this also implies the disjointness of levels.

There are two simple corollaries of Theorem 1: terms with the same type have the same class, and convertible terms do, too.

Lemma 33. *Whenever* $\Gamma \vdash t : T$ *and* $\Gamma \vdash t' : T$*, we have* $\mathrm{class}_\Gamma(t) = \mathrm{class}_\Gamma(t')$.

Proof. Since t and t' have the same type in Γ, they have the same level, too. Terms that have the same level also have the same class by Theorem 1 and Lemma 32. □

Lemma 34. *Whenever* $\Gamma \vdash t : T$ *and* $\Gamma \vdash t' : T'$*,* $t \approx_\beta t'$ *implies* $\mathrm{class}_\Gamma(t) = \mathrm{class}_\Gamma(t')$.

Proof. By Lemma 23, we have $T = \mathrm{Type}_0$ or $\Gamma \vdash T : s$ for some s.

$T = \mathrm{Type}_0$: We can easily argue that $T' = \mathrm{Type}_0$ in this case. Thus Lemma 33 applies.

$\Gamma \vdash T : s$ *for some* s : By Lemma 26, we have $T \approx_\beta T'$. We prove $\Gamma \vdash t' : T$ by the conversion rule. Thus Lemma 33 applies.

□

3.2 Candidates of Reducibility

Candidates of reducibility have been introduced by Tait in 1967 and many technical improvements have been introduced by Girard in 1972 [16, p. 41].

Definition 21. We say a set X is a candidate of reducibility, if it has the following three properties:

$$X \subseteq \mathrm{SN} \qquad \text{(termination)}$$

$$\forall t \in X.\beta_t \subseteq X \qquad \text{(closure under reduction)}$$

$$\mathrm{neutral}\, t \;\rightarrow\; \beta_t \subseteq X \;\rightarrow\; t \in X \qquad \text{(closure under expansion)}$$

SN is such a candidate of reducibility, and because of termination it is the largest one. Since candidates of reducibility are closed under expansion for neutral terms, every candidate of reducibility includes all neutral normal terms, e.g., all the variables.

3.3 Extending Candidates of Reducibility

While this notion of candidates of reducibility serves us well to prove termination of polymorphic lambda calculus, in order to go to CC or even CC_Σ, we need to extend candidates to work with type operators. We can think of types as sets of terms, i.e. the set of terms that have that type in some Γ. Type operators however map types to types, so using the same perspective we can think of them as somehow related to functions that map sets of terms to sets of terms.

We will hence extend candidates of reducibility to function spaces. We define CR to be the set of all sets that have the reducibility properties, and $(\mathrm{CR})^*$ to be CR closed under function spaces. In order for this definition to work, we will first define the inductive skeletal structure of the elements of $(\mathrm{CR})^*$, and then map these skeletal structures back to function types.

Note that these definitions are a mix of Geuvers and Barras: Geuvers does not use skeletons and uses saturated sets [13, p. 21], and Barras uses the set of all sets of terms as a base (i.e., 2^Λ, not CR). He then uses a predicate that describes those elements of $\left(2^\Lambda\right)^*$ which are in $(\mathrm{CR})^*$. This has technical reasons, as $(\mathrm{CR})^*$ is hard to directly formalize in Coq.

Definition 22.

$$\mathrm{CR} := \{X \subseteq \Lambda \mid X \text{ is a reducibility candidate}\}$$

Definition 23. We inductively define the set of skeletons, which represent the skeletal structure of kinds:

$$\mathrm{Skel} := \bigstar \mid s_1 \to_S s_2$$

where s_1 and s_2 are skeletons.

Definition 24. We define *parents* of order s recursively along the skeleton s:

$$\begin{aligned} \mathrm{can}(\bigstar) &:= \mathrm{CR} \\ \mathrm{can}(s_1 \to_S s_2) &:= \{f \mid f : \mathrm{can}(s_1) \to \mathrm{can}(s_2)\} \end{aligned}$$

We call the elements of the parent of order s the candidates of order s. Note that candidates of order $\bigstar$ are exactly the reducibility candidates, while candidates of higher orders are functions that map from parents into parents, i.e., candidates to candidates.

Definition 25. We define $(\mathrm{CR})^*$ as CR, closed under function spaces:

$$(\mathrm{CR})^* := \bigcup_{s \in \mathrm{Skel}} (\mathrm{can}(s))$$

3.4 Type Skeletons

With this definition of skeleton we can extract the structure of a term K with relation to the type operators occurring in K.

Definition 26. For η and K we define $\text{skeleton}_\eta(K) \in \text{Skel}$:

$$\begin{aligned}
\text{skeleton}_\eta(\text{Prop}) &:= \bigstar \\
\text{skeleton}_\eta(\forall x : T.U) &:= \text{skeleton}_\eta(T) \to_S \text{skeleton}_{\eta^x_{\text{class}_\eta(T)}}(U) && \text{if } T \in \mathcal{K}_\eta \\
\text{skeleton}_\eta(\forall x : T.U) &:= \text{skeleton}_{\eta^x_{\text{class}_\eta(T)}}(U) && \text{if } T \in \mathcal{C}_\eta
\end{aligned}$$

Note that this is a partial function. We will have to prove that this is defined for all $K \in \text{Kind}_\Gamma$ for some Γ such that $\eta \ltimes \Gamma$, and we will use it only then.

Lemma 35. *When $\eta \ltimes \Gamma$ and $K \in \text{Kind}_\Gamma$, $\text{skeleton}_\eta(K)$ is defined.*

Proof. We prove this by induction on the typing derivation of $\Gamma \vdash K : \text{Type}_0$, using Lemma 28.

prop : Suppose the typing derivation ends with

$$prop \quad \frac{\Gamma \text{ is well-formed}}{\Gamma \vdash \text{Prop} : \text{Type}_0}$$

$\text{skeleton}_\eta(\text{Prop}) = \bigstar$ and is thus defined.

product where $U \in \text{Kind}_{\Gamma,x:T}$ and $s_2 = \text{Type}_0$: Suppose the typing derivation ends with

$$product \quad \frac{\Gamma \vdash T : s_1 \qquad \Gamma, x : T \vdash U : s_2}{\Gamma \vdash (\forall x : T.U) : s_2}$$

We have $\Gamma \vdash T : s_1$ as an assumption. By a case distinction on s_1, we get the following two cases:

$s_1 = \text{Type}_0$ *and thus $T \in \text{Kind}_\Gamma$* : Thus by Theorem 1, $T \in \mathcal{K}_\eta$. By definition,

$$\text{skeleton}_\eta(\forall x : T.U) = \text{skeleton}_\eta(T) \to_S \text{skeleton}_{\eta^x_{\text{class}_\eta(T)}}(U)$$

By the induction hypotheses for $\Gamma \vdash T : \text{Type}_0$ and $\Gamma, x : T \vdash U : \text{Type}_0$, this is defined if we can prove $\eta^x_{\text{class}_\eta(T)} \ltimes \Gamma, x : T$. The proof of this goes analogous to that in Lemma 32.

$s_1 =$ Prop *and thus* $T \in \mathrm{Constr}_\Gamma$: Thus by Theorem 1, $T \in \mathcal{C}_\eta$. By definition,

$$\mathrm{skeleton}_\eta\left(\forall x : T.U\right) = \mathrm{skeleton}_{\eta^x_{\mathrm{class}_\eta(T)}}\left(U\right)$$

By the induction hypothesis for $\Gamma, x : T \vdash U : \mathrm{Type}_0$, this is defined if we can prove $\eta^x_{\mathrm{class}_\eta(T)} \ltimes \Gamma, x : T$. The proof of this goes analogous to that in Lemma 32.

conversion where $U = \mathrm{Type}_0$: Suppose the typing derivation ends with

$$\textit{conversion}\quad \frac{\Gamma \vdash t : T \qquad \Gamma \vdash U : s \qquad T \approx_\beta U}{\Gamma \vdash t : U}$$

We have as an assumption that $\Gamma \vdash \mathrm{Type}_0 : s$, which by Lemma 18.2 is a contradiction.

□

When we have a $T \in \mathrm{Constr}_\Gamma$ that we want to interpret, we will have $[\![T]\!]_\xi$ map into $\mathrm{can}(\mathrm{skeleton}_{\eta_\Gamma}(K))$ where K is the type of T and ξ somehow relates to Γ, and for a $T \in \mathrm{Kind}_\Gamma$ we will have $[\![T]\!]_\xi$ map into CR. Recall that all of this is to make sure that if $\Gamma \vdash t : T$, $[\![T]\!]_\xi \subseteq \mathrm{SN}$. For those $[\![T]\!]_\xi$ that map into CR, or $\mathrm{can}(\bigstar)$, we get this for free. For those T where $[\![T]\!]_\xi$ ends up in a different parent, we can easily assure ourselves that they are not inhabited: in order to map into a parent that is not $\mathrm{can}(\bigstar)$, the type of T must be a product. We know that $\Gamma \vdash t : T$ implies that either $T = \mathrm{Type}_0$ or T lives in some universe (Lemma 23). In the first case, T is not in Constr_Γ, and in the second case the type of T is a universe, not a product.

Definition 27. We define the *set-interpretation* of a kind:

$$\mathcal{I}_\eta\left(K\right) := \mathrm{can}(\mathrm{skeleton}_\eta\left(K\right))$$

Similar to our previous notation for, e.g., $\mathcal{C}_{\eta_\Gamma}$, we will write $\mathcal{I}_\Gamma\left(K\right)$ for $\mathcal{I}_{\eta_\Gamma}\left(K\right)$. Note that again $\mathcal{I}_\eta\left(K\right)$ is a partial function, which is defined if and only if $\mathrm{skeleton}_\eta\left(K\right)$ is defined. Thus we will only use $\mathcal{I}_\eta\left(K\right)$ if $K \in \mathrm{Kind}_\Gamma$ for some Γ such that $\eta \ltimes \Gamma$. In order for the upcoming proofs to go through, we need to prove that $\mathcal{I}_{\Gamma,x:U}\left(T\right) = \mathcal{I}_\Gamma\left(T^x_u\right)$ for every $T \in \mathrm{Kind}_\Gamma$ and $\Gamma \vdash u : U$. Similarly we need that $\mathcal{I}_\Gamma\left(T\right) = \mathcal{I}_\Gamma\left(U\right)$ when $T \approx_\beta U$. The reason why we need stability under substitution is that some of typing derivations will include a substitution that we will have to eliminate in order to apply the inductive hypothesis. One such example is $t\,u : U^x_u$, where $t : \forall x : T.U$ and $u : T$. The inductive hypothesis on the product will give us (after some work) the property for U, but not for U^x_u - but because $\mathcal{I}_\Gamma\left(\cdot\right)$ is stable under substitution, this will be enough. Likewise, we need stability under beta reduction because of the conversion rule: When we have a λ-term $\lambda x : T.t$ with $\Gamma, x : T \vdash t : U$ and $\Gamma \vdash \lambda x : T.t : V$, we cannot argue that $V = \forall x : T.U$. We can only argue that

$V \approx_\beta \forall x : T.U$. The induction hypothesis will only give us the properties for $\forall x : T.U$, not for V. By using the stability result, we can go from one to the other.

Lemma 36. *For every* Γ, Γ' *and* $T \in \mathrm{Kind}_\Gamma$, *and for every* ρ *such that* $\rho \vDash \Gamma \to \Gamma'$, $\mathcal{I}_\Gamma(T)$ *and* $\mathcal{I}_{\Gamma'}(\rho(T))$ *are defined and*

$$\mathcal{I}_\Gamma(T) = \mathcal{I}_{\Gamma'}(\rho(T))$$

Proof. The definedness follows directly from Lemma 35. Note that $\Gamma' \vdash \rho(T) : \mathrm{Type}_0$ by Lemma 21.

We prove the equality by induction on the typing derivation of $\Gamma \vdash T : \mathrm{Type}_0$, using Lemma 28.

prop : Suppose the typing derivation ends with

$$\textit{prop}\ \frac{\Gamma \text{ is well-formed}}{\Gamma \vdash \mathrm{Prop} : \mathrm{Type}_0}$$

Since $\rho(\mathrm{Prop}) = \mathrm{Prop}$,

$$\mathcal{I}_\Gamma(\mathrm{Prop}) = \mathrm{CR} = \mathcal{I}_{\Gamma'}(\rho(\mathrm{Prop}))$$

product where $U \in \mathrm{Kind}_{\Gamma,x:T}$ *and* $s_2 = \mathrm{Type}_0$: Suppose the typing derivation ends with

$$\textit{product}\ \frac{\Gamma \vdash T : s_1 \qquad \Gamma, x : T \vdash U : s_2}{\Gamma \vdash (\forall x : T.U) : s_2}$$

We have $\Gamma \vdash T : s_1$ as an assumption. By a case distinction on s_1, we get the following two cases:

$s_1 = \mathrm{Type}_0$ *and thus* $T \in \mathrm{Kind}_\Gamma$: By Lemma 21 and $\rho(\mathrm{Type}_0) = \mathrm{Type}_0$, $\rho(T) \in \mathrm{Kind}_{\Gamma'}$. Thus by Theorem 1, $T \in \mathcal{K}_\Gamma$ and $\rho(T) \in \mathcal{K}_{\Gamma'}$.
Therefore,

$$\mathcal{I}_\Gamma(\forall x : T.U) = \mathcal{I}_\Gamma(T) \to \mathcal{I}_{\Gamma,x:T}(U)$$

and

$$\mathcal{I}_{\Gamma'}(\rho(\forall x : T.U)) = \mathcal{I}_{\Gamma'}(\rho(T')) \to \mathcal{I}_{\Gamma',z:\rho(T)}(\rho_z^x(U))$$

By the induction hypotheses for $\Gamma \vdash T : \mathrm{Type}_0$ and $\Gamma, x : T \vdash U : \mathrm{Type}_0$, these are equal if we can prove that $\rho_z^x \vDash \Gamma, x : T \to \Gamma', z : \rho(T)$. This follows from $\rho \vDash \Gamma \to \Gamma'$ and Lemma 20 ; the details are left as an exercise for the reader.

$s_1 = \mathrm{Prop}$ *and thus* $T \in \mathrm{Constr}_\Gamma$: The case is similar to the one above and is left as an exercise for the reader.

conversion where $U = \mathrm{Type}_0$: Suppose the typing derivation ends with

$$\textit{conversion}\quad \frac{\Gamma \vdash t : T \qquad \Gamma \vdash U : s \qquad T \approx_\beta U}{\Gamma \vdash t : U}$$

We have as an assumption that $\Gamma \vdash \mathrm{Type}_0 : s$, which by Lemma 18.2 is a contradiction.

□

Lemma 37. *For every* Γ *and* $T, U \in \mathrm{Kind}_\Gamma$ *such that* $T \approx_\beta U$, $\mathcal{I}_\Gamma(T)$ *and* $\mathcal{I}_\Gamma(U)$ *are defined and*

$$\mathcal{I}_\Gamma(T) = \mathcal{I}_\Gamma(U)$$

Proof. The definedness follows directly from Lemma 35. Because of Lemma 15 and the fact that equality is an equivalence relation, we can reduce convertibility to one step beta-reduction.

For every Γ, $T \in \mathrm{Kind}_\Gamma$ and V such that $T \to_\beta V$, $\mathcal{I}_\Gamma(T)$ and $\mathcal{I}_\Gamma(V)$ are defined and

$$\mathcal{I}_\Gamma(T) = \mathcal{I}_\Gamma(V)$$

Note that $V \in \mathrm{Kind}_\Gamma$ by Lemma 25. Thus the well-definedness follows directly from Lemma 35.

We prove the equality by induction over the derivation of $\Gamma \vdash T : \mathrm{Type}_0$. We use Lemma 28.

prop : Suppose the typing derivation ends with

$$\textit{prop}\quad \frac{\Gamma \text{ is well-formed}}{\Gamma \vdash \mathrm{Prop} : \mathrm{Type}_0}$$

Since universes are normal, this case can not appear.

product where $s_2 = \mathrm{Type}_0$: Suppose the typing derivation ends with

$$\textit{product}\quad \frac{\Gamma \vdash T : s_1 \qquad \Gamma, x : T \vdash U : s_2}{\Gamma \vdash (\forall x : T.U) : s_2}$$

We have $\Gamma \vdash T : s_1$ as an assumption. By a case distinction on s_1, we get the following two cases:

$s_1 = \mathrm{Type}_0$ *and thus* $T \in \mathrm{Kind}_\Gamma$: $\mathcal{I}_\Gamma(\forall x : T.u) = \mathcal{I}_\Gamma(T) \to \mathcal{I}_{\Gamma, x:T}(V)$ by definition. Case analysis of $\forall x : T.U \to_\beta V$, leaves us in one of two cases:

$\forall x : T.U \to_\beta \forall x : T'.U$ where $T \to_\beta T'$: By the induction hypothesis for $\Gamma \vdash T : \text{Type}_0$, $\mathcal{I}_\Gamma(T) = \mathcal{I}_\Gamma(T')$. By Lemma 34, $\text{class}_\Gamma(T) = \text{class}_\Gamma(T')$. So:

$$\begin{aligned}\mathcal{I}_\Gamma(T) \to \mathcal{I}_{\Gamma,x:T}(U) &= \mathcal{I}_\Gamma(T') \to \mathcal{I}_{\Gamma,x:T'}(U) \\ &= \mathcal{I}_\Gamma(\forall x : T'.U)\end{aligned}$$

$\forall x : T.U \to_\beta \forall x : T.U'$ where $U \to_\beta U'$: By the same line of reasoning,

$$\begin{aligned}\mathcal{I}_\eta(T) \to \mathcal{I}_{\Gamma,x:T}(U) &= \mathcal{I}_\eta(T) \to \mathcal{I}_{\Gamma,x:T}(U') \\ &= \mathcal{I}_\eta(\forall x : T.U')\end{aligned}$$

$s_1 = \text{Prop}$ *and thus* $T \in \text{Constr}_\Gamma$: The case is similar to the one above and is left as an exercise for the reader.

conversion where $U = \text{Type}_0$: Suppose the typing derivation ends with

$$\textit{conversion}\quad \frac{\Gamma \vdash t : T \qquad \Gamma \vdash U : s \qquad T \approx_\beta U}{\Gamma \vdash t : U}$$

We have as an assumption that $\Gamma \vdash \text{Type}_0 : s$, which by Lemma 18.2 is a contradiction.

□

3.5 Interpretation

We need to define our interpretation such that we map into the right element of $(\text{CR})^*$ and also that we can prove the soundness property. Before we can do that though, we will make a couple of helpful definitions which will be the basis for our interpretation of products and sigmas. These definitions come from [13, Lemma 3.5 on p. 21 and Definition 5.3 on p. 29].

Definition 28.

$$X \rightrightarrows Y := \{m \in \Lambda \mid \forall t \in X.m\,t \in Y\}$$

Definition 29.

$$X \times Y := \{m \in \Lambda \mid \pi_1(m) \in X \wedge \pi_2(m) \in Y\}$$

We will need to get the class of a term with relation to a $(\text{CR})^* \cup \{\mathcal{C}, \mathcal{O}\}$-valuation ξ. For that we can easily translate ξ to a class-valuation by mapping the elements of $(\text{CR})^*$ to $\mathcal{K}$ and keeping the other symbols:

Definition 30.

$$\eta_\xi(x) := \begin{cases} \mathcal{K} & \xi(x) \in (\mathrm{CR})^* \\ \xi(x) \end{cases}$$

Similar to how we often use Γ when we mean η_Γ, we will also use ξ in those places when we mean η_ξ. Now we can finally define how we interpret the terms of CC_Σ:

Definition 31.

$$\begin{aligned}
[\![s]\!]_\xi &:= \mathrm{SN} \\
[\![x]\!]_\xi &:= \xi(x) \\
[\![t\,u]\!]_\xi &:= [\![t]\!]_\xi([\![u]\!]_\xi) && \text{if } u \in \mathcal{C}_\xi \\
[\![t\,u]\!]_\xi &:= [\![t]\!]_\xi && \text{if } u \in \mathcal{O}_\xi \\
[\![\lambda x : T.u]\!]_\xi &:= \left(\Phi \in \mathcal{I}_\xi\,(T) \mapsto [\![U]\!]_{\xi^x_\Phi}\right) && \text{if } T \in \mathcal{K}_\xi \\
[\![\lambda x : T.U]\!]_\xi &:= [\![U]\!]_{\xi^x_\mathcal{C}} && \text{if } T \in \mathcal{C}_\xi \\
[\![\forall x : T.U]\!]_\xi &:= [\![T]\!]_\xi \rightrightarrows \cap_{\Phi \in \mathcal{I}_\xi(T)} [\![U]\!]_{\xi^x_\Phi} && \text{if } T \in \mathcal{K}_\xi \\
[\![\forall x : T.U]\!]_\xi &:= [\![T]\!]_\xi \rightrightarrows [\![U]\!]_{\xi^x_\mathcal{C}} && \text{if } T \in \mathcal{C}_\xi \\
[\![\Sigma x : T.U]\!]_\xi &:= [\![T]\!]_\xi \times [\![U]\!]_{\xi^x_\mathcal{C}} \\
[\![(t,u)_{\Sigma x:T.U}]\!]_\xi &:= \mathrm{SN} \\
[\![\pi_1(t)]\!]_\xi &:= \mathrm{SN} \\
[\![\pi_2(t)]\!]_\xi &:= \mathrm{SN}
\end{aligned}$$

Note that this is not a total definition. As an example, consider the application case $[\![t\,u]\!]_\xi$ where $t \in \mathcal{C}_\xi$: $[\![t]\!]_\xi([\![u]\!]_\xi)$ is only a valid definition of $[\![t]\!]_\xi : A \to B$ and $[\![u]\!]_\xi \in A$ for some sets A and B. We will later prove that if T is a kind or constructor in some environment Γ, $[\![T]\!]_\xi$ is well-defined if the ξ relates to the Γ in a way we will yet define.

We prove that the values of variables which are not free in T don not matter for $[\![T]\!]_\xi$.

Lemma 38. *If for all ξ, ξ' and T such that for all $x \in \mathcal{F}\,(T)$, $\xi(x) = \xi'(x)$, both $[\![T]\!]_\xi$ and $[\![T]\!]_{\xi'}$ are defined and $[\![T]\!]_\xi = [\![T]\!]_{\xi'}$, or $[\![T]\!]_\xi$ and $[\![T]\!]_{\xi'}$ are both undefined.*

Proof. The proof is via straightforward induction over T and similar to the proof of Lemma 30. As an example, we will sketch the proof for $t\,u$. By definition, $[\![t\,u]\!]_\xi = [\![t]\!]_\xi([\![u]\!]_\xi)$. This is defined only if $[\![t]\!]_\xi$ and $[\![u]\!]_\xi$ are defined, $[\![t]\!]_\xi$ is a function, and $[\![u]\!]_\xi$ is in the domain of that function. Similarly, $[\![t\,u]\!]_{\xi'} = [\![t]\!]_{\xi'}([\![u]\!]_{\xi'})$. Again, this is defined only if $[\![t]\!]_{\xi'}$ and $[\![u]\!]_{\xi'}$ are defined, $[\![t]\!]_{\xi'}$ is a function, and $[\![u]\!]_{\xi'}$ is in the domain of that function. Now by the induction hypothesis for t, $[\![t]\!]_\xi$ equals $[\![t']\!]_\xi$ or both are undefined. This means that either

both are a function or $[\![t\,u]\!]_\xi$ and $[\![t\,u]\!]_{\xi'}$ are undefined. Similarly, either both $[\![u]\!]_\xi$ and $[\![u]\!]_{\xi'}$ are defined and in the domain of that function or $[\![t\,u]\!]_\xi$ and $[\![t\,u]\!]_{\xi'}$ are undefined. Now if both $[\![t\,u]\!]_\xi$ and $[\![t\,u]\!]_{\xi'}$ are defined, we need to prove they are equal. This is true since $[\![t]\!]_\xi = [\![t]\!]_{\xi'}$ and $[\![u]\!]_\xi = [\![u]\!]_{\xi'}$.

□

Note that we never put $\mathcal{O}$ into ξ. In fact, we could do with $(\mathrm{CR})^* \cup \{\mathcal{C}\}$-valuations, but for now we will use the more general ξ.

We have to prove that this interpretation behaves appropriately. However, not every ξ is sensible. Similar to preservation, we need a predicate that makes sure that the ξ fits the environment. We say ξ *respects* Γ or write $\xi \Vdash \Gamma$ if ξ sends the variables to the right element of $(\mathrm{CR})^*$:

Definition 32. We define $\xi \Vdash \Gamma$ inductively.

$$\frac{}{(x \mapsto \mathrm{SN}) \Vdash \emptyset} \qquad \frac{\xi \Vdash \Gamma \qquad T \in \mathrm{Kind}_\Gamma \qquad \Phi \in \mathcal{I}_\Gamma(T)}{\xi^x_\Phi \Vdash \Gamma, x : T}$$

$$\frac{\xi \Vdash \Gamma \qquad T \in \mathrm{Constr}_\Gamma}{\xi^x_\mathcal{C} \Vdash \Gamma, x : T}$$

It is easy to prove that if $\xi \Vdash \Gamma$, Γ is well-formed, $\eta_\Gamma = \eta_\xi$ and for all prefixes $\Gamma', x : T$ of Γ, $T \in \mathrm{Kind}_{\Gamma'}$ implies $\xi(x) \in \mathcal{I}_{\Gamma'}(T)$; the details are left as an exercise for the reader.

Similar to this, we will define a relation between assignments and $(\mathrm{CR})^* \cup \{\mathcal{C}, \mathcal{O}\}$-valuations.

Definition 33. We define $\rho \Vdash_\Gamma \xi$ inductively.

$$\frac{}{\rho \Vdash_\emptyset \xi} \qquad \frac{\rho \Vdash_\Gamma \xi \qquad T \in \mathrm{Kind}_\Gamma \qquad t \in [\![T]\!]_\xi \qquad \Phi \in \mathcal{I}_\xi(T)}{\rho^x_t \Vdash_{\Gamma, x:T} \xi^x_\Phi}$$

$$\frac{\rho \Vdash_\Gamma \xi \qquad T \in \mathrm{Constr}_\Gamma \qquad t \in [\![T]\!]_\xi}{\rho^x_t \Vdash_{\Gamma, x:T} \xi^x_\mathcal{C}}$$

Note that it is not clear yet that this is a valid definition, as we have not proven yet that $[\![T]\!]_\xi$ is actually defined in those cases. We will do so in Theorem 2 and we will not use this definition until then. With the help of Lemma 38 it is then easy to prove that if $\rho \Vdash_\Gamma \xi$ and $\Gamma', x : T$ is a prefix of Γ, then $[\![T]\!]_\xi$ is defined and $\rho(x) \in [\![T]\!]_\xi$.

Both definitions have been taken from Barras [4, Definition 67, p. 22], but have been modified to fit the mathematical context of this proof.

3.6 Candidate Interpretation

The first thing we have to prove is that the interpretation always ends up in the right subset of $(\mathrm{CR})^*$. We will prove this by induction over the term. In order to simplify the proof, we will first state five lemmas, two of which deal with the two extra definitions from above, and two of which help us deal with updated valuations. The remaining lemma deals with with the intersection that we use when we interpret product types. The statements of the following three lemmata come from Geuvers [13, Lemma 3.5 on p. 21 and Lemma 5.4 on p. 29], but he gives no proof.

Lemma 39. *Candidates of reducibility are closed under arbitrary intersections.*

$$\forall I \neq \emptyset, (\forall i \in I. X_i \in \mathrm{CR}) \ \rightarrow \ \bigcap_{i \in I} X_i \in \mathrm{CR}$$

Proof. We need to show that $\bigcap_{i \in I} X_i$ satisfies the three CR properties.

1. Termination: since all the X_i were subsets of SN, so is the intersection.
2. Closure under beta reduction: if a term t is in the intersection, it was in all the X_i. Since all the X_i were closed under beta reduction, β_t is a subset of all the X_i, hence also of the intersection.
3. Closure under beta expansion: let t be a neutral term such that β_t is a subset of the intersection. Hence β_t is a subset of all the X_i. Thus by closure under beta expansion, t was an element of all the X_i. Therefore it must be an element of the intersection as well.

□

Lemma 40. *If X and Y are candidates of reducibility, so is $X \rightrightarrows Y$.*

Proof. We need to show that $X \rightrightarrows Y$ satisfies the three CR properties. Note that all variables are in X.

Termination
Let t be an element of $X \rightrightarrows Y$. Hence for the variable x in X, $t\,x$ is in Y. Y is a subset of SN therefore $t\,x$ terminates. Thus t must also terminate because it is a subterm of $t\,x$.

Closure under beta reduction
Let t be an element of $X \rightrightarrows Y$ and $t' \in \beta_t$. For all u in X, $t\,u$ is in Y. Since Y is closed under beta reduction, $t'\,u$ is also in Y for all u in X. Thus t' is in $X \rightrightarrows Y$.

Closure under beta expansion
Let t be a neutral term such that β_t is a subset of $X \rightrightarrows Y$ and u be some element of X. We must show that $t\,u$ is in Y. Note that because X is a subset of SN, u terminates. By induction on the termination order of

u, we get the hypothesis that $\forall u' \in \beta_u . t\, u' \in Y$. Y is closed under beta expansion and $t\, u$ is neutral, so it suffices to show that $\beta_{t\, u} \subseteq Y$. Let us now do a case analysis on the elements of $\beta_{t\, u}$.

$t\, u \to_\beta t'\, u$ where $t \to_\beta t'$
$t'\, u \in Y$ because $t' \in \beta_t$ and $\beta_t \subseteq X \rightrightarrows Y$ by assumption.

$t\, u \to_\beta t\, u'$ where $u \to_\beta u'$
In this case, $t\, u' \in Y$ follows directly from the inductive hypothesis.

$(\lambda x : T.U)u \to_\beta U^x_u$ where $t = \lambda x : T.U$
This can be ruled out because t is neutral.

□

Lemma 41. *If X and Y are candidates of reducibility, so is $X \times Y$.*

Proof. We need to show that $X \times Y$ satisfies the three CR properties.

Termination
Let t be an element of $X \times Y$. Thus $\pi_1(t)$ in X. X is a subset of SN hence $\pi_1(t)$ terminates. Therefore t must also terminate because it is a subterm of $\pi_1(t)$.

Closure under beta reduction
Let t be an element of $X \times Y$ and $t' \in \beta_t$. We need to show that $\pi_1(t') \in X$ and $\pi_2(t') \in Y$. We will only show the first, the second one is analogous. Since $t \in X \times Y$, we know that $\pi_1(t) \in X$. Since X is closed under beta reduction, and $\pi_1(t') \in \beta_{\pi_1(t)}$, $\pi_1(t')$ is also an element of X.

Closure under beta expansion
Let t be a neutral term such that β_t is a subset of $X \times Y$. We need to show that $\pi_1(t) \in X$ and $\pi_2(t) \in Y$. We will only show the first, the second one is analogous. Because X is closed under beta expansion and $\pi_1(t)$ is neutral, it suffices to show that $\beta_{\pi_1(t)} \subseteq X$. Let us do a case analysis on the elements of $\beta_{\pi_1(t)}$.

$\pi_1(t) \to_\beta \pi_1(t')$ where $t \to_\beta t'$
Therefore $\pi_1(t') \in X$ since $t' \in \beta_t$ and $\beta_t \subseteq X \times Y$ by assumption.

$\pi_1\left((t_1, t_2)_{\Sigma x:T_1.T_2}\right) \to_\beta t_1$ where $t = (t_1, t_2)_{\Sigma x:T_1.T_2}$
This case can be ruled out because t is neutral.

□

Now let us prove that we map to the right element of $(\mathrm{CR})^*$:

Theorem 2. *If $\xi \Vdash \Gamma$ and $\Gamma \vdash T : K$, then both following statements are true:*

$$K = \mathrm{Type}_0 \;\to\; [\![T]\!]_\xi \text{ is defined} \wedge [\![T]\!]_\xi \in \mathrm{CR} \qquad \text{(IHKind)}$$

$$\Gamma \vdash T : K \;:\; \mathrm{Type}_0 \;\to\; [\![T]\!]_\xi \text{ is defined} \wedge [\![T]\!]_\xi \in \mathcal{I}_\Gamma(K) \qquad \text{(IHConstructor)}$$

Proof. Proof by induction on the derivation of $\Gamma \vdash T : K$. Note that since η_Γ preserves Γ by Lemma 32, we get the properties of Theorem 1. We will use them implicitly in this proof in conjunction with Lemmata 28 and 29. Since we have $\xi \Vdash \Gamma$, we can conclude $\eta_\Gamma = \eta_\xi$. We will thus not distinguish between $\mathcal{K}_\Gamma$ and $\mathcal{K}_\xi$, etc.

We will only show the prop, variable, application, product and conversion cases; the other cases are similar or trivial.

prop : Suppose the typing derivation ends with

$$\textit{prop}\quad \frac{\Gamma \text{ is well-formed}}{\Gamma \vdash \text{Prop} : \text{Type}_0}$$

Thus $K = \text{Type}_0$. $[\![\text{Prop}]\!]_\xi = \text{SN}$. Thus $[\![\text{Prop}]\!]_\xi$ is defined and since $\text{SN} \in \text{CR}$, $[\![\text{Prop}]\!]_\xi \in \text{CR}$.

variable : Suppose the typing derivation ends with

$$\textit{variable}\quad \frac{\Gamma \text{ is well-formed} \qquad \Gamma', x : T \subseteq \Gamma \text{ for some } \Gamma'}{\Gamma \vdash x : T}$$

Thus $K = T$. We have to prove that if $\Gamma \vdash T : \text{Type}_0$, $[\![x]\!]_\xi$ is defined and in $\mathcal{I}_\Gamma(T)$[1]. By definition, $[\![x]\!]_\xi = \xi(x)$.

We prove that $\Gamma' \vdash T : \text{Type}_0$. By Lemma 17, we have $\Gamma' \vdash T : s$ for some s. By Lemma 20 and Lemma 24, we can follow $\Gamma \vdash T : s$ and thus $s \approx_\beta \text{Type}_0$. As convertible normal terms are equal, $s = \text{Type}_0$.

From $\xi \Vdash \Gamma$ we can therefore conclude that $\xi(x) \in \mathcal{I}_{\Gamma'}(T)$. We prove that $\mathcal{I}_{\Gamma'}(T) = \mathcal{I}_\Gamma(T)$. By Lemma 2, we can prove instead $\mathcal{I}_=(T)\,\mathcal{I}_\Gamma(\rho_I(T))$. This follows from Lemma 21 if $\rho_I \vDash \Gamma' \to \Gamma$, which can be proven by Lemma 20. The details are left as an exercise to the reader.

application : Suppose the typing derivation ends with

$$\textit{application}\quad \frac{\Gamma \vdash t : (\forall x : T.U) \qquad \Gamma \vdash u : T}{\Gamma \vdash t\,u : U_u^x}$$

Thus $K = U_u^x$. We have to prove that if $\Gamma \vdash U_u^x : \text{Type}_0$, $[\![t\,u]\!]_\xi$ is defined and in $\mathcal{I}_\Gamma(U_u^x)$[2].

By Lemma 23 and $\Gamma \vdash t : \forall x : T.U$, we know $\Gamma \vdash \forall x : T.U : s_2$ for some s_2. Thus by Lemma 18.6 and the fact that convertible normal terms are equal, we get $\Gamma \vdash T : s_1$ and $\Gamma, x : T \vdash U : s_2$ for some s_1. It is easy to prove by Lemma 24 and Lemma 21 that $s_2 = \text{Type}_0$, as $\Gamma \vdash U_u^x : \text{Type}_0$. The details are left as an exercise to the reader.

We make a case distinction on s_1.

[1] It is easy to show that $T \neq \text{Type}_0$ Lemma 28.
[2] It is easy to show that $U_u^x \neq \text{Type}_0$ by Lemma 28.

$s_1 = \mathrm{Type}_0$: Thus $u \in \mathrm{Constr}_\Gamma$ and by Theorem 1, $u \in \mathcal{C}_\Gamma$. Thus by definition, $[\![t\,u]\!]_\xi = [\![t]\!]_\xi([\![u]\!]_\xi)$. Also by definition, $\mathcal{I}_\Gamma(\forall x : T.U) = \mathcal{I}_\Gamma(T) \to \mathcal{I}_{\Gamma,x:T}(U)$. By (IHConstructor) for $\Gamma \vdash u : T$, $[\![u]\!]_\xi$ is defined and in $\mathcal{I}_\Gamma(T)$. By (IHKind) for $\Gamma \vdash t : \forall x : T.U$, $[\![t]\!]_\xi$ is defined and in $\mathcal{I}_\Gamma(T) \to \mathcal{I}_{\Gamma,x:T}(U)$.

Thus $[\![t]\!]_\xi([\![u]\!]_\xi)$ is defined and in $\mathcal{I}_{\Gamma,x:T}(U)$, which by Lemma 36 equals $\mathcal{I}_\Gamma(U^x_u)$.

$s_1 = \mathrm{Prop}$: Thus $u \in \mathrm{Obj}_\Gamma$ and by Theorem 1, $u \in \mathcal{O}_\Gamma$. Thus by definition, $[\![t\,u]\!]_\xi = [\![t]\!]_\xi$. Also by definition, $\mathcal{I}_\Gamma(\forall x : T.U) = \mathcal{I}_{\Gamma,x:T}(U)$. By (IHKind) for $\Gamma \vdash t : \forall x : T.U$, $[\![t]\!]_\xi$ is defined and in $\mathcal{I}_{\Gamma,x:T}(U)$. This by Lemma 36 equals $\mathcal{I}_\Gamma(U^x_u)$.

product : Suppose the typing derivation ends with

$$\textit{product}\quad \frac{\Gamma \vdash T : s_1 \qquad \Gamma, x : T \vdash U : s_2}{\Gamma \vdash (\forall x : T.U) : s_2}$$

Thus $K = s_2$. Note that $\mathcal{I}_\Gamma(\mathrm{Prop}) = \mathrm{CR}$. We thus need to prove $[\![\forall x : T.U]\!]_\xi$ is defined and in CR. We make a case distinction on s_1, but will only show the case where $s_1 = \mathrm{Type}_0$. The other case is similar. We know $T \in \mathcal{K}_\eta$ by Theorem 1. By definition,

$$[\![\forall x : T.U]\!]_\xi = [\![T]\!]_\xi \rightrightarrows \bigcap_{\Phi \in \mathcal{I}_\xi(T)} [\![U]\!]_{\xi^x_\Phi}$$

By Lemma 40 we only need to show that both $[\![T]\!]_\xi$ and $\bigcap_{\Phi \in \mathcal{I}_\xi(T)} [\![U]\!]_{\xi^x_t}$ are defined and in CR.

$[\![T]\!]_\xi \in \mathrm{CR}$: Because $s_1 = \mathrm{Type}_0$, this follows from (IHKind) for $\Gamma \vdash T : s_1$.

$\bigcap_{\Phi \in \mathcal{I}_\xi(T)} [\![U]\!]_{\xi^x_\Phi} \in \mathrm{CR}$: Note that $\mathcal{I}_\xi(T)$ is defined by Lemma 35 and $\Gamma \vdash T : \mathrm{Type}_0$. By Lemma 39 we only have to prove that for every $\Phi \in \mathcal{I}_\xi(T)$, $[\![U]\!]_{\xi^x_\Phi}$ is defined and in CR. Let Φ be an element of $\mathcal{I}_\xi(T)$. We know s_2 is either Type_0 or Prop. We will show the proof when s_2 is Prop, the other case is similar. By (IHConstructor) for $\Gamma, x : T \vdash U : \mathrm{Prop}$, $[\![U]\!]_{\xi^x_\Phi}$ is defined and in CR if $\xi^x_\Phi \Vdash \Gamma, x : T$. This is true because T is in Kind_Γ, $\xi \Vdash \Gamma$ and $\Phi \in \mathcal{I}_\xi(T)$.

conversion : Suppose the typing derivation ends with

$$\textit{conversion}\quad \frac{\Gamma \vdash t : T \qquad \Gamma \vdash U : s \qquad T \approx_\beta U}{\Gamma \vdash t : U}$$

We need to prove $[\![t]\!]_\xi$ is defined and in CR (if $U = \mathrm{Type}_0$) or in $\mathcal{I}_\Gamma(U)$ (if $\Gamma \vdash U : \mathrm{Type}_0$). We only prove the case where $\Gamma \vdash U : \mathrm{Type}_0$, since the other case is trivial.

We first prove that $\Gamma \vdash T : \mathrm{Type}_0$. By Lemma 23, $T = \mathrm{Type}_0$ or $\Gamma \vdash T : s'$ for some s'.

$T = \mathrm{Type}_0$: In this case we get from $T \approx_\beta U$, Lemma 15 and the fact that Type_0 is normal that $U \to_\beta^* T$. Thus by Lemma 25, $\Gamma \vdash T : \mathrm{Type}_0$.

$\Gamma \vdash T : s'$ *for some* s' : By Lemma 26, we have $\mathrm{Type}_0 \approx_\beta s'$. As normal convertible terms are equal, we have $\Gamma \vdash T : \mathrm{Type}_0$.

We use (IHConstructor) for $\Gamma \vdash t : T$ and get $[\![t]\!]_\xi$ is defined and in $\mathcal{I}_\Gamma(T)$. We know that $\mathcal{I}_\Gamma(T)$ equals $\mathcal{I}_\Gamma(U)$ by Lemma 37. Thus $[\![t]\!]_\xi \in \mathcal{I}_\Gamma(U)$.

□

This was a large proof, but now we have the first important result about our interpretation: if $t : T$ and $t \in [\![T]\!]_\xi$, then t terminates. Now we have to prove that if $\Gamma \vdash t : T$, t actually is an element of $[\![T]\!]_\xi$. In order to get there, we will have to prove stability of that interpretation with relation to substitution and conversion as well. The reason that we need stability under substitution is the same as the reason that we needed it for $\mathcal{I}_\Gamma(\cdot)$, just that the same thing will now occur with sigma-types as well. Similarly, the conversion rule will again introduce a typing where $t : U$ and $U \approx_\beta V$. The inductive hypothesis will only work for V, but as interpretations are stable under conversion we can easily get it for U as well.

3.7 Stability Results

We will not prove the following two lemmata since the proofs are extremely long and tedious. Proofs can be found in the formalization.

Lemma 42. *When* $\Gamma \vdash u : U$ *and* $\xi \Vdash \Gamma$, *and* $\Gamma, x : U \vdash T : s$ *for some* s,

$$[\![T^x_u]\!]_\xi = \begin{cases} [\![T]\!]_{\xi^x_{[\![u]\!]_\xi}} & \text{when } U \in \mathrm{Kind}_\Gamma \\ [\![T]\!]_{\xi^x_C} & \text{when } U \in \mathrm{Constr}_\Gamma \end{cases}$$

Note that all interpretations are defined by Theorem 2.

Lemma 43. *For* T *and* T' *such that* $T \approx_\beta T'$, $\Gamma, x : U \vdash T : s$ *and* $\Gamma \vdash T' : s$ *for some* s, *and* $\xi \Vdash \Gamma$, $[\![T]\!]_\xi$ *and* $[\![T']\!]_\xi$ *are defined and equal.*

In addition to these lemmata we will prove a few small lemmata. The basic idea for these lemmata is that we have a complex term (like an application or a projection) of which all subterms terminate, and we want to prove that on all reduction paths some invariant is satisfied. Recall the one-step reduction rules: we grouped them into congruence rules (which handle the reduction of a subterm) and the main rules (which handle beta reduction, etc.). Now since all the subterms terminate, proving the invariant for the congruence rules is trivial. The lemmata state that it suffices to prove the invariant for the main reduction rules. We will show the proof only for the first lemma.

Lemma 44. *Let $l := (\lambda x : T.u)$ where u and T terminates. When m is a terminating term and P has the* CR*-properties (and is thus a set of terminating terms),*

$$u^x_m \in P \implies (l\, m) \in P$$

Proof. By induction on the termination order of T, m, and u. The induction hypotheses look as follows:

$$\begin{aligned} \forall T' \in \beta_T, \forall m \in \mathrm{SN}, \forall u \in \mathrm{SN}, u^x_m \in P &\implies ((\lambda x : T'.u)\, m) \in P \\ \forall m' \in \beta_m, \forall u \in \mathrm{SN}, u^x_m \in P &\implies ((\lambda x : T.u)\, m') \in P \\ \forall u' \in \beta_u, {u'}^x_m \in P &\implies ((\lambda x : T.u')\, m) \in P \end{aligned}$$

As $l\, m$ is neutral and P satisfies the CR-properties, we only need to prove $\beta_{l\, m} \subseteq P$ by closure under expansion.

Now we make a case split over $\beta_{(l\, m)}$. There are four cases, three of which correspond to a congruence rule. We will only show the case for $((\lambda x : T.u')\, m)$ where $u \to_\beta u'$ and for the beta rule.

$(l\, m) \to_\beta ((\lambda x : T.u')\, m)$ *where* $u' \in \beta_u$: We have to show $((\lambda x : T.u')\, m) \in P$ which is exactly the induction hypothesis for $u\!\downarrow$. We still need to show that ${u'}^x_m \in P$. By closure under reduction, we only need to show $u^x_m \to^*_\beta {u'}^x_m$ [3] and $u^x_m \in P$. The first follows directly from Lemma 13, Lemma 11 and Lemma 9. The second we have as an assumption.

$(l\, m) \to_\beta u^x_m$: We have to show $u^x_m \in P$ which we have by assumption.

□

Lemma 45. *Let $p := (t, u)_{\Sigma x:T.U}$ where t, u, T and U terminate. If P has the* CR*-properties,*

$$t \in P \implies \beta_{\pi_1(p)} \subseteq P$$

and

$$u \in P \implies \beta_{\pi_2(p)} \subseteq P$$

3.8 Soundness

We now prove the essential theorem: that interpretation is sound with relation to the typing relation. In other words, $\rho(t) \in [\![T]\!]_\xi$ whenever $\Gamma \vdash t : T$. The main result, namely that all typed terms in CC_Σ are strongly normalizing, is merely a corollary. In the last chapters and sections, we built a lot of heavy machinery to make the proof of this theorem possible.

[3] Note that closure under reduction actually requires a $t \in P$ such that $t \to_\beta {u'}^x_m$. However it can be proven that this is enough by induction on the number of reduction steps.

Theorem 3. *When $\xi \Vdash \Gamma$ and $\rho \Vdash_\Gamma \xi$, then for any t, T and Γ such that $\Gamma \vdash t : T$, we have $\rho(t) \in [\![T]\!]_\xi$*[4].

Proof. Via induction on the typing derivation.

prop : Suppose the typing derivation ends with

$$\textit{prop}\quad \frac{\Gamma \text{ is well-formed}}{\Gamma \vdash \text{Prop} : \text{Type}_0}$$

We need to prove $\rho(\text{Prop}) \in [\![\text{Type}_0]\!]_\xi$, which is equivalent to proving $\text{Prop} \in \text{SN}$. This is trivial.

variable : Suppose the typing derivation ends with

$$\textit{variable}\quad \frac{\Gamma \text{ is well-formed} \qquad \Gamma', x : T \subseteq \Gamma \text{ for some } \Gamma'}{\Gamma \vdash x : T}$$

We need to prove $\rho(x) \in [\![T]\!]_\xi$, where $\Gamma', x : T \subseteq \Gamma$. This result follows from $\rho \Vdash_\Gamma \xi$.

application : Suppose the typing derivation ends with

$$\textit{application}\quad \frac{\Gamma \vdash t : (\forall x : T.U) \qquad \Gamma \vdash u : T}{\Gamma \vdash t\, u : U_u^x}$$

We need to prove $\rho(t\,u) \in [\![U_u^x]\!]_\xi$ where $\Gamma \vdash t : (\forall x : T.U)$ and $\Gamma \vdash u : T$. By definition, $\rho(t\,u) = \rho(t)\,\rho(u)$. By the induction hypothesis for $\Gamma \vdash t : \forall x : T.U$, $\rho(t) \in [\![\forall x : T.U]\!]_\xi$. Now we need to do a case split, since $[\![\forall x : T.U]\!]_\xi$ is defined depending on the class of T under ξ. By Lemma 23 and Lemma 18.6, we get s_1 and s_2 such that $\Gamma \vdash T : s_1$ and $\Gamma, x : T \vdash U : s_2$. By case analysis on s_1 and using Theorem 1, we have either $s_1 = \text{Type}_0$ and $T \in \mathcal{K}_\xi$, or $s_1 = \text{Prop}$ and $T \in \mathcal{C}_\xi$. We will only treat the case where $T \in \mathcal{K}_\xi$, the other case is much easier.

We have:

$$\begin{aligned}[\![\forall x : T.U]\!]_\xi &= [\![T]\!]_\xi \rightrightarrows \bigcap_{\Phi \in \mathcal{I}_\xi(T)} [\![U]\!]_{\xi_\Phi^x} \\ &= \{t' \in \Lambda \mid \forall u' \in [\![T]\!]_\xi . t'\,u' \in \bigcap_{\Phi \in \mathcal{I}_\xi(T)} [\![U]\!]_{\xi_\Phi^x}\}\end{aligned}$$

So for $u' \in [\![T]\!]_\xi$,

$$\rho(t)\,u' \in \bigcap_{\Phi \in \mathcal{I}_\xi(T)} [\![U]\!]_{\xi_\Phi^x}$$

[4] Note that $[\![T]\!]_\xi$ is defined by Lemma 23 and Theorem 2.

By the induction hypothesis for $\Gamma \vdash u : T$, we have $\rho(u) \in [\![T]\!]_\xi$. Hence,

$$\rho(t)\,\rho(u) \in \bigcap_{\Phi \in \mathcal{I}_\xi(T)} [\![U]\!]_{\xi^x_\Phi}$$

So in order to prove $\rho(t)\,\rho(u) \in [\![U^x_u]\!]_\xi$, it suffices to prove:

$$\bigcap_{\Phi \in \mathcal{I}_\xi(T)} [\![U]\!]_{\xi^x_\Phi} \subseteq [\![U^x_u]\!]_\xi$$

We argue that $[\![U^x_u]\!]_\xi = [\![U]\!]_{\xi^x_{[\![u]\!]_\xi}}$ by Lemma 42, $\Gamma \vdash u : T$ and $\Gamma \vdash T : \mathrm{Type}_0$. We only need to prove that $[\![U]\!]_{\xi^x_{[\![u]\!]_\xi}}$ is an instance of $[\![U]\!]_{\xi^x_\Phi}$ with $\Phi \in \mathcal{I}_\xi(T)$: by Theorem 2, $\xi \Vdash \Gamma$, $\Gamma \vdash u : T$, and $\Gamma \vdash T : \mathrm{Type}_0$, we know that $[\![u]\!]_\xi \in \mathcal{I}_\xi(T)$.

lambda : Suppose the typing derivation ends with

$$\textit{lambda}\quad \frac{\Gamma \vdash T : s_1 \qquad \Gamma, x : T \vdash u : U \qquad \Gamma, x : T \vdash U : s_2}{\Gamma \vdash (\lambda x : T.u) : (\forall x : T.U)}$$

We have to show that $\rho(\lambda x : T.u) \in [\![\forall x : T.U]\!]_\xi$. By definition, this is equivalent to showing $\lambda z : \rho(T).\rho^x_z(u) \in [\![\forall x : T.U]\!]_\xi$. Now we need to do a case split since the definition of $[\![\forall x : T.U]\!]_\xi$ depends on the class of T under ξ. Analogous to the application case, we get two cases: one where $s_1 = \mathrm{Type}_0$ and $T \in \mathcal{K}_\xi$, and one where $s_2 = \mathrm{Prop}$ and $T \in \mathcal{C}_\xi$. Again, we only show the case where $s_1 = \mathrm{Type}_0$ and $T \in \mathcal{K}_\xi$.

We thus need to show:

$$\lambda z : \rho(T).\rho^x_z(u) \in \{t' \in \Lambda \mid \forall u' \in [\![T]\!]_\xi, t'\,u' \in \bigcap_{\Phi \in \mathcal{I}_\xi(T)} [\![U]\!]_{\xi^x_\Phi}\}$$

So we need to show that for all $u' \in [\![T]\!]_\xi$ and $\Phi \in \mathcal{I}_\xi(T)$,

$$(\lambda z : \rho(T).\rho^x_z(u))\,u' \in [\![U]\!]_{\xi^x_\Phi}$$

By Theorem 2, we know $[\![T]\!]_\xi$ and $[\![U]\!]_{\xi^x_\Phi}$ are in CR. By closure under expansion, we only need to show:

$$\beta_{(\lambda z:\rho(T).\rho^x_z(u))\,u'} \subseteq [\![U]\!]_{\xi^x_\Phi}$$

By Lemma 44, it suffices to show that $(\rho^x_z(u))^z_{u'} \in [\![U]\!]_{\xi^x_\Phi}$. By Lemma 6, this is equivalent to proving $\rho^x_{u'}(u) \in [\![U]\!]_{\xi^x_\Phi}$. This is the induction hypothesis for $\Gamma, x : T \vdash u : U$ if we can prove that $\rho^x_{u'} \Vdash_{\Gamma, x:T} \xi^x_\Phi$ and $\xi^x_\Phi \Vdash \Gamma, x : T$. We need to prove $u' \in [\![T]\!]_\xi$ and $\Phi \in \mathcal{I}_\xi(T)$, which we both have by assumption.

product : Suppose the typing derivation ends with

$$\textit{product}\quad \frac{\Gamma \vdash T : s_1 \qquad \Gamma, x : T \vdash U : s_2}{\Gamma \vdash (\forall x : T.U) : s_2}$$

We have to show that $\rho(\forall x : T.U) \in [\![s_2]\!]_\xi$. By definition, this means we have to show $\forall z : \rho(T).\rho_z^x(U) \in \mathrm{SN}$. It suffices to show that $\rho(T)$ and $\rho_z^x(U)$ are in SN. For this we use the induction hypothesis for $\Gamma \vdash T : s_1$ and $\Gamma, x : T \vdash U : s_2$. We only need to show that $\rho_z^x \Vdash_{\Gamma,x:T} \xi^x_{[\![T]\!]_\xi}$ and $\xi^x_{[\![T]\!]_\xi} \Vdash \Gamma, x : T$. We make a case distinction on s_1, but only treat the case where $s_1 = \mathrm{Type}_0$ and thus $T \in \mathrm{Kind}_\Gamma$. We will prove that $[\![T]\!]_\xi$ is defined and in CR, which follows from Theorem 2, thus $\xi^x_{[\![T]\!]_\xi} \Vdash \Gamma, x : T$ by definition. Since every set in CR contains at least all variables, $z \in [\![T]\!]_\xi$, and thus $\rho_z^x \Vdash_{\Gamma,x:T} \xi^x_{[\![T]\!]_\xi}$ by definition.

sigma : Suppose the typing derivation ends with

$$\textit{sigma}\quad \frac{\Gamma \vdash T : \mathrm{Prop} \qquad \Gamma, x : T \vdash U : \mathrm{Prop}}{\Gamma \vdash (\Sigma x : T.U) : \mathrm{Prop}}$$

We need to show that $\rho(\Sigma x : T.U) \in [\![\mathrm{Prop}]\!]_\xi$. By definition this is equivalent to $\Sigma z : \rho(T).\rho_z^x(U) \in \mathrm{SN}$ For this it suffices to show that $\rho(T)$ and $\rho_z^x(U)$ are in SN. Both follow from the induction hypothesis for $\Gamma \vdash T : \mathrm{Prop}$ and $\Gamma, x : T \vdash U : \mathrm{Prop}$, respectively. It remains to be shown that $\rho_z^x \Vdash_{\Gamma,x:T} \xi^x_{\mathcal{C}}$ and $\xi^x_{\mathcal{C}} \Vdash \Gamma, x : T$. Both follow by definition and $T \in \mathrm{Constr}_\Gamma$. Note that since $[\![T]\!]_\xi \in \mathrm{CR}$ by Theorem 2, all variables are in $[\![T]\!]_\xi$.

pair : Suppose the typing derivation ends with

$$\textit{pair}\quad \frac{\Gamma \vdash t : T \qquad \Gamma \vdash u : U_t^x \qquad \Gamma \vdash T : \mathrm{Prop} \qquad \Gamma, x : T \vdash U : \mathrm{Prop}}{\Gamma \vdash (t, u)_{\Sigma x:T.U} : (\Sigma x : T.U)}$$

We need to show:

$$\rho((t, u)_{\Sigma x:T.U}) \in [\![\Sigma x : T.U]\!]_\xi$$

By definition, this is equivalent to showing:

$$(\rho(t), \rho(u))_{\Sigma z:\rho(T).\rho_z^x U} \in [\![T]\!]_\xi \times [\![U]\!]_{\xi^x_{\mathcal{C}}}$$

For the sake of brevity, let $m := (\rho(t), \rho(u))_{\Sigma z:\rho(T).\rho_z^x U}$. If we unwrap the definition of $[\![T]\!]_\xi \times [\![U]\!]_{\xi^x_{\mathcal{C}}}$, we see we need to show that $\pi_1(m) \in [\![T]\!]_\xi$ and $\pi_2(m) \in [\![U]\!]_{\xi^x_{\mathcal{C}}}$. We only show the second, the first case is easier.

By Theorem 2, we know $[\![T]\!]_\xi$ and $[\![U]\!]_{\xi^x_{\mathcal{C}}}$ are in CR. By closure under expansion, we only need to show $\beta_{\pi_2(m)} \subseteq [\![U]\!]_{\xi^x_{\mathcal{C}}}$.

By Lemma 45, it suffices to show $\rho(u) \in [\![U]\!]_{\xi^x_{\mathcal{C}}}$. Our induction hypothesis for $\Gamma \vdash u : U_t^x$ gives us $\rho(u) \in [\![U_t^x]\!]_\xi$. Furthermore, $[\![U_t^x]\!]_\xi = [\![U]\!]_{\xi^x_{\mathcal{C}}}$ by Lemma 42 since $\Gamma \vdash t : T$ and $T \in \mathcal{C}_\Gamma$.

proj1 and proj2 : These are extremely similar. We will only show the case of proj2, since the other case is easier.

Suppose the typing derivation ends with

$$\textit{proj2}\quad \frac{\Gamma \vdash t : (\Sigma x : T.U)}{\Gamma \vdash \pi_2 t : U^x_{\pi_1(t)}}$$

We need to show $\rho(\pi_2(t)) \in [\![U^x_{\pi_1(t)}]\!]_\xi$ where $\Gamma \vdash t : \Sigma x : T.U$. By the induction hypothesis, we have $\rho(t) \in [\![\Sigma x : T.U]\!]_\xi$. This allows us to conclude $\rho(\pi_2(t)) \in [\![U]\!]_{\xi^x_{\mathcal{C}}}$. By Lemma 42, $[\![U]\!]_{\xi^x_{\mathcal{C}}} = [\![U^x_{\pi_1(t)}]\!]_\xi$ if $\Gamma \vdash \pi_1(t) : T$ and $T \in \mathcal{C}_\Gamma$. The first follows from the proj1 rule. The second is proven by Lemma 18.7, which yields $\Gamma \vdash T : \mathrm{Prop}$.

conversion : Suppose the typing derivation ends with

$$\textit{conversion}\quad \frac{\Gamma \vdash t : T \qquad \Gamma \vdash U : s \qquad T \approx_\beta U}{\Gamma \vdash t : U}$$

We need to prove $\rho(t) \in \mathcal{I}_\xi(U)$, where $\Gamma \vdash t : T$, $\Gamma \vdash U : s$, and $T \approx_\beta U$. Our induction hypothesis for $\Gamma \vdash t : T$ gives us that $\rho(t) \in \mathcal{I}_\xi(U)$. By Lemma 43 and $T \approx_\beta U$, $\mathcal{I}_\xi(T) = \mathcal{I}_\xi(U)$ if we can prove that $\Gamma \vdash T : s$. This, however, follows directly from Lemma 27.

□

We will now prove that for well-formed Γ we can actually find a ξ and ρ such that $\xi \Vdash \Gamma$ and $\rho \Vdash_\Gamma \xi$.

Definition 34. We define default elements $\mathrm{D}_s \in \mathrm{can}(s)$.

$$\mathrm{D}_\bigstar := \mathrm{SN}$$
$$\mathrm{D}_{s_1 \to_S s_2} := (\_ \in \mathrm{can}(s_1) \mapsto \mathrm{D}_{s_2})$$

Definition 35. We define ξ_Γ by recursion on Γ.

$$\xi_\emptyset := (x \mapsto \mathrm{SN})$$
$$\xi_{\Gamma,x:T} := \begin{cases} \xi_\Gamma{}^x_{\mathrm{D}_s} & \text{when } T \in \mathrm{Kind}_\Gamma, \text{ and where } s = \mathrm{skeleton}_\Gamma(T) \\ \xi_\Gamma{}^x_{\mathcal{C}} & \text{otherwise} \end{cases}$$

Now we prove that $\xi_\Gamma \Vdash \Gamma$ and $\rho_I \Vdash_\Gamma \xi_\Gamma$.

Lemma 46. *For all well-formed Γ, $\xi_\Gamma \Vdash \Gamma$.*

Proof. We prove this by induction on Γ.

$\emptyset$: This is true by definition.

$\Gamma, x : T$: As $\Gamma, x : T$ is well-formed, we have $\Gamma \vdash T : s$ for some s. This leaves us in one of two cases: either $s = \text{Type}_0$ or $s = \text{Prop}$.

$s = \text{Type}_0$: In this case, $\xi_{\Gamma,x:T} = \xi_\Gamma{}^x_{\mathrm{D}_{s'}}$ where $s' = \text{skeleton}_\Gamma(T)$. We need to prove $\xi_\Gamma{}^x_{\mathrm{D}_{s'}} \Vdash \Gamma, x : T$. As $\xi_\Gamma \Vdash \Gamma$ by the induction hypothesis and $\Gamma \vdash T : \text{Type}_0$, it suffices to prove that $\mathrm{D}_{s'} \in \mathcal{I}_\Gamma(T)$. This is true by definition.

$s = \text{Prop}$: In this case, $\xi_{\Gamma,x:T} = \xi_\Gamma{}^x_{\mathcal{C}}$. We need to prove $\xi_\Gamma{}^x_{\mathcal{C}} \Vdash \Gamma, x : T$. As $\xi_\Gamma \Vdash \Gamma$ by the induction hypothesis and $\Gamma \vdash T : \text{Prop}$, this is true by definition.

□

Lemma 47. *For all well-formed* Γ, $\rho_I \Vdash_\Gamma \xi_\Gamma$.

Proof. We prove this by induction on Γ.

$\emptyset$: This is true by definition.

$\Gamma, x : T$: As $\Gamma, x : T$ is well-formed, we have $\Gamma \vdash T : s$ for some s. This leaves us in one of two cases: either $s = \text{Type}_0$ or $s = \text{Prop}$.

$s = \text{Type}_0$: In this case, $\xi_{\Gamma,x:T} = \xi_\Gamma{}^x_{\mathrm{D}_{s'}}$ where $s' = \text{skeleton}_\Gamma(T)$.
We need to prove $\rho_I \Vdash_{\Gamma,x:T} \xi_\Gamma{}^x_{\mathrm{D}_{s'}}$. We will prove instead that $\rho_I{}^x_x \Vdash_{\Gamma,x:T} \xi_\Gamma{}^x_{\mathrm{D}_{s'}}$. As $\rho_I \Vdash_\Gamma \xi_\Gamma$ by the induction hypothesis and $\Gamma \vdash T : \text{Type}_0$, it suffices to prove that $x \in [\![T]\!]_\xi$ and $\mathrm{D}_{s'} \in \mathcal{I}_\Gamma(T)$. By Theorem 2, we know $[\![T]\!]_\xi$ is in CR. Thus at least all variables are in $[\![T]\!]_\xi$; and $\mathrm{D}_{s'} \in \mathcal{I}_\Gamma(T)$ is true by definition.

$s = \text{Prop}$: In this case, $\xi_{\Gamma,x:T} = \xi_\Gamma{}^x_{\mathcal{C}}$. We need to prove $\rho_I \Vdash_{\Gamma,x:T} \xi_\Gamma{}^x_{\mathcal{C}}$. We will prove instead that $\rho_I{}^x_x \Vdash_{\Gamma,x:T} \xi_\Gamma{}^x_{\mathcal{C}}$. As $\rho_I \Vdash_\Gamma \xi_\Gamma$ by the induction hypothesis and $\Gamma \vdash T : \text{Prop}$, it suffices to prove that $x \in [\![T]\!]_\xi$. The proof goes analogous to the case where $s = \text{Type}_0$.

□

3.9 Termination

Now we prove the main result, i.e., that all typed terms in CC_Σ are strongly normalizing.

Corollary 1. *For any* Γ, t *and* T *such that* $\Gamma \vdash t : T$, $t{\downarrow}$.

Proof. We prove that $\rho_I(t) \in [\![T]\!]_{\xi_\Gamma}$. This follows from Theorem 3 if we can prove $\xi_\Gamma \Vdash \Gamma$ and $\rho_I \Vdash_\Gamma \xi_\Gamma$, which follow directly from Lemma 46 and Lemma 47.

By Lemma 2, we now have $t \in [\![T]\!]_\xi$. We want to prove $[\![T]\!]_\xi \subseteq \text{SN}$. By Lemma 23, we have two cases: $T = \text{Type}_0$ or $\Gamma \vdash T : s$ for some s.

$T = \mathrm{Type}_0$: In this case $[\![T]\!]_\xi = \mathrm{SN}$ by definition.

$\Gamma \vdash T : s$ *for some* s : By Theorem 2, we know $[\![T]\!]_\xi \in \mathrm{CR}$. By termination, $[\![T]\!]_\xi \subseteq \mathrm{SN}$.

Now by $t \in [\![T]\!]_\xi$ and $[\![T]\!]_\xi \subseteq \mathrm{SN}$, we have $t \in \mathrm{SN}$. Consequently $t{\downarrow}$. □

4. FORMALIZATION

We started from Barras' formalization of CC [4]. We extended the set of terms, the reduction rules and the typing rules to CC_Σ. In order to deal with these changes, we had to prove additional cases in existing lemmas, add some cases in existing definitions, and even add a couple of new definitions and lemmas. In most of the following code, we will only consider the extra cases we added in existing definitions as well as the new definitions.

4.1 Preliminaries

The file `termes.v` covers the most part of Chapter 2. The set of terms Λ, the substitution of a single free variable u_t^x and one-step-reduction are introduced.

`Inductive sort` introduces the universes `set` (Type_0) and `prop` (Prop). `Inductive term` introduces the set of all lambda terms. We use de Bruijn indices for variables [12]. Recall that in the mathematical proofs, we considered terms to be equal up to the name of bound variables. In the formalization, we get the same behavior for free, since the names of variables disappear altogether.

```
Inductive sort : Set :=
  | prop : sort
  | set : sort.

Inductive term : Set :=
  .
  .
  .
  | Sigma : term -> term -> term
  | Pair : term -> term -> term -> term -> term
  | Proj1 : term -> term
  | Proj2 : term -> term.
```

Since we use de Bruijn indices, we need `lift_rec n t x`, which increases all variables with index at least `x` in the term `t` by `n`. Instead of defining parallel substitution first and using it to define the substitution of a single variable, we define `subst_rec t u x` to be the substitution of `x` in `u` by `t`. Recall that in the lambda reduction rule, a binder gets eliminated and the variable gets substituted. Since we are dealing with de Bruijn indices, we will have to compensate the elimination of the binder by decreasing the index of

all the variables greater than `x` in `u` by one. We define `lift` and `subst` to be short-hand for lifting where we start with `0` and substitution of `0`, respectively.

```
Fixpoint lift_rec n t {struct t} : nat -> term :=
  fun k =>
  match t with
  .
  .
  .
  | Sigma A B => Sigma (lift_rec n A k) (lift_rec n B (S k))
  | Pair A B u v =>
    Pair (lift_rec n A k) (lift_rec n B (S k))
      (lift_rec n u k) (lift_rec n v k)
  | Proj1 u => Proj1 (lift_rec n u k)
  | Proj2 u => Proj2 (lift_rec n u k)
  end.

Definition lift n t := lift_rec n t 0.


Fixpoint subst_rec N M {struct M} : nat -> term :=
  fun k =>
  match M with
  .
  .
  .
  | Sigma A B => Sigma (subst_rec N A k) (subst_rec N B (S k))
  | Pair A B u v =>
    Pair (subst_rec N A k) (subst_rec N B (S k))
      (subst_rec N u k) (subst_rec N v k)
  | Proj1 u => Proj1 (subst_rec N u k)
  | Proj2 u => Proj2 (subst_rec N u k)
  end.

Definition subst N M := subst_rec N M 0.
```

Now the various reduction relations are defined. `red1` is one-step-reduction, while `red` and `conv` are the reflexive transitive and the reflexive transitive symmetric closures of `red1`. `par_red1` is parallel reduction and `par_red` is its reflexive transitive closure. Since we define `red`, `conv` and `par_red` manually, we will have to prove that they are actually the relations we intend them to be (e.g that `red` actually is the reflexive transitive closure of `red1`). Note that we do not show the extension of the congruence rules.

```
Inductive red1 : term -> term -> Prop :=
  .
  .
```

```
  .
  | proj1 : forall A B M N, red1 (Proj1 (Pair A B M N)) M
  | proj2 : forall A B M N, red1 (Proj2 (Pair A B M N)) N
  .
  .
  .

Inductive red M : term -> Prop :=
  | refl_red : red M M
  | trans_red : forall (P : term) N, red M P -> red1 P N -> red M N.

Inductive conv M : term -> Prop :=
  | refl_conv : conv M M
  | trans_conv_red : forall (P : term) N,
      conv M P -> red1 P N -> conv M N
  | trans_conv_exp : forall (P : term) N,
      conv M P -> red1 N P -> conv M N.

Inductive par_red1 : term -> term -> Prop :=
  .
  .
  .

  | par_proj1 :
      forall A B M M' N, par_red1 M M' ->
        par_red1 (Proj1 (Pair A B M N)) M'
  | par_proj2 :
      forall A B M N N',
      par_red1 N N' -> par_red1 (Proj2 (Pair A B M N)) N'
  .
  .
  .

Definition par_red := clos_trans term par_red1.
```

With `normal` we formalize what it means for a term to be normal and `sn` is the set of all terminating terms. Note that we use `Acc` and `transp` which come from the `Relations` package. The reason we have to transpose the reduction relation lies in the definition of `Acc`, which for our purposes is defined "backwards".

```
Inductive Acc (A : Type) (R : A -> A -> Prop) (x : A) : Prop :=
  Acc_intro : (forall y : A, R y x -> Acc R y) -> Acc R x

Definition normal t : Prop := forall u, ~ red1 t u.

Definition sn : term -> Prop := Acc (transp _ red1).
```

Now we prove a series of laws about lifting and substitution. Since we use de Bruijn indices in the formalization, only one of these lemmas has a counterpart

in the mathematical proof: `distr_subst_rec` corresponds to Lemma 5. Note that the mathematical version is more general in that it allows the second substitution to substitute multiple variables at once, while here we only allow the substitution of a single variable.

```
Lemma distr_subst_rec :
  forall M N (P : term) n p,
  subst_rec P (subst_rec N M p) (p + n) =
  subst_rec (subst_rec P N n) (subst_rec P M (S (p + n))) p.
```

We prove that `par_red1` is reflexive and that it lies between `red1` and its reflexive transitive closure in the lemmas `refl_par_red1`, `red1_par_red1`, `red_par_red` and `par_red_red`. Recall that we did a similar thing in the mathematical proof in order to apply Lemma 7 and Lemma 8. Here we prove directly that the closures are equivalent.

```
Lemma refl_par_red1 : forall M, par_red1 M M.

Lemma red1_par_red1 : forall M N, red1 M N -> par_red1 M N.

Lemma red_par_red : forall M N, red M N -> par_red M N.

Lemma par_red_red : forall M N, par_red M N -> red M N.
```

We formalize Lemma 11 in `par_red1_subst`.

```
Lemma par_red1_subst :
  forall c d : term,
  par_red1 c d ->
  forall a b : term,
  par_red1 a b -> forall k, par_red1 (subst_rec a c k) (subst_rec b d k).
```

4.2 Confluence

The file `conv.v` has the proof of confluence. As before, we first prove that `par_red1` has the diamond property; then we infer the same for `red`. Then we prove a couple of properties about conversion which we also use in the mathematical proof, but we do not prove separate lemmas for these.

The first definition to consider is the diamond property, `str_confluent`. Note that it is specialized to work with relations between terms and is hence called confluence, while in the mathematical proof we were using the more general notion of $\diamondsuit$. The diamond property is formalized as the commutativity of a term relation with its own inverse. It is trivial to check that this is, in fact, the diamond property.

```
Definition commut (A : Type) (R1 R2 : relation A) :=
  forall x y : A,
  R1 y x -> forall z : A, R2 z y -> exists2 y' : A, R2 y' x & R1 z y'.

Definition str_confluent (R : term -> term -> Prop) :=
  commut _ R (transp _ R).
```

In `str_confluence_par_red`, we prove that $\Rightarrow_\beta$ is confluent. Then we prove $\Rightarrow_\beta^*$ is confluent in two steps, analogously to the mathematical proof: in `strip_lemma` we prove that we can fill a single row of the diamond, in `confluence_par_red` we prove that this suffices to fill the diamond row by row.

```
Lemma str_confluence_par_red1 : str_confluent par_red1.

Lemma strip_lemma : commut _ par_red (transp _ par_red1).

Lemma confluence_par_red : str_confluent par_red.
```

As before, it directly follows that $\rightarrow_\beta^*$ is confluent, which we show in `confluence_red`. We can also infer the Church-Rosser property, which we formalize in the theorem `church_rosser`.

```
Lemma confluence_red : str_confluent red.

Theorem church_rosser :
  forall u v, conv u v -> ex2 (fun t => red u t) (fun t => red v t).
```

What now follows is a series of helpful lemmas, most of which appear implicitly in the mathematical proof but are made explicit in the formalization. Examples are `nf_uniqueness`, which proves two convertible normal terms are equal, and its corollary `conv_sort` which says convertible universes are equal. This proves `conv_set_prop`, which says `set` and `prop` are not convertible.

```
Lemma nf_uniqueness : forall u v,
  conv u v ->
  normal u ->
  normal v ->
    u = v.

Lemma conv_sort : forall s1 s2, conv (Srt s1) (Srt s2) -> s1 = s2.

Lemma conv_set_prop : ~ conv (Srt set) (Srt prop).
```

4.3 Assignments

In the file `int_term.v` we introduce parallel substitution and some of its properties. We formalize assignments as functions `intt` which map indices to terms. The identity assignment is the one that maps `x` to `Ref x`.

```
Definition intt := nat -> term.
```

The application of an assignment to a term is defined by `int_term`. Note that `int_term` takes an additional parameter `k` which tells us which variables not to change and where to look up all the other variables. Whenever a binder is introduced, we compensate by increasing `k` by one.

```
Fixpoint int_term (t : term) : intt -> nat -> term :=
  fun (I : intt) (k : nat) =>
  match t with
  .
  .
  .
  | Sigma A B => Sigma (int_term A I k) (int_term B I (S k))
  | Pair A B u v =>
    Pair (int_term A I k) (int_term B I (S k))
      (int_term u I k) (int_term v I k)
  | Proj1 M => Proj1 (int_term M I k)
  | Proj2 M => Proj2 (int_term M I k)
  end.
```

With `shift_intt`, we can compensate for a binder by assigning the new variable a value. Using this, we prove `int_term_subst` which closely corresponds to Lemma 6: when we substitute a single variable after doing a parallel substitution, we can use a single, larger substitution. Note that in the formalization, the preconditions of Lemma 6 are automatically met by the magic of de Bruijn indices.

```
Definition shift_intt (i : intt) (t : term) : intt :=
  fun n : nat => match n with
                 | O => t
                 | S k => i k
                 end.

Lemma int_term_subst :
  forall (t : term) (it : intt) (k : nat) (x : term),
    subst_rec x (int_term t it (S k)) k = int_term t (shift_intt it x) k.
```

4.4 Typing Rules

The file `types.v` provides the typing rules and many of the properties of typing, e.g., subject reduction and weakening.

We begin with the definition of environments `env`, which are lists of terms. Recall that in the mathematical proof, we used lists of pairs $(x : T)$, where x is a variable and T is a term. Since we use de Bruijn Indices in the formalization, this would correspond to pairs `(x,T)`, where `x` is the index of a variable and `T` is a term. However, whenever a binding is introduced (e.g., in the lambda

case), the indices of all free variables in gamma would need to be increased by 1 (including the free variables in the terms). The list would hence have the form (using some properties of lift):

```
gamma = [(0,T_0), (1, lift 1 T_1), (2, lift 2 T_2), ...]
```

In general, the k-th element of gamma is (k, lift k T_k). This allows us to derive two conclusions: first, the k in the pair is redundant information, so we can just leave it out, and second, we can defer the lift k until we extract the element. In other words, our gamma will be [T_0, T_1, T_2, ...], and when we want to get the type of the variable with index x, we take T_x, which is the x-th element of gamma, and then lift all indices of free variables in T_x by x.

In order to formalize this, we define the predicate item_lift T gamma x, which is true if and only if T = lift x T_x, where T_x is the x-th element of gamma.

```
Definition env := list term.

Definition item_lift t e n :=
  ex2 (fun u => t = lift (S n) u)
      (fun u => item term u (e:list term) n).
```

As in the mathematical version, we define wf gamma and typ gamma t T by mutual induction. Note the variable rule, which uses item_lift T gamma x, for the reasons that we discussed above.

```
Inductive wf : env -> Prop :=
  | wf_nil : wf nil
  | wf_var : forall e T s, typ e T (Srt s) -> wf (T :: e)
with typ : env -> term -> term -> Prop :=
  .
  .
  .
  | type_var :
      forall e,
      wf e -> forall (v : nat) t, item_lift t e v -> typ e (Ref v) t
  .
  .
  .
  | type_sigma :
      forall e A B,
      typ e A (Srt prop) ->
      typ (A::e) B (Srt prop) ->
        typ e (Sigma A B) (Srt prop)
  | type_pair :
      forall e A B M N,
      typ e A (Srt prop) ->
      typ (A::e) B (Srt prop) ->
      typ e M A ->
```

```
      typ e N (subst M B) ->
        typ e (Pair A B M N) (Sigma A B)
  | type_proj1 :
      forall e M A B,
      typ e M (Sigma A B) -> typ e (Proj1 M) A
  | type_proj2 :
      forall e M A B,
      typ e M (Sigma A B) -> typ e (Proj2 M) (subst (Proj1 M) B).
```

The following two lemmata correspond directly to Lemma 19 and Lemma 16, respectively.

```
Lemma typ_free_db : forall e t T, typ e t T -> free_db (length e) t.

Lemma typ_wf : forall e t T, typ e t T -> wf e.
```

The next theorem formalizes that if $\Gamma', x : T \subseteq \Gamma$ and Γ is well-formed, $\Gamma' \vdash T : s$ for some s. Note that the usage of `trunc _ (S x) gamma gamma'` and `item _ T gamma x` in conjunction, which is used to formalize $\Gamma', x : T \subseteq \Gamma$.

```
Lemma wf_sort :
  forall n e f,
  trunc _ (S n) e f ->
    wf e -> forall t, item _ t e n -> exists s : sort, typ f t (Srt s).
```

Now we prove the inversion lemma, Lemma 18.

We first formalize the inversion property, `inv_type P gamma t T`. Note that the property is defined negatively ; the best way to understand how it works is to look at `typ_inversion P gamma t T`, which formalizes the inversion lemma. It says that in order to prove `P`, it suffices to prove `inv_type P gamma t T`.

The lemma `inv_type_conv` corresponds to the proof step in the proof sketch of Lemma 18.

```
Definition inv_type (P : Prop) e t T : Prop :=
  match t with
  .
  .
  .
  | Sigma A B =>
    typ e A (Srt prop) ->
    typ (A::e) B (Srt prop) ->
    conv T (Srt prop)
      -> P
  | Pair A B M N =>
    typ e A (Srt prop) ->
    typ (A::e) B (Srt prop) ->
    typ e M A ->
    typ e N (subst M B) ->
    conv T (Sigma A B)
      -> P
```

```
  | Proj1 M =>
    forall A B,
    typ e M (Sigma A B) -> conv T A -> P
  | Proj2 M =>
    forall A B,
    typ e M (Sigma A B) -> conv T (subst (Proj1 M) B) -> P
  end.

Lemma inv_type_conv :
  forall (P : Prop) e t (U V : term),
    conv U V -> inv_type P e t U -> inv_type P e t V.

Theorem typ_inversion :
  forall (P : Prop) e t T, typ e t T -> inv_type P e t T -> P.
```

The formalization of weakening (Lemma 20) is called `thinning`. Note that since the environment was extended, we need to increase the indices of all free variables in `t` and `T`.

```
Theorem thinning :
  forall e t T,
  typ e t T -> forall A, wf (A :: e) ->
    typ (A :: e) (lift 1 t) (lift 1 T).
```

Next we have the formalizations of Lemma 21, Lemma 24, Lemma 23, Lemma 25 and Lemma 26. Note that `substitution` uses single variable substitution instead of parallel substitution, and that `type_case` is formulated differently. Instead of $T \neq \mathrm{Type}_0 \implies \exists s, \Gamma \vdash T : s$, it is formulated as $T = \mathrm{Type}_0 \vee \exists s, \Gamma \vdash T : s$. Since term equality with `(Srt set)` is decidable, both versions are equivalent even in the formalization.

```
Theorem substitution :
  forall e t u (U : term),
  typ (t :: e) u U ->
    forall d : term, typ e d t -> typ e (subst d u) (subst d U).

Theorem typ_unique :
  forall e t T, typ e t T -> forall U : term, typ e t U -> conv T U.

Theorem type_case :
  forall e t T,
    typ e t T -> (exists s : sort, typ e T (Srt s)) \/ T = Srt set.

Theorem subject_reduction :
  forall e t u, red t u -> forall T, typ e t T -> typ e u T.

Lemma typ_conv_conv :
  forall e u (U : term) v (V : term),
    typ e u U -> typ e v V -> conv u v -> conv U V.
```

4.5 Classification

Skeletons, classes, class valuations, the class and skeleton of a term, and several stability results are defined and proven in `class.v`. The class-level correspondence is also proven. Note however that the proof uses two steps: at first, only the class-level correspondence and loose stability results are proven. Those results are then used to prove the strict results. According to Barras, the dependencies go as follows (Barras, personal communication, November 11, 2012): the correspondence between Kind_Γ and $\mathcal{K}_\eta$ is easy to prove. This can be used to prove the correctness of skeletons (Lemma 37, etc.). Those results can be used to prove the correspondence between Constr_Γ and $\mathcal{C}_\eta$. Which finally yields the correctness of skeletons on the constructor level. Note that this last step is not included in the mathematical proof since we defined skeletons only for kinds, not for constructors. Thanks to the introduction of preservation, those dependencies disappear; the class-level correspondence can be proven in a single step, omitting skeletons altogether.

The first definitions in the file are `skel`, which corresponds directly to our skeletons, and `class`, which corresponds to a pair of a skeleton and a class: `Trm` corresponds to $\mathcal{O}$, `Typ` corresponds to $\mathcal{C}$ and a skeleton, and `Knd` corresponds to $\mathcal{K}$ and a skeleton.

```
Inductive skel : Type :=
  | PROP : skel
  | PROD : skel -> skel -> skel.


Inductive class : Type :=
  | Trm : class
  | Typ : skel -> class
  | Knd : skel -> class.
```

The definition of `cls` corresponds to our class-valuation η, but with a few important differences. Note that in the formalization, `cls` is not a function, but a list. Since we only need these valuations over finite ranges of variables, i.e. the free variables of a term, it makes sense to use lists. Note also that η only remembered the classes of variables, while `cls` also remembers the skeletons.

```
Definition cls := TList class.
```

The procedure `cl_term t e` combines $\mathrm{skeleton}_\eta(t)$ and $\mathrm{class}_\eta(t)$ into an element of `class`. Note that in the mathematical proof, $\mathrm{skeleton}_\eta(t)$ was only defined when t was in $\mathcal{K}_\eta$. In the formalization, we will also receive a skeleton for t in $\mathcal{C}_\eta$; the definition has been chosen to select the same skeleton for t and T where $t \in \mathcal{C}_\eta$, $T \in \mathcal{K}_\eta$ and $\Gamma \vdash t : T$ when $\eta \ltimes \Gamma$. This is needed later for technical reasons. The combination of skeletons and classification has two advantages. First, we receive implicit typing information about the terms at hand when we want to determine their skeletons. In those cases that we can rule out by Lemma 28 and Lemma 29, we can choose skeletons that will

make later proofs go through easily, without having to point at the fact that those cases can not occur. Second, recall the dependencies listed by Barras. By combining classification and skeletons, two steps of this dependency can be tackled at once: both the correspondence and the correctness of skeletons can be proven at the same time for kinds, and then again for constructors. The downside is that sometimes, we will want to talk only about the skeleton or only about the class of a term; and this becomes more complicated.

Barras also states that in an earlier development [3, p. 66], `cl_term` was separated in two distinct procedures - just as in the mathematical proof.

```
Fixpoint cl_term (t : term) : cls -> class :=
  fun i : cls =>
  match t with
  .
  .
  .
  | Sigma T U => Typ PROP
  | Pair A B u v => Trm
  | Proj1 M => Trm
  | Proj2 M => Trm
  end.
```

In the same way η_Γ chooses an η for Γ, `class_env g` produces a `cls` for some environment `g`.

```
Fixpoint class_env (e : env) : cls :=
  match e with
  | nil => TNl _
  | t :: f => TCs _ (cl_term t (class_env f)) (class_env f)
  end.
```

4.5.1 Loose Stability

We first define `loose_eqc`, or class equality: two elements of `class` are considered equal under `loose_eqc` if they have the same class, and if that class is `Knd`, also the same skeleton. Then we define `adj_cls`, or the class hierarchy: the lowest class is `Trm`, followed by `Typ`. The highest class in the hierarchy is `Knd`.

Note that for both class equality and the class hierarchy, the skeletons are largely ignored. For example, two `class`-objects `Typ s1` and `Typ s2` are considered equal independent of `s1` and `s2`. The reason for this is that the classification of terms in the `Typ` class depends on the skeleton of their type (which is itself a `Knd` term). As a consequence we can prove similar stability results for the `Typ` class only when we have them for the `Knd` class. We will soon consider strict equality and the strict hierarchy, where skeletons are also considered for the `Typ` class.

```
Inductive loose_eqc : class -> class -> Prop :=
  | leqc_trm : loose_eqc Trm Trm
```

```
  | leqc_typ : forall s1 s2 : skel, loose_eqc (Typ s1) (Typ s2)
  | leqc_ord : forall s : skel, loose_eqc (Knd s) (Knd s).

Inductive adj_cls : class -> class -> Prop :=
  | adj_t : forall s : skel, adj_cls Trm (Typ s)
  | adj_T : forall s1 s2 : skel, adj_cls (Typ s1) (Knd s2).
```

A slightly stricter version of the class-level correspondence, or Theorem 1, is formalized in `cl_term_sound`. In fact, the three subcases are proven individually in `class_knd`, `class_typ` and `class_trm`. The version in the formalization is stricter since it requires $\eta = \eta_\Gamma$, instead of just $\eta \ltimes \Gamma$. Note that only the direction from the level to the class is proven. We call `cv_skel (cl_term t eta)` the kind-skeleton of `t` and `typ_skel (cl_term t eta)` the constructor-skeleton of `t`. Note that the constructor-skeleton of a `t` is `PROP` if `t` is a `Knd` in `eta`, and we will later prove that it is the kind-skeleton of `T` if `t` is a `Typ` in `class_env gamma` and `typ gamma t T`. Note that the kind-skeleton of `t` in `class_env gamma` corresponds to $\text{skeleton}_\Gamma(t)$ if `typ gamma t (Srt set)`.

```
Definition cv_skel (c : class) : skel :=
  match c with
  | Knd s => s
  | _ => PROP
  end.

Definition typ_skel (c : class) : skel :=
  match c with
  | Typ s => s
  | _ => PROP
  end.

Lemma class_knd :
  forall (e : env) (t T : term),
  typ e t T ->
  T = Srt set ->
  cl_term t (class_env e)
    = Knd (cv_skel (cl_term t (class_env e))).

Lemma class_typ :
  forall (e : env) (t T : term),
  typ e t T ->
  typ e T (Srt set) ->
  cl_term t (class_env e)
    = Typ (typ_skel (cl_term t (class_env e))).

Lemma class_trm :
  forall (e : env) (t T : term) (s : sort),
  s = prop -> typ e t T -> typ e T (Srt s) ->
    cl_term t (class_env e) = Trm.
```

```
Lemma cl_term_sound :
  forall (e : env) (t T : term),
  typ e t T ->
  forall K : term,
  typ e T K ->
    adj_cls (cl_term t (class_env e)) (cl_term T (class_env e)).
```

As mentioned before, `cl_term` assigns the same skeleton to constructors and their kinds. This is proven in `skel_sound`.

```
Lemma skel_sound :
  forall (e : env) (t T : term),
  typ e t T ->
    cv_skel (cl_term T (class_env e)) = typ_skel (cl_term t (class_env e)).
```

4.5.2 Strict Stability Results

From the loose stability results and `skel_sound` in particular, the strict results follow. We replace `loose_eqc` by conventional equality and `adj_cls` by `typ_cls`. In both cases, we now consider the skeletons.

The definition of `typ_cls` hence formalizes the subclassing relation, which is a stronger version of the class hierarchy. `Trm` is a subclass of `Typ PROP` only, and `Typ s` is a subclass of `Knd s`.

```
Inductive typ_cls : class -> class -> Prop :=
  | tc_t : typ_cls Trm (Typ PROP)
  | tc_T : forall s : skel, typ_cls (Typ s) (Knd s).
```

The proof that this `cl_term` respects this hierarchy is formalized in `class_sound`. To be exact, the class of t under Γ is a subclass of the class of T under Γ if $\Gamma \vdash t : T$ and T has a type in Γ. The latter condition is to make sure, generally speaking, that we are not trying to relate the classes of Prop and Type_0 which are both $\mathcal{K}_\Gamma$.

```
Lemma class_sound :
  forall (e : env) (t T : term),
  typ e t T ->
  forall K : term,
    typ e T K -> typ_cls (cl_term t (class_env e)) (cl_term T (class_env e)).
```

The last lemma in this file is `class_red`, which combines Lemma 34 and Lemma 37. Note that Lemma 37 reasons about $\mathcal{I}_\eta(t)$, while this lemma reasons about $\mathrm{skeleton}_\eta(t)$; it is easy to check that those two are equivalent, though.

```
Lemma class_red :
  forall (e : env) (T U K : term),
  typ e T K -> red T U -> cl_term T (class_env e) = cl_term U (class_env e).
```

4.6 Candidates of Reducibility

In the next file, `can.v`, we formalize candidates of reducibility and the interpretations for products and sigmas. Note that our mathematical proof assumes set extensionality, an axiom which does not hold in our formalization. We will often be in a situation where we can only prove extensional equality, not Leibniz equality, of two candidates. Hence we formalize extensional equality of candidates of reducibility. Also the default candidates are defined in this file.

The fixpoint `Can` corresponds loosely to our function can. Note that `Can s` is weaker than can(s). While `Can PROP` is the type of sets of terms, can($\bigstar$) is the set of sets of terms with the reducibility properties. We will later have to prove that the result of the interpretation, which will be an element of one of the `Can s`, does in fact satisfy the reducibility properties. For this reason, Barras calls `Can s` a *scheme of order* `s`, not a candidate. We also formalize extensional equality of schemes of order `s` with `eq_can`. The base case is common set extensionality, which is formalized in `eq_cand`. It is notable that the formalization does not prove that this is an equivalence relation. In fact, reflexivity is not proven. Instead, we only consider those schemes which behave reflexive, i.e. schemes `X` of order `s` where `eq_can s X X`. Barras calls these schemes *invariant*. In many definitions that will follow, there will statements that limit the domain to invariant schemes. The first such example is `eq_can` (below), which in the recursive definition requires `eq_can s1 X1 X1` and `eq_can s1 X2 X2`.

```
Fixpoint Can (K : skel) : Type :=
  match K with
  | PROP => term -> Prop
  | PROD s1 s2 => Can s1 -> Can s2
  end.

Definition eq_cand (X Y : term -> Prop) : Prop :=
  forall t : term, X t <-> Y t.

Fixpoint eq_can (s : skel) : Can s -> Can s -> Prop :=
  match s as s0 return (Can s0 -> Can s0 -> Prop) with
  | PROP => eq_cand
  | PROD s1 s2 =>
      fun C1 C2 : Can (PROD s1 s2) =>
      forall X1 X2 : Can s1,
      eq_can s1 X1 X2 ->
      eq_can s1 X1 X1 -> eq_can s1 X2 X2 -> eq_can s2 (C1 X1) (C2 X2)
  end.
```

The next definition that we recognize is `neutral`, which formalizes neutrality. Recall that a term is neutral if it is not a lambda or a pair.

```
Inductive neutral : term -> Prop :=
  .
  .
```

```
  .
  | neutral_sigma T U : neutral (Sigma T U)
  | neutral_proj1 M : neutral (Proj1 M)
  | neutral_proj2 M : neutral (Proj2 M).
```

Recall that `Can PROP` was less specific than can(Prop). We will now define what it means for a reducibility scheme to be a reducibility candidate. For schemes of order `PROP`, this corresponds directly to the three reducibility properties. For schemes of order `PROD s1 s2`, the extension is similar to what we did in the mathematical proof; but for reasons that were mentioned before, we only consider invariant candidates. We also prove that every reducibility candidate includes at least all variables.

```
Record is_cand (X : term -> Prop) : Prop :=
  {incl_sn : forall t : term, X t -> sn t;
   clos_red : forall t : term, X t -> forall u : term, red1 t u -> X u;
   clos_exp :
    forall t : term, neutral t ->
      (forall u : term, red1 t u -> X u) -> X t}.

Lemma var_in_cand :
  forall (n : nat) (X : term -> Prop), is_cand X -> X (Ref n).

Fixpoint is_can (s : skel) : Can s -> Prop :=
  match s as s0 return (Can s0 -> Prop) with
  | PROP => fun X : term -> Prop => is_cand X
  | PROD s1 s2 =>
      fun C : Can s1 -> Can s2 =>
      forall X : Can s1, is_can s1 X -> eq_can s1 X X -> is_can s2 (C X)
  end.
```

Now we formalize the default candidates, D_s. The default candidate of order `PROP` is the set of strongly normalizing terms, `sn`. We prove that the default candidates are actually invariant candidates of the respective order.

Note that default candidates are used much earlier in the formalization than in the mathematical proof. One of the reasons is that in the mathematical proof, definitions over terms in $\mathcal{K}_\Gamma$ or $\mathcal{C}_\Gamma$ use Lemma 28 or Lemma 29, respectively. In the formalizations, this is not easily possible. Hence we need values for the cases that are exempt by aforementioned lemmas. For some definitions, D_s will do.

```
Fixpoint default_can (s : skel) : Can s :=
  match s as ss return (Can ss) with
  | PROP => sn
  | PROD s1 s2 => fun _ : Can s1 => default_can s2
  end.


Lemma def_can_cr : forall s : skel, is_can s (default_can s).
```

```
Lemma def_inv : forall s : skel, eq_can s (default_can s) (default_can s).
```

The next definitions, `Pi` and `Cartesian`, correspond to our definitions of $X \rightrightarrows Y$ and $X \times Y$. Note the quantification `C : Can s`. The first thing that we have to consider here is that again, `C : Can s` is too large, so we restrict it to the invariant candidates in this scheme. The second thing is that this intersection does not appear in the definition of $X \rightrightarrows Y$. It corresponds closely, however, to our definition of $[\![\forall x : U.V]\!]_\xi$ where $U \in \mathcal{K}_\xi$. Note that we could define $[\![\forall x : U.V]\!]_\xi$ where $U \in \mathcal{C}_\xi$ in a similar fashion: $[\![V]\!]_{\xi^x_C} = \bigcap_{u \in [\![U]\!]_\xi} [\![V]\!]_{\xi^x_C}$ is a trivial equality when $[\![U]\!]_\xi$ is non-empty. In fact, this is exactly what we will later do when we formalize $[\![T]\!]_\xi$.

```
Definition Pi (s : skel) (X : term -> Prop) (F : Can (PROD s PROP))
  (t : term) : Prop :=
  forall u : term,
  X u -> forall C : Can s, is_can s C -> eq_can s C C -> F C (App t u).

Definition Cartesian (X Y : Can PROP)
  (t : term) : Prop :=
  X (Proj1 t) /\ Y (Proj2 t).
```

Lemmas 40 and 41 are formalized in `is_can_Pi` and `cartesian_is_cand`, respectively. In addition to this, we have to prove that `Pi` and `Cartesian` preserve invariance. Those proofs are formalized in `eq_can_Pi` and `cartesian_eq_can`.

```
Lemma eq_can_Pi :
  forall (s : skel) (X Y : term -> Prop) (F1 F2 : Can (PROD s PROP)),
  eq_can PROP X Y ->
  eq_can (PROD s PROP) F1 F2 ->
    eq_can PROP (Pi s X F1) (Pi s Y F2).

Lemma is_can_Pi :
  forall (s : skel) (X : term -> Prop),
  is_cand X ->
    forall F : Can (PROD s PROP),
      is_can (PROD s PROP) F -> is_cand (Pi s X F).


Lemma cartesian_eq_can X1 Y1 X2 Y2 :
  eq_can PROP X1 X2 ->
  eq_can PROP Y1 Y2 ->
    eq_can PROP (Cartesian X1 Y1) (Cartesian X2 Y2).

Lemma cartesian_is_cand X Y :
  is_can PROP X ->
  is_can PROP Y ->
    is_can PROP (Cartesian X Y).
```

4.7 Interpretation

In the file `int_typ.v`, we formalize $(\mathrm{CR})^* \cup \{\mathcal{C}, \mathcal{O}\}$-valuations ξ. We define a method to get a class-valuation η_ξ for some ξ. We lift our definition of extensional equality and invariance on schemes to ξ. And finally, we formalize the interpretation $[\![t]\!]_\xi$.

The first two definitions correspond to $(\mathrm{CR})^* \cup \{\mathcal{C}, \mathcal{O}\}$-valuations. Note that again, we use a list to formalize the valuation. Recall that in the mathematical proof, we said that it was possible to use $(\mathrm{CR})^* \cup \{\mathcal{C}\}$-valuations instead. In the formalization, we do exactly that. `Int_K` formalizes the set $(\mathrm{CR})^* \cup \{\mathcal{C}\}$, and `intP` formalizes a valuation of this set.

```
Inductive Int_K : Type :=
  | iK : forall s : skel, Can s -> Int_K
  | iT : Int_K.

Definition intP := TList Int_K.
```

The next two definitions correspond to η_ξ. The idea is to extract the class out of the object of `Int_K`. As in the formalization classes and skeletons are combined, we need to put in a skeleton in the `iT` case as well. Since we are actually only interested in the class here, we choose `PROP`. We will later prove that this is acceptable, and that the class extracted of $[\![T]\!]_\xi$ is the class of T under ξ.

```
Definition class_of_ik (ik : Int_K) :=
  match ik with
  | iK s _ => Knd s
  | iT => Typ PROP
  end.

Definition cls_of_int : intP -> cls := Tmap _ _ class_of_ik.
```

We define extensional equality of the objects of `Int_K`. This is formalized in `ik_eq`. Then we lift this definition to `intP` in `int_eq_can`, and define invariance of `intP` to mean that all the elements in that list are extensionally equal to themselves.

```
Inductive ik_eq : Int_K -> Int_K -> Prop :=
  | eqi_K :
      forall (s : skel) (X Y : Can s),
      eq_can s X X ->
      eq_can s Y Y -> eq_can s X Y -> ik_eq (iK s X) (iK s Y)
  | eqi_T : ik_eq iT iT.

Definition int_eq_can : intP -> intP -> Prop := Tfor_all2 _ _ ik_eq.
Definition int_inv (i : intP) := int_eq_can i i.
```

In `int_eq_can_cls`, we prove that extensionally equal `intP` produce the same `cls`. Note again that none of this is necessary when extensional equality is assumed, as is the case in the mathematical proof.

```
Lemma int_eq_can_cls :
  forall i i' : intP, int_eq_can i i' -> cls_of_int i = cls_of_int i'.
```

Recall that in the recursive definition of $[\![T]\!]_\xi$, we deal with binders $x : U$ by updating ξ with either $\mathcal{C}$ or $[\![U]\!]_\xi$, depending on the class of U. In the formalization, we have a procedure which does this case distinction and returns the correct element of $(\mathrm{CR})^* \cup \{\mathcal{C}\}$, and another one which performs the update. Those procedures are `ext_ik` and `int_cons`, respectively. Note that when $\Phi = [\![U]\!]_\xi$, `ext_ik U xi s Phi` is equal to $[\![U]\!]_\xi$ when $U \in \mathcal{K}_\xi$ and equal to $\mathcal{C}$ when $U \in \mathcal{C}_\xi$.

```
Definition ext_ik (T : term) (ip : intP) (s : skel)
  (C : Can s) :=
  match cl_term T (cls_of_int ip) with
  | Knd _ => iK s C
  | _ => iT
  end.

Definition int_cons (T : term) (ip : intP) (s : skel)
  (C : Can s) := TCs _ (ext_ik T ip s C) ip.
```

The definition of `coerce_CR` looks very complicated, but it does a very simple thing: it checks whether `i : Int_K` behaves appropriately. If `i` was constructed with `iK` and has the skeleton we want (`s`), it extracts the candidate stored in `i`. Otherwise, it returns the default candidate of order `s`. The innermost `match`-expression is there for technical reasons only.

```
Definition coerce_CR (s : skel) (i : Int_K) : Can s :=
  match i with
  | iK si Ci =>
      match EQ_skel si s with
      | left y =>
          match y in (_ = x) return (Can x) with
          | refl_equal => Ci
          end
      | _ => default_can s
      end
  | _ => default_can s
  end.
```

We now formalize $[\![T]\!]_\xi$, `int_typ T xi s`. Note the additional argument `s`, which allows us to choose the parent into which we will map. This is required for technical reasons, and it is inherently different from the way we defined $[\![\cdot]\!]_\xi$. $[\![T]\!]_\xi$ maps to some subset of $(\mathrm{CR})^*$. One of the key theorems of the mathematical proof was to prove that when $T \in \mathrm{Kind}_\Gamma$ or $T \in \mathrm{Constr}_\Gamma$, $[\![T]\!]_\xi$ is defined and maps to the correct subset of $(\mathrm{CR})^*$ - namely $\mathcal{I}_\Gamma(K)$, where $\Gamma \vdash T : K$.

In the formalization, we need `int_typ T xi s` to be a total function. In some cases, this requires that we know that the recursive calls have the correct

type. Consider the case of $[\![t\,u]\!]_\xi$ where $t \in \mathcal{K}_\xi$: $[\![t]\!]_\xi([\![u]\!]_\xi)$ is defined only if $[\![t]\!]_\xi : A \to B$ and $[\![u]\!]_\xi \in A$ for some sets A and B. In the formalization, we will be able to choose `s1` and `s2` such that `int_typ t xi s1 : Can s1` and `int_typ t xi s2 : Can s2`. We will hence choose `PROD s2 s` for `s1`, which will ensure that `(int_typ t xi (PROD s2 s)) : Can s2 -> Can s`, and justifies the application.

We will also choose `s2` to be the constructor-skeleton of `u` in `xi`. Recall that this equals to the kind-skeleton of `T` when `cls_of_int xi = class_env gamma`, `typ gamma u T` and `typ gamma T (Srt set)`. If you compare this with our proof of Theorem 2, you will notice that this is exactly the situation we find ourselves in in the application case.

Thus for $T \in \mathrm{Kind}_\Gamma$, we will choose `s` to be `PROP`, and for $T \in \mathrm{Constr}_\Gamma$ where $\Gamma \vdash T : K$, we will choose `s` to be `cv_skel (cl_term K (class_env gamma))`, that we get Theorem 2 for free. Note that by `cl_term_sound`, both equal to the constructor-skeleton of `T` in `class_env gamma`. This is why Barras states "this definition is relevant only when T is a well-formed [constructor or kind] in [an environment] Γ and [`s` is the constructor-skeleton of `T` in `class_env gamma`]" [4, p. 20]. It might not be obvious that if $T \in \mathrm{Constr}_\Gamma$, this invariant will actually be enforced by the way we apply `int_typ` recursively. It is not hard to prove, though.

```
Fixpoint int_typ (T : term) : intP -> forall s : skel, Can s :=
  fun (ip : intP) (s : skel) =>
  match T with
  .
  .
  .
  | App u v =>
        match cl_term v (cls_of_int ip) with
        | Trm => int_typ u ip s
        | Typ sv => int_typ u ip (PROD sv s) (int_typ v ip sv)
        | _ => default_can s
        end
  .
  .
  .
  | Sigma A B =>
      match s as x return (Can x) with
      | PROP =>
          let intA := (int_typ A ip PROP) in
          Cartesian intA (int_typ B (int_cons A ip PROP intA) PROP)
      | PROD s1 s2 => default_can (PROD s1 s2)
      end
  | _ => default_can s
  end.
```

4.8 Stability Results

The file `int_stab.v` proves the stability results for interpretations. It begins with a few lemmata which are trivial in the mathematical setting, e.g., the fact that if ξ_1 and ξ_2 contain extensionally equal candidates, then $\mathrm{class}_{\xi_1}(T) = \mathrm{class}_{\xi_2}(T)$.

Recall the definition of `ext_ik` on p. 74, which returned the correct element Φ of $(\mathrm{CR})^* \cup \{\mathcal{C}\}$ to interpret the type introduced in a binder $x : U$ under some ξ. Now we will look at the formalization of the phrase "correct interpretation" for terms of that type. An example for this is Lemma 42, where u is a term of type U. We extend ξ either by $[\![u]\!]_\xi$ when $U \in \mathrm{Kind}_\Gamma$ or by $\mathcal{C}$ when $U \in \mathrm{Constr}_\Gamma$. By Theorem 1 and the fact that the class of u is a subclass of the class of U in this case, it would be possible to perform the case distinction on the class of u instead. The declaration of `int_var_sound u xi Phi` thus formalizes what it means for Φ to be the correct interpretation of u under ξ: when u is in $\mathcal{C}_\xi$, Φ must be $[\![u]\!]_\xi$, and when u is in $\mathcal{O}_\xi$, Φ must be $\mathcal{C}$. Note however that the class extracted from Φ[1] is then $\mathcal{K}$ when $u \in \mathcal{C}_\xi$ and $\mathcal{C}$ when $u \in \mathcal{O}_\xi$. Thus the class of u is a subclass of the class extracted from Φ; a proof of this is formalized in `int_var_sound_lift`.

```
Inductive int_var_sound (t : term) (ip : intP) : Int_K -> Prop :=
  | ivs_K :
      forall s : skel,
      cl_term t (cls_of_int ip) = Typ s ->
      int_var_sound t ip (iK _ (int_typ t ip s))
  | ivs_T : cl_term t (cls_of_int ip) = Trm -> int_var_sound t ip iT.


Lemma int_var_sound_lift :
  forall (t : term) (ip : intP) (i : Int_K),
  int_var_sound t ip i ->
    typ_cls (cl_term t (cls_of_int ip)) (class_of_ik i).
```

The next lemma we recognize, since it corresponds closely to Lemma 42. Note the lack of a case split due to `int_var_sound`.

Note also the predicates `TIns` and `TTrunc`, as well as the three valuations `ipe`, `ipf`, and `ipg`. These are required due to the fact that we represent $(\mathrm{CR})^* \cup \{\mathcal{C}\}$ valuations using lists. In the mathematical proof we would simply write $\xi(x := C)$ for `ipe` and ξ for `ipf` and `ipg`, but in the formalization this is much harder to write down. `TIns _ i k ipf ipe` states that `ipf` is `ipe` where the value of the variable with index `k` has been updated to `i`. `TTrunc _ k ipf ipg` states that `ipg` is the prefix of length `k` of `ipf`.

There is another important difference to the mathematical version, in which the mathematical proof uses $\xi \Vdash \Gamma$ while the formalization only requires $\eta_\xi = \eta_\Gamma$. In fact, there is no statement corresponding to the other consequence of $\xi \Vdash \Gamma$[2]

[1] See `class_of_ik` on p. 73

[2] I.e., that for all prefixes $\Gamma', x : T$ of Γ, $T \in \mathrm{Kind}_\Gamma$ implies $\xi(x) \in \mathcal{I}_T(\Gamma')$

in the formalization. The reason for this lies in the additional parameter `s` that `int_typ` takes when compared to its mathematical counterpart, $[\![\cdot]\!]_\xi$. By definition, we could restate the missing consequence as follows:

$$\forall (x : T) \in \Gamma, T \in \mathcal{K}_\Gamma \implies [\![x]\!]_\xi \in \mathrm{can}(\mathrm{skeleton}_\Gamma(T))$$

In the formalization, this would require us to prove that `int_typ T xi s : Can s`, where `s` would be the constructor-skeleton of T - which is true by definition. Hence, in the formalization, $\eta_\xi = \eta_\Gamma$ is equivalent to $\xi \Vdash \Gamma$. We will thus not distinguish between them while describing the formalization.

```
Lemma subst_int_typ :
  forall (v : term) (ipg : intP) (i : Int_K),
  int_var_sound v ipg i ->
  forall (e : env) (T K : term),
  typ e T K ->
  forall (k : nat) (ipe ipf : intP),
  TIns _ i k ipf ipe ->
  TTrunc _ k ipf ipg ->
  cls_of_int ipe = class_env e ->
  int_inv ipe ->
  cl_term T (cls_of_int ipe) <> Trm ->
  eq_can (skel_int T ipe) (int_typ T ipe _)
    (int_typ (subst_rec v T k) ipf _).
```

Now we consider the lemma `int_cons_equal`. This lemma proves that if $\eta \ltimes \Gamma$ and we extend ξ and Γ consistently, that $\eta_{\xi^x_C} \Vdash \eta_{\Gamma,x:T}$. Recall that in the mathematical proof, we simply defined $\eta_{\xi^x_C} \Vdash \eta_{\Gamma,x:T}$ this way, and needed two rules, one where $T \in \mathrm{Kind}_\Gamma$ and one where $T \in \mathrm{Constr}_\Gamma$. We only require a single lemma in the formalization since this case split is hidden in `int_cons`, which uses `ext_ik`.

```
Lemma int_cons_equal :
  forall (ip : intP) (e : env),
  cls_of_int ip = class_env e ->
  forall (N : term) (s1 : sort),
  typ e N (Srt s1) ->
  forall C : Can (cv_skel (cl_term N (class_env e))),
    cls_of_int (int_cons N ip _ C) = class_env (N :: e).
```

The last important lemma in this file is `conv_int_typ`, which straightforwardly formalizes Lemma 43.

```
Lemma conv_int_typ :
  forall (e : env) (U V K : term),
  conv U V ->
  typ e U K ->
  typ e V K ->
```

```
  forall ip : intP,
  cls_of_int ip = class_env e ->
  int_inv ip ->
  cl_term U (class_env e) <> Trm ->
    eq_can (skel_int U ip) (int_typ U ip _) (int_typ V ip _).
```

4.9 Termination

The proof of Lemma 1 is contained in `strong_norm.v`. A few of the lemmata and properties which have been defined earlier in the mathematical proof are defined in this file, too.

The first definition formalizes $\rho \Vdash_\Gamma \xi$. On first glance, it is strange that the value with which we extend ξ does not matter for the correctness of this definition. This is due to the fact that if $\Gamma \vdash t : T$ and x is $\Gamma-$fresh, x can not be free in T. Hence the value of x in ξ does not matter for $[\![T]\!]_\xi$. Finally, `int_adapt gamma xi rho` couples $\xi \Vdash \Gamma$ and $\rho \Vdash_\Gamma \xi$.

```
Inductive trm_in_int : env -> intP -> intt -> Prop :=
  | int_nil : forall itt : intt, trm_in_int nil (TNl _) itt
  | int_cs :
      forall e (ip : intP) (itt : intt),
      trm_in_int e ip itt ->
      forall (y : Int_K) t T,
      int_typ T ip PROP t ->
      trm_in_int (T :: e) (TCs _ y ip) (shift_intt itt t).


Record int_adapt e (ip : intP) (itt : intt) : Prop :=
  {adapt_trm_in_int : trm_in_int e ip itt;
   int_can_adapt : can_adapt ip;
   adapt_class_equal : cls_of_int ip = class_env e}.
```

The lemma `extend_int` proves that if we extend Γ with $x : T$, and ξ and ρ using some default values (which depend on the class and skeleton of A), `int_adapt` is preserved.

```
Lemma extend_int A e ip it :
  typ e A (Srt prop) ->
  int_adapt e ip it ->
    int_adapt (A :: e) (def_cons A ip) (shift_intt it (Ref 0)).
```

Now we formalize Theorem 3, in a very straightforward way.

```
Lemma int_sound :
  forall e t T,
  typ e t T ->
  forall (ip : intP) (it : intt),
    int_adapt e ip it -> int_typ T ip PROP (int_term t it 0).
```

The next two definitions formalize ξ_Γ and a default assignment, `def_intt e k`. Note that `def_intt` appears to depend on Γ and some index k. However, the Γ is not meaningful to the computation of the result ; the function is merely stated in this way to ease the proof of `def_adapt`, which is the lemma that formalizes Lemma 47.

The lemma `def_intt_id` formalizes a proof that `def_intt` can be stated as the function which maps the variable with index `p` to the variable with index `p + k`. Obviously for $k = 0$, this function behaves like the identity assignment ; a proof of this claim is given in `id_int_term`. We need the `k` only for technical reasons which are due to the usage of de Bruijn indices.

```
Fixpoint def_intp e : intP :=
  match e with
  | nil => TNil _
  | t :: f => def_cons t (def_intp f)
  end.

Fixpoint def_intt e : nat -> intt :=
  fun k =>
  match e with
  | nil => fun p => Ref (k + p)
  | _ :: f => shift_intt (def_intt f (S k)) (Ref k)
  end.

Lemma def_adapt :
  forall e, wf e -> forall k, int_adapt e (def_intp e) (def_intt e k).

Lemma def_intt_id :
  forall n e k, def_intt e k n = Ref (k + n).

Lemma id_int_term :
  forall e t k, int_term t (def_intt e 0) k = t.
```

And finally we can formalize the proof of Lemma 1, as well as the corollary that if $\Gamma \vdash t : T$, $T\downarrow$.

```
Theorem str_norm : forall e t T, typ e t T -> sn t.

Lemma type_sn : forall e t T, typ e t T -> sn T.
```

5. CONCLUSION AND FUTURE WORK

We have given a mathematical description and a formalization of proofs for confluence and termination of CC_Σ. Coq lends itself to formalizing type theories and many properties thereof. While other proof assistants might have given us stronger automatization tools, we were very comfortable with the level of automatization that Coq gave to us.

Luckily, Barras dealt with the only burden that the environment of Coq put on us, and that was the formalization of candidates of a higher order. The intensional nature of Coq called for the formalization of extensional equivalence in `eq_can` and the introduction of invariance.

A perk of formalizing these proofs in Coq without supposing any further axioms is that the formal proofs are constructive. This means that the proof of termination could easily be turned into a program that implements a reduction strategy, i.e., an interpreter of CC_Σ.

There are a few things that we would do different if we had to redo this project. Recall the four main differences between the formalization and the mathematical description:

- The usage of de Bruijn indices
- Classification and skeletons are combined
- Constructors have skeletons
- There is an additional parameter `s` in `int_typ T xi s`

As we mentioned in the introduction, we believe that a locally nameless approach would have greatly simplified the formalization and unified the mathematical presentation and the formalization. As a simple example, consider the formalization of Lemma 20 on p. 65. Compare it with what the statement could look like in the context of a locally nameless approach, where `(x & T) :: gamma` corresponds to $\Gamma, x : T$:

```
Theorem thinning :
  forall e x A,
    wf ((x & A) :: e) ->
    forall t T,
    typ e t T ->
    typ ((x & A) :: e) t T.
```

We also believe that the combination of classification with skeletons creates more confusion than it entails benefits. We thus formalized a standalone version of classification, which is a direct translation of the mathematical proof found in this thesis. However, due to our limited time frame, we did not adapt the rest of the formal proof to fit this environment. In addition to this, we believe that this separation is a necessity when wanting to formalize vertical extensions. We will go into more detail later.

We have a more ambivalent opinion on constructor-skeletons and their role as the additional parameter `s`. On one hand, we see them as a huge technical improvement in the setting of CC_Σ (and, for that matter, CC). Compare, for example, the cumbersome mathematical statement "$[\![T]\!]_\xi \in \mathcal{I}_\Gamma(K)$ when $\Gamma \vdash T : K$ and $\xi \Vdash \Gamma$" to its formal version, i.e., `int_typ T xi s : Can s` for all `s`. Note that one requires an exceedingly long proof (Theorem 2), while we get the other for free. On the other hand, we doubt that this notion can be translated to vertical extensions of CC_Σ.

Note that we did not mention horizontal extensions yet as we believe they can be easily dealt with in the current environment by following the proof sketches given in [13, pp. 13–22]. The interesting question is whether this proof technique can be used for vertical extensions of CC (or CC_Σ). The mathematical proof of termination presented here can be divided into three segments: classification, skeletons, and interpretation. It is our belief that classification is trivial to extend to arbitrary numbers of universes, as long as there is no subtyping rule which allows us to treat types from one universe as types from a higher universe.

For skeletons and interpretation, the situation is a little more complicated. Recall our current definitions of $\mathcal{I}_\eta(\cdot)$ and $[\![\cdot]\!]_\xi$, of which the well-definedness depends strongly on the possible syntactic forms of kinds and constructors. In stronger vertical extensions of CC_Σ like ECC, kinds can also have other forms, e.g., applications. Luo deals with this by proving that all types can be reduced to a type which has a form similar to the one presented in Lemma 28[23, p. 77, Corollary 4.14]. This result might be sufficient to define skeletons and interpretation in these theories.

Another possibility would be to extend skeletons to work with application, abstraction, application and pairs, and certainly a few ways for this are thinkable. Good ideas for this could come from the formalization of constructor-skeletons from Barras, since constructors in CC already do not enjoy the privilege of Lemma 28.

When it comes to extending the formalization itself, however, the situation appears to be quite different. We have already mentioned that it is possible to rectify many of the differences between the formalization and the mathematical presentation. We will now give a short discussion as to why we think this is desirable.

We said that it is possible to separate skeletons from classification. Note that in the current formalization, there has to be one stability result for every universe, like loose stability for kinds (corresponding to Type_0) and strict stability for constructors (corresponding to Prop). If we were to add further universes, this would require further stability results; an extension to an arbi-

trary number of universes (e.g., ECC) seems out of reach. While these stability results might be needed to deal with constructor-skeletons, we believe that they are a consequence of the combination of skeletons and classification.

We also said that we doubt that the notion of constructor-skeletons can be translated to vertical extensions of CC_Σ. Let us take the perspective that constructor-skeleton and kind-skeleton are the polarized lenses of movie theater glasses. When we write down a type hierarchy, e.g., $\emptyset \vdash I : \mathrm{True} : \mathrm{Prop} : \mathrm{Type}_0$ (Section 2.8, p. 19), we can align the lenses such that through the left lense we see a term, and through the right lense we see a type. In above example, this would give us three possible positions for our glasses, for example, one where I is seen through the constructor lense and True is seen through the kind lense; or another where True is seen through the constructor lense and Prop is seen through the kind lense.

Now we replace every term in the hierarchy by two polarized and overlapping pictures, such that one picture is visible through the constructor lense and the other through the kind lense. Let us call these pictures the constructor picture and the kind picture, respectively. There are a couple of properties that this setup should satisfy, we will mention three and then discuss why we doubt that these properties can be satisfied in some vertical extensions. First, no matter which position we choose, the pictures that are seen through the two lenses must be the same. Second, when we look at a product, the kind picture must look like $s_1 \to_S s_2$ for some smaller pictures s_1 and s_2. Third, when we look at a universe, both pictures must look like $\bigstar$.

Now it appears that there are two ways to generate such a setup. The first is to use two functions, one which generates the kind pictures and another which generates the constructor pictures. The other option is to use a single function that generates a picture, and then use two functions that decide whether to take this picture or use a different one.

It seems impossible to use the first way and satisfy all properties. To be more precise, in order to prove the first property, one would need to prove that the kind picture of U^x_u in the environment Γ equals to the kind picture of U in the environment $\Gamma, x : T$, where $\Gamma \vdash x : T$. Let us give two instances that together seem insolvable. Let $U := x$ and $u_1 := \mathrm{Prop} \to \mathrm{Prop}$ and $u_2 := \mathrm{Prop}$ and $\Gamma := \emptyset$ and $T := \mathrm{Type}_0$. It is easy to verify that $\Gamma \vdash u_1 : T$ and $\Gamma \vdash u_2 : T$. If the above property would be true, the kind picture of $U^x_{u_1}$ would have to equal the kind picture of $U^x_{u_2}$. Thus the kind picture for $\mathrm{Prop} \to \mathrm{Prop}$ needs to equal the kind picture for Prop, which by properties two and three can not be the case.

It seems impossible to use the second way in a vertical extension and satisfy all properties. A consequence of the second and third property is that a product needs to have $s_1 \to s_2$ as the kind picture and $\bigstar$ as the constructor picture. This would mean that the single function would need to know in advance whether the product is to be looked at through the kind lense or through the constructor lense. In CC_Σ, this is basically the case. With a little bit of simplification, we can say that we look at products with the kind lense only if they are kinds, and with the constructor lense only when they are constructors. In many vertical

extensions it seems absurd to have a function that will know in advance which lense we will use to look at any given product.

APPENDIX

A. COQ SYNTAX

Since we show definitions and statements of lemmata in this thesis, a high level introduction into Coq Syntax is in order.

Basics

`Prop` is the type of propositions and `Type` is the type of datatypes. `X -> Y` is a function type or implication, which are the same in Coq.

Since Coq uses type inference, types can be largely omitted. This affects both binders, which are usually of the form `forall (x : T), ...` or `exists (x : T), ...`(short `forall x, ...`, if `T` can be inferred), as well as function application which is of the form `f T t` (short `f _ t`, if `T` can be inferred).

Function application is left associative while function types are right associative, i.e., `f a b = (f a) b` while `A -> B -> C = A -> (B -> C)`.

Inductive Types

Inductive types are defined via the intro rules:

```
Inductive name : type :=
| rule1 :  ...
| rule2 :  ...
...
```

Recursion and induction schemes are automatically derived by the Coq system.

Recursive Functions

Recursive functions (fixpoints) are defined as follows:

```
Fixpoint name (x : type) : type := ...
```

where `x` is the variable we do recursion on and thus needs to have an inductive type.

These definitions are often followed by a case distinction on `x`, where the fixpoint is then applied to the structurally smaller components.

Definitions

Other simple definitions have the form:

```
Definition name : type := ...
```

Lemmata

A lemma is introduced with the `Lemma` keyword:

```
Lemma name : type.
```

Record

For the purpose of this thesis, a record can be seen as a conjunction of any number of named properties.

```
Record int_adapt binders : type :=
{ name1 : property1;
  name2 : property2;
  ...
  }.
```

The full Coq development is located at `http://ps.uni-saarland.de/~jonas/ccsigma_Oct4.tar`.

B. DECLARATIONS

B.1 Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

B.2 Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis

B.3 Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

B.4 Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, _____ _____

(Datum/Date) (Unterschrift/Signature)

BIBLIOGRAPHY

[1] Robin Adams. "Lambda-free logical frameworks". In: *arXiv preprint arXiv:0804.1879* (2008).

[2] Brian Aydemir et al. "Engineering formal metatheory". In: *ACM SIGPLAN Notices* 43.1 (2008), pp. 3–15.

[3] B. Barras. *Coq en Coq*. Rapport de Recherche 3026. INRIA, Oct. 1996.

[4] B. Barras. "Coq in Coq". 1997.

[5] Hendrik Pieter Barendregt. *The lambda calculus: Its syntax and semantics*. Vol. 103. North Holland, 1984.

[6] Henk Barendregt. "Introduction to generalized type systems". In: *Journal of functional programming* 1.2 (1991), pp. 125–154.

[7] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. springer, 2004.

[8] Nicolaas Govert de Bruijn. "Reflections on automath". In: *Nederpelt et al.[1994]* (1990), pp. 201–228.

[9] Alonzo Church. "A formulation of the simple theory of types". In: *J. Symb. Log.* 5.2 (1940), pp. 56–68.

[10] Alonzo Church. "A set of postulates for the foundation of logic". In: *Annals of mathematics* 33.2 (1932), pp. 346–366.

[11] Thierry Coquand, Gerard Huet, et al. "The calculus of constructions". In: (1986).

[12] Nicolaas Govert De Bruijn. "Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem". In: *Indagationes Mathematicae (Proceedings)*. Vol. 75. 5. Elsevier. 1972, pp. 381–392.

[13] Herman Geuvers. "A short and flexible proof of Strong Normalization for the Calculus of Constructions". In: *TYPES*. 1994, pp. 14–38.

[14] Jean-Yves Girard. "Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur". PhD thesis. PhD thesis, Universit e Paris VII, 1972.

[15] Jean-Yves Girard. "The system F of variable types, fifteen years later". In: *Theoretical computer science* 45 (1986), pp. 159–192.

[16] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and types*. Vol. 7. Cambridge University Press Cambridge, 1989.

[17] Andrew Gordon. "A mechanisation of name-carrying syntax up to alpha-conversion". In: *Higher Order Logic Theorem Proving and Its Applications* (1994), pp. 413–425.

[18] Robert Harper, Furio Honsell, and Gordon Plotkin. "A framework for defining logics". In: *Journal of the ACM (JACM)* 40.1 (1993), pp. 143–184.

[19] William A Howard. "The formulae-as-types notion of construction". In: *To HB Curry: essays on combinatory logic, lambda calculus and formalism* 44 (1980), pp. 479–490.

[20] S.C. Kleene and J.B. Rosser. "The inconsistency of certain formal logics". In: *Annals of mathematics* 36.3 (1935), pp. 630–636.

[21] GR Renardel de Lavalette. "Strictness analysis for POLYREC, a language with polymorphic and recursive types". In: *Logic Group Preprint Series* 33 (1988).

[22] G Longo and E Moggi. *Constructive natural deduction and its modest interpretation*. Tech. rep. 1988.

[23] Z. Luo. *Computation and Reasoning: A Type Theory for Computer Science*. International Series of Monographs on Computer Science. Clarendon Press, 1994. ISBN: 9780198538356. URL: `http://books.google.de/books?id=z3uicYzJR1MC`.

[24] Per Martin-Löf. *A theory of types*. 1971.

[25] Michael Norrish. "Mechanising λ-calculus using a classical first order theory of terms with permutations". In: *Higher-Order and Symbolic Computation* 19.2 (2006), pp. 169–195.

[26] Frank Pfenning and Conal Elliot. "Higher-order abstract syntax". In: *ACM SIGPLAN Notices* 23.7 (1988), pp. 199–208.

[27] John Reynolds. "Towards a theory of type structure". In: *Programming Symposium*. Springer. 1974, pp. 408–425.

[28] Allen Stoughton. "Substitution revisited". In: *Theoretical Computer Science* 59.3 (1988), pp. 317–325.

[29] Masako Takahashi. "Parallel reductions in λ-calculus". In: *Information and computation* 118.1 (1995), pp. 120–127.

[30] Christian Urban. "Nominal unification revisited". In: *arXiv preprint arXiv:1012.4890* (2010).

[31] Freek Wiedijk and Dana S Scott. *The seventeen provers of the world*. Springer, 2006.