BAMBERGER BEITRÄGE ZUR
WIRTSCHAFTSINFORMATIK UND ANGEWANDTEN INFORMATIK
ISSN 0937-3349

Nr. 91/2013

On the Computational Interpretation of CK_n for Contextual Information Processing – Ancillary Material –

Michael Mendler and Stephan Scheele

FAKULTÄT FÜR
WIRTSCHAFTSINFORMATIK UND ANGEWANDTE INFORMATIK
OTTO-FRIEDRICH-UNIVERSITÄT BAMBERG

On the Computational Interpretation of CK_n for Contextual Information Processing – Ancillary Material –

Michael Mendler, Stephan Scheele

Faculty of Information Systems and Applied Computer Sciences
The Otto-Friedrich-University of Bamberg, Germany
{michael.mendler,stephan.scheele}@uni-bamberg.de

May 4, 2013

Abstract

In the journal article [8] we introduce a modal λ-calculus $\lambda\mathsf{CK}_n$ whose type system corresponds to the constructive multi-modal logic CK_n. This logic is a constructive refinement of the classical multi-modal logic K_n which forms the heart of the class of description logics used in semantic information processing. $\lambda\mathsf{CK}_n$ constitutes the core of a functional language for information processing in this application domain. Being strongly typed, it offers static type checking to support safe contextual reasoning in relational structures like those treated by description logics.

Here we provide detailed mathematical proofs for the results presented in [8]. Accordingly, this report is not meant as a self-contained technical exposition of the whys and hows of $\lambda\mathsf{CK}_n$. It is merely a collection of ancillary material to complement the main article [8] to which the reader is referred for more information.

An early version of this work appeared at the 3rd International Workshop on Logics, Agents and Mobility (LAM 2010). We are grateful for the support by the German Research Council (DFG) who funded this research as part of the project SPACMODL under grant No. ME 1427/4-1.

1 The Structure of Normal Form Proofs

Proposition 1. Every expression typeable under the rules of Fig. 1 and Fig. 2 is in normal form.

Proof. The proof proceeds by induction on the structure of the typing derivation. The statement is obvious for the base case of rule Ax_m and the induction steps for left rules $\vee L$, $\exists L$, the right rules $\wedge R$, $\vee R_1$, $\vee R_2$, $\supset R$, $\exists R$ as well as $\forall R$. For the left rules $\wedge L_1$, $\wedge L_2$, $\supset L$, $\forall L$ we use the fact that if nf is a normal form and y a free variable in nf, then the substitutions $nf[\![\pi_1\, x/y]\!]$, $nf[\![\pi_2\, x/y]\!]$, $nf[\![x\, nf'/y]\!]$ and $nf[\![x@b/y]\!]$ are again normal forms. For rule Ax_f we observe that if nf is a normal form, then so is $nf[\![\hat{\Gamma}/\Gamma]\!]$. □

$$\frac{}{\Sigma, x : C \vdash x : C}\,Ax_m \qquad \frac{\Sigma \vdash e_1 : D_1 \qquad \Sigma \vdash e_2 : D_2}{\Sigma \vdash (e_1, e_2) : D_1 \wedge D_2}\,\wedge R$$

$$\frac{\Sigma_1, y : C_1, \Sigma_2 \vdash e : \Psi}{\Sigma_1, x : C_1 \wedge C_2, \Sigma_2 \vdash e[\![\pi_1\, x/y]\!] : \Psi}\,\wedge L_1 \qquad \frac{\Sigma_1, y : C_2, \Sigma_2 \vdash e : \Psi}{\Sigma_1, x : C_1 \wedge C_2, \Sigma_2 \vdash e[\![\pi_2\, x/y]\!] : \Psi}\,\wedge L_2$$

$$\frac{\Sigma \vdash e : D_1}{\Sigma \vdash \iota_1\, e : D_1 \vee D_2}\,\vee R_1 \qquad \frac{\Sigma \vdash e : D_2}{\Sigma \vdash \iota_2\, e : D_1 \vee D_2}\,\vee R_2$$

$$\frac{\Sigma_1, y_1 : C_1, \Sigma_2 \vdash e_1 : \Psi \qquad \Sigma_1, y_2 : C_2, \Sigma_2 \vdash e_2 : \Psi}{\Sigma_1, x : C_1 \vee C_2, \Sigma_2 \vdash \texttt{case}\ x\ \texttt{of}\ [\iota_1\, y_1 \to e_1 \mid \iota_2\, y_2 \to e_2] : \Psi}\,\vee L$$

$$\frac{\Sigma_1 \vdash e_1 : C_1 \qquad \Sigma_1, y : C_2, \Sigma_2 \vdash e : \Psi}{\Sigma_1, x : C_1 \supset C_2, \Sigma_2 \vdash e[\![x\, e_1/y]\!] : \Psi}\,\supset L$$

$$\frac{\Sigma, y : C \vdash e : D}{\Sigma \vdash \lambda y.\, e : C \supset D}\,\supset R$$

The variables y, y_1, y_2 in rules $\wedge L_1$, $\wedge L_2$, $\vee L$, $\supset L$, $\supset R$ must be fresh.

Figure 1: Lambda Typing Rules.

2 Admissibility of Structural Rules

Lemma 1 (Admissibility of Structural Rules)**.**

1. (Contraction) For every derivation ϕ of a typing sequent $\Sigma \oplus^d \{z : D, x : D\} \vdash e : \Psi$ there is a derivation ϕ^* of $\Sigma \oplus^d z : D \vdash e\{z/x\} : \Psi$. Secondly, for every derivation ϕ of a typing sequent $\Sigma \oplus^d R?a\langle \hat{z} : D, \hat{x} : D\rangle \vdash e : \Psi$ there is a derivation ϕ^* of $\Sigma \oplus^d R?a\langle \hat{z} : D\rangle \vdash e\{\hat{z}/\hat{x}\} : \Psi$.
2. (Strengthening) Given a derivation ϕ of $\Sigma \oplus^d x : C \vdash e : E$ such that $x \notin FV(e)$. Then, we can type e without x, i.e., there is a derivation ϕ^* of $\Sigma \vdash e : E$. Secondly, given a derivation ϕ of $\Sigma \oplus^d R?a\langle \hat{x} : C\rangle \vdash e : E$ such that $\hat{x} \notin FV(e)$. Then, we can type e without $\hat{x}$, i.e., there is a derivation ϕ^* of $\Sigma \vdash e : E$.
3. (Weakening) Given a derivation ϕ of $\Sigma \vdash e : E$. Then, for an arbitrary (valid) context extension $\Sigma \preceq \Sigma'$ there is a derivation ϕ^* of $\Sigma' \vdash e : E$.

Additionally, in each case, the size of ϕ^* (i.e., the number of rule applications in the derivation tree) is no larger than that of ϕ, and if ϕ is a cut-free derivation (i.e., without applications of the substitution rules cut_1 or cut_2), then so is ϕ^*.

Proof. The three statements are shown by a straightforward induction on the structure of the typing derivation ϕ. One observes that in each case the induction step does not involve additional typing rules. In particular, no extra substitution rules are needed to transform ϕ into ϕ^*.

Technically, the proof of contraction admissibility depends on the fact that the contraction $\{z/x\}$ commutes with the capture avoiding substitutions in the typing rules $\wedge L_i$, $\supset L$, $\forall L$, cut_1

$$\frac{\Sigma \gg_{R!a} \Gamma \vdash e : \Psi}{\Sigma, R?a\langle\hat{\Gamma}\rangle \vdash \emptyset \gg_{R!a} e[\![\hat{\Gamma}/\Gamma]\!] : \Psi}\, Ax_f$$

$$\frac{\Sigma \vdash \emptyset \gg_{R!a} e : D}{\Sigma \vdash\ !a.\, e : \exists R.D}\, \exists R \qquad \frac{\Sigma_1, R?a\langle \hat{y} : C\rangle, \Sigma_2 \vdash e : \Psi}{\Sigma_1, x : \exists R.C, \Sigma_2 \vdash \mathtt{let}\ !a.\, \hat{y} = x\ \mathtt{in}\, e : \Psi}\, \exists L$$

$$\frac{\Sigma_1 \ \gg_{R!a}\ y : C, \Sigma_2 \vdash e : \Psi}{\Sigma_1, x : \forall R.C \ \gg_{R!a}\ \Sigma_2 \vdash e[\![x@a/y]\!] : \Psi}\, \forall L \qquad \frac{\Sigma \ \gg_{R!a}\ \emptyset \vdash e : D}{\Sigma \vdash\ ?a.\, e : \forall R.D}\, \forall R$$

The scope name a in rules $\forall R$, $\exists L$, trunk variable y in $\forall L$ and branch variable $\hat{y}$ in $\exists L$ must be fresh.

Figure 2: Modal Typing Rules.

and cut_2. For instance, regarding $\forall L$ we have $e[\![x@a/y]\!]\{z/x\} = e\{z/x\}[\![z@a/y]\!]$ because z cannot be bound by e at y. We also note that admissibility of weakening involves implicit α-conversion. In general, if $\Sigma \vdash e : E$ and the context is extended to Σ', then $\Sigma' \vdash e' : E$ where $e' \equiv_\alpha e$. The reason is that for validity of Σ' all we require is that the new variables added to Σ' do not occur in Σ, i.e., that they are not free in e. Yet, they may be bound in e. Strictly, these bound occurrences need to be named apart to make the typing of e' work in Σ'. However, as we identify expressions up to $\equiv_\alpha$, this may be left implicit in the statement (3) of Lem. 1. □

3 Extensional Soundness and Completeness

Theorem 1 (Extensional Soundness and Completeness)**.** If proposition D does not contain negation or falsity, then D is a theorem of CK_n iff $\emptyset \vdash^{norm} D$.

Proof. We use a direct syntactic technique to prove equivalence of the two derivation systems. One direction (completeness) is to show that every derivation $\vdash_H E$, where $\vdash_H$ denotes Hilbert derivability in CK_n, can be simulated in $\lambda\mathsf{CK}_n$. First, we observe that there are proof terms $K_{\forall R}$ and $K_{\exists R}$ for the axiom schemes $\forall R.(A \supset B) \supset (\forall R.A \supset \forall R.B)$ and $\forall R.(A \supset B) \supset (\exists R.A \supset \exists R.B)$ respectively. The K-combinators of the Hilbert system for CK_n are the typed normal form terms

$$\begin{aligned} \mathsf{K}_{\exists R} &= \lambda x.\, \lambda z.\, \mathtt{let}\ !a.\, \hat{y} = z\ \mathtt{in}\ !a.\, (x@a)\, \hat{y} && : \forall R.(C \supset D) \supset (\exists R.C) \supset (\exists R.D) \\ \mathsf{K}_{\forall R} &= \lambda x.\, \lambda z.\, ?a.\, (x@a)(z@a) && : \forall R.\, (C \supset D) \supset (\forall R.C) \supset (\forall R.D) \end{aligned}$$

obtained from the following derivations (without terms, for conciseness):

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\emptyset \gg_{R!a} C \vdash C}\,Ax_m \quad \dfrac{}{\emptyset \gg_{R!a} D, C \vdash D}\,Ax_m}{\emptyset \gg_{R!a} C \supset D, C \vdash D}\,{\supset}L}{\forall R.\,(C \supset D) \gg_{R!a} C \vdash D}\,\forall L}{\forall R.\,(C \supset D), R?a\langle C\rangle \vdash \emptyset \gg_{R!a} D}\,Ax_f}{\forall R.\,(C \supset D), R?a\langle C\rangle \vdash \exists R.D}\,\exists R}{\forall R.\,(C \supset D), \exists R.C \vdash \exists R.D}\,\exists L}{\forall R.\,(C \supset D) \vdash (\exists R.C) \supset (\exists R.D)}\,{\supset}R}{\emptyset \vdash \forall R.\,(C \supset D) \supset (\exists R.C) \supset (\exists R.D)}\,{\supset}R$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\emptyset \gg_{R!a} C \vdash C}\,Ax_m \quad \dfrac{}{\emptyset \gg_{R!a} D, C \vdash D}\,Ax_m}{\emptyset \gg_{R!a} C \supset D, C \vdash D}\,{\supset}L}{\forall R.\,(C \supset D) \gg_{R?a} C \vdash D}\,\forall L}{\forall R.\,(C \supset D), \forall R.C \gg_{R!a} \emptyset \vdash D}\,\forall L}{\forall R.\,(C \supset D), \forall R.C \vdash \forall R.D}\,\forall R}{\forall R.\,(C \supset D) \vdash (\forall R.C) \supset (\forall R.D)}\,{\supset}R}{\emptyset \vdash \forall R.\,(C \supset D) \supset (\forall R.C) \supset (\forall R.D)}\,{\supset}R$$

Also, every theorem of minimal intuitionistic propositional logic (IPL) is derivable. This follows from the observation that the derivation rules, Fig. 1, specialised to sequents of the form $\Gamma \vdash e : E$, where Γ is a single scope, provide a complete axiomatisation of IPL. Further, the Hilbert rules of Modus Ponens and Necessitation are admissible, too, by the Structural Rules, Lem. 1, and the admissibility of cut elimination which follows from Prop. 2, Strong Normalisation Prop. 9 and Subject Reduction Prop. 6, as follows:

First, Modus Ponens, which is function application, is representable as a combinator $\mathsf{MP} = \lambda y.\, \lambda x.\, y\, x$ for which $\emptyset \vdash \mathsf{MP} : C \supset (C \supset D) \supset D$ is derivable. Then, if $\emptyset \vdash m_1 : C \supset D$ and $\emptyset \vdash m_2 : C$ we have $\emptyset \vdash \mathsf{MP}\, m_1\, m_2 : D$ by the substitution typing rule cut_1. Second, regarding Necessitation, context weakening (Lem. 1) guarantees that $\emptyset \vdash m : D$ implies $\emptyset \gg_{R?a} \emptyset \vdash m : D$ and thus $\emptyset \vdash ?a.\, m : \forall R.D$, for any role R. Hence, $\mathsf{Nec}_R\, m$ is an abbreviation for $?a.\, m$.

This proves that $\lambda\mathsf{CK}_n$ is at least as rich as CK_n.

Next, we argue the converse (soundness) direction, that $\lambda\mathsf{CK}_n$ does not include more theorems. We show that every derivation $\emptyset \vdash^{norm} E$ in the Gentzen-style system of Figs. 1 and 2 can be simulated by Hilbert to give $\vdash_H E$. We achieve this through a structural transformation of sequents into propositions and of sequent rules into Hilbert derivations. This translation is an adaptation of the interpretation suggested in [7] for a related sequent calculus. In this translation we remove all term annotations and fold up the tree structure of a sequent $\Sigma \vdash \Psi$ into a single formula. We define a wrapping translation $[\Sigma \vdash \Psi]^h$ for sequents with path contexts Σ as follows:

$$\begin{array}{rcll}
[x : C, \Sigma \vdash \Psi]^h & =_{df} & C \supset [\Sigma \vdash \Psi]^h & \\
[R?a\langle\hat{\Gamma}\rangle, \Sigma \vdash \Psi]^h & =_{df} & (\exists R.\hat{\Gamma}) \supset [\Sigma \vdash \Psi]^h & (\Sigma \text{ does not contain scope } a) \\
[\emptyset \gg_{R!a} \Sigma \vdash \Psi]^h & =_{df} & \forall R.\, [\Sigma \vdash \Psi]^h & \\
[\emptyset \vdash \emptyset \gg_{R!a} \Psi]^h & =_{df} & \exists R.[\emptyset \vdash \Psi]^h & \\
[\emptyset \vdash e : E]^h & =_{df} & E &
\end{array}$$

where $\hat{\Gamma}$ denotes the conjunction of all types in Γ. As an example consider the sequent $\Sigma \vdash e : E$ with context $\Sigma = \{x : A, y : B, R?b\langle \hat{z} : C\rangle, S!d[v : B, T?c\langle \hat{w} : D\rangle]\}$, or in path notation $x : A, y : B, R?b\langle \hat{z} : C\rangle \gg_{S!d} v : B, T?c\langle \hat{w} : D\rangle \vdash e : E$. It wraps into the proposition

$$[\Sigma \vdash e : E]^h \;\; = \;\; A \supset B \supset \exists R.C \supset \forall S.(B \supset \exists T.D \supset E),$$

where, as usual, nested implications associate to the right. One proves by induction that for every derivation $\Sigma \vdash^{norm} \Psi$ of a sequent using the rules in Figs. 1 and 2 there exists a Hilbert proof of $\vdash_H [\Sigma \vdash \Psi]^h$. Thus, if $\emptyset \vdash^{norm} e : E$ we get $\vdash_H [\emptyset \vdash e : E]^h$, which is the same as $\vdash_H E$.

In order to simulate the sequent rules, which act arbitrarily deep inside the formula structure, we need to exploit the extensionality principle of Hilbert calculus. Specifically, if $\vdash_H \phi[D_1]$ where ϕ is a *positive*[1] propositional context with a selected occurrence of a sub-proposition D_1 and $\vdash_H D_1 \supset D_2$ then $\vdash_H \phi[D_2]$. For intuitionistic propositional logic this is well known, for CK_n the two characteristic axioms $K_{\forall R} : \forall R.(A \supset B) \supset (\forall R.A \supset \forall R.B)$ and $K_{\exists R} : \forall R.(A \supset B) \supset (\exists R.A \supset \exists R.B)$ are instrumental for this. In fact, preserving this extensionality principle is the main and only job done by these axioms. It suffices to look at the modal typing rules in Fig. 2. All the lambda typing rules in Fig. 1 are handled analogously.

- The premise of Ax_f translated (essentially) gives $[\Sigma \gg_{R!a} \Gamma \vdash \Psi]^h = \phi[\forall R.(\hat{\Gamma} \supset \psi)]$ where $\psi = [\emptyset \vdash \Psi]^h$ and $\phi[\cdot]$ subsumes the (positive) context arising from the translation of the Σ prefix. Here and in the following we leave out the proof terms from the sequents since they get dropped in the translation, anyway. Now, the implication $\forall R.(\hat{\Gamma} \supset \psi) \supset (\exists R.\hat{\Gamma}) \supset \exists R.\psi$ is an instance of $K_{\exists R}$. But the formula $\phi[(\exists R.\hat{\Gamma}) \supset \exists R.\psi]$ is the translation of the Ax_f rule's conclusion. Hence, by the extensionality principle, Ax_f is admissible.

- Regarding rule $\exists R$, consider a translation of a typical premise $[\Sigma \vdash \emptyset \gg_{R!a} D]^h = \phi[\exists R.D]$. Obviously, the conclusion $\Sigma \vdash \exists R.D$ translates into exactly the same formula. Hence $\exists R$ is trivially admissible.

- The wrapping up of the premise of $\exists L$ is of the shape $[\Sigma_1, R?a\langle C\rangle, \Sigma_2 \vdash \Psi]^h = \phi[\exists R.C \supset \psi]$ where ϕ captures the prefix Σ_1 and ψ takes care of the suffix sequent $\psi = [\Sigma_2 \vdash \Psi]^h$. This is precisely the translation of the conclusion $[\Sigma_1, \exists R.C, \Sigma_2 \vdash \Psi]^h = \phi[\exists R.C \supset \psi]$.

- An application of rule $\forall L$ takes a premise which translates as $[\Sigma_1 \gg_{R!a} C, \Sigma_2 \vdash \Psi]^h = \phi[\forall R.(C \supset \psi)]$. An application of the CK_n axiom scheme $K_{\forall R} : \forall R.(A \supset B) \supset (\forall R.A \supset \forall R.B)$ and extensionality permits us to derive the translation of the conclusion $[\Sigma_1, \forall R.C \gg_{R!a} \Sigma_2 \vdash \Psi]^h = \phi[\forall R.C \supset \forall R.\psi]$.

- Finally, consider the rule $\forall R$. Suppose we have derived the wrap-up of a premise $[\Sigma \gg_{R!a} \emptyset \vdash D]^h = \phi[\forall R.D]$. Then, the translation of the conclusion can be obtained, too, because it is the same.

□

4 Characterisation of Well-typed Irreducibles

Proposition 2. Suppose $\Sigma \vdash e : \Psi$. Then e is irreducible iff $\Sigma \vdash^{norm} e : \Psi$.

[1] A context $\phi[X]$ is *positive* if X occurs in positive position, i.e., nested behind an even number of negations.

Proof. For the first part of the proposition recall (Prop. 1) that all terms typed under the rules of Fig. 1 and Fig. 2 are in *normal form*, given as

$$\begin{array}{rcl} \mathit{nf} & ::= & g \mid \mathtt{let}\,!a.\,\hat{y} = g\,\mathtt{in}\,\mathit{nf} \mid (\mathit{nf}_1, \mathit{nf}_2) \mid \iota_1\,\mathit{nf} \mid \iota_2\,\mathit{nf} \mid \\ & & \mathtt{case}\,g\,\mathtt{of}[\iota_1\,y_1 \to \mathit{nf}_1 \mid \iota_2\,y_2 \to \mathit{nf}_2] \mid \lambda x.\mathit{nf} \mid ?a.\,\mathit{nf} \mid !a.\,\mathit{nf} \\ g & ::= & x \mid \hat{x} \mid g@a \mid \pi_1\,g \mid \pi_2\,g \mid g\,\mathit{nf}. \end{array}$$

A special class of normal form terms are those of form g, called *neutral* terms. Each neutral term g consists of a sequence of destructors applied to a unique variable, called the *spine variable* of g. Every occurrence of a free variable inside nf is the spine variable of a neutral term g by which the variable is destructed to some depth, from which a sequence of constructors then builds the normal form nf. Also, every sub-expression f in a normal form $\mathit{nf}\{f/x\}$ that starts with a destructor $f = \pi_1\,g$, $f = \pi_2\,g$, $f = g@a$, $f = g\,\mathit{nf}$, $f = \mathtt{case}\,g\,\mathtt{of}[\iota_1\,y_1 \to \mathit{nf}_1 \mid \iota_2\,y_2 \to \mathit{nf}_2]$ or $f = \mathtt{let}\;!a.\,\hat{y} = g\,\mathtt{in}\,f'$ has a neutral term g as its reduction object and immediate sub-expression. This means that normal forms do not contain any redexes for either β-contractions or (commuting) γ-contractions. Thus, if $\Sigma \vdash^{norm} e : \Psi$, then e is irreducible.

For the second part of the proposition we must show that every well-typed irreducible expression can be typed with the typing rules of Figs. 1 and 2 alone, i.e., without cut_1 or cut_2. The proof proceeds by induction on the size of typing derivations (i.e., the number of rules) to show that each application of a cut_1 or cut_2 in the typing of an irreducible term can be eliminated.

Rule $[cut_1]$**.** We begin by looking at a typing tree which ends in rule cut_1, i.e., a derivation tree ϕ of the following form:

$$\frac{\begin{array}{c}\vdots\,\phi_1 \\ \Sigma\lceil d \vdash e : D\end{array} \qquad \begin{array}{c}\vdots\,\phi_2 \\ \Sigma \oplus^d x : D \vdash f : \Psi\end{array}}{\Sigma \vdash f[\![e/x]\!] : \Psi}\; cut_1\ (x \text{ fresh})$$

where $f[\![e/x]\!]$ is irreducible and ϕ_1 and ϕ_2 are the two sub-proofs on which the cut is performed. Without loss of generality we may assume that (i) the cut variable x occurs free in f and (ii) we are eliminating an innermost cut, i.e., ϕ_1 and ϕ_2 do not contain cuts themselves. We first observe that all the cases where ϕ_1 ends in one of the rules $\wedge L_i$, $\supset L$, $\forall L$ commute with the cut, so that cut elimination follows by induction hypothesis. If ϕ_1 is an application of Ax_m then $e = y$ where $y : D$ is a trunk variable in the active scope of $\Sigma\lceil d$, i.e., at depth d in Σ. Because the cut variable x is fresh, we must have $y \neq x$. Thus, $f[\![e/x]\!] = f[\![y/x]\!]$. By assumption, $\Sigma \oplus^d x : D \vdash f : \Psi$ is cut-free. Hence we can use the admissibility of contraction of Lem 1(1) to conclude that $\Sigma \vdash f[\![y/x]\!] : \Psi$ can be typed without cut. Since the end rule of ϕ_1 cannot be Ax_f it remains to treat the cases where

- ϕ_1 ends in one of the right rules $\wedge R$, $\vee R_i$, $\supset R$, $\exists R$, $\forall R$, or the left rules $\vee L$ or $\exists L$.

All these are special in the sense that they introduce the top-level operator of expression e which gets substituted. These we cannot commute with the last rule of ϕ_2, without destroying the structure of the resulting term $f[\![e/x]\!]$, unless the last rule of ϕ_2 is Ax_m. If $\phi_2 = Ax_m \cdot \phi_2'$ and $f = x$ is the cut variable then $f[\![e/x]\!] = e$ and d is the depth of Σ, i.e., $\Sigma\lceil d = \Sigma$. Then ϕ_1 is the desired typing without the cut. If $\phi_2 = Ax_m \cdot \phi_2'$ and $f = y \neq x$ is some other variable in the active scope of $\Sigma \oplus^d x : D$ then $f[\![e/x]\!] = y[\![e/x]\!] = y$ and ϕ_2 is the desired cut-free typing. If ϕ_2 ends in a right rule $\wedge R$, $\vee R_i$, $\supset R$, $\exists R$, $\forall R$ then this rule application can be pushed down across the cut, too, without changing the typed expression. Just as easy is the case where $\phi_2 = Ax_f \cdot \phi_2'$:

$$\dfrac{\begin{array}{c}\vdots\,\phi_1\\ (\Sigma', R?a\langle\hat{\Gamma}\rangle)\lceil d \vdash e : D\end{array} \qquad \dfrac{\begin{array}{c}\vdots\,\phi_2'\\ \Sigma' \oplus^d x : D \gg_{R!a} \Gamma \vdash f' : F\end{array}}{(\Sigma', R?a\langle\hat{\Gamma}\rangle) \oplus^d x : D \vdash \emptyset \gg_{R!a} f'[\![\hat{\Gamma}/\Gamma]\!] : F}\, Ax_f}{\Sigma', R?a\langle\hat{\Gamma}\rangle \vdash \emptyset \gg_{R!a} f'[\![\hat{\Gamma}/\Gamma]\!][\![e/x]\!] : F}\, cut_1$$

If needed, we can weaken ϕ_2' to a typing ϕ_2^* of $(\Sigma', R?a\langle\hat{\Gamma}\rangle) \oplus^d x : D \gg_{R!a} \Gamma \vdash f' : F$ and then apply cut_1 between ϕ_1 and ϕ_2^* which obtains $\Sigma', R?a\langle\hat{\Gamma}\rangle \gg_{R!a} \Gamma \vdash f'[\![e/x]\!] : F$ from which Ax_f generates the conclusion $\Sigma', R?a\langle\hat{\Gamma}\rangle \vdash \emptyset \gg_{R!a} f'[\![e/x]\!][\![\hat{\Gamma}/\Gamma]\!] : F$ which is what we want because $f'[\![e/x]\!][\![\hat{\Gamma}/\Gamma]\!] = f'[\![\hat{\Gamma}/\Gamma]\!][\![e/x]\!]$. The equality holds since no variable of Γ is free in e and trunk variable x is distinct from all branch variables in $\hat{\Gamma}$.

It remains to see how we get rid of cut_1 when ϕ_2 ends in a left rule. It is not difficult to show that every application of a left rule in ϕ_2 which does not involve the cut variable x can be permuted with the cut. So, it is enough to consider the cases where

- ϕ_2 ends in $\wedge L_i$, $\vee L$, $\supset L$, $\exists L$, $\forall L$ which operates on the cut variable $x : D$.

The type D now restricts the possible combination of the end rules in ϕ_1 and in ϕ_2. Specifically, we can only have ($\wedge R$, $\wedge L_i$), ($\vee R_i$, $\vee L$), ($\supset R$, $\supset L$), ($\exists R$, $\exists L$), ($\forall R$, $\forall L$) as well as all combinations of $\vee L$, $\exists L$ in ϕ_1 with any of $\wedge L_i$, $\vee L$, $\supset L$, $\exists L$, $\forall L$ for ϕ_2. Hence, we are down to 19 cases. At this point, the assumption that $f[\![e/x]\!]$ is irreducible comes into play. For none of these cases can actually occur in an irreducible expression as they would generate a redex in $f[\![e/x]\!]$.

For instance, if ϕ_1 ends in $\wedge R$ and $\phi_2 = \wedge L_i \cdot \phi_2'$ where $\wedge L_i$ works on $x : D$, we must have $D = D_1 \wedge D_2$, $e = (e_1, e_2)$ as well as $f = f'[\![\pi_i\, x/y]\!]$ with ϕ_2' being a typing of $\Sigma \oplus^d \{y : D_1, x : D\} \vdash f' : \Psi$. Thus, $f[\![e/x]\!] = (f'[\![\pi_i\, x/y]\!])[\![(e_1, e_2)/x]\!] = (f'[\![(e_1, e_2)/x]\!])[\![\pi_i(e_1, e_2)/y]\!]$. But then $f[\![e/x]\!]$ contains the β-redex $\pi_i(e_1, e_2)$ unless y is not free in f'. Yet, if this is so, the end rule $\wedge L_i$ of ϕ_2 is redundant, because $f = f'$ and thence sub-derivation ϕ_2' without $\wedge L_i$ (using admissibility of strengthening Lem 1(2)) already is a typing of $\Sigma \oplus^d x : D \vdash f : \Psi$. We can then invoke the induction hypothesis and conclude that the cut_1 between ϕ_1 typing $\Sigma\lceil d \vdash e : D$ and ϕ_2', which also types $\Sigma \vdash f[\![e/x]\!] : \Psi$, is eliminable. Essentially the same argument applies to the combinations ($\vee R_i$, $\vee L$), ($\supset R$, $\supset L$), ($\exists R$, $\exists L$), ($\forall R$, $\forall L$). They all generate β-redexes.

Now what about ϕ_1 ending in $\vee L$, $\exists L$? Then, the top-level operator of e is a `case` or `let` construct. Combine this with an application of $\wedge L_i$, $\vee L$, $\supset L$, $\exists L$ or $\forall L$ as the end rule in ϕ_2 acting on the cut variable x. We get a destructor applied to a `case` or `let` inside $f[\![e/x]\!]$. This results in one of the 10 γ-redexes of Fig. 4. For example, suppose $\phi_1 = \exists L \cdot \phi_1'$ and $\phi_2 = \exists L \cdot \phi_2'$ where $\exists L$ in ϕ_2 decomposes $x : D$ with $D = \exists R.D'$. Then, $e = \texttt{let}\ !b.\,\hat{z} = e_1\ \texttt{in}\ e_2$ and $f = \texttt{let}\ !a.\,\hat{y} = x\ \texttt{in}\ f'$ such that ϕ_2' types $\Sigma \oplus^d \{x : D, R?a\langle\hat{y} : C\rangle\} \vdash f' : \Psi$. We calculate $f[\![e/x]\!] = \texttt{let}\ !a.\,\hat{y} = (\texttt{let}\ !b.\,\hat{z} = e_1\ \texttt{in}\ e_2)\ \texttt{in}\ f'$. This is a redex for γ-contraction $\gamma\texttt{let}_5$ (see Fig. 4). The same reasoning shows that in all the other cases, too, γ-redexes would arise, if the last left rule in ϕ_2 is not redundant. But redexes for commuting contractions do not exist in $f[\![e/x]\!]$ by assumption. This completes the proof that cut_1 can be eliminated.

Rule $[cut_2]$**.** Next, we come to look at an application of cut_2. Again, let ϕ_1 and ϕ_2 be two cut-free (by virtue of the induction hypothesis) sub-proofs, then perform cut_2 on their conclusion sequents. This yields a derivation tree ϕ as follows:

$$\dfrac{\begin{array}{c}\vdots\,\phi_1\\ \Sigma\lceil d \vdash \emptyset \gg_{R!a} e : D\end{array} \qquad \begin{array}{c}\vdots\,\phi_2\\ \Sigma \oplus^d R?a\langle \hat{x} : D\rangle \vdash f : \Psi\end{array}}{\Sigma \vdash f[\![e/\hat{x}]\!] : \Psi}\; cut_2 \text{ ($\hat{x}$ fresh)}$$

We can easily interchange cut_2 with the last rules of ϕ_2, if this is a right rule $\wedge R$, $\supset R$, $\exists R$, $\forall R$, $\vee R_i$ or any of the left rules. This is also true for Ax_m because it cannot involve the cut variable $\hat{x} : D$, which is a branch not a trunk variable. Let us look at the case of $\supset L$ for a typical example. The typing derivation ϕ of such a cut looks like this

$$\dfrac{\begin{array}{c}\vdots\,\phi_1\\ \Sigma\lceil d \vdash \emptyset \gg_{R!a} e : D\end{array} \qquad \dfrac{\begin{array}{c}\vdots\,\phi_{21}\\ (\Sigma \oplus^d R?a\langle \hat{x} : D\rangle)\lceil k \vdash e_1 : C_1\end{array} \qquad \begin{array}{c}\vdots\,\phi_{22}\\ \Sigma \oplus^d R?a\langle \hat{x} : D\rangle \oplus^k z : C_2 \vdash f : \Psi\end{array}}{\Sigma \oplus^d R?a\langle \hat{x} : D\rangle \vdash f[\![y\, e_1/z]\!] : \Psi}\supset L}{\Sigma \vdash (f[\![y\, e_1/z]\!])[\![e/\hat{x}]\!] : \Psi}\; cut_2$$

where $y : C_1 \supset C_2$ is a trunk variable at depth k in $\Sigma \oplus^d R?a\langle \hat{x} : D\rangle$. Here, as with most other cuts in which $\supset L$ appears on the right branch, the cut distributes over the two sub-derivations ϕ_{21} and ϕ_{22}, to give ϕ^*_{21} and ϕ^*_{22} on which we can use the induction hypothesis because they are each smaller in size.

Suppose first that $d \leq k$, i.e., $(\Sigma \oplus^d R?a\langle \hat{x} : D\rangle)\lceil k = \Sigma\lceil k \oplus^d R?a\langle \hat{x} : D\rangle$ and $\Sigma\lceil k\lceil d = \Sigma\lceil d$. Then we can perform cut_2 between ϕ_1 and ϕ_{21} obtaining a cut-free typing ϕ^*_{21} of $\Sigma\lceil k \vdash e_1[\![e/\hat{x}]\!] : C_1$. Now consider that $(\Sigma \oplus^k z : C_2)\lceil d = \Sigma\lceil d$ (z is a trunk variable and cannot be captured as a branch) so the induction hypothesis is applicable to ϕ_1 and ϕ_{22} to build a cut-free typing ϕ^*_{22} of $\Sigma \oplus^k z : C_2 \vdash f[\![e/\hat{x}]\!] : \Psi$. Both ϕ^*_{21} and ϕ^*_{22} recombine under $\supset L$ to yield $\Sigma \vdash (f[\![e/\hat{x}]\!])[\![y(e_1[\![e/\hat{x}]\!])/z]\!] : \Psi$, this time cut-free. This is what we desire since $(f[\![e/\hat{x}]\!])[\![y(e_1[\![e/\hat{x}]\!])/z]\!] = (f[\![y\, e_1/z]\!])[\![e/\hat{x}]\!]$. Note that this syntactic equality guarantees that both $e_1[\![e/\hat{x}]\!]$ and $f[\![e/\hat{x}]\!]$ must be irreducible because $(f[\![y\, e_1/z]\!])[\![e/\hat{x}]\!]$ is irreducible. This is why the induction hypotheses apply.

The case $d > k$ is even simpler, since then $\hat{x}$ does not occur free in e_1. We only need to eliminate the cut_2 of ϕ_1 with ϕ_{22} and then apply $\supset L$ to the result. This completes the case where ϕ_2 ends in $\supset L$. The only remaining, and most interesting, case is when

- ϕ_2 ends in Ax_f.

If $\phi_2 = Ax_f \cdot \phi'_2$ and the branch $R?a\langle \hat{x} : D\rangle$ containing the cut variable is not introduced by

Ax_f,

$$\dfrac{\begin{array}{c}\vdots\,\phi_1\\ (\Sigma', S?b\langle\hat{\Gamma}\rangle)\lceil d \vdash \emptyset \gg_{R!a} e : D\end{array} \qquad \dfrac{\begin{array}{c}\vdots\,\phi_2'\\ \Sigma' \oplus^d R?a\langle\hat{x} : D\rangle \gg_{S!b} \Gamma \vdash f : \Psi'\end{array}}{(\Sigma', S?b\langle\hat{\Gamma}\rangle) \oplus^d R?a\langle\hat{x} : D\rangle \vdash \emptyset \gg_{S!b} f[\![\hat{\Gamma}/\Gamma]\!] : \Psi'}\,Ax_f}{\Sigma', S?b\langle\hat{\Gamma}\rangle \vdash \emptyset \gg_{S!b} f[\![\hat{\Gamma}/\Gamma]\!][\![e/\hat{x}]\!] : \Psi'}\,cut_2 \ (\hat{x} \text{ fresh})$$

then we have direct commutation between cut_2 and Ax_f. Again, if needed, derivation ϕ_2' can be weakened to a typing ϕ_2^* of $(\Sigma', S?b\langle\hat{\Gamma}\rangle) \oplus^d R?a\langle\hat{x} : D\rangle \gg_{S!b} \Gamma \vdash f : \Psi'$. Considering that d is at most the depth of Σ', one can show that $(\Sigma', S?b\langle\hat{\Gamma}\rangle \gg_{S!b} \Gamma)\lceil d = (\Sigma', S?b\langle\hat{\Gamma}\rangle)\lceil d$, so that cut_2 combines ϕ_1 and ϕ_2^* to obtain $\Sigma', S?b\langle\hat{\Gamma}\rangle \gg_{S?b} \Gamma \vdash f[\![e/\hat{x}]\!] : \Psi'$ from which an application of Ax_f gives us $\Sigma', S?b\langle\hat{\Gamma}\rangle \vdash \emptyset \gg_{S!b} f[\![e/\hat{x}]\!][\![\hat{\Gamma}/\Gamma]\!] : \Psi'$. Since $\hat{x}$ is different from all $\hat{\Gamma}$ and no variable from Γ can be free in e this expression is the same as $f[\![\hat{\Gamma}/\Gamma]\!][\![e/\hat{x}]\!]$.

If Ax_f introduces the cut variable, then d is the depth of Σ, $\Sigma\lceil d = \Sigma$ where ϕ_2' types $\Sigma \gg_{R!a} x : D \vdash f : \Psi'$ with $\Psi = \emptyset \gg_{R!a} \Psi'$. Here we apply the Scope Shift Prop. 5 to ϕ_1 which gives $\Sigma = \Sigma', R?a\langle\hat{\Gamma}\rangle$ and a derivation ϕ_1^* of $\Sigma \gg_{R!a} \Gamma \vdash e' : D$ with $e = e'[\![\hat{\Gamma}/\Gamma]\!]$. Close inspection of the proof of Prop. 5 shows that for this "transformation" from ϕ_1 to ϕ_1^* we do not need to introduce any cut rules and also do not increase the size of the derivation. Since no trunk variable of Γ occurs in Σ, we can use Lem. 1(3) to weaken ϕ_2' to derive a typing ϕ_2^* of $\Sigma \gg_{R!a} x : D, \Gamma \vdash f : \Psi'$ of size not greater than ϕ_2'. Using the induction hypothesis for cut_1 on ϕ_1^* and ϕ_2^* we construct a cut free typing derivation of $\Sigma \gg_{R!a} \Gamma \vdash f[\![e'/\hat{x}]\!] : \Psi'$ from where an application of Ax_f takes us back to the original typing $\Sigma \vdash f[\![e'/\hat{x}]\!][\![\hat{\Gamma}/\Gamma]\!] : \Psi$ in view of $\Psi = \emptyset \gg_{R!a} \Psi'$ and $f[\![e'/\hat{x}]\!][\![\hat{\Gamma}/\Gamma]\!] = f[\![\hat{\Gamma}/\Gamma]\!][\![e/\hat{x}]\!]$. This completes the proof of elimination for cut_2 and thus the proof of Prop. 2. □

5 Destructor Completeness

Proposition 3 (Destructor Completeness)**.** 1. $\Sigma \vdash \texttt{let}\ !a.\,\hat{y} = e\,\texttt{in}\,f : F$ iff for some depth k of Σ, $\Sigma\lceil k \vdash e : \exists R.D$ and $\Sigma \oplus^k R?a\langle\hat{y} : D\rangle \vdash f : F$.

2. $\Sigma \vdash e@a : E$ iff Σ is of the form $\Sigma = \Sigma' \gg_{R!a} \Gamma$ and $\Sigma' \vdash e : \forall R.E$.

3. $\Sigma \vdash e_1\, e_2 : E$ iff $\Sigma \vdash e_1 : D \supset E$ and $\Sigma \vdash e_2 : D$ for some type D.

4. $\Sigma \vdash \pi_1\, e : E$ iff $\Sigma \vdash e : E \wedge D$ for some type D.

5. $\Sigma \vdash \pi_2\, e : E$ iff $\Sigma \vdash e : D \wedge E$ for some type D.

6. $\Sigma \vdash \iota_1\, e : F$ iff $F = D \vee E$ for some types D, E and $\Sigma \vdash e : D$.

7. $\Sigma \vdash \iota_2\, e : F$ iff $F = D \vee E$ for some types D, E and $\Sigma \vdash e : E$.

Proof. The ($\Rightarrow$) direction of all claims are obtained by induction on the structure of typing derivations and the ($\Leftarrow$) direction uses the typing rules including the cut rules to build the desired typing. We show the inversion for the modal destructors and function application, cases (1)-(3) of Prop. 3. The other cases (4)-(7) for projections and injections are standard.

(1) $\exists R.D$. We start with the ($\Rightarrow$) direction of the statement. Let ϕ be a typing derivation of form $\Sigma \vdash \texttt{let}\ !a.\,\hat{y} = e\,\texttt{in}\,f : F$. We show that there is some depth d such that $\Sigma{\restriction}d \vdash e : \exists R.D$ and $\Sigma \oplus^d R?a\langle \hat{y} : D\rangle \vdash f : F$. The argument proceeds by induction on ϕ. Obviously, ϕ cannot end in any of the rules Ax_f, Ax_m, $\supset R$, $\wedge R$, $\vee R_i$, $\vee L$, $\exists R$ or $\forall R$. What we need to check are the rules $\exists L$, $\supset L$, $\wedge L_i$, $\forall L$ and cut_1, cut_2.

• If $\phi = \exists L \cdot \phi_1$ the statement is trivial since then $e = x$ is a variable, $\Sigma = \Sigma_1, x : \exists R.D, \Sigma_2$ and ϕ_1 types $\Sigma_1, x : \exists R.D, R?a\langle \hat{y} : D\rangle, \Sigma_2 \vdash f : F$. Clearly, $\Sigma{\restriction}d \vdash x : \exists R.D$ by Ax_m where d is the depth of x in Σ.

• The cases $\supset L$, $\wedge L_i$, $\forall L$ are easy to argue as they commute with the inversion. Consider $\supset L$ in detail, $\phi = \forall L \cdot \phi_1$ and $\phi = \wedge L_i \cdot \phi_1$ are handled in the same fashion:

$$\dfrac{\begin{array}{c}\vdots\,\phi_1\\ \Sigma{\restriction}k \vdash e_1 : C_1\end{array} \qquad \begin{array}{c}\vdots\,\phi_2\\ \Sigma \oplus^k z : C_2 \vdash \texttt{let}\ !a.\,\hat{y} = e'\,\texttt{in}\,f' : F\end{array}}{\Sigma \vdash \texttt{let}\ !a.\,\hat{y} = e'[\![x\,e_1/z]\!]\,\texttt{in}\,f'[\![x\,e_1/z]\!] : F} \supset L \quad (z \text{ fresh})$$

where $e = e'[\![x\,e_1/z]\!]$, $f = f'[\![x\,e_1/z]\!]$ and $x : C_1 \supset C_2$ is a trunk variable at depth k in Σ. One can show that a and $\hat{y}$ must be fresh (in ϕ_2) for Σ and thus for $\Sigma{\restriction}k$. Therefore, they cannot be free in e_1, so that $(\texttt{let}\ !a.\,\hat{y} = e'\,\texttt{in}\,f')[\![x\,e_1/z]\!] = \texttt{let}\ !a.\,\hat{y} = e'[\![x\,e_1/z]\!]\,\texttt{in}\,f'[\![x\,e_1/z]\!]$. By induction hypothesis on ϕ_2, there are typings ϕ_{21} of $(\Sigma \oplus^k z : C_2){\restriction}d \vdash e' : \exists R.D$ and ϕ_{22} of $(\Sigma \oplus^k z : C_2) \oplus^d R?a\langle \hat{y} : D\rangle \vdash f' : F$ at some depth d. Note that the context of ϕ_{22} can be rearranged as $(\Sigma \oplus^k z : C_2) \oplus^d R?a\langle \hat{y} : D\rangle = (\Sigma \oplus^d R?a\langle \hat{y} : D\rangle) \oplus^k z : C_2$ which prepares ϕ_{22} for application with $\supset L$.

1. If $k \leq d$, then $(\Sigma \oplus^k z : C_2){\restriction}d = \Sigma{\restriction}d \oplus^k z : C_2$ as well as $\Sigma{\restriction}d{\restriction}k = \Sigma{\restriction}k$. Thus, the typings ϕ_1 and ϕ_{21} can be combined by $\supset L$ obtaining $\Sigma{\restriction}d \vdash e : \exists R.D$.
2. If $d < k$ then $(\Sigma \oplus^k z : C_2){\restriction}d = \Sigma{\restriction}d$ and e', according to ϕ_{21} does not contain z free. It follows $e = e'$ and ϕ_{21} is the required typing $\Sigma{\restriction}d \vdash e : \exists R.D$ for e.

This extracts the typing for e. Regarding the typing for f we argue as follows:

1. If $d \leq k$ then $(\Sigma \oplus^d R?a\langle \hat{y} : D\rangle){\restriction}k = \Sigma{\restriction}k \oplus^d R?a\langle \hat{y} : D\rangle$. Combining ϕ_1, weakened by $R?a\langle \hat{y} : D\rangle$ at depth d (Lem. 1(3)), with ϕ_{22} under $\supset L$ then yields $\Sigma \oplus^d R?a\langle \hat{y} : D\rangle \vdash f : F$ as required.
2. If $k < d$ then $(\Sigma \oplus^d R?a\langle \hat{y} : D\rangle){\restriction}k = \Sigma{\restriction}k$. Now we can directly combine ϕ_1 with ϕ_{22} under $\supset L$ to infer $\Sigma \oplus^d R?a\langle \hat{y} : D\rangle \vdash f : F$.

• The end rules cut_1 and cut_2 are not difficult either. Let us begin with cut_1 which gives subderivations ϕ_1 of $\Sigma{\restriction}k \vdash g_1 : C$ and ϕ_2 for $\Sigma \oplus^k x : C \vdash g_2 : F$ with $g_2[\![g_1/x]\!] = \texttt{let}\ !a.\,\hat{y} = e\,\texttt{in}\,f$.

A special case of this is where $g_2 = x$ and $g_1 = \texttt{let}\ !a.\,\hat{y} = e\,\texttt{in}\,f$. For the typing ϕ_2 to be valid, variable x must be a (trunk) variable in the active scope of $\Sigma \oplus^k x : C$ which means that k is identical to the depth of Σ, whence $\Sigma{\restriction}k = \Sigma$. The statement then follows directly from the induction hypothesis applied to ϕ_1. So, suppose $g_2 \neq x$, i.e., $g_2 = \texttt{let}\ !a.\,\hat{y} = e'\,\texttt{in}\,f'$ with $e'[\![g_1/x]\!] = e$ and $f'[\![g_1/x]\!] = f$. We invoke the induction hypothesis on ϕ_2. This gives a depth d within context $\Sigma \oplus^k x : C$ such that $(\Sigma \oplus^k x : C){\restriction}d \vdash e' : \exists R.C$ and $(\Sigma \oplus^k x : C) \oplus^d R?a\langle \hat{y} : D\rangle \vdash f' : F$. The latter can directly be combined with ϕ_1, possibly weakened (Lem. 1(3)), by

cut_1 to give $\Sigma \oplus^d R?a\langle\hat{y} : D\rangle \vdash f'[\![g_1/x]\!] : F$ or in fact $\Sigma \oplus^d R?a\langle\hat{y} : D\rangle \vdash f : F$. Regarding the typing of e we make a case distinction: If $d \geq k$, we can form cut_1 between ϕ_1 and the typing $(\Sigma \oplus^k x : C)\lceil d \vdash e' : \exists R.C$ to get $\Sigma\lceil d \vdash e'[\![g_1/x]\!] : \exists R.C$. This is what we need. If $d < k$ then $(\Sigma \oplus^k x : C)\lceil d = \Sigma\lceil d$ and thus e' does not have x free. Then, $e = e'$ and we are done.

- Finally, regarding cut_2 we are looking at sub-typings ϕ_1 of $\Sigma\lceil k \vdash \emptyset \gg_{S!c} g_1 : C$ and ϕ_2 for $\Sigma \oplus^k S?c\langle\hat{x} : C\rangle \vdash g_2 : F$ with $g_2[\![g_1/\hat{x}]\!] = \mathtt{let}\ !a.\,\hat{y} = e\,\mathtt{in}\,f$. Here we can immediately exclude the case $g_2 = \hat{x}$ because the typing $\Sigma \oplus^k S?c\langle\hat{x} : C\rangle \vdash \hat{x} : F$ would require that $\hat{x}$ is a trunk variable in the active scope of the context, while in fact it is a branch variable (at depth $k+1$). Thus, $g_2 \neq \hat{x}$ and $g_2 = \mathtt{let}\ !a.\,\hat{y} = e'\,\mathtt{in}\,f'$ with $e'[\![g_1/\hat{x}]\!] = e$ and $f'[\![g_1/\hat{x}]\!] = f$.

The induction hypothesis on ϕ_2 yields a depth d in context $\Sigma \oplus^k S?c\langle\hat{x} : C\rangle$ together with a typing derivation ϕ_{21} for

$$(\Sigma \oplus^k S?c\langle\hat{x} : C\rangle)\lceil d \vdash e' : \exists R.D$$

and ϕ_{22} of $\Sigma \oplus^k S?c\langle\hat{x} : C\rangle \oplus^d R?a\langle\hat{y} : D\rangle \vdash f' : F$. It is possible to use cut_2 with ϕ_1 (suitably weakened by $R?a\langle\hat{y} : D\rangle$ if $d \leq k$) on ϕ_{22}, with the result $\Sigma \oplus^d R?a\langle\hat{y} : D\rangle \vdash f'[\![g_1/\hat{x}]\!] : F$ which is $\Sigma \oplus^d R?a\langle\hat{y} : D\rangle \vdash f : F$ as desired.

Given $d \geq k$ it holds that $(\Sigma \oplus^k R?a\langle\hat{x} : C\rangle)\lceil d = \Sigma\lceil d \oplus^k R?a\langle\hat{x} : C\rangle$ and $\Sigma\lceil d\lceil k = \Sigma\lceil k$. The application of cut_2 on ϕ_1 with ϕ_{21} produces $\Sigma\lceil d \vdash e'[\![g_1/\hat{x}]\!] : \exists R.D$. But this is exactly the typing $\Sigma\lceil d \vdash e : \exists R.D$ that we want. What if $d < k$? Then, e' cannot have branch variable $\hat{x}$ free, whence $e' = e$ and typing ϕ_{21} already amounts to $\Sigma\lceil d \vdash e : \exists R.D$.

It remains to argue the converse direction ($\Leftarrow$) of the statement. To this end suppose there is some depth d such that $\Sigma\lceil d \vdash e : \exists R.D$ and $\Sigma \oplus^d R?a\langle\hat{y} : D\rangle \vdash f : F$. First, we can weaken (Lem. 1(3)) the typing of f to $\Sigma \oplus^d \{x : \exists R.D, R?a\langle\hat{y} : D\rangle\} \vdash f : F$ for a suitable fresh variable x, and then apply $\exists L$ to construct $\Sigma \oplus^d x : \exists R.D \vdash \mathtt{let}\ !a.\,\hat{y} = x\,\mathtt{in}\,f : F$. Finally, we invoke cut_1 with $\Sigma\lceil d \vdash e : \exists R.D$ and get $\Sigma \vdash \mathtt{let}\ !a.\,\hat{y} = e\,\mathtt{in}\,f : F$ as desired.

Case (2) $\forall R.E$**.** First consider the ($\Rightarrow$) direction, assuming a derivation ϕ of $\Sigma \vdash e@a : E$. We show that $\Sigma = \Sigma' \gg_{R!a} \Gamma$ and $\Sigma' \vdash e : \forall R.E$. We proceed by induction on the structure of the typing derivation. Clearly, the last rule cannot be Ax_f or any of the right rules $\forall R$, $\exists R$, $\supset R$, $\wedge R$, $\vee R_i$ nor the left rule $\exists L$ or $\vee L$. We only need to treat the end rules $\supset L$, $\wedge L_i$, $\forall L$, cut_1 and cut_2.

- If $\phi = \supset L\cdot(\phi_1, \phi_2)$ then ϕ_1 types $\Sigma\lceil k \vdash e_1 : C_1$ and ϕ_2 is a typing for $\Sigma \oplus^k w : C_2 \vdash e' : E$ such that $e'[\![y\,e_1/w]\!] = e@a$ for some trunk variable $y : C_1 \supset C_2$ at depth k in Σ. The trivial case $e' = w$ is excluded since always $y\,e_1 \neq e@a$. We must have $e' = e''@a$ and $e''[\![y\,e_1/w]\!] = e$. The induction hypothesis applied to ϕ_2 guarantees that $\Sigma \oplus^k w : C_2 = \Sigma' \gg_{R!a} \Gamma$ and $\Sigma' \vdash e'' : \forall R.E$. We observe that $\Sigma' \gg_{R!a} \Gamma = (\Sigma'' \gg_{R!a} \Gamma') \oplus^k w : C_2$ and $\Sigma = \Sigma'' \gg_{R!a} \Gamma'$. We claim that $\Sigma'' \vdash e : \forall R.E$. But this follows directly by assumption if $w \notin FV(e'')$, because then $\Sigma' = \Sigma''$ and $e = e''$, or by applying $\supset L$ to ϕ_1 and the typing $\Sigma' \vdash e'' : \forall R.E$ if $w \in FV(e'')$, since then $w\ \varepsilon^k\ \Sigma'$, $\Sigma' = \Sigma'' \oplus w : C_2$ and $\Sigma''\lceil k = \Sigma\lceil k$.

- The next cases are $\phi = \wedge L_i\cdot\phi_1$ and $\phi = \forall L\cdot\phi_1$. We only treat the latter, as the former proceeds analogously. Suppose, then, ϕ_1 types $\Sigma \oplus^{k+1} w : C \vdash e' : E$ where $\Sigma = \Sigma_1, y : \forall S.C \gg_{S!b} \Sigma_2$ with trunk variable y at level k and $e@a = e'[\![y@b/w]\!]$.

The matching may come about because $e' = w$, where we have $e = y$ and $a = b$. Also, the typing ϕ_1 turns into $\Sigma \oplus^{k+1} w : C \vdash w : E$ which means that $E = C$ as well as that Σ has depth $k+1$ and $\Sigma_2 = \Gamma$ is of depth 0. Then, obviously, $\Sigma = \Sigma' \gg_{S!a} \Gamma$ with $\Sigma' = \Sigma_1, y : \forall S.C$ and $\Sigma' \vdash y : \forall S.C$ by rule Ax_m as desired.

The other possibility is that $e@a = e'[\![y@b/w]\!]$ holds because there is an expression e'' with $e' = e''@a$ and $e = e''[\![y@b/w]\!]$. Then ϕ_1 types $\Sigma \oplus^{k+1} w : C \vdash e''@a : E$. Applying the induction hypothesis yields $\Sigma \oplus^{k+1} w : C = \Sigma'' \gg_{R!a} \Gamma$ and $\Sigma'' \vdash e'' : \forall R.E$. We can extract the trunk variable $w : C$ at level $k+1$ in $\Sigma'' \gg_{R!a} \Gamma$ and find Σ''', Γ' so that $\Sigma'' \gg_{R!a} \Gamma = (\Sigma''' \gg_{R!a} \Gamma') \oplus^{k+1} w : C$ and also $\Sigma = \Sigma''' \gg_{R!a} \Gamma'$.

If $w \notin FV(e'')$, then $e = e''$ and we can strengthen (Lem. 1(2)) the typing $\Sigma'' \vdash e'' : \forall R.E$ to $\Sigma''' \vdash e'' : \forall R.E$. Thus, we are done. If $w \in FV(e'')$, then w must appear at depth $k+1$ in Σ'', i.e., $\Sigma'' = \Sigma''' \oplus^{k+1} w : C$ and $\Gamma' = \Gamma$. We still have variable $y : \forall S.C$ at depth k in Σ''' since it occurs in Σ at depth k. Therefore, we can apply $\forall L$ to $\Sigma'' \vdash e'' : \forall R.E$ to derive $\Sigma''' \vdash e''[\![y@b/w]\!] : \forall R.E$ or rather $\Sigma''' \vdash e : \forall R.E$.

• If ϕ ends in cut_1 with sub-derivations ϕ_1 and ϕ_2, then $e@a = g_2[\![g_1/y]\!]$ such that ϕ_1 types $\Sigma\lceil k \vdash g_1 : C$ and ϕ_2 is a typing of $\Sigma \oplus^k y : C \vdash g_2 : E$.

The operator @ is either part of g_2 or introduced with the substitution of g_1 for y. In the latter case we must have $g_1 = e@a$ as well as $g_2 = y$, $E = C$, and Σ is of depth k. Thus, $\Sigma\lceil k = \Sigma$ which means our claim follows from the induction hypothesis applied to ϕ_1.

If however @ is part of g_2 then $g_2 = g_2'@a$, $e = g_2'[\![g_1/y]\!]$ and the typing ϕ_2 becomes $\Sigma \oplus^k y : C \vdash g_2'@a : E$. The induction hypothesis can now be applied to ϕ_2. It tells us that $\Sigma \oplus^k y : C = \Sigma' \gg_{R!a} \Gamma$ and $\Sigma' \vdash g_2' : \forall R.E$. As before we extract the trunk variable $y : C$ from Σ' and Γ so that $\Sigma' \gg_{R!a} \Gamma = (\Sigma'' \gg_{R!a} \Gamma') \oplus^k y : C$ and $\Sigma = \Sigma'' \gg_{R!a} \Gamma'$. If $y \notin FV(g_2')$ then $e = g_2'$ and the typing $\Sigma' \vdash g_2' : \forall R.E$ may be strengthened (Lem. 1(2)) to $\Sigma'' \vdash g_2' : \forall R.E$, i.e., $\Sigma'' \vdash e : \forall R.E$, which completes the proof. If $y \in FV(g_2')$ then y appears at depth k in Σ', specifically $\Sigma' = \Sigma'' \oplus^k y : C$ as well as $\Sigma\lceil k = (\Sigma'' \gg_{R!a} \Gamma')\lceil k = \Sigma''\lceil k$. This means that cut_1 with ϕ_1 is applicable to the typing $\Sigma' \vdash g_2' : \forall R.E$ to yield $\Sigma'' \vdash g_2'[\![g_1/y]\!] : \forall R.E$ which is nothing but $\Sigma'' \vdash e : \forall R.E$ as required.

• The last case to treat is where $\phi = cut_2 \cdot (\phi_1, \phi_2)$ with $e@a = g_2[\![g_1/\hat{y}]\!]$ such that ϕ_1 types $\Sigma\lceil k \vdash \emptyset \gg_{S!b} g_1 : C$ and ϕ_2 is a typing of $\Sigma \oplus^k S?b\langle \hat{y} : C\rangle \vdash g_2 : E$. Note that $g_2 \neq \hat{y}$ since there is no typing $\Sigma \oplus^k S?b\langle \hat{y} : C\rangle \vdash \hat{y} : E$. Since $g_2 \neq \hat{y}$ we have $g_2 = g_2'@a$, $e = g_2'[\![g_1/\hat{y}]\!]$ and the typing ϕ_2 becomes $\Sigma \oplus^k S?b\langle \hat{y} : C\rangle \vdash g_2'@a : \Psi$. It is not difficult, then, to obtain the statement by induction hypothesis on typing ϕ_2. The proof works exactly like in case of cut_1.

Finally we come to tackle the ($\Leftarrow$) direction of the statement. Suppose, $\Sigma = \Sigma' \gg_{R!a} \Gamma$ and $\Sigma' \vdash e : \forall R.E$. Clearly, $\Sigma', x : \forall R.E \gg_{R!a} y : E, \Gamma \vdash y : E$ by Ax_m from which an application of $\forall L$ gives $\Sigma', x : \forall R.E \gg_{R!a} \Gamma \vdash x@a : E$. Into this derivation we can cut_1 the well-typed expression $\Sigma' \vdash e : \forall R.E$ and get $\Sigma \vdash e@a : E$. This proves the claim and completes the proof of Case (2).

Case (3) $D \supset E$**.** The direction ($\Leftarrow$) is trivial. For if $\Sigma \vdash e_1 : D \supset E$ and $\Sigma \vdash e_2 : D$ then we take the derivation $\supset\!L \cdot (Ax_m, Ax_m)$ for $\Sigma, x : D \supset E, y : D \vdash x\,y : E$ and cut_1 with the typings of e_1, e_2, suitably weakened (Lem. 1(3)), to obtain $\Sigma \vdash e_1\,e_2 : E$.

Now we address the ($\Rightarrow$) direction which is the more difficult part. Assume we have a derivation of $\Sigma \vdash e_1\,e_2 : E$. The end rule cannot be Ax_f, Ax_m, $\supset\!R$, $\wedge R$, $\vee R_i$, $\vee L$, $\forall R$, $\exists R$ or $\exists L$ but must be one of $\supset\!L$, $\wedge L_i$, $\forall L$, cut_1, cut_2. Of these we only present $\supset\!L$ and $\forall L$. The other cases are analogous.

- Suppose the derivation is $\phi = \,\supset\!L \cdot (\phi_1, \phi_2)$ where ϕ_1 types $\Sigma\lceil k \vdash g_1 : C_1$ while ϕ_2 is a typing derivation of $\Sigma \oplus^k w : C_2 \vdash g_2 : E$ for some trunk variable $y : C_1 \supset C_2$ at depth k in Σ, and moreover $e_1\,e_2 = g_2[\![y\,g_1/w]\!]$.

The easy case is where $g_2 = w$, so that $e_1\,e_2 = y\,g_1$, or rather $e_1 = y$ and $e_2 = g_1$. Now, typing ϕ_2 specialises to $\Sigma \oplus^k w : C_2 \vdash w : E$ which implies that $E = C_2$ and w is a variable in the active scope, i.e. Σ has depth k. But then also y is a variable in the active scope of Σ and we trivially have $\Sigma \vdash y : C_1 \supset C_2$ on the one hand and $\Sigma \vdash g_1 : C_1$, since $\Sigma\lceil k = \Sigma$. This is what we are after.

The inductive case is when g_2 is not the variable w but when the matching $e_1\,e_2 = g_2[\![y\,g_1/w]\!]$ holds because $g_2 = g_{21}\,g_{22}$ and $e_1 = g_{21}[\![y\,g_1/w]\!]$, $e_2 = g_{22}[\![y\,g_1/w]\!]$. Here we invoke the induction hypothesis on the typing derivation ϕ_2 for g_2. This gives us $\Sigma \oplus^k w : C_2 \vdash g_{21} : D \supset E$ and $\Sigma \oplus^k w : C_2 \vdash g_{22} : D$ for some type D. Now we may apply $\supset\!L$ with ϕ_1 to both derivations to get $\Sigma \vdash e_1 : D \supset E$ and $\Sigma \vdash e_2 : D$ as desired.

- If the derivation is $\phi = \forall L \cdot \phi_1$, the sub-derivation ϕ_1 is of the shape $\Sigma \oplus^{k+1} y : C \vdash e : E$ where $\Sigma = \Sigma_1, x : \forall S.C \gg_{S!b} \Sigma_2$ with trunk variable x at depth k and $e_1\,e_2 = e[\![x@b/y]\!]$. Obviously, the expression e cannot be simply the variable y. Rather, $e = e_1'\,e_2'$ with $e_1 = e_1'[\![x@b/y]\!]$ and $e_2 = e_2'[\![x@b/y]\!]$. We use the induction hypothesis on ϕ_1 which gives us derivations $\Sigma \oplus^{k+1} y : C \vdash e_1' : D \supset E$ and $\Sigma \oplus^{k+1} y : C \vdash e_2' : D$. To both of these typings the rule $\forall L$ is applicable, obtaining $\Sigma \vdash e_1 : D \supset E$ and $\Sigma \vdash e_2 : D$. This is what we need. □

6 Constructor Completeness

Proposition 4 (Constructor Completeness)**.**

1. $\Sigma \vdash\, !a.\, e : F$ iff $F = \exists R.E$ and $\Sigma \vdash \emptyset \gg_{R!a} e : E$.
2. $\Sigma \vdash\, ?a.\, e : F$ iff $F = \forall R.E$ and $\Sigma \gg_{R!a} \emptyset \vdash e : E$.
3. $\Sigma \vdash \lambda y.e : F$ iff $F = D \supset E$ and $\Sigma, y : D \vdash e : E$.
4. $\Sigma \vdash (e_1, e_2) : F$ iff $F = D \wedge E$ and $\Sigma \vdash e_1 : D$ and $\Sigma \vdash e_2 : E$.
5. $\Sigma \vdash \texttt{case}\ e\ \texttt{of}\ [\iota_1\,x_1 \to e_1 \mid \iota_2\,x_2 \to e_2] : F$ iff for some depth k of Σ we have $\Sigma\lceil k \vdash e : D \vee E$ and $\Sigma \oplus^k x_1 : D \vdash e_1 : F$ and $\Sigma \oplus^k x_2 : D \vdash e_2 : F$.

Proof. The proof proceeds by induction on the structure of typing derivations and uses the cut rules. As before, only the modal constructors and function abstraction, cases (1)-(3) of Prop. 4 are discussed. The other cases (4)-(5) for pairing and case analysis are standard.

Case (1) $\exists R.E$**.** The ($\Leftarrow$) direction of the statement is trivial since it is the same as rule $\exists R$. For the ($\Rightarrow$) direction suppose ϕ is the derivation of $\Sigma \vdash\, !a.\, e : F$. We claim that $F = \exists R.D$ and $\Sigma \vdash \emptyset \gg_{R!a} e : D$. Clearly, the last rule of ϕ cannot be Ax_f, Ax_m, $\exists L$, $\forall R$, $\supset R$, $\wedge R$, $\vee R_i$ or $\vee L$. If the last rule is $\exists R$ the claim is immediate. All the left rules $\supset L$, $\wedge L_i$, $\forall L$ as well as the cut rules, which are generic in the typing statement, easily commute with the inversion.

Case (2) $\forall R.E$**.** The direction ($\Leftarrow$) from $\Sigma \gg_{R!a} \emptyset \vdash e : E$ to $\Sigma \vdash\, ?a.\, e : \forall R.E$ is precisely the rule $\forall R$. For ($\Rightarrow$) we proceed by induction on the last rule of the typing derivation $\Sigma \vdash\, ?a.\, e : F$. This can only be $\forall L$, $\supset L$, $\wedge L_i$ or $\forall R$ and the cut rules cut_1, cut_2. If it is $\forall R$, then the claim follows immediately. The rules $\forall L$, $\supset L$ and $\wedge L_i$ commute with the statement of the claim and thus can be obtained by induction hypothesis. The same applies to cut_1 and cut_2 since both are generic in the depth of the sequents. The extension of the context from Σ to $\Sigma \gg_{R!a} \emptyset$ in the inversion does not prevent the application of the cut rules. Thus, they follow by induction hypothesis, too.

Case (3) $D \supset E$**.** Again, the direction ($\Leftarrow$) is simply the rule $\supset R$. To argue the converse direction ($\Rightarrow$) suppose we are given a typing derivation ϕ of $\Sigma \vdash \lambda y.e : F$. The end rules Ax_f, Ax_m, $\forall R$, $\exists R$, $\exists L$, $\wedge R$, $\vee R_i$ and $\vee L$ are excluded. If the last rule of ϕ is $\supset R$ then we are done immediately. The rules $\forall L$, $\supset L$ and $\wedge L_i$ commute with the inversion, too, and hence they follow by induction hypothesis. Similarly, the end rules cut_1, cut_2 can be pushed up the tree as they remain applicable under extension of the active scope.

Observe that if $\lambda y.e = g_2[\![g_1/\hat{x}]\!]$ arises from an application of cut_2, then necessarily $g_2 \neq \hat{x}$, because $\hat{x}$ is a branch variable while g_2 has a trunk type. This means the constructor λy is not substituted by g_1, but rather $g_2 = \lambda y.e'$ with $e = e'[\![g_1/\hat{x}]\!]$. Thus, we can use the induction hypothesis for the trunk typing $\Sigma \oplus^k R?a\langle \hat{x} : C\rangle \vdash \lambda y.e' : F$ rather than a branch type $\Sigma\lceil k \vdash \emptyset \gg_{R!a} \lambda y.e : C$, for which we have not stated it. □

7 Scope Shift

Proposition 5 (Scope Shift)**.** $\Sigma \vdash \emptyset \gg_{R!a} e : E$ iff $\Sigma = \Sigma', R?a\langle\hat{\Gamma}\rangle$ and $\Sigma \gg_{R!a} \Gamma \vdash e' : E$ for some e' with $e = e'[\![\hat{\Gamma}/\Gamma]\!]$.

Proof. For ease of notation let us abbreviate the substitution $e[\![\hat{\Gamma}/\Gamma]\!]$ by $e|_\Gamma$ for any set of typed trunk variables Γ. Observe that the ($\Leftarrow$) direction of Prop. 5 is simply an application of rule Ax_f (notice the implicit contraction permitted by the rule). For the ($\Rightarrow$) direction we show by induction on derivations that for all typings ϕ of $\Sigma \vdash \emptyset \gg_{R!a} e : E$ the context Σ must contain a branch $R?a\langle\hat{\Gamma}\rangle$ in the active scope, i.e., $\Sigma = \Sigma', R?a\langle\hat{\Gamma}\rangle$ and that ϕ can be reduced to a typing derivation ϕ^* for $\Sigma \gg_{R!a} \Gamma \vdash e' : E$ of size not larger than that of ϕ such that $e = e'|_\Gamma$. Moreover, ϕ^* is cut-free whenever ϕ is.

The proof is based on the observation that the branch typing $\emptyset \gg_{R!a} e : E$ must have been introduced by rule Ax_f which depends on such a branch $R?a\langle\hat{\Gamma}\rangle$ in the active scope and none of the left rules applicable to branch typings destroys it. In addition, all the left rules remain applicable after the scope shift $\Sigma \gg_{R!a} \Gamma \vdash e' : E$. Note that we can assume, without loss of generality, that $\hat{\Gamma}$ includes *all* branch variables of scope a in Σ (by weakening Lem. 1(3)).

Consider the last rule of the typing ϕ for $\Sigma \vdash \emptyset \gg_{R!a} e : E$, which can only be Ax_f, one of the left rules $\wedge L_i$, $\vee L$, $\supset L$, $\exists L$, $\forall L$ or one of the cut rules cut_1, cut_2. If the last rule is Ax_f then the statement follows immediately, $\phi = Ax_f \cdot \phi^*$ and ϕ^* is strictly smaller. Similarly, all of the left rules $\wedge L_i$, $\vee L$, $\supset L$, $\forall L$ permute with the scope shift, because (i) they do not involve the branch variables $R?a\langle\hat{\Gamma}\rangle$ and (ii) they remain applicable after the context extension from Σ to $\Sigma \gg_{R!a} \Gamma$.

As an example for how left rules permute with the scope shift, suppose $\Sigma \vdash \emptyset \gg_{R!a} e : E$ is derived by $\supset L$, i.e., there is a derivation ϕ_1 of $\Sigma\lceil k \vdash e_1 : C_1$ and ϕ_2 of $\Sigma \oplus^k z : C_2 \vdash \emptyset \gg_{R!a} e' : E$ such that $e = e'[\![y\, e_1/z]\!]$ and $y : C_1 \supset C_2$ is a trunk variable in Σ at depth k. By induction hypothesis we know that $\Sigma = \Sigma', R?a\langle\hat{\Gamma}\rangle$ and ϕ_2 can be transformed (without size increase or additional cut rules) into ϕ_2^* for $\Sigma \oplus^k z : C_2 \gg_{R!a} \Gamma \vdash e'' : E$ with $e' = e''|_\Gamma$. The rule $\supset L$ remains applicable for the typing ϕ_2^*, thereby generating $\Sigma \gg_{R!a} \Gamma \vdash e''[\![y\, e_1/z]\!] : E$. This proves the claim since $(e''[\![y\, e_1/z]\!])|_\Gamma = (e''|_\Gamma)[\![y\, e_1/z]\!] = e'[\![y\, e_1/z]\!] = e$. Observe that the trunk variables Γ do not occur in the context Σ, by validity, and hence not in $\Sigma\lceil k$. Also, by validity of contexts, they are different from the variables y and z used in the rule $\supset L$. Therefore, the substitutions $(\cdot)[\![y\, e_1/z]\!]$ and $(\cdot)|_\Gamma$ commute.

The only left rule which does touch a branch variable is $\exists L$. However, it cannot remove the context branch $R?a\langle\hat{\Gamma}\rangle$, seen in forward direction, because the filler reference a appears free in the branch typing $\Sigma \vdash \emptyset \gg_{R!a} e : E$ and rule $\exists L$ requires the branch reference that it closes not to be free in the typing. More precisely, suppose the derivation $\phi = \exists L \cdot \phi_1$ in question is

$$\frac{\begin{array}{c}\vdots\,\phi_1\\ \Sigma_1, x : \exists S.C, S?b\langle\hat{y} : C\rangle, \Sigma_2 \vdash \emptyset \gg_{R!a} e' : E\end{array}}{\Sigma_1, x : \exists S.C, \Sigma_2 \vdash \emptyset \gg_{R!a} \mathtt{let}\,!b.\,\hat{y} = x\,\mathtt{in}\,e' : E}\ \exists L\ (b,\ \hat{y}\ \text{fresh})$$

where $\Sigma = \Sigma_1, x : \exists S.C, \Sigma_2$ and $e = \mathtt{let}\,!b.\,y = x\,\mathtt{in}\,e'$. Then, since a and $\hat{y}$ are required to be fresh for the context and statement, we must have $a \neq b$. Now, when we apply the induction hypothesis to ϕ_1 we get a derivation ϕ_1^* of $\Sigma', R?a\langle\hat{\Gamma}\rangle \gg_{R!a} \Gamma \vdash e'' : E$ with $\Sigma_1, x : \exists S.C, S?b\langle\hat{y} : C\rangle, \Sigma_2 = \Sigma', R?a\langle\hat{\Gamma}\rangle$ and $e' = e''|_\Gamma$. The transformed typing ϕ_1^* is not larger than ϕ_1 and cut-free if ϕ_1 is. Since $a \neq b$ we can be sure that the branch variables $R?a\langle\hat{\Gamma}\rangle$ are different from $S?b\langle\hat{y} : C\rangle$ (though we may have $R = S$). This means that $R?a\langle\hat{\Gamma}\rangle$ is contained in the active scope of Σ_1 or Σ_2, i.e., $\Sigma_1, \Sigma_2 = \Sigma_1', \Sigma_2', R?a\langle\hat{\Gamma}\rangle$ and $\Sigma' = \Sigma_1', x : \exists S.C, S?b\langle\hat{y} : C\rangle, \Sigma_2'$. Hence rule $\exists L$ remains applicable to ϕ_1^*, yielding $\Sigma_1, x : \exists S.C, \Sigma_2 \gg_{R!a} \Gamma \vdash \mathtt{let}\,!b.\,\hat{y} = x\,\mathtt{in}\,e'' : E$ as required, considering that $\Sigma = \Sigma_1, x : \exists S.C, \Sigma_2 = \Sigma_1', x : \exists S.C, \Sigma_2', R?a\langle\hat{\Gamma}\rangle$ and $(\mathtt{let}\,!b.\,\hat{y} = x\,\mathtt{in}\,e'')|_\Gamma = \mathtt{let}\,!b.\,\hat{y} = x\,\mathtt{in}\,(e''|_\Gamma) = \mathtt{let}\,!b.\,\hat{y} = x\,\mathtt{in}\ e' = e$. The reduced typing derivation in this case, thus, is $\phi^* = \exists L \cdot \phi_1^*$.

The cut rule cut_1 is handled trivially. It cannot touch the branch variables, whence it commutes with the scope shift. The same applies to cut_2 if the branch variables are not involved. The only interesting case deserving separate attention is an application of cut_2 where the branch

$R?a$ is actively involved in the cut. This is where $\Sigma \vdash \emptyset \gg_{R!a} e : E$ arises from a derivation

$$\dfrac{\begin{array}{c}\vdots\,\phi_1\\ \Sigma\lceil k \vdash \emptyset \gg_{R!a} f : F\end{array} \qquad \begin{array}{c}\vdots\,\phi_2\\ \Sigma \oplus^k R?a\langle \hat{x} : F\rangle \vdash \emptyset \gg_{R!a} e' : E\end{array}}{\Sigma \vdash \emptyset \gg_{R!a} e'[\![f/\hat{x}]\!] : E}\ cut_2$$

so that $e = e'[\![f/\hat{x}]\!]$. In addition, $\hat{x}$ is fresh and so does not occur as a branch variable in the scope a of Σ.

The induction hypothesis applied to ϕ_2 brings us $\Sigma \oplus^k R?a\langle \hat{x} : F\rangle = \Sigma', R?a\langle\hat{\Gamma}\rangle$ together with a typing ϕ_2^* of $\Sigma', R?a\langle\hat{\Gamma}\rangle \gg_{R!a} \Gamma \vdash e'' : E$ and $e' = e''|_{\Gamma}$ and such that $\hat{\Gamma}$ includes all branch variables in scope a of Σ. The match $\Sigma \oplus^k R?a\langle \hat{x} : F\rangle = \Sigma', R?a\langle\hat{\Gamma}\rangle$ implies that k must be the depth of Σ, because the same scope reference a cannot appear at different depths simultaneously. Thus, ϕ_1 is $\Sigma \vdash \emptyset \gg_{R!a} f : F$ and $\Sigma \oplus^k R?a\langle \hat{x} : F\rangle = \Sigma, R?a\langle \hat{x} : F\rangle = \Sigma', R?a\langle\hat{\Gamma}\rangle$.

Moreover, the equation $\Sigma \oplus^k R?a\langle \hat{x} : F\rangle = \Sigma', R?a\langle\hat{\Gamma}\rangle$ implies that $\hat{\Gamma} = \hat{x} : F, \hat{\Gamma}'$ and $\Sigma = \Sigma', R?a\langle\hat{\Gamma}'\rangle$, where $\hat{\Gamma}'$ does not contain $\hat{x}$. In particular, this means the typing ϕ_2^* for e'' is actually of the form $\Sigma', R?a\langle\hat{\Gamma}\rangle \gg_{R!a} x : F, \Gamma' \vdash e'' : E$. Now we apply the induction hypothesis to the typing ϕ_1, obtaining $\Sigma = \Sigma'', R?a\langle\hat{\Gamma}''\rangle$ and a typing ϕ_1' of $\Sigma'', R?a\langle\hat{\Gamma}''\rangle \gg_{R!a} \Gamma'' \vdash f' : F$ with $f = f'|_{\Gamma''}$. Again, we may assume that Σ'' does not contain branch variables in scope a, making them all part of $\hat{\Gamma}''$. Then, the separations $\Sigma = \Sigma', R?a\langle\hat{\Gamma}'\rangle$ and $\Sigma = \Sigma'', R?a\langle\hat{\Gamma}''\rangle$ are unique, so that $\Sigma'' = \Sigma'$ and $\Gamma'' = \Gamma'$. Thus ϕ_1', by weakening with $\hat{x} : F$ (Lem. 1(3)), turns into a derivation ϕ_1^* of $\Sigma', R?a\langle\hat{\Gamma}\rangle \gg_{R!a} \Gamma' \vdash f' : F$. We may now cut_1 the derivation ϕ_1^* and ϕ_2^* to obtain $\Sigma \gg_{R!a} \Gamma' \vdash e''[\![f'/x]\!] : E$. This is what we want because $(e''[\![f'/x]\!])|_{\Gamma'} = (e''|_{\Gamma})[\![f'|_{\Gamma'}/\hat{x}]\!] = e'[\![f'|_{\Gamma''}/\hat{x}]\!] = e'[\![f/\hat{x}]\!] = e$. Notice that $\phi^* = cut_1 \cdot (\phi_1^*, \phi_2^*)$ is not larger than $\phi = cut_2 \cdot (\phi_1, \phi_2)$ since each ϕ_i^* is at most of the size of ϕ_i, and it does not involve any extra uses of cut_1 or cut_2 over and above what is contained in ϕ_i. □

8 Subject Reduction

$$\begin{array}{rcll}
\pi_1(e_1, e_2) & \longrightarrow_\beta & e_1 & \beta\pi_1\\
\pi_2(e_1, e_2) & \longrightarrow_\beta & e_2 & \beta\pi_2\\
\texttt{case}\ \iota_1\, e\ \texttt{of}\ [\iota_1\, x_1 \to e_1 \mid \iota_2\, x_2 \to e_2] & \longrightarrow_\beta & e_1[\![e/x_1]\!] & \beta\texttt{case}_1\\
\texttt{case}\ \iota_2\, e\ \texttt{of}\ [\iota_1\, x_1 \to e_1 \mid \iota_2\, x_2 \to e_2] & \longrightarrow_\beta & e_2[\![e/x_2]\!] & \beta\texttt{case}_2\\
(\lambda x.\, e_1)\, e_2 & \longrightarrow_\beta & e_1[\![e_2/x]\!] & \beta\lambda\\
(?a.\, e)@b & \longrightarrow_\beta & e[\![b/a]\!] & [\beta]\\
\texttt{let}\ !a.\, \hat{y} = !b.\, e_1\, \texttt{in}\, e_2 & \longrightarrow_\beta & e_2[\![e_1/\hat{y}, b/a]\!] & \langle\beta\rangle
\end{array}$$

Figure 3: β-Contraction Rules.

Proposition 6 (Subject Reduction)**.** If $\Sigma \vdash e : \Psi$ and $e \twoheadrightarrow_{\beta\gamma} e'$ then $\Sigma \vdash e' : \Psi$.

The proof of subject reduction proceeds with the help of two auxiliary results. First, we prove Subject Contraction Prop. 7 which states that all basic contractions preserve typing. Second, we prove the Subject Inversion Prop. 8 which generalises this to arbitrary contexts.

$$
\begin{array}{llll}
\pi_i(\mathtt{let}\ !a.\,\hat{y} = e_1\,\mathtt{in}\,e_2) & \longrightarrow_\gamma & \mathtt{let}\ !a.\,\hat{y} = e_1\,\mathtt{in}\,\pi_i\,e_2 & \gamma\mathtt{let}_1 \\
(\mathtt{let}\ !a.\,\hat{y} = e_1\,\mathtt{in}\,e_2)\,e_3 & \longrightarrow_\gamma & \mathtt{let}\ !a.\,\hat{y} = e_1\,\mathtt{in}\,e_2\,e_3 & \gamma\mathtt{let}_2 \\
(\mathtt{let}\ !a.\,\hat{y} = e_1\,\mathtt{in}\,e_2)@b & \longrightarrow_\gamma & \mathtt{let}\ !a.\,\hat{y} = e_1\,\mathtt{in}\,(e_2@b) & \gamma\mathtt{let}_3 \\
\mathtt{case}\,(\mathtt{let}\ !a.\,\hat{y} = e_1\,\mathtt{in}\,e_2) & & & \\
\quad\mathtt{of}\,[\iota_1\,x_1 \to e_3 \mid \iota_2\,x_2 \to e_4] & \longrightarrow_\gamma & \mathtt{let}\ !a.\,\hat{y} = e_1\,\mathtt{in} & \\
 & & \quad\mathtt{case}\,e_2\,\mathtt{of}\,[\iota_1\,x_1 \to e_3 \mid \iota_2\,x_2 \to e_4] & \gamma\mathtt{let}_4 \\
\mathtt{let}\ !a.\,\hat{y} = \mathtt{let}\ !b.\,\hat{z} = e_1\,\mathtt{in}\,e_2\,\mathtt{in}\,e_3 & \longrightarrow_\gamma & \mathtt{let}\ !b.\,\hat{z} = e_1\,\mathtt{in}\,\mathtt{let}\ !a.\,\hat{y} = e_2\,\mathtt{in}\,e_3 & \gamma\mathtt{let}_5 \\
\pi_i(\mathtt{case}\,e\,\mathtt{of}\,[\iota_1\,x_1 \to e_1 \mid \iota_2\,x_2 \to e_2]) & \longrightarrow_\gamma & \mathtt{case}\,e\,\mathtt{of}\,[\iota_1\,x_1 \to \pi_i\,e_1 \mid \iota_2\,x_2 \to \pi_i\,e_2] & \gamma\,\mathtt{case}_1 \\
(\mathtt{case}\,e\,\mathtt{of}\,[\iota_1\,x_1 \to e_1, \iota_2\,x_2 \to e_2])\,e_3 & \longrightarrow_\gamma & \mathtt{case}\,e\,\mathtt{of}\,[\iota_1\,x_1 \to e_1\,e_3 \mid \iota_2\,x_2 \to e_2\,e_3] & \gamma\,\mathtt{case}_2 \\
(\mathtt{case}\,e\,\mathtt{of}\,[\iota_1\,x_1 \to e_1 \mid \iota_2\,x_2 \to e_2])@b & \longrightarrow_\gamma & \mathtt{case}\,e\,\mathtt{of}\,[\iota_1\,x_1 \to e_1@b \mid \iota_2\,x_2 \to e_2@b] & \gamma\,\mathtt{case}_3 \\
\mathtt{case}\,(\mathtt{case}\,e\,\mathtt{of} & & \mathtt{case}\,e\,\mathtt{of} & \\
\qquad[\iota_1\,y_1 \to e_1 \mid \iota_2\,y_2 \to e_2]) & \longrightarrow_\gamma & \quad[\iota_1\,y_1 \to \mathtt{case}\,e_1\,\mathtt{of} & \\
\quad\mathtt{of}\,[\iota_1\,x_1 \to e_3 \mid \iota_2\,x_2 \to e_4] & & \qquad\qquad[\iota_1\,x_1 \to e_3 \mid \iota_2\,x_2 \to e_4], & \\
 & & \quad\ \iota_2\,y_2 \to \mathtt{case}\,e_2\,\mathtt{of} & \\
 & & \qquad\qquad[\iota_1\,x_1 \to e_3 \mid \iota_2\,x_2 \to e_4]] & \gamma\,\mathtt{case}_4 \\
\mathtt{let}\ !a.\,\hat{y} = & & & \\
\quad\mathtt{case}\,e\,\mathtt{of}\,[\iota_1\,x_1 \to e_1 \mid \iota_2\,x_2 \to e_2]\,\mathtt{in}\,e_3 & \longrightarrow_\gamma & \mathtt{case}\,e\,\mathtt{of} & \\
 & & \quad[\iota_1\,x_1 \to \mathtt{let}\ !a.\,\hat{y} = e_1\,\mathtt{in}\,e_3 \mid & \\
 & & \quad\ \iota_2\,x_2 \to \mathtt{let}\ !a.\,\hat{y} = e_2\,\mathtt{in}\,e_3] & \gamma\,\mathtt{case}_5
\end{array}
$$

Side condition: a, $\hat{y}$ not free in e_3 or e_4 in $\gamma\mathtt{let}_2$ or $\gamma\mathtt{let}_4$; $a \neq b$ in $\gamma\mathtt{let}_3$; b, $\hat{z}$ not free in e_3 in $\gamma\mathtt{let}_5$; y_1, y_2 not free in e_3 or e_4 in $\gamma\,\mathtt{case}_4$; x_1, x_2 not free in e_3 in $\gamma\,\mathtt{case}_2$ or $\gamma\,\mathtt{case}_5$.

Figure 4: γ-Contraction Rules ("Commuting Conversions").

Proposition 7 (Subject Contraction)**.** All β-contractions (Fig. 3) and γ-contractions (Fig. 4) preserve typing, i.e., if $\Sigma \vdash e : \Psi$ and $e \longrightarrow_\beta e'$ or $e \longrightarrow_\gamma e'$, then $\Sigma \vdash e' : \Psi$.

Proof. We show that the β-contractions in Fig. 3 and commuting contractions of Fig. 4 preserve typing. This follows directly from the local completeness properties Props. 3, 4 as well as scope shift Prop. 5.

Before we go through the different redexes let us observe that it suffices to prove subject contraction for trunk typings. For if we have a redex typed as a branch $\Sigma \vdash \emptyset \gg_{S!c} e_1 : E$ we can always apply a scope shift (Prop. 5) of the branch typing to get $\Sigma = \Sigma', S?c\langle\hat{\Gamma}\rangle$ and $\Sigma \gg_{S!c} \Gamma \vdash e_1' : E$ such that $e_1 = e_1'|_\Gamma$. Then, subject contraction for trunk typings is applicable (in the advanced context) which gives $\Sigma \gg_{S!c} \Gamma \vdash e_2' : E$, where e_2' is the redex of e_1', with $e_2 = e_2'|_\Gamma$, in each case. Finally, from there, rule Ax_f lifts this to the branch typing $\Sigma \vdash \emptyset \gg_{S!c} e_2 : E$. So, from now on we only consider subject contraction for trunk typings.

Case $[\beta]$**.** Suppose $\Sigma \vdash (?a.\,e)@b : E$. We claim that also $\Sigma \vdash e[\![b/a]\!] : E$. By destructor inversion (Prop. 3(2)), we get $\Sigma = \Sigma' \gg_{S!b} \Gamma$ and $\Sigma' \vdash ?a.\,e : \forall S.E$. Further, constructor inversion (Prop. 4(2)) permits us to conclude that $R = S$ and $\Sigma' \gg_{R!a} \emptyset \vdash e : E$. We now weaken the typing to introduce Γ (Lem. 1(3)) and translate the scope reference by variable renaming into $\Sigma' \gg_{R!b} \Gamma \vdash e[\![b/a]\!] : E$ by a combination of weakening and contraction (Lem. 1(1,3)). This is nothing but $\Sigma \vdash e[\![b/a]\!] : E$ as desired.

Case $\langle\beta\rangle$**.** Suppose $\Sigma \vdash \texttt{let}\ !a.\,x = !b.\,e_1\,\texttt{in}\,e_2 : E$. We apply the destructor inversion (Prop. 3(1)) on this typing and obtain $\Sigma{\upharpoonright}k \vdash !b.\,e_1 : \exists R.D$ and $\Sigma \oplus^k R?a\langle\hat{x} : D\rangle \vdash e_2 : E$ at some depth k. Constructor inversion (Prop. 4(1)) on the former implies $\Sigma{\upharpoonright}k \vdash \emptyset \gg_{R!b} e_1 : D$. In the latter we systematically rename a by b through a combination of weakening and contractions on scope references. Then, we can apply cut_2 to substitute this into the typing for e_2, which yields $\Sigma \vdash e_2[\![e_1/\hat{x}, b/a]\!] : E$ as required.

Case $\beta\lambda$**.** The proof of subject reduction for standard β-reduction does not pose any difficulties either. Given a typing $\Sigma \vdash (\lambda x.e_1)\,e_2 : E$ we first invoke destructor completeness (Prop. 3(3)) for $\Sigma \vdash \lambda x.e_1 : D \supset E$ and $\Sigma \vdash e_2 : D$ and then constructor completeness (Prop. 4(3)) to generate a typing $\Sigma, x : D \vdash e_1 : E$. Both typings cut_1 together lead us to the conclusion $\Sigma \vdash e_1[\![e_2/x]\!] : E$.

Case $\gamma\texttt{let}_2$**.** Let us start from a well-typed expression $\Sigma \vdash (\texttt{let}\ !a.\,\hat{y} = e_1\,\texttt{in}\,e_2)\,e_3 : E$, where a and $\hat{y}$ are not free in e_3. We claim that $\Sigma \vdash \texttt{let}\ !a.\,\hat{y} = e_1\,\texttt{in}\,(e_2\,e_3) : E$. By inversion of the destructor (Prop. 3(3)) we have $\Sigma \vdash \texttt{let}\ !a.\,\hat{y} = e_1\,\texttt{in}\,e_2 : D \supset E$ and $\Sigma \vdash e_3 : D$. We invert the $\texttt{let}$ destructor (Prop. 3(1)) to extract typings $\Sigma{\upharpoonright}k \vdash e_1 : \exists R.C$ and $\Sigma \oplus^k R?a\langle\hat{y} : C\rangle \vdash e_2 : D \supset E$. The type of e_3 can be weakened (Lem. 1(3)) to $\Sigma \oplus^k R?a\langle\hat{y} : C\rangle \vdash e_3 : D$ so that $\Sigma \oplus^k R?a\langle\hat{y} : C\rangle \vdash e_2\,e_3 : E$ by destructor completeness (Prop. 3(3)). From here we easily assemble the desired typing $\Sigma \vdash \texttt{let}\ !a.\,\hat{y} = e_1\,\texttt{in}\,(e_2\,e_3) : E$ using destructor completeness (Prop. 3(1)).

Case $\gamma\texttt{let}_3$**.** Let us assume a well-typed expression $\Sigma \vdash (\texttt{let}\ !a.\,\hat{y} = e_1\,\texttt{in}\,e_2)@b : E$ where $b \neq a$. We wish to prove that also $\Sigma \vdash \texttt{let}\ !a.\,\hat{y} = e_1\,\texttt{in}\,(e_2@b) : E$. We go through the inversions as follows: First, destructor inversion (Prop. 3(2)) yields $\Sigma = \Sigma' \gg_{S!b} \Gamma$ and $\Sigma' \vdash \texttt{let}\ !a.\,\hat{y} = e_1\,\texttt{in}\,e_2 : \forall S.E$. From there, another destructor inversion (Prop. 3(1)) yields $\Sigma'{\upharpoonright}k \vdash e_1 : \exists R.D$ as well as $\Sigma' \oplus^k R?a\langle\hat{y} : D\rangle \vdash e_2 : \forall S.E$ at some depth k in Σ'. Notice that $\Sigma' \oplus^k R?a\langle\hat{y} : D\rangle \gg_{S!b} \Gamma = \Sigma \oplus^k R?a\langle\hat{y} : D\rangle$ is a valid context since $b \neq a$. We can apply destructor completeness again (Prop. 3(2), really an application of weakening Lem. 1(3), Ax_m, cut_1 and rule $\forall L$) to derive $\Sigma \oplus^k R?a\langle\hat{y} : D\rangle \vdash e_2@b : E$, or with weakening $\Sigma \oplus^k \{x : \exists R.D, R?a\langle\hat{y} : D\rangle\} \vdash e_2@b : E$. This can be subjected to an application of $\exists L$ which gives $\Sigma \oplus^k x : \exists R.D \vdash \texttt{let}\ !a.\,\hat{y} = x\,\texttt{in}\,(e_2@b) : E$. Observe that $\Sigma{\upharpoonright}k = \Sigma'{\upharpoonright}k$, whence we can cut_1 with the typing $\Sigma'{\upharpoonright}k \vdash e_1 : \exists R.D$ and so finally arrive at $\Sigma \vdash \texttt{let}\ !a.\,\hat{y} = e_1\,\texttt{in}\,(e_2@b) : E$.

Case $\gamma\texttt{let}_5$**.** Suppose $\texttt{let}\ !a.\,\hat{y} = (\texttt{let}\ !b.\,\hat{z} = e_1\,\texttt{in}\,e_2)\,\texttt{in}\,e_3$ is of type E in context Σ and b, $\hat{z}$ not free in e_3. By destructor inversion (Prop. 3(1)) the sub-expression $\texttt{let}\ !b.\,\hat{z} = e_1\,\texttt{in}\,e_2$ has a type $\exists R.D$ in a pruned context $\Sigma{\upharpoonright}k$ at some depth k, and $e_3 : E$ in context $\Sigma \oplus^k R?a\langle\hat{y} : D\rangle$. Since b, $\hat{z}$ not free in e_3 they are fresh for $\Sigma \oplus^k R?a\langle\hat{y} : D\rangle$. Further, by a second application of destructor inversion (Prop. 3(1), e_1 has an existential type $\exists S.C$ at some prefix $\Sigma{\upharpoonright}k{\upharpoonright}d = \Sigma{\upharpoonright}d$ ($d \leq k$) of $\Sigma{\upharpoonright}k$ so that $e_2 : \exists R.D$ in the extended context $\Sigma{\upharpoonright}k \oplus^d S?b\langle\hat{z} : C\rangle$. Note that $e_3 : E$ in the (valid) weakened context $\Sigma \oplus^k R?a\langle\hat{y} : D\rangle \oplus^d S?b\langle\hat{z} : C\rangle$, by Lem. 1(3), and $(\Sigma \oplus^d S?b\langle\hat{z} : C\rangle){\upharpoonright}k = \Sigma{\upharpoonright}k \oplus^d S?b\langle\hat{z} : C\rangle$. Hence by destructor completeness (Prop. 3(1)) the expression $\texttt{let}\ !a.\,\hat{y} = e_2\,\texttt{in}\,e_3$ has type E in context $\Sigma \oplus^d S?b\langle\hat{z} : C\rangle$. Finally, $\Sigma{\upharpoonright}d \vdash e_1 : \exists S.C$ plus destructor completeness (Prop. 3(1)) yields the desired result $\Sigma \vdash \texttt{let}\ !b.\,\hat{z} = e_1\,\texttt{in}\,(\texttt{let}\ !a.\,\hat{y} = e_2\,\texttt{in}\,e_3) : E$. □

Prop. 7 does not say that reduction $\twoheadrightarrow_{\beta\gamma}$ preserves typing, i.e., when contractions are performed in arbitrary syntactic contexts. However, this follows from the *substitution inversion* property (Prop. 8) to be stated and proved next. Substitution Inversion is another important consequence of Props. 3, 4 and 5.

To explain substitution inversion, consider a well-typed expression $\Sigma \vdash f\{e/\vec{z}\} : \Psi$ arising from a raw substitution of an expression e for a variable $\vec{z}$ with a *single* occurrence in another expression f which singles out a unique occurrence of e in $f\{e/\vec{z}\}$. We write the variable $\vec{z}$ in bold face to indicate that z has multiplicity 1. Substitution inversion separates the typing $\Sigma \vdash f\{e/\vec{z}\} : \Psi$ of the composite expression from the typing of the sub-expression e and the typing of the surrounding "context" expression f.

Let us look at ordinary typed λ-calculus (i.e., without modalities) first. There, the type context Σ is a single (active) scope $\Sigma = \Gamma$ and the typing statement a trunk typing $\Psi = F$. Then, every occurrence of a sub-expression e in $f\{e/\vec{z}\}$ is typeable in an extension of the active scope Γ, i.e., there exists Γ' such that $\Gamma, \Gamma' \vdash e : E$. The extension Γ' contains all the local variables that are free in e but bound in f at $\vec{z}$. Expression e may be replaced by any other expression e' well-typed in the same way: More precisely, if $\Gamma, \Gamma' \vdash e' : E$ then $\Gamma \vdash f\{e'/\vec{z}\} : F$. We can express this by saying that the place holder or context hole $\vec{z}$ has type E with an additional local scope Γ' and write $\vec{z} : E[\Gamma']$ and $\Gamma, \vec{z} : E[\Gamma'] \vdash f : F$. In this way, the variable $\vec{z}$ turns into a *meta-variable* to abstract the raw occurrence of the sub-expression e in its local context inside f. The meta-variable $\vec{z}$ is specified with a *contextual type* [9] $E[\Gamma']$ to express that it represents a context hole that can be filled with any expression of type E in the extended context Γ, Γ'. We call $\Gamma, \vec{z} : E[\Gamma'] \vdash f : F$ a *substitution inversion* because we have removed e from $f\{e/\vec{z}\}$ and typed the syntactic context f all by itself. It is important to note that the meta-variable $\vec{z}$ abstracts a *raw* occurrence of $\vec{z}$ in f, to be filled by raw substitution $f\{e/\vec{z}\}$ which binds all free occurrences of local variables Γ' in e, as opposed to *capture avoiding* substitution $f[\![e/\vec{z}]\!]$ which does not.

In $\lambda\mathsf{CK}_n$ we can do the same inversion of raw substitution but with a richer notion of local context. Since an expression $f\{e/\vec{z}\}$ wraps up different computational contexts, the sub-expression e must be typed in a local context Σ' which not only extends but possibly also branches off from the main context Σ at some depth of scoping. In addition, by moving into the local context Σ' of e, we may have to activate some set $\hat{\Gamma}'$ of branch variables of e, i.e., replace some occurrences of branch variables $\hat{y}$ in e by their trunk versions y. The typing of a meta-variable $\vec{z}$ in $\lambda\mathsf{CK}_n$ may thus be written as follows

$$\Sigma_1, \vec{z} : \Psi'[\Sigma_1']_{\Gamma'}; \Sigma_2 \vdash f : \Psi, \tag{1}$$

where $\Sigma_1 = \Sigma{\upharpoonright}n$ is some initial prefix of the global context, Σ_1' is a local context of variables by which Σ_1 must be extended and Γ' is the set of trunk variables to be activated at $\vec{z}$ in f. Formally, the (meta) typing (1) states that whenever $\Sigma_1 \oplus \Sigma_1' \vdash e' : \Psi'$ implies $\Sigma_1, \Sigma_2 \vdash f\{e'[\![\hat{\Gamma}'/\Gamma']\!]/\vec{z}\} : \Psi$. Observe that the depth of local context Σ_1' can be larger than that of Σ_1, so that $\Sigma_1 \oplus \Sigma_1'$ is an extended context path which branches off from the main context path Σ at depth k. In this way, different occurrences of sub-expressions e_1, e_2 may live in different parts (active nodes) of a global context tree wrapped up inside $f\{e_1/\vec{z}_1\}\{e_2/\vec{z}_2\}$. Note the semi-colon in (1) between the context scopes $\Sigma_1, \vec{z} : \Psi'[\Sigma_1']_{\Gamma'}$ relevant for sub-expression e and the context Σ_2 which is only relevant for the composite expression $f\{e'[\![\hat{\Gamma}'/\Gamma']\!]/\vec{z}\}$ at top level.

The semicolon prevents us from exchanging typings in the root of Σ_2 with those in the active scope of Σ_1.

Example 1. Consider the context $\Sigma = f : \exists S.C \supset \forall R.D, S?c\langle \hat{y} : C\rangle$ and the typing

$$\Sigma \vdash ?b.\, f(!c.\, \hat{y})@b : \forall R.D \tag{2}$$

which can be constructed without cut rules. In the typing derivation of (2) using the sequent rules for $\vdash^{norm}$ (Figs. 1 and 2) the sub-expression $f(!c.\, \hat{y})$ does not appear and thus is not typed. However, it is easy to see that the sub-expression does have a type, viz. $\Sigma \vdash f(!c.\, \hat{y}) : \forall R.D$. This follows directly from the admissible inversion rules of Props. 3, 4. From (2) and Prop. 4(2) we infer that $\Sigma \gg_{R!b} \emptyset \vdash f(!c.\, \hat{y})@b : D$. From here, Prop. 3(2) implies that $\Sigma \vdash f(!c.\, \hat{y}) : \forall R.D$. Indeed, we can build the typing of (2) in a purely syntax-driven fashion along the sub-expressions of $?b.\, f(!c.\, \hat{y})@b$:

$$\dfrac{\dfrac{\dfrac{\Sigma \vdash f : \exists S.C \supset \forall R.D \qquad \dfrac{\dfrac{\Sigma \gg_{S!c} y : C \vdash y : C}{\Sigma \vdash \emptyset \gg_{S!c} \hat{y} : C}\,\text{Prop. 5}}{\Sigma \vdash !c.\, \hat{y} : \exists S.C}\,\text{Prop. 4(1)}}{\Sigma \vdash f(!c.\, \hat{y}) : \forall R.D}\,\text{Prop. 3(3)}}{\Sigma \gg_{R!b} \emptyset \vdash f(!c.\, \hat{y})@b : D}\,\text{Prop. 3(2)}}{\Sigma \vdash ?b.\, f(!c.\, \hat{y})@b : \forall R.D}\,\text{Prop. 4(2)}$$

Notice that all these admissible typing rules are invertible and thus can be used equally well for type analysis as for type synthesis.

Example 2 (Substitution Inversion)**.** Consider the expression from Ex. 1, i.e., the typing

$$\Sigma \vdash ?b.\, f(!c.\, \hat{y})@b : \forall R.D \tag{3}$$

in context $\Sigma = f : \exists S.C \supset \forall R.D, S?c\langle \hat{y} : C\rangle$. We have seen that the sub-expression $f(!c.\, \hat{y})@b : D$ is typeable in the context $\Sigma \oplus (\emptyset \gg_{R!b} \emptyset) = \Sigma \gg_{R!b} \emptyset = f : \exists S.C \supset \forall R.D, S?c\langle \hat{y} : C\rangle \gg_{R!b} \emptyset$. Obviously, any other expression $e_1 : D$ typeable in this extended context $\Sigma \oplus (\emptyset \gg_{R!b} \emptyset)$ gives rise to $\Sigma \vdash ?b.\, e_1 : \forall R.D$ by rule $\forall R$ (or Prop. 4(2)). We can denote this as

$$\Sigma, \vec{z} : D[\emptyset \gg_{R!b} \emptyset]_{\emptyset}; \emptyset \vdash ?b.\, \vec{z} : \forall R.D, \tag{4}$$

where the meta-variable $\vec{z}$ is the place holder for the raw occurrence of the sub-expression $f(!c.\, y)@b$ in (3). The meta-variable $\vec{z}$ is specified by a *contextual* type $D[\emptyset \gg_{R!b} \emptyset]_{\emptyset}$ which states that $\vec{z}$ can be substituted by any well-formed term of type D in the context Σ extended by $\emptyset \gg_{R!b} \emptyset$. Observe that $f(!c.\, y)@b$ has a free scope reference b that gets captured by the raw substitution, i.e., b is *bound* by $?b.\, \vec{z}$ *at* $\vec{z}$.

In contrast, the sub-expressions $f(!c.\, \hat{y}) : \forall R.D$ and $!c.\, \hat{y} : \exists S.C$ in (3) can be typed in the outer context Σ. Given any $\Sigma \vdash e_2 : \forall R.D$ one obtains $\Sigma \vdash ?b.\, e_2@b : \forall R.D$ and every $\Sigma \vdash e_3 : \exists S.C$ can be extended to type $\Sigma \vdash ?b.\, (f e_3)@b : \forall R.D$. Using a meta-variable $\vec{z}$ we can state this as

$$\Sigma, \vec{z} : \forall R.D[\emptyset]_{\emptyset}; \emptyset \vdash ?b.\, \vec{z}@b : \forall R.D \text{ and } \Sigma, \vec{z} : \exists S.C[\emptyset]_{\emptyset}; \emptyset \vdash ?b.\, (f\,\vec{z})@b : \forall R.D.$$

An interesting case is the branch variable $\hat{y}$ as a sub-expression of (3) whose typing can be inverted both as a branch typing $\Sigma \vdash \emptyset \gg_{S!c} \hat{y} : C$ or a trunk typing $\Sigma \gg_{S!c} y : C \vdash y : C$. Taking the former, we can see that for every branch expression $\Sigma \vdash \emptyset \gg_{S!c} e_4 : C$ we can construct a typing $\Sigma \vdash ?b.\, f(!c.\, e_4)@b : \forall R.D$. This means we have

$$\Sigma, \vec{z} : (\emptyset \gg_{S!c} C)[\emptyset]_\emptyset; \emptyset \vdash ?b.\, f(!c.\, \vec{z})@b : \forall R.D$$

as a statement of admissibility, On the other hand, given any $\Sigma \gg_{S!c} y : C \vdash e_5 : C$ we also obtain $\Sigma \vdash ?b.\, f(!c.\, e_5[\![\hat{y}/y]\!])@b : \forall R.D$. Notice how the free references to the trunk variable y inside e_5 must be abstracted as branch variables $\hat{y}$. This is reflected in the associated meta typing

$$\Sigma, \vec{z} : C[\emptyset \gg_{S!c} y : C]_{\{y:C\}}; \emptyset \vdash ?b.\, f(!c.\, \vec{z})@b : \forall R.D.$$

In the statement and proof of the Substitution Inversion Prop. 8 it is convenient to consider a trunk variable x bound in an expression f at $\vec{z}$, not only if $\vec{z}$ occurs in the scope of a binder for x but also for its branch partner $\hat{x}$. The idea is that when we substitute an expression d with x free, as in $f\{d/\vec{z}\}$, then the trunk variable x must be renamed (”deactivated”) to $\hat{x}$ and therefore it becomes bound. Again, $e|_\Gamma$ stands for $e[\![\hat{\Gamma}/\Gamma]\!]$, which we use for abbreviation heavily in the proof of Prop. 8.

Proposition 8 (Substitution Inversion). Let $\Sigma \vdash f\{d/\vec{z}\} : \Psi$ where $\vec{z}$ has multiplicity 1. Then, there exists a context split $\Sigma = \Sigma_1, \Sigma_2$ where $\Sigma_1 = \Sigma\lceil n$ for some n and typing contexts Σ', Ψ', a set of trunk variables Γ' and an expression d' such that $d = d'|_{\Gamma'}$ and $\Sigma_1 \oplus \Sigma' \vdash d' : \Psi'$ and $\Sigma_1, \vec{z} : \Psi'[\Sigma']_{\Gamma'}; \Sigma_2 \vdash f : \Psi$. Moreover, the following ”locality conditions” hold:

1. the extension context Σ' has at least the depth of Σ_1 and is compatible with Σ_1, i.e., $\Sigma_1 \oplus \Sigma'$ is valid;
2. each variable of Σ' is bound in f at $\vec{z}$ or contained in Γ';
3. each variable in Γ' which is not bound in f at $\vec{z}$ is both a branch variable in the active scope of Σ_1 and a trunk variable in Σ' at the same depth.

Proof. We argue by induction on the structure of the expression f, more precisely the number of operators, using the constructor and destructor inversion stated in Props. 3 and 4 as well as Scope Shift Prop. 5. The latter permits us to reduce the statement of Prop. 8 for a branch typing $\Psi = \emptyset \gg_{R!a} F$ to that for trunk typings $\Psi = F$, albeit without reducing the size of the expression. This is well-founded since the only case of a trunk typing to recur to branch typing is in the case of an expression $f = !a.\, e$. However, by Prop. 4(1), this reduces to the *strictly smaller* sub-expression e (which is then dealt with by Scope Shift Prop. 5).

• First, observe that the statement of the proposition is immediate for variables, i.e., if $f = \vec{z}$. Then, the typing ϕ gives $\Sigma \vdash d : \Psi$ directly. Trivially, for any well-typed $\Sigma \vdash e : \Psi$ we have $\Sigma \vdash f\{e/\vec{z}\} : \Psi$ and thus $\Sigma, \vec{z} : \Psi[\emptyset]_\emptyset; \emptyset \vdash f : \Psi$. The locality conditions are obvious.

We begin by first handling all cases of trunk typings, i.e., where $\Sigma \vdash f\{d/\vec{z}\} : F$.

• The cases where $f = \texttt{let}\ !a.\, \hat{y} = f_1\ \texttt{in}\ f_2$ and $f = \texttt{case}\ f_1\ \texttt{of}\ [\iota_1\, x_1 \to f_2 \mid \iota_2\, x_2 \to f_3]$ are essentially the same. We only cover the former case. Suppose $f\{d/\vec{z}\} = \texttt{let}\ !a.\, \hat{y} = g_1\ \texttt{in}\ g_2$ and $\Sigma \vdash f\{d/\vec{z}\} : F$. By Lem. 3(1) we have $\Sigma\lceil k \vdash g_1 : \exists R.D$ for some depth k of Σ and $\Sigma \oplus^k R?a\langle \hat{y} : D\rangle \vdash g_2 : F$.

Let us assume first that $g_1 = f_1$, $g_2 = f_2\{d/\vec{z}\}$ and $\vec{z}$ has a unique occurrence in f_2. The induction hypothesis can thus be applied to obtain a split $(\Sigma \oplus^k R?a\langle\hat{y} : D\rangle) = \Sigma_1, \Sigma_2$ and admissible context extension Σ', Ψ', Γ', d' with $d = d'|_{\Gamma'}$ as well as $\Sigma_1 \oplus \Sigma' \vdash d' : \Psi'$ and

$$\Sigma_1, \vec{z} : \Psi'[\Sigma']_{\Gamma'}; \Sigma_2 \vdash f_2 : \Psi \tag{5}$$

where each variable in Σ' (compatible with Σ_1 and of at least same depth) is bound in f_2 at $\vec{z}$ or included in Γ'. Further, every variable in Γ' that is not bound in f_2 at $\vec{z}$ is a branch variable in the active scope of Σ_1 and also a trunk variable in Σ' at the same depth. Obviously, every variable bound in f_2 at $\vec{z}$ is bound in f at $\vec{z}$.

1. If the branch $R?a\langle\hat{y} : D\rangle$ occurs in Σ_1 then $\Sigma_1 = \Sigma_1' \oplus^k R?a\langle\hat{y} : D\rangle$ and $\Sigma = \Sigma_1', \Sigma_2$. Moreover, defining $\Sigma'' =_{df} \Sigma' \oplus^k R?a\langle\hat{y} : D\rangle$ we get $\Sigma_1 \oplus \Sigma' = \Sigma_1' \oplus \Sigma''$ and $\Sigma_1' \oplus \Sigma'' \vdash d' : \Psi'$. Then, given a well-typed expression $\Sigma_1' \oplus \Sigma'' \vdash e : \Psi'$ the induction hypothesis (5) implies $\Sigma_1, \Sigma_2 \vdash f_2\{e|_{\Gamma'}/\vec{z}\} : \Psi$ to which we apply rule $\exists L$, obtaining $\Sigma_1', \Sigma_2 \vdash \texttt{let}\ !a.\,\hat{y} = f_1\ \texttt{in}\,(f_2\{e|_{\Gamma'}/\vec{z}\}) : \Psi$. This implies the admissibility of

$$\Sigma_1', \vec{z} : \Psi'[\Sigma'']_{\Gamma'}; \Sigma_2 \vdash f : \Psi$$

since $\Sigma = \Sigma_1', \Sigma_2$ and

$$f\{e|_{\Gamma'}/\vec{z}\} = (\texttt{let}\ !a.\,\hat{y} = f_1\ \texttt{in}\ f_2)\{e|_{\Gamma'}/\vec{z}\} = \texttt{let}\ !a.\,\hat{y} = f_1\ \texttt{in}\,(f_2\{e|_{\Gamma'}/\vec{z}\}).$$

Regarding the locality condition, consider any variable $u \mathrel{\varepsilon} \Sigma''$: either (i) $u \mathrel{\varepsilon} \Sigma'$ and thus, by induction hypothesis (5), u is bound in f_2 at $\vec{z}$ (hence also in f at $\vec{z}$) or $u \in \Gamma'$, or (ii) u is one of the variables a, $\hat{y}$ of the branch $R?a\langle\hat{y} : D\rangle$ and thus bound in f at $\vec{z}$. Finally, consider any variable $u \in \Gamma'$ not bound in f at $\vec{z}$, which means $u \neq y$. Such u is not bound in f_2 at $\vec{z}$ either, so that the induction hypothesis (5) implies $\hat{u}$ is a branch variable in the active scope of Σ_1 and u a trunk variable in Σ' at the same depth. But since $u \neq y$ also $\hat{u} \neq \hat{y}$ which means $\hat{u}$ must be a branch variable in Σ_1'. Also, no trunk variables have been modified from Σ' and Σ'', so that u remains trunk in Σ'' at the same depth as $\hat{u}$ in Σ_1'. The depths of Σ_1' and Σ_1 are the same as well as those of Σ'' and Σ'.

2. The other scenario is when the branch variable $R?a\langle\hat{y} : D\rangle$ is fully inside Σ_2, say $\Sigma_2 = \Sigma_2' \oplus^{k-n} R?a\langle\hat{y} : D\rangle$ where n is the depth of Σ_1. In this case, $\Sigma = \Sigma_1, \Sigma_2'$ and the induction hypothesis (5) implies $\Sigma_1, \vec{z} : \Psi'[\Sigma']_{\Gamma'}; \Sigma_2' \vdash \texttt{let}\ !a.\,\hat{y} = f_1\ \texttt{in}\ f_2 : \Psi$, which is the same as

$$\Sigma_1, \vec{z} : \Psi'[\Sigma']_{\Gamma'}; \Sigma_2' \vdash f : \Psi.$$

Here, the locality condition holds directly by induction hypothesis (5).

Next, consider the case that $g_2 = f_2$, $g_1 = f_1\{d/\vec{z}\}$ and $\vec{z}$ has a unique occurrence in f_1. The induction hypothesis can thus be applied to obtain a split $\Sigma\lceil k = \Sigma_1, \Sigma_2$ together with an admissible context extension Σ', Ψ', Γ', d' with $d = d'|_{\Gamma'}$ as well as $\Sigma_1 \oplus \Sigma' \vdash d' : \Psi'$ and

$$\Sigma_1, \vec{z} : \Psi'[\Sigma']_{\Gamma'}; \Sigma_2 \vdash f_1 : \exists R.D \tag{6}$$

for which also the locality conditions hold. Thus, whenever $\Sigma_1 \oplus \Sigma' \vdash e : \Psi'$ this implies $\Sigma_1, \Sigma_2 \vdash f_1\{e|_{\Gamma'}/\vec{z}\} : \exists R.D$. Since $\Sigma_1, \Sigma_2 = \Sigma\lceil k$ and $\Sigma \oplus^k R?a\langle\hat{y} : D\rangle \vdash f_2 : F$ the application of Prop. 3(1) gives us $\Sigma \vdash \texttt{let}\ !a.\,\hat{y} = f_1\{e|_{\Gamma'}/\vec{z}\}\ \texttt{in}\ f_2 : F$ which is the same as $\Sigma \vdash f\{e|_{\Gamma'}/\vec{z}\} : F$

as desired. Overall, this shows

$$\Sigma_1, \vec{z} : \Psi'[\Sigma']_{\Gamma'}; \Sigma_2, \Sigma_3 \vdash f : F,$$

where Σ_3 is chosen so that $\Sigma = \Sigma\lceil k, \Sigma_3 = \Sigma_1, \Sigma_2, \Sigma_3$. The locality conditions can be verified from the induction hypothesis (6).

• The cases where $f = \iota_i\, f'$ or $f = \pi_i\, f'$ are trivial, by induction hypothesis on the subexpression f'.

• Suppose $f = !a.\, f'$ where $\vec{z}$ has multiplicity 1 in f' and $\Sigma \vdash !a.\,(f'\{d/\vec{z}\}) : F$. By Prop. 4(1) we must have $F = \exists R.E$ and $\Sigma \vdash \emptyset \gg_{R!a} f'\{d/\vec{z}\} : E$. By induction hypothesis then

$$\Sigma_1, \vec{z} : \Psi'[\Sigma']_{\Gamma'}; \Sigma_2 \vdash \emptyset \gg_{R!a} f' : E, \tag{7}$$

for Σ', Ψ', Γ', d' with $\Sigma_1 \oplus \Sigma' \vdash d' : \Psi'$ and $d = d'|_{\Gamma'}$ as well as $\Sigma = \Sigma_1, \Sigma_2 = \Sigma\lceil n, \Sigma_2$. Σ' is compatible with Σ_1 and of depth no smaller than Σ_1. Also, all variables in Σ' are bound in f' at $\vec{z}$ or contained in Γ'. Also, each variable in Γ' which is not bound in f' at $\vec{z}$ appears as a branch variable in the active scope of Σ_1 and as a trunk variable in Σ' at the same depth.

The typing (7) gives $\Sigma_1, \vec{z} : \Psi'[\Sigma']_{\Gamma'}; \Sigma_2 \vdash f : \exists R.E$, directly, by application of $\exists R$. More precisely, assuming $\Sigma_1 \oplus \Sigma' \vdash e : \Psi'$ is a typing of any other expression e, the induction hypothesis (7) implies $\Sigma_1, \Sigma_2 \vdash \emptyset \gg_{R!a} f'\{e|_{\Gamma'}/\vec{z}\} : E$. To this we can apply rule $\exists R$, which gives $\Sigma_1, \Sigma_2 \vdash !a.\,(f'\{e|_{\Gamma'}/\vec{z}\}) : \exists R.G$, which is the same as $\Sigma_1, \Sigma_2 \vdash f\{e|_{\Gamma'}/\vec{z}\} : \exists R.E$ as $!a.\,(f'\{e|_{\Gamma'}/\vec{z}\}) = (!a.\, f')\{e|_{\Gamma'}/\vec{z}\}$. This proves that

$$\Sigma_1, \vec{z} : \Psi'[\Sigma']_{\Gamma'}; \Sigma_2 \vdash f : \exists R.E$$

as claimed. The locality condition is trivially satisfied by induction hypothesis (7) and since all variables in Σ' bound in f' at $\vec{z}$ are also bound in f at $\vec{z}$.

• Suppose $f = f'@a$ and $\vec{z}$ has multiplicity 1 in f'. Suppose $\Sigma \vdash (f'\{d/\vec{z}\})@a : F$. By Prop. 3(2) we have $\Sigma = \Sigma'' \gg_{R!a} \Gamma$ such that $\Sigma'' \vdash f'\{d/\vec{z}\} : \forall R.F$. The induction hypothesis on f' yields a split $\Sigma'' = \Sigma_1, \Sigma_2$ together with local contexts Σ', Ψ' and an expression d' and trunk variables Γ' such that $d = d'|_{\Gamma'}$ and $\Sigma_1 \oplus \Sigma' \vdash d' : \Psi'$. Moreover,

$$\Sigma_1, \vec{z} : \Psi'[\Sigma']_{\Gamma'}; \Sigma_2 \vdash f' : \forall R.F \tag{8}$$

together with the respective locality conditions. Any other typing $\Sigma_1 \oplus \Sigma' \vdash e : \Psi'$ may be put through the induction hypothesis (8) to generate a well-formed expression $\Sigma_1, \Sigma_2 \vdash f'\{e|_{\Gamma'}/\vec{z}\} : \forall R.F$. Another application of Prop. 3(2), this time the other direction, obtains $\Sigma \vdash f\{e|_{\Gamma'}/\vec{z}\} : F$ since $(f'\{e|_{\Gamma'}/\vec{z}\})@a = (f'@a)\{e|_{\Gamma'}/\vec{z}\}$. Overall, this proves

$$\Sigma_1, \vec{z} : \Psi'[\Sigma']_{\Gamma'}; \Sigma_2 \gg_{R!a} \Gamma \vdash f : F.$$

as required. The locality conditions follow from (8).

• The treatment of constructors $f = \lambda y.f'$ and $f = ?a.\, f'$ is not difficult, either. These are more interesting, though, since they extend the context as we dive down into the expression f'. We only discuss the latter, the former proceeds analogously. So, suppose $f = ?b.\, f'$ with

$\Sigma \vdash\ ?b.\,(f'\{d/\vec{z}\}) : F$ and $\vec{z}$ occurs free exactly once in f'. From Prop. 4(2) we infer that $F = \forall S.E$ and $\Sigma \gg_{S!b} \emptyset \vdash f'\{d/\vec{z}\} : E$, where b is fresh for Σ, by validity of the context.

By induction hypothesis we have $\Sigma \gg_{S!b} \emptyset = \Sigma_1, \Sigma_2$ and contexts Σ', Ψ', Γ', d' with $d = d'|_{\Gamma'}$, $\Sigma_1 \oplus \Sigma' \vdash d' : \Psi'$ and

$$\Sigma_1, \vec{z} : \Psi'[\Sigma']_{\Gamma'}; \Sigma_2 \vdash f' : E \tag{9}$$

such that Σ' is compatible with Σ_1 and the locality conditions hold. This implies that for all typings $\Sigma_1 \oplus \Sigma' \vdash e : \Psi'$ we have $\Sigma_1, \Sigma_2 \vdash f'\{e|_{\Gamma'}/\vec{z}\} : E$. To this we can apply $\forall R$ and conclude $\Sigma \vdash f\{e|_{\Gamma'}/\vec{z}\} : \forall S.E$ because $?b.\,(f'\{e|_{\Gamma'}/\vec{z}\}) = (?b.\,f')\{e|_{\Gamma'}/\vec{z}\} = f\{e|_{\Gamma'}/\vec{z}\}$. As an admissibility statement for meta-variable $\vec{z}$ this gives two cases:

1. If $\Sigma_1 = \Sigma \gg_{S!b} \emptyset$ and $\Sigma_2 = \emptyset$, then (by compatibility of Σ' with Σ_1) $\Sigma \oplus \Sigma' = \Sigma_1 \oplus \Sigma'$, whence
$$\Sigma, \vec{z} : \Psi'[\Sigma']_{\Gamma'}; \emptyset \vdash f : \forall S.E.$$
Regarding the locality conditions observe that every variable in Γ' which is not bound in f at $\vec{z}$ is not bound in f' at $\vec{z}$. Hence, by induction hypothesis (9) this variable must be a branch variable in the active scope of Σ_1. However, the active scope of Σ_1 is empty. This proves that all variables in Γ' are bound in f' at $\vec{z}$ and therefore in f at $\vec{z}$, too. All other locality conditions follow directly from the induction hypothesis (9). Notice, the depth of Σ is smaller than that of Σ_1.

2. If $\Sigma = \Sigma_1, \Sigma_2'$ and $\Sigma_2' \gg_{S!b} \emptyset = \Sigma_2$ then we must write the meta-variable typing as
$$\Sigma_1, \vec{z} : \Psi'[\Sigma']_{\Gamma'}; \Sigma_2' \vdash f : \forall S.E.$$
and the locality conditions follow directly from (9).

• The constructor $f = (f_1, f_2)$ and destructor $f = f_1\, f_2$ are handled in a similar fashion. We only discuss the latter. Let $f = f_1\, f_2$ and $f\{d/\vec{z}\} = g_1\, g_2$ with $\Sigma \vdash g_1\, g_2 : F$. By Prop. 3(3), $\Sigma \vdash g_1 : D \supset F$ and $\Sigma \vdash g_2 : D$.

We distinguish the two cases, depending on whether $\vec{z}$ is free in f_1 or f_2. First suppose $\vec{z}$ occurs free in f_1 exactly once but not in f_2, i.e., $g_2 = f_2$ and $g_1 = f_1\{d/\vec{z}\}$. The induction hypothesis breaks up the outer context Σ as $\Sigma = \Sigma_1, \Sigma_2$ and provides local contexts Σ', Ψ' together with an expression d' and trunk variables Γ' such that $d = d'|_{\Gamma'}$, $\Sigma_1 \oplus \Sigma' \vdash d' : \Psi'$ and

$$\Sigma_1, \vec{z} : \Psi'[\Sigma']_{\Gamma'}; \Sigma_2 \vdash f_1 : D \supset F \tag{10}$$

plus locality conditions. For any well-typed expression $\Sigma_1 \oplus \Sigma' \vdash e : \Psi'$ the induction hypothesis (10) generates the typing $\Sigma_1, \Sigma_2 \vdash f_1\{e|_{\Gamma'}/\vec{z}\} : D \supset F$ from which an application of Prop. 3(3) in backwards direction yields $\Sigma \vdash (f_1\{e|_{\Gamma'}/\vec{z}\})f_2 : F$. But this is the same as $\Sigma \vdash f\{e|_{\Gamma'}/\vec{z}\} : F$, which proves

$$\Sigma_1, \vec{z} : \Psi'[\Sigma']_{\Gamma'}; \Sigma_2 \vdash f : F$$

for which the locality conditions follow directly from (10). Finally, note that the case where $\vec{z}$ occurs free in f_2 exactly once but not in f_1, i.e., $g_1 = f_1$ and $g_2 = f_2\{d/\vec{z}\}$ is essentially the same.

• Finally, we treat all the cases where the expression at hand has a branch typing $\Sigma \vdash \emptyset \gg_{R!a} f\{d/\vec{z}\} : F$ wholesale, irrespective of the structure of f. We apply the Scope Shift Prop. 5 and get $\Sigma = \Sigma'', R?a\langle\hat{\Gamma}\rangle$ and $\Sigma \gg_{R!a} \Gamma \vdash g : F$ with $f\{d/\vec{z}\} = g|_\Gamma$. Hence, $f'\{d'/\vec{z}\} = g$ such that $f = f'|_\Gamma$ and $d = d'|_{\Gamma'}$, where $\Gamma' \subseteq \Gamma$ is the subset of variables from Γ which are not bound in f' at $\vec{z}$ and occur free in d'.

Now we invoke (by way of implicit induction hypothesis) the appropriate case proved above for the trunk typing $\Sigma \gg_{R!a} \Gamma \vdash f'\{d'/\vec{z}\} : F$, depending on the top-level operator of f'. Therefore, we can split the context $\Sigma \gg_{R!a} \Gamma = \Sigma_1, \Sigma_2$ and extract Σ', Ψ'', d'', Γ'' so that $d' = d''|_{\Gamma''}$, together with $\Sigma_1 \oplus \Sigma' \vdash d'' : \Psi''$ and

$$\Sigma_1, \vec{z} : \Psi''[\Sigma']_{\Gamma''}; \Sigma_2 \vdash f' : F, \tag{11}$$

such that Σ' is compatible with Σ_1 and at least of the same depth. Moreover, all variables in Σ' are bound in f' at $\vec{z}$ or contained in Γ'', and each variable in Γ'' which is not bound in f' at $\vec{z}$ is both a branch variable in the active scope of Σ_1 and a trunk variable in Σ' at the same depth.

Note that $d = d'|_{\Gamma'} = (d''|_{\Gamma''})|_{\Gamma'} = d''|_{\Gamma'' \cup \Gamma'}$. Given any other typing $\Sigma_1 \oplus \Sigma' \vdash e : \Psi''$ the statement (11) implies $\Sigma_1, \Sigma_2 \vdash f'\{e|_{\Gamma''}/\vec{z}\} : F$. To this we apply Ax_f and conclude $\Sigma \vdash \emptyset \gg_{R!a} (f'\{e|_{\Gamma''}/\vec{z}\})|_\Gamma : F$. One shows $(f'\{e|_{\Gamma''}/\vec{z}\})|_\Gamma = (f'|_\Gamma)\{e|_{\Gamma''\cup\Gamma'}/\vec{z}\} = f\{e|_{\Gamma''\cup\Gamma'}/\vec{z}\}$ where we remember that $\Gamma' \subseteq \Gamma$ is the subset of variables from Γ that are not bound in f' at $\vec{z}$. In sum,

$$\Sigma_1 \oplus \Sigma' \vdash e : \Psi'' \quad \Rightarrow \quad \Sigma \vdash \emptyset \gg_{R!a} f\{e|_{\Gamma''\cup\Gamma'}/\vec{z}\} : F. \tag{12}$$

To turn (12) into a typing of f with meta-variable $\vec{z}$ we must consider the context split $\Sigma \gg_{R!a} \Gamma = \Sigma_1, \Sigma_2$ with $\Sigma = \Sigma'', R?a\langle\hat{\Gamma}\rangle$. There are two cases, depending on whether the scoping step $\gg_{R!a}$ is part of Σ_1 or Σ_2. Note that in any case $\Sigma_1 = (\Sigma \gg_{R!a} \Gamma)\lceil n$ for some n, i.e., Σ_1 is a full prefix of $\Sigma \gg_{R!a} \Gamma$.

(1) Suppose that $\Sigma_1 = \Sigma \gg_{R!a} \Gamma$ and $\Sigma_2 = \emptyset$. Then, $n-1$ is the depth of Σ and scope Γ at depth n in Σ_1. Since Σ' is compatible with Σ_1 and has at least the same depth, it can be split up as $\Sigma' = \Sigma'_1 \gg_{R!a} \Sigma'_2$ so that $\Sigma_1 \oplus \Sigma' = \Sigma \oplus (\Sigma'_1 \gg_{R!a} \Gamma, \Sigma'_2)$. Then, by (12) we have shown

$$\Sigma, \vec{z} : \Psi''[\Sigma'_1 \gg_{R!a} \Gamma, \Sigma'_2]_{\Gamma''\cup\Gamma'}; \emptyset \vdash \emptyset \gg_{R!a} f : F.$$

We claim that the locality condition is preserved. Since Γ (i.e., the active scope of Σ_1) does not contain branch variables, by the locality condition that comes with the induction hypothesis (11), all variables in Γ'' must be bound in f' at $\vec{z}$ and thus also in f at $\vec{z}$. The variables in Γ' are not bound in f' at $\vec{z}$, but they are (i) branch variables in the active scope of Σ (because $\Sigma = \Sigma'', R?a\langle\hat{\Gamma}\rangle$ and $\Gamma' \subseteq \Gamma$) and (ii) trunk variables at the same depth in $\Sigma'_1 \gg_{R!a} \Gamma, \Sigma'_2$ (because the depth of Σ'_1 is identical to that of Σ and $\Gamma' \subseteq \Gamma$). Further, by induction hypothesis (11) all variables in $\Sigma'_1 \gg_{R!a} \Sigma'_2 = \Sigma'$ which are not bound in f at $\vec{z}$ (and thus not bound in f' at $\vec{z}$) are contained in $\Gamma'' \subseteq \Gamma' \cup \Gamma''$. By definition, all variables in Γ that are not bound in f at $\vec{z}$ are included in $\Gamma' \subseteq \Gamma' \cup \Gamma''$.

(2) The remaining case is $\Sigma_2 = \Sigma'_2 \gg_{R!a} \Gamma$. Then, $\Sigma = \Sigma'', R?a\langle\hat{\Gamma}\rangle = \Sigma_1, \Sigma'_2$ and $d = d'|_{\Gamma'} =$

$(d''|_{\Gamma''})|_{\Gamma'} = d''|_{\Gamma'' \cup \Gamma'}$. Then, (12) means

$$\Sigma_1, \vec{z} : \Psi''[\Sigma']_{\Gamma'' \cup \Gamma'}; \Sigma_2' \vdash \emptyset \gg_{R!a} f : F.$$

We claim that $\Gamma' \subseteq \Gamma''$ and thus the locality condition holds for this admissible typing by induction hypothesis (11). Take any variable $x \in \Gamma'$. By definition x is occurs free in d'. If it does not occur free in d'' then we must have $x \in \Gamma''$ because $d' = d''|_{\Gamma''}$. If x is free in d'' then it must be in Σ_1 or in Σ' because of the typing $\Sigma_1 \oplus \Sigma' \vdash d'' : \Psi''$. But $x \in \Gamma' \subseteq \Gamma$ cannot be in Σ_1 because all trunk variables Γ are in fact part of $\Sigma_2 = \Sigma_2' \gg_{R!a} \Gamma$ by assumption, since the context $\Sigma \gg_{R!a} \Gamma = \Sigma_1, \Sigma_2$ is well-formed. Hence, x is a trunk variable in Σ' and (11) implies $x \in \Gamma''$ as claimed. □

With substitution inversion Prop. 8 it is straight-forward to complete the proof of Subject Reduction Prop. 6, i.e., that contractions preserve typing in arbitrary syntactic contexts. Let $e_1 \longrightarrow_{\beta\gamma} e_2$ be a β- or γ-contraction and $\Sigma \vdash f\{e_1/\vec{z}\} : \Psi$ a single occurrence of the redex e_1 inside a well-typed syntactic context f. By Prop. 8, $e_1 = e_1'[\![\hat{\Gamma}'/\Gamma']\!]$ for some set of trunk variables Γ' such that e_1' is typeable as $\Sigma_1 \oplus \Sigma' \vdash e_1' : \Psi'$ in some local context $\Sigma_1 \oplus \Sigma'$ extending an initial prefix $\Sigma_1 = \Sigma \lceil n$ of Σ. Moreover, we have $\Sigma_1, \vec{z} : \Psi'[\Sigma']_{\Gamma'}; \Sigma_2 \vdash f : \Psi$ where $\Sigma = \Sigma_1, \Sigma_2$. Since contractions commute with the (capture avoiding) substitution $[\![\hat{\Gamma}'/\Gamma']\!]$ we have $e_1' \longrightarrow_{\beta\gamma} e_2'$ for some e_2' with $e_2 = e_2'[\![\hat{\Gamma}'/\Gamma']\!]$. On the other hand, basic contractions are type preserving (Prop. 7), which means $\Sigma_1 \oplus \Sigma' \vdash e_2' : \Psi'$. Then, the context typing $\Sigma_1, \vec{z} : \Psi'[\Sigma']_{\Gamma'}; \Sigma_2 \vdash f : \Psi$ with meta-variable $\vec{z}$ ensures that $\Sigma \vdash f\{e_2/\vec{z}\} : \Psi$ as desired.

9 Strong Normalisation

To show strong normalisation we use the standard technique of embedding $\lambda\mathsf{CK}_n$ faithfully into the simply typed λ-calculus with finite products and sums so that reductions in the former are bounded by the length of reductions in the latter. Since the latter is strongly normalising [3], the former must be, too. A natural embedding is to map all universal modalities $\forall R.C$ into function types $\mathsf{Ct}_R \supset C$ and all existentials $\exists R.C$ into product types $(\mathsf{Ct}_R \times C) + (\mathsf{Ct}_R \times C)$ where Ct_R represents some fixed but arbitrary type, parametrised in $R \in N_R$ to encode contexts explicitly. In other words, we identify contexts with the objects of type Ct_R and take scope abstractions to be functional abstractions over this parameter. Context closures are objects paired with elements from Ct_R, plus a choice in order to interpret the commuting conversion of the `let`. The syntactic translation on types is

$$\begin{array}{rclrcl}
A^{st} & = & A & (C \supset D)^{st} & = & C^{st} \to D^{st} \\
(C \wedge D)^{st} & = & C^{st} \times D^{st} & (\forall R.C)^{st} & = & \mathsf{Ct}_R \to D^{st} \\
(C \vee D)^{st} & = & C^{st} + D^{st} & (\exists R.C)^{st} & = & (\mathsf{Ct}_R \times C^{st}) + (\mathsf{Ct}_R \times C^{st})
\end{array}$$

where A is an atomic type. On terms we put

$$\begin{array}{rcl@{\qquad}rcl}
x^{st} &=& x & (e_1, e_2)^{st} &=& ({e_1}^{st}, {e_2}^{st}) \\
a^{st} &=& a & (\pi_i\, e)^{st} &=& \pi_i\, e^{st} \\
(\lambda x.e)^{st} &=& \lambda x.e^{st} & (\iota_i\, e)^{st} &=& \iota_i\, e^{st} \\
(e_1\, e_2)^{st} &=& {e_1}^{st}\, {e_2}^{st} & \multicolumn{3}{l}{(\texttt{case}\, e\, \texttt{of}\, [\iota_1\, y_1 \to e_1 \mid \iota_2\, y_2 \to e_2])^{st}} \\
(!b.\, e)^{st} &=& \iota_1(b, e^{st}) & &=& \texttt{case}\, e^{st}\, \texttt{of}\, [\iota_1\, y_1 \to {e_1}^{st} \mid \iota_2\, y_2 \to {e_2}^{st}] \\
(?a.\, e)^{st} &=& \lambda a.e^{st} & \multicolumn{3}{l}{(\texttt{let}\ !a.\, \hat{x} = e_1\, \texttt{in}\, e_2)^{st}} \\
(e@a)^{st} &=& e^{st}\, a & &=& \texttt{case}\, {e_1}^{st}\, \texttt{of}[\iota_1(a, x) \to {e_2}^{st} \mid \iota_2(a, x) \to {e_2}^{st}]
\end{array}$$

where $\hat{x}, x$ and a are branch, trunk and filler scope variables, respectively.

Proposition 9 (Strong Normalisation)**.** The reduction relation $\longrightarrow\!\!\!\!\rightarrow_{\beta\gamma}$ is strongly normalising on well-typed expressions.

Proof. The interpretation $(\cdot)^{st}$ is extended naturally to typing contexts and typing statements. Every path context Σ is abstracted into a simply typed context Σ^{st} by putting $(x : C, \Sigma)^{st} = x : C^{st}, \Sigma^{st}$, $(R?a\langle\hat{\Gamma}\rangle, \Sigma)^{st} = \Gamma^{st}, a : \mathsf{Ct}_R, \Sigma^{st}$ and $(R!a[\Sigma])^{st} = a : \mathsf{Ct}_R, \Sigma^{st}$, which removes all prefixes $R?a\langle\cdot\rangle$ and $R!a[\cdot]$ and creates a single scope. Note that if a variable occurs both as a trunk and a branch, which is possible only at the same depth, by validity of the context both typings have the same type and thus can be collapsed. For instance, the context $\Sigma = \{x : A, y : B, R?b\langle\hat{z} : C\rangle, R!b[v : B, z : C, T?c\langle\hat{w} : D\rangle]\}$ turns into

$$\Sigma^{st} = \{x : A^{st}, y : B^{st}, b : \mathsf{Ct}_R, z : C^{st}, v : B^{st}, c : \mathsf{Ct}_T, w : D^{st}\}$$

and a typing $\Psi = \emptyset \gg_{R!a} E$ becomes $\Psi^{st} = E^{st}$. One then shows by induction on the structure of typing trees that whenever $\Sigma \vdash e : \Psi$ we have $\Sigma^{st} \vdash_\lambda e^{st} : \Psi^{st}$ in the simply typed λ-calculus with finite products and sums.

The next step is to show that every contraction $e_1 \longrightarrow_{\beta\gamma} e_2$ is preserved across the collapse, i.e., ${e_1}^{st} \longrightarrow_\lambda {e_2}^{st}$, where $\longrightarrow_\lambda$ denotes a rewriting step in the simply typed λ-calculus. We do this for the basic reductions from Fig. 3. For $\beta\lambda$ this is immediate because $((\lambda x.e_1)\, e_2)^{st} = (\lambda x.{e_1}^{st})\, {e_2}^{st} \longrightarrow_\lambda {e_1}^{st}[\![{e_2}^{st}/x]\!] = (e_1[\![e_2/x]\!])^{st}$ since the collapse function $(\cdot)^{st}$ distributes over substitution. Next, consider a reduction $[\beta]$ of the form $(?a.\, e)@b \longrightarrow_\beta e[\![b/a]\!]$. We have $((?a.\, e)@b)^{st} = (\lambda a.e^{st})\, b \longrightarrow_\lambda e^{st}[\![b/a]\!] = (e[\![b/a]\!])^{st}$. Finally a $\langle\beta\rangle$-contraction $\texttt{let}\ !a.\, \hat{x} = !b.\, e_1\, \texttt{in}\, e_2 \longrightarrow_\beta e_2[\![e_1/x, b/a]\!]$ also translates into a λ-reduction, viz.

$$\begin{array}{ll}
\multicolumn{2}{l}{(\texttt{let}\ !a.\, \hat{x} = !b.\, e_1\, \texttt{in}\, e_2)^{st}} \\
= & \texttt{case}\ (!b.\, e_1)^{st}\ \texttt{of}\ [\iota_1(a, x) \to {e_2}^{st} \mid \iota_2(a, x) \to {e_2}^{st}] \\
= & \texttt{case}\ \iota_1(b, {e_1}^{st})\ \texttt{of}\ [\iota_1(a, x) \to {e_2}^{st} \mid \iota_2(a, x) \to {e_2}^{st}] \\
\longrightarrow_\lambda & {e_2}^{st}[\![b/a, {e_1}^{st}/x]\!]\ =\ (e_2[\![b/a, e_1/x]\!])^{st}.
\end{array}$$

Commuting contractions are also preserved. As an example consider the γ-contraction $\gamma\mathtt{let}_5$:

$$
\begin{array}{ll}
\multicolumn{2}{l}{(\mathtt{let}\ !a.\,\hat{y} = \mathtt{let}\ !b.\,\hat{z} = e_1\,\mathtt{in}\,e_2\,\mathtt{in}\,e_3)^{st}} \\
= & \mathtt{case}\ (\mathtt{let}\ !b.\,\hat{z} = e_1\,\mathtt{in}\,e_2)^{st}\ \mathtt{of}\ [\iota_1(a,y) \to {e_3}^{st} \mid \iota_2(a,y) \to {e_3}^{st}] \\
= & \mathtt{case}\,(\mathtt{case}\ {e_1}^{st}\ \mathtt{of}\ [\iota_1(b,z) \to {e_2}^{st} \mid \iota_2(b,z) \to {e_2}^{st}])\ \mathtt{of}\ [\iota_1(a,y) \to {e_3}^{st} \mid \iota_2(a,y) \to {e_3}^{st}] \\
\longrightarrow_\lambda & \mathtt{case}\ {e_1}^{st}\ \mathtt{of}\ [\iota_1(b,z) \to \mathtt{case}\ {e_2}^{st}\ \mathtt{of}\ [\iota_1(a,y) \to {e_3}^{st} \mid \iota_2(a,y) \to {e_3}^{st}] \\
 & \qquad\qquad\qquad \mid \iota_2(b,z) \to \mathtt{case}\ {e_2}^{st}\ \mathtt{of}\ [\iota_1(a,y) \to {e_3}^{st} \mid \iota_2(a,y) \to {e_3}^{st}]\,] \\
= & \mathtt{case}\ {e_1}^{st}\ \mathtt{of}\ [\iota_1(b,z) \to (\mathtt{let}\ !a.\,\hat{y} = e_2\,\mathtt{in}\,e_3)^{st} \mid \iota_2(b,z) \to (\mathtt{let}\ !a.\,\hat{y} = e_2\,\mathtt{in}\,e_3)^{st}] \\
= & (\mathtt{let}\ !b.\,\hat{z} = e_1\,\mathtt{in}\,\mathtt{let}\ !a.\,\hat{y} = e_2\,\mathtt{in}\,e_3)^{st}.
\end{array}
$$

The commutation $\longrightarrow_\lambda$ is sound under the side condition of $\gamma\mathtt{let}_5$, viz. that b, z are not free in e_3. □

10 Confluence

Proposition 10 (Confluence)**.** Let e, e_1, e_2 be $\lambda\mathsf{CK}_n$ expressions. Then, if $e \twoheadrightarrow_\beta e_1$ and $e \twoheadrightarrow_\beta e_2$ then there exists e' such that $e_1 \twoheadrightarrow_\beta e'$ and $e_2 \twoheadrightarrow_\beta e'$.

Proof. The proof follows the standard method introduced by Tait and Martin-Löf [2]. Its main tool is the definition of parallel β-reduction. $e \longrightarrow_{p\beta} e'$ in which a set of existing redexes in e are reduced in *parallel* in one step to e'. The proof of confluence consists of three parts. First, one shows that $\longrightarrow_{p\beta}$ enjoys the diamond property, i.e., if $e \longrightarrow_{p\beta} e_1$ and $e \longrightarrow_{p\beta} e_2$ then there exists e' such that $e_1 \longrightarrow_{p\beta} e'$ and $e_2 \longrightarrow_{p\beta} e'$. Secondly, one verifies that the diamond property of $\longrightarrow_{p\beta}$ implies the diamond property of its reflexive, transitive closure $\twoheadrightarrow_{p\beta}$ by a straightforward induction and application of a strip lemma. The final step is to deduce the confluence of $\twoheadrightarrow_\beta$ from the diamond property of $\twoheadrightarrow_{p\beta}$ by proving $\twoheadrightarrow_{p\beta} = \twoheadrightarrow_\beta$.

We have formalised the proof of confluence in the proof-assistant Abella [4] which supports higher-order abstract syntax (HOAS) [10] and has been developed by Andrew Gacek, based on the work of Gacek, Miller and Nadathur [5, 6]. In HOAS-based proof-assistants induction usually requires to consider predicates inside contexts of local assumptions which formalise the structure of typed settings. Since the type information is unnecessary in the confluence proof, our development adopts the technique of Accattoli [1] which allows us to circumvent contexts (which are artifacts in the untyped case, anyway). The Abella source code can be found in the appendix, Sec. A. We greatfully acknowledge Alberto Momigliano's contribution who suggested the use of Abella and assisted us in getting started with the formalisation. □

References

[1] B. Accattoli. Proof Pearl: Abella Formalization of λ-Calculus Cube Property. In Ch. Hawblitzel and D. Miller, editors, *CPP*, volume 7679 of *Lecture Notes in Computer Science*, pages 173–187. Springer, 2012.

[2] H. P. Barendregt. *The Lambda Calculus – Its Syntax and Semantics*, volume 103. North-Holland, 1984.

[3] P. de Groote. On the strong normalisation of intuitionistic natural deduction with permutation-conversions. *Information and Computation*, 178(2):441–464, 2002.

[4] A. Gacek. The Abella Interactive Theorem Prover (System Description). In *Proceedings of the 4th international joint conference on Automated Reasoning*, IJCAR '08, pages 154–161, Berlin, Heidelberg, 2008. Springer-Verlag.

[5] A. Gacek, D. Miller, and G. Nadathur. Combining Generic Judgments with Recursive Definitions. In *Proceedings of the 2008 23rd Annual IEEE Symposium on Logic in Computer Science*, LICS '08, pages 33–44, Washington, DC, USA, 2008. IEEE Computer Society.

[6] A. Gacek, D. Miller, and G. Nadathur. Reasoning in Abella about Structural Operational Semantics Specifications. *Electronic Notes in Theoretical Computer Science*, 228:85–100, January 2009.

[7] M. Mendler and S. Scheele. Cut-free Gentzen calculus for multimodal CK. *Information and Computation*, 209:1465–1490, 2011.

[8] M. Mendler and S. Scheele. On the computational interpretation of CK_n for contextual information processing. *Fundamenta Informaticae*, 130:1–39, 2014.

[9] A. Nanevski, F. Pfenning, and B. Pientka. Contextual modal type theory. *ACM Transactions on Computational Logic*, 9(3):1–49, June 2008.

[10] F. Pfenning and C. Elliot. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, pages 199–208, New York, NY, USA, 1988. ACM.

A Abella Code

A.1 Abella Signature for $\lambda\mathsf{CK}_n$

```
%%% Tait/Martin-Loef style proof of Church-Rosser using parallel reductions
%%
%%% Stephan Scheele, Alberto Momigliano :: Dec. 2012
%
%% This proof shows confluence for the (monomodal fragment of) lambda-CKn
%% for closed terms. Commuting conversions are omitted.
%
%% This work extends the work by Accattoli who gave an Abella proof of
%% confluence for the simple lambda calculus. Here, we extend this by
%% products, sums and contextual terms and associated reductions.

sig crcknV2.

kind     tm                 type.
kind     sc                 type.

type     app                tm -> tm -> tm.
type     abs                (tm -> tm) -> tm.

% products
type     pair               tm -> tm -> tm.
type     fst                tm -> tm.
type     snd                tm -> tm.

% sums
type      case              tm -> (tm -> tm) -> (tm -> tm) -> tm.
type      inl               tm -> tm.
type      inr               tm -> tm.

% diamond contextual type
type     let                tm -> (sc -> tm -> tm) -> tm.
type     dia                sc -> tm -> tm.

% box contextual type
type     box                (sc -> tm) -> tm.
type     atw              sc -> tm -> tm.
```

A.2 Abella Proof Script

```
%%% Tait/Martin-Loef style proof of Church-Rosser using parallel reduction
%
%%% Stephan Scheele, Alberto Momigliano :: Dec. 2012
%
%% This proof shows confluence for the (monomodal fragment of) lambda-CKn
%% for closed terms. Commuting conversions are omitted.
%
%% This work extends the work by Accattoli who gave an Abella proof of
%% confluence for the simple lambda calculus. Here, we extend this by
%% products, sums and contextual terms and associated reductions.

Specification "crcknV2".

% enforce subordination rel.
Close tm, sc.

% set level of subgoals presented by Abella
Set subgoals 2.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% The following part is copycat + extensions of Accattoli's Abella proof of confluence of beta
%% reduction for the simple lambda calculus.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
%% We prove confluence of beta reduction through various steps:
%%   - we define the term predicate to do induction on terms (Abella currently does
%%     not allow induction on types) and prove a substitution lemma for it
%%   - we prove the diamond property for the transitive closure of par1 (noted parN). The proof uses
%%     the diamond property of par1 and a strip lemma.
%%   - we define usual beta reduction (called beta_red1) and its transitive closure (called beta_redn).
%%   - we relate parN and beta_redn, obtaining confluence (from the diamond property of parN).
%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%



%%%%%%%
%%% The term and scope predicates
%%%%%%

Define term : tm -> prop,  scope : sc -> prop by
        nabla x, term x;
        nabla s, scope s;
term (app M N)   := term M /\ term N;
term (abs FM)  := nabla x, term (FM x);
% contextual terms
term (let M FN)  := nabla s, nabla x, term (FN s x) /\ term M;
term (dia S M)  := scope S /\ term M;
term (box FN)  := nabla s, term (FN s);
term (atw R M)  := scope R /\ term M;
% product and sums
        term (pair L R)  := term L /\ term R;
        term (fst L)     := term L;
        term (snd R)     := term R;
        term (case C FN FM) := term C /\ nabla x, term (FN x) /\ nabla y, term (FM y);
        term (inl N)     := term N;
        term (inr M)     := term M.



% term substitution lemma
Theorem tm_sub : forall M N, nabla x,
term (M x) ->
term N -> term (M N).
induction on 1. intros. case H1.
search.
search.
                        apply IH to H3 H2. apply IH to H4 H2. search.
apply IH to H3 H2. search.
                        apply IH to H3 H2. apply IH to H4 H2. search.
                        apply IH to H4 H2. case H3. search.
                        apply IH to H3 H2. search.
                        apply IH to H4 H2. case H3. search.
                        apply IH to H3 H2. apply IH to H4 H2. search.
                        apply IH to H3 H2. search.
                        apply IH to H3 H2. search.
                        apply IH to H3 H2. apply IH to H4 H2. apply IH to H5 H2. search.
                        apply IH to H3 H2. search.
                        apply IH to H3 H2. search.


% scope substitution lemma
Theorem sc_sub : forall M S, nabla x,
term (M x) ->
```

```
scope S -> term (M S).
induction on 1. intros. case H1.
search.
                        apply IH to H3 H2. apply IH to H4 H2. search.
apply IH to H3 H2. search.
                        apply IH to H3 H2. apply IH to H4 H2. search.
                        apply IH to H4 H2. case H3. search.search.
                        apply IH to H3 H2. search.
                        apply IH to H4 H2. case H3. search.search.
                       % products and sums
                        apply IH to H3 H2. apply IH to H4 H2. search.
                        apply IH to H3 H2. search.
                        apply IH to H3 H2. search.
                        apply IH to H3 H2. apply IH to H4 H2. apply IH to H5 H2. search.
                        apply IH to H3 H2. search.
                        apply IH to H3 H2. search.



% definition of parallel reduction including let, dia, box and atw (con/de)-structors
Define par1 : tm -> tm -> prop, pars1: sc -> sc -> prop by
       nabla x , par1 x x;
       nabla s, pars1 s s;
       par1 (app T1 S1) (app T2 S2)  := par1 T1 T2 /\ par1 S1 S2;
       par1 (abs S1) (abs S2) := nabla x,  par1 (S1 x) (S2 x);
       par1 (app (abs T1) S1) (T2 S2) := nabla x, par1 (T1 x) (T2 x) /\ par1 S1 S2;
% contextual
       par1 (let M FN) (let M' FN') := par1 M M' /\ nabla s, nabla x, par1 (FN s x) (FN' s x);
       par1 (dia S M) (dia S' M') := pars1 S S' /\ par1 M M';
       par1 (let (dia S M) FP) (FP' S' M') := pars1 S S' /\ par1 M M' /\ nabla s, nabla x, par1 (FP s x) (FP' s x);
       par1 (box FN) (box FN') :=  nabla s, par1 (FN s) (FN' s);
       par1 (atw R (box FN)) (FN' R') := pars1 R R' /\  nabla s, par1 (FN s) (FN' s);
       par1 (atw R M) (atw R' N) := pars1 R R' /\ par1 M N;
% products
       par1 (fst M) (fst N)      := par1 M N;
       par1 (snd M) (snd N)      := par1 M N;
       par1 (pair T1 S1) (pair T2 S2) := par1 T1 T2 /\ par1 S1 S2;
       par1 (fst (pair T1 S1)) T2     := par1 T1 T2 /\ term S1;
       par1 (snd (pair T1 S1)) S2     := par1 S1 S2 /\ term T1;
% sums
       par1 (inl M) (inl N)      := par1 M N;
       par1 (inr M) (inr N)      := par1 M N;
       par1 (case T1 FN FM) (case T2 FN' FM')  := par1 T1 T2 /\ nabla x, par1 (FN x) (FN' x) /\
                                                    nabla y, par1 (FM y) (FM' y);
       par1 (case (inl M) FN FM) (FN' M')      := par1 M M' /\ nabla x, par1 (FN x) (FN' x) /\ nabla y, term (FM y);
       par1 (case (inr M) FN FM) (FM' M')      := par1 M M' /\ nabla x, par1 (FM x) (FM' x) /\ nabla y, term (FN y).



% the result of a par1 is a term/scope
Theorem par1_tm_sc :
(forall M N, par1 M N -> term M /\ term N) /\
(forall R S, pars1 R S -> scope R /\ scope S).
induction on 1 1. split. intros. case H1.
search.
apply IH to H2. apply IH to H3. search.
apply IH to H2. search.
apply IH to H2. apply IH to H3. apply tm_sub to H5 H7. search.
                        apply IH to H2. apply IH to H3. apply tm_sub to H6 H4. apply tm_sub to H7 H5.
                              search.
                        apply IH to H3. case H2.search.
                        apply IH to H3. apply IH to H4. apply IH1 to H2. apply sc_sub to H7 H9.
                              apply sc_sub to H8 H10. apply tm_sub to H11 H5.
                              apply tm_sub to H12 H6. search.
                        apply IH to H2. search.
                        apply IH1 to H2. apply IH to H3. apply sc_sub to H6 H4. apply sc_sub to H7 H5. search.
                        apply IH to H3. apply IH1 to H2. search.
                        % products and sums
                        apply IH to H2. search.
                        apply IH to H2. search.
                        apply IH to H2. apply IH to H3. search.
                        apply IH to H2. search.
                        apply IH to H2. search.
                        apply IH to H2. search.
                        apply IH to H2. search.
                        apply IH to H2. apply IH to H3. apply IH to H4. search.
                        apply IH to H2. apply IH to H3. apply tm_sub to H8 H6. search.
                        apply IH to H2. apply IH to H3. apply tm_sub to H8 H6. search.


                        intros. case H1. search.


Split par1_tm_sc as par1_tm, par1_sc.


% substitution lemma for parallel reduction par1
Theorem par1_pars1_subst_lem:
(forall M N K1 K2, nabla x,
par1 (M x) (N x) -> par1 K1 K2 -> par1 (M K1) (N K2)) /\
(forall M N K1 K2, nabla x,
par1 (M x) (N x) -> pars1 K1 K2 -> par1 (M K1) (N K2)).
```

```
induction on 1 1. split.
        intros. case H1. search. search.
        apply IH to H3 H2. apply IH to H4 H2. search.
        apply IH to H3 H2. search.
        apply IH to H4 H2. apply IH to H3 H2. search.
        apply IH to H3 H2. apply IH to H4 H2. search.
        apply IH to H4 H2. unfold. search.
        search.
        apply IH to H4 H2. apply IH to H5 H2. search.
        apply IH to H3 H2. search.
        apply IH to H4 H2. case H3. search.
        apply IH to H4 H2. case H3. search.
% products and sums
        apply IH to H3 H2. search.
        apply IH to H3 H2. search.
        apply IH to H3 H2. apply IH to H4 H2. search.
        apply IH to H3 H2. apply par1_tm to H2. apply tm_sub to H4 H6. search.
        apply IH to H3 H2. apply par1_tm to H2. apply tm_sub to H4 H6. search.
        apply IH to H3 H2. search.
        apply IH to H3 H2. search.
        apply IH to H3 H2. apply IH to H4 H2. apply IH to H5 H2. search.
        apply IH to H3 H2. apply IH to H4 H2. apply par1_tm to H2. apply tm_sub to H5 H8. search.
        apply IH to H3 H2. apply IH to H4 H2. apply par1_tm to H2. apply tm_sub to H5 H8.  search.


        intros. case H1. search.
        apply IH1 to H3 H2. apply IH1 to H4 H2. search.
        apply IH1 to H3 H2. search.
        apply IH1 to H4 H2. apply IH1 to H3 H2. search.
        apply IH1 to H3 H2. apply IH1 to H4 H2. search.
        apply IH1 to H4 H2. unfold. case H3. search. search. search.
        apply IH1 to H4 H2. apply IH1 to H5 H2. case H3. search.  search.
        apply IH1 to H3 H2. search.
        apply IH1 to H4 H2. case H2. case H3.  search. search. search.
        apply IH1 to H4 H2. case H3. search. search.
% products and sums
        apply IH1 to H3 H2. search.
        apply IH1 to H3 H2. search.
        apply IH1 to H3 H2. apply IH1 to H4 H2. search.
        apply IH1 to H3 H2. apply par1_sc to H2. apply sc_sub to H4 H6. search.
        apply IH1 to H3 H2. apply par1_sc to H2. apply sc_sub to H4 H6. search.
        apply IH1 to H3 H2. search.
        apply IH1 to H3 H2. search.
        apply IH1 to H3 H2. apply IH1 to H4 H2. apply IH1 to H5 H2. search.
        apply IH1 to H3 H2. apply IH1 to H4 H2. apply par1_sc to H2. apply sc_sub to H5 H8. search.
        apply IH1 to H3 H2. apply IH1 to H4 H2. apply par1_sc to H2. apply sc_sub to H5 H8.  search.


Split par1_pars1_subst_lem as par1_subst_lem, pars1_subst_lem.


% diamond property for par1
Theorem par1_diamond : forall T A1 A2,
par1 T A1 -> par1 T A2 -> exists V, par1 A1 V /\ par1 A2 V.
induction on 1. intros. case H1.
% var
          case H2. search.
% app
          case H2.
% app app
          apply IH to H3 H5. apply IH to H4 H6. search.
% app abs
          case H3. apply IH to H4 H6. apply IH to H7 H5. apply par1_subst_lem to H11 H9. search.
% abs
          case H2. apply IH to H3 H4. search.
% app abs
          case H2. case H5. apply IH to H3 H7. apply IH to H4 H6. apply par1_subst_lem to H8 H10.
               apply par1_subst_lem to H9 H11. search.
% app
          apply IH to H3 H5. apply IH to H4 H6. apply par1_subst_lem to H7 H9.
                apply par1_subst_lem to H8 H10. search.
% let
          case H2. apply IH to H3 H5. apply IH to H4 H6. search.
% let dia
          case H3. apply IH to H4 H7. apply IH to H9 H6. apply pars1_subst_lem to H10 H5.
               apply pars1_subst_lem to H11 H5. apply par1_subst_lem to H14 H12.
               apply par1_subst_lem to H15 H13. case H8. case H5. search.
% dia dia
          case H2. apply IH to H4 H6. case H3. case H5. search.
% app let dia
          case H2. apply IH to H5 H7. case H6.apply IH to H4 H11. apply pars1_subst_lem to H8 H10.
               apply pars1_subst_lem to H9 H10. apply par1_subst_lem to H14 H12.
               apply par1_subst_lem to H15 H13. case H3. case H10. search.
% FP case
          apply IH to H4 H7. apply IH to H5 H8. apply pars1_subst_lem to H11 H6.
                apply pars1_subst_lem to H12 H6. apply par1_subst_lem to H13 H9.
                apply par1_subst_lem to H14 H10. case H3. case H6. search.
% box new
          case H2. apply IH to H3 H4. search.
% atw
          case H2. apply IH to H4 H6. apply pars1_subst_lem to H7 H5.
          apply pars1_subst_lem to H8 H5. case H3. case H5. search.
          case H6. apply IH to H4 H7. apply pars1_subst_lem to H8 H3. apply pars1_subst_lem to H9 H5.
              case H3. case H5. search.
          case H2. case H4. apply IH to H7 H6. apply pars1_subst_lem to H8 H3.
               apply pars1_subst_lem to H9 H5. case H3. case H5. search.
          apply IH to H4 H6. case H3. case H5. search.
% fst
          case H2. apply IH to H3 H4. search.
% fst pair
          case H3. apply IH to H6 H4. apply par1_tm to H7. search.
% snd
          case H2. apply IH to H3 H4. search.
% snd pair
          case H3. apply IH to H7 H4. apply par1_tm to H6. search.
% pair pair
          case H2. apply IH to H3 H5. apply IH to H4 H6. search.
% fst pair
          case H2. case H5. apply IH to H3 H6. apply par1_tm to H7. search.
          apply IH to H3 H5. search.
% snd pair
          case H2. case H5. apply IH to H3 H7. apply par1_tm to H6. search.
          apply IH to H3 H5. search.
% inl
          case H2. apply IH to H3 H4. search.
% inr
          case H2. apply IH to H3 H4. search.
% case
          case H2. apply IH to H3 H6. apply IH to H4 H7. apply IH to H5 H8. search.
% case inl
          case H3. apply IH to H9 H6. apply IH to H4 H7. apply par1_subst_lem to H12 H10.
          apply par1_subst_lem to H13 H11. apply par1_tm to H5. search.
% case inr
          case H3. apply IH to H9 H6. apply IH to H5 H7. apply par1_subst_lem to H12 H10.
          apply par1_subst_lem to H13 H11. apply par1_tm to H4. search.
% case inl R
          case H2. case H6. apply IH to H3 H9. apply IH to H4 H7. apply par1_subst_lem to H12 H10.
          apply par1_subst_lem to H13 H11. apply par1_tm to H8. search.
          apply IH to H3 H6. apply IH to H4 H7. apply par1_subst_lem to H11 H9. apply par1_subst_lem to H12 H10. search.
% case inr R
          case H2. case H6. apply IH to H3 H9. apply IH to H4 H8. apply par1_subst_lem to H12 H10.
          apply par1_subst_lem to H13 H11. apply par1_tm to H7. search.
          apply IH to H3 H6. apply IH to H4 H7. apply par1_subst_lem to H11 H9. apply par1_subst_lem to H12 H10. search.


%%%%%%%%
%% The reflexive and transitive closure parN of par1
%%%%%%%%

Define parN : tm -> tm -> prop, parsN : sc -> sc -> prop by
parN M M  := term M;
parN M N  := par1 M N;
parN M N  := exists S, parN M S /\ parN S N;
        parsN R R := scope R;
        parsN R T := pars1 R T;
        parsN R T := exists S, parsN R S /\ parsN S T.



Theorem parN_parsN_tm_sc :
(forall M N, parN M N -> term M /\ term N) /\
(forall R S, parsN R S -> scope R /\ scope S).
induction on 1 1. split. intros. case H1.
search.
apply par1_tm_sc. apply H3 to H2. search.
                        apply IH to H2. apply IH to H3. search.

                        intros. case H1. search. apply par1_tm_sc. apply H4 to H2. search.
                        apply IH1 to H2. apply IH1 to H3. search.

% strip lemma
Theorem parN_parsN_strip :
(forall X Y Z, parN X Y  -> par1 X Z  -> exists W, parN Z W  /\ par1 Y W) /\
(forall X Y Z, parsN X Y -> pars1 X Z -> exists W, parsN Z W /\ pars1 Y W).
induction on 1 1. split. intros. case H1.
apply par1_tm_sc. apply H4 to H2. search.
apply par1_diamond to H2 H3. search.
apply IH to H3 H2. apply IH to H4 H6. search.

                        intros. case H1. apply par1_tm_sc. apply H5 to H2. search.
                        case H2. case H3.search.
                        apply IH1 to H3 H2. apply IH1 to H4 H6. search.


Theorem parN_diamond : forall X Y Z,
parN X Y ->
parN X Z -> exists U, parN Z U /\ parN Y U.
induction on 1. intros. case H1.
apply parN_parsN_tm_sc. apply H4 to H2. assert parN Z Z. search.
                        apply parN_parsN_strip. apply H4 to H2 H3. search.
apply IH to H3 H2. apply IH to H4 H6. search.
```

```
%%%%%%%%%%%%%%%%%%%%%
%%% beta reduction (one step)
%%%%%%%%%%%%%%%%%%%%%


% define contextual 1-step beta reduction, beta for beta reductions and ...
Define beta : tm -> tm -> prop by
       beta (app (abs M) N) (M N)           := nabla x, term (M x) /\ term N;
       beta (let (dia S M) FP) (FP S M)     := nabla s, nabla x, term (FP s x) /\ scope S /\ term M;
       beta (atw R (box FN)) (FN R)         := nabla s, term (FN s) /\ scope R;
% products and sums
       beta (fst (pair L R)) L              := term L /\ term R;
       beta (snd (pair L R)) R              := term L /\ term R;
% sums
       beta (case (inl M) FN FM) (FN M)     := term M /\ nabla x, term (FN x) /\ nabla y, term (FM y);
       beta (case (inr M) FN FM) (FM M)     := term M /\ nabla x, term (FN x) /\ nabla y, term (FM y).



% ... beta_red1 captures the congruences + beta
Define beta_red1 : tm -> tm -> prop, beta_reds1 : sc -> sc -> prop by
       beta_reds1 R S := scope R /\ scope S;
       beta_red1 T S  := beta T S;
       beta_red1 (app M N) (app S N)                := beta_red1 M S /\ term N;
       beta_red1 (app N M) (app N S)             := beta_red1 M S /\ term N;
       beta_red1 (abs FM) (abs FN)             := nabla x, beta_red1 (FM x) (FN x);
       beta_red1 (let M FN) (let M FS)             := nabla s, nabla x, beta_red1 (FN s x) (FS s x) /\
                                                       term M;
       beta_red1 (let M FN) (let N FN)             := nabla s, nabla x, beta_red1 M N /\ term (FN s x);
       beta_red1 (dia S M) (dia S N)             := beta_red1 M N /\ scope S;
       beta_red1 (box FN) (box FM)             := nabla s, beta_red1 (FN s) (FM s);
       beta_red1 (atw R M) (atw R N)                := beta_red1 M N /\ scope R;
% products
       beta_red1 (fst M) (fst N)                    := beta_red1 M N;
       beta_red1 (snd M) (snd N)                    := beta_red1 M N;
       beta_red1 (pair L R) (pair L’ R)             := beta_red1 L L’ /\ term R;
       beta_red1 (pair L R) (pair L  R’)            := beta_red1 R R’ /\ term L;
% sums
       beta_red1 (inl M) (inl N)                    := beta_red1 M N;
       beta_red1 (inr M) (inr N)                    := beta_red1 M N;
       beta_red1 (case C FN FM) (case C’ FN FM)     := beta_red1 C C’ /\ nabla x, term (FN x) /\
                                                       nabla y, term (FM y);
       beta_red1 (case C FN FM) (case C FN’ FM)     := term C /\ nabla x, beta_red1 (FN x) (FN’ x) /\
                                                       nabla y, term (FM y);
       beta_red1 (case C FN FM) (case C FN FM’)     := term C /\ nabla x, beta_red1 (FM x) (FM’ x) /\
                                                       nabla y, term (FN y).

Set subgoals 1.

% beta_red1 of a term is a term
Theorem beta_red1_tm : forall M N,
beta_red1 M N -> term M /\ term N.
induction on 1. intros. case H1.
case H2. apply tm_sub to H3 H4. search.
                        apply sc_sub to H3 H4. apply tm_sub to H6 H5. search.
                        apply sc_sub to H3 H4. search.
                        search. search. apply tm_sub to H4 H3. search.
                        apply tm_sub to H5 H3. search.
                        apply IH to H2. search.
apply IH to H2. search.
apply IH to H2. search.
apply IH to H2. search.
                        apply IH to H2. search.
                        apply IH to H2. search.
                        apply IH to H2. search.
                        apply IH to H2. search.
                        apply IH to H2. search.
                        apply IH to H2. search.
                        apply IH to H2. search.
                        apply IH to H2. search.
                        apply IH to H2. search.
                        apply IH to H2. search.
                        apply IH to H2. search.
                        apply IH to H3. search.
                        apply IH to H3. search.

%%%%%%%%
%% The context and transitive closure beta_redn of beta_red1, and the lemmas showing that beta_redn passes to the context.
%%%%%%%%

Define beta_redn : tm -> tm -> prop by
beta_redn T T := term T;
beta_redn T S := beta_red1 T S;
beta_redn T S := exists M, beta_redn T M /\ beta_redn M S.

Theorem beta_redn_tm : forall M N,
beta_redn M N -> term M /\ term N.
induction on 1. intros. case H1.
search.
apply beta_red1_tm to H2. search.
apply IH to H2. apply IH to H3. search.
```

```
Theorem beta_redn_abs : forall M N, nabla x,
beta_redn (M x) (N x) -> beta_redn (abs M) (abs N).
induction on 1. intros. case H1.
search.
search.
apply IH to H2. apply IH to H3. search.

Theorem beta_redn_appl : forall M N S,
beta_redn M N ->
term S -> beta_redn (app M S) (app N S).
induction on 1. intros. case H1.
search.
search.
apply IH to H3 H2. apply IH to H4 H2. search.

Theorem beta_redn_appr : forall M N S,
beta_redn M N ->
term S -> beta_redn (app S M) (app S N).
induction on 1. intros. case H1.
search.
search.
apply IH to H3 H2. apply IH to H4 H2. search.

Theorem beta_redn_app : forall M M' N N',
beta_redn M M' -> beta_redn N N' -> beta_redn (app M N) (app M' N').
intros. apply beta_redn_tm to H1. apply beta_redn_tm to H2.
                        apply beta_redn_appl to H1 H5.
                        apply beta_redn_appr to H2 H4. search.

Theorem beta_redn_reduct_abs : forall M M' N N', nabla x,
beta_redn (M x) (M' x) -> beta_redn N N' -> beta_redn (app (abs M) N) (M' N').
intros. apply beta_redn_tm to H1. apply beta_redn_tm to H2. apply beta_redn_abs to H1.
                        apply beta_redn_app to H7 H2. search.


% contextual terms of ckn
Theorem beta_redn_letr : forall FM FN M,
nabla s, nabla x,
        beta_redn (FM s x) (FN s x) ->
term M -> beta_redn (let M FM) (let M FN).
        induction on 1. intros. case H1. search.
        search.
        apply IH to H3 H2 with s = n1, x = n2.
        apply IH to H4 H2 with s = n1, x = n2.
        search.

Theorem beta_redn_letl :  forall M N FM, nabla s, nabla x,
        beta_redn M N -> term (FM s x) -> beta_redn (let M FM) (let N FM).
        induction on 1. intros. case H1. search.
        search.
        apply IH to H3 H2 with s = n1, x = n2.
        apply IH to H4 H2 with s = n1, x = n2.
        search.

Theorem beta_redn_let : forall M M' FN FN', nabla s, nabla x,
beta_redn M M' ->  beta_redn (FN s x) (FN' s x) -> beta_redn (let M FN) (let M' FN').
        intros.
        apply beta_redn_tm to H1. apply beta_redn_tm to H2.
        apply beta_redn_letl to H1 H5 with s = n1, x = n2.
        apply beta_redn_letr to H2 H4 with s = n1, x = n2.
        search.

Theorem beta_redn_dia :  forall M N S,
beta_redn M N ->
scope S -> beta_redn (dia S M) (dia S N).
        induction on 1. intros. case H1. search.
        apply beta_red1_tm to H3. search.
        apply IH to H3 H2. apply IH to H4 H2. search.


Theorem beta_redn_box : forall FM FN,
nabla s, beta_redn (FM s) (FN s)
-> beta_redn (box FM) (box FN).
        induction on 1. intros. case H1. search.
        apply beta_red1_tm to H2. search.
        apply IH to H2. apply IH to H3. search.

Theorem beta_redn_reduct_let : forall FM FM' N N' S, nabla s, nabla x,
beta_redn (FM s x) (FM' s x) -> beta_redn N N' ->
        scope S -> beta_redn (let (dia S N) FM) (FM' S N').
        intros. apply beta_redn_tm to H1. apply beta_redn_tm to H2. apply beta_redn_dia to H2 H3.
                apply beta_redn_let to H8 H1 with s = n1, x = n2. search.


Theorem beta_redn_atw : forall M M' S,
beta_redn M M' ->  scope S -> beta_redn (atw S M) (atw S M').
        induction on 1. intros. case H1. search. search. apply IH to H3 H2. apply IH to H4 H2. search.

Theorem beta_redn_reduct_atw : forall FM FM' S, nabla s,
        beta_redn (FM s) (FM' s) ->  scope S -> beta_redn (atw S (box FM)) (FM' S).
        intros. apply beta_redn_tm to H1. apply beta_redn_box to H1. apply beta_redn_atw to H5 H2. search.
```

```
% products
Theorem beta_redn_fst : forall M N,
        beta_redn M N -> beta_redn (fst M) (fst N).
        induction on 1. intros. case H1. search. search.
        apply IH to H2. apply IH to H3. search.

Theorem beta_redn_snd : forall M N,
        beta_redn M N -> beta_redn (snd M) (snd N).
        induction on 1. intros. case H1. search. search.
        apply IH to H2. apply IH to H3. search.

Theorem beta_redn_pairl : forall L L’ R,
        beta_redn L L’ -> term R -> beta_redn (pair L R) (pair L’ R).
        induction on 1. intros. case H1. search. search.
        apply IH to H3 H2. apply IH to H4 H2. search.

Theorem beta_redn_pairr : forall L R R’,
        beta_redn R R’ -> term L -> beta_redn (pair L R) (pair L R’).
        induction on 1. intros. case H1. search. search.
        apply IH to H3 H2. apply IH to H4 H2. search.

Theorem beta_redn_pair : forall L L’ R R’,
beta_redn L L’ -> beta_redn R R’ -> beta_redn (pair L R) (pair L’ R’).
intros. apply beta_redn_tm to H1. apply beta_redn_tm to H2.
                        apply beta_redn_pairl to H1 H5.
                        apply beta_redn_pairr to H2 H4. search.

Theorem beta_redn_reduct_pairl : forall L L’ R,
        beta_redn L L’ ->  term R -> beta_redn (fst (pair L R)) L’.
        intros. apply beta_redn_tm to H1. apply beta_redn_pairl to H1 H2. apply beta_redn_fst to H5. search.

Theorem beta_redn_reduct_pairr : forall L R R’,
        beta_redn R R’ -> term L -> beta_redn (snd (pair L R)) R’.
        intros. apply beta_redn_tm to H1. apply beta_redn_pairr to H1 H2. apply beta_redn_snd to H5. search.

% sums
Theorem beta_redn_inl : forall M N,
        beta_redn M N -> beta_redn (inl M) (inl N).
        induction on 1. intros. case H1. search. search.
        apply IH to H2. apply IH to H3. search.

Theorem beta_redn_inr : forall M N,
        beta_redn M N -> beta_redn (inr M) (inr N).
        induction on 1. intros. case H1. search. search.
        apply IH to H2. apply IH to H3. search.

Theorem beta_redn_casec : forall C C’ FN FM, nabla x,
        beta_redn C C’ -> term (FN x) -> term (FM x) ->
        beta_redn (case C FN FM) (case C’ FN FM).
        induction on 1. intros. case H1. search. search.
        apply IH to H4 H2 H3 with x = n1. apply IH to H5 H2 H3 with x = n1. search.

Theorem beta_redn_casel : forall C FN FN’ FM, nabla x,
        beta_redn (FN x) (FN’ x) -> term C -> term (FM x) ->
        beta_redn (case C FN FM) (case C FN’ FM).
        induction on 1. intros. case H1. search. search.
        apply IH to H4 H2 H3 with x = n1. apply IH to H5 H2 H3 with x = n1. search.

Theorem beta_redn_caser : forall C  FN FM FM’,
        nabla x, beta_redn (FM x) (FM’ x) -> term C -> term (FN x) ->
        beta_redn (case C FN FM) (case C FN FM’).
        induction on 1. intros. case H1. search. search.
        apply IH to H4 H2 H3 with x = n1. apply IH to H5 H2 H3 with x = n1. search.

Theorem beta_redn_casecl : forall C C’ FN FN’ FM, nabla x,
        beta_redn C C’ -> beta_redn (FN x) (FN’ x) -> term (FM x) ->
        beta_redn (case C FN FM) (case C’ FN’ FM).
        intros. apply beta_redn_tm to H1. apply beta_redn_tm to H2.
        apply beta_redn_casec to H1 H6 H3.
        apply beta_redn_casel to H2 H5 H3. search.

Theorem beta_redn_casecr : forall C C’ FN FM FM’, nabla x,
        beta_redn C C’ -> beta_redn (FM x) (FM’ x) -> term (FN x) ->
        beta_redn (case C FN FM) (case C’ FN FM’).
        intros. apply beta_redn_tm to H1. apply beta_redn_tm to H2.
        apply beta_redn_casec to H1 H3 H6.
        apply beta_redn_caser to H2 H5 H3. search.

Theorem beta_redn_caselr : forall C FN FN’ FM FM’, nabla x,
        term C -> beta_redn (FN x) (FN’ x) -> beta_redn (FM x) (FM’ x) ->
        beta_redn (case C FN FM) (case C FN’ FM’).
        intros. apply beta_redn_tm to H2. apply beta_redn_tm to H3.
        apply beta_redn_casel to H2 H1 H6.
        apply beta_redn_caser to H3 H1 H5. search.

Theorem beta_redn_caseclr : forall C C’ FN FN’ FM FM’, nabla x,
        beta_redn C C’ -> beta_redn (FN x) (FN’ x) -> beta_redn (FM x) (FM’ x) ->
        beta_redn (case C FN FM) (case C’ FN’ FM’).
        intros. apply beta_redn_tm to H1. apply beta_redn_tm to H2. apply beta_redn_tm to H3.
        apply beta_redn_caselr to H4 H2 H3. apply beta_redn_casec to H1 H7 H9. search.

Theorem beta_redn_reduct_casel : forall M M' FN FN' FM,nabla x,
        beta_redn M M' -> beta_redn (FN x) (FN' x) -> term (FM x) ->
        beta_redn (case (inl M) FN FM) (FN' M').
        intros. apply beta_redn_tm to H1. apply beta_redn_tm to H2.
        apply beta_redn_inl to H1. apply beta_redn_tm to H8.
        apply beta_redn_casecl to H8 H2 H3. search.

Theorem beta_redn_reduct_caser : forall M M' FN FM FM',nabla x,
        beta_redn M M' -> beta_redn (FM x) (FM' x) -> term (FN x) ->
        beta_redn (case (inr M) FN FM) (FM' M').
        intros. apply beta_redn_tm to H1. apply beta_redn_tm to H2.
        apply beta_redn_inr to H1. apply beta_redn_tm to H8.
        apply beta_redn_casecr to H8 H2 H3. search.




%%%%%%%%
%% relation between beta_redn and parN, proving confluence of beta_redn via confluence of parN
%%%%%%%%

% reflexivity of par1
Theorem par1_refl : forall M,
term M -> par1 M M.
induction on 1. intros. case H1.
search.
apply IH to H2. apply IH to H3. search.
apply IH to H2. search.
                        apply IH to H2. apply IH to H3. search.
                        apply IH to H3. case H2. search.
                        apply IH to H2. search.
                        apply IH to H3. case H2. search.
                        apply IH to H2. apply IH to H3. search.
                        apply IH to H2. search.
                        apply IH to H2. search.
                        apply IH to H2. apply IH to H3. apply IH to H4. search.
                        apply IH to H2. search.
                        apply IH to H2. search.

%beta_red1 contained in par1/parN
Theorem beta_red1_parN : forall M N,
beta_red1 M N -> par1 M N /\ parN M N.
induction on 1. intros. case H1.
case H2. apply par1_refl to H3.  apply par1_refl to H4. search.
       apply par1_refl to H3. apply par1_refl to H5. case H4. search.
       apply par1_refl to H3. case H4. search.
                       apply par1_refl to H3. search.
                       apply par1_refl to H3. apply par1_refl to H4. search.
                       apply par1_refl to H3. apply par1_refl to H4. apply par1_refl to H5. search.
                       apply par1_refl to H3. apply par1_refl to H4.apply par1_refl to H5. search.

       apply par1_refl to H3. apply IH to H2. search.
       apply par1_refl to H3. apply IH to H2. search.
       apply IH to H2. search.
       apply IH to H2. apply par1_refl to H3. search.
       apply IH to H2. apply par1_refl to H3. search.
       apply IH to H2. case H3. search.
       apply IH to H2. search.
                       apply IH to H2. case H3. search.
                       apply IH to H2. search.
                       apply IH to H2. search.
                       apply IH to H2. apply par1_refl to H3. search.
                       apply IH to H2. apply par1_refl to H3. search.
                       apply IH to H2. search.
                       apply IH to H2. search.
                       apply IH to H2. apply par1_refl to H3. apply par1_refl to H4. search.
                       apply IH to H3. apply par1_refl to H2. apply par1_refl to H4. search.
                       apply IH to H3. apply par1_refl to H2. apply par1_refl to H4. search.

% beta_redn contained in parN
Theorem beta_redn_parN : forall M N,
beta_redn M N -> parN M N.
induction on 1. intros. case H1.
search.
apply beta_red1_parN to H2. search.
apply IH to H2. apply IH to H3. search.

% par1 contained in beta_redn
Theorem par1_beta_redn : forall M N,
par1 M N -> beta_redn M N.
        induction on 1. intros. case H1. search.
                apply IH to H2. apply IH to H3. apply beta_redn_app to H4 H5. search.
                apply IH to H2. apply beta_redn_abs to H3. search.
                apply IH to H2. apply IH to H3. apply beta_redn_reduct_abs to H4 H5. search.
                apply IH to H2. apply IH to H3. apply beta_redn_let to H4 H5 with s = n1, x = n2.
                      search.
                apply IH to H3. apply par1_tm_sc. apply H6 to H2. apply beta_redn_dia to H4 H7.
                      case H7. case H2. case H8. search.
                apply IH to H3. apply IH to H4. apply par1_tm_sc. apply H8 to H2.
                      apply beta_redn_reduct_let. apply H11 to H6 H5 H9 with s = n1, x = n2. case H9.
                      case H10. case H2.case H2. search.
                apply IH to H2. apply beta_redn_box to H3. search.
                apply IH to H3. apply par1_tm_sc. apply H6 to H2. apply beta_redn_reduct_atw to H4 H7.
                      case H2. case H7. case H8. search.
                apply IH to H3. apply par1_tm_sc. apply H6 to H2. apply beta_redn_atw to H4 H7. case H2.
                      case H7. case H8. search.
                apply IH to H2. apply beta_redn_fst to H3. search.
                apply IH to H2. apply beta_redn_snd to H3. search.
                apply IH to H2. apply IH to H3. apply beta_redn_pair to H4 H5. search.
                apply IH to H2. apply par1_tm to H2. apply beta_redn_pairl to H4 H3. apply beta_redn_fst to H7. search.
                apply IH to H2. apply beta_redn_pairr to H4 H3. apply beta_redn_snd to H5. apply par1_tm to H2. search.
                apply IH to H2. apply beta_redn_inl to H3. search.
                apply IH to H2. apply beta_redn_inr to H3. search.
                apply IH to H2. apply IH to H3. apply IH to H4. apply beta_redn_caseclr to H5 H6 H7. search.
                apply IH to H2. apply IH to H3. apply beta_redn_reduct_casel to H5 H6 H4. search.
                apply IH to H2. apply IH to H3. apply beta_redn_reduct_caser to H5 H6 H4. search.


% parN contained in beta_redn
Theorem parN_beta_redn : forall M N,
parN M N -> beta_redn M N.
induction on 1. intros. case H1.
search.
apply par1_beta_redn to H2. search.
apply IH to H2. apply IH to H3. search.


% confluence of beta_redn (via confluence of parN)
Theorem lck_confluence : forall M N S,
beta_redn S M ->
beta_redn S N -> exists U, beta_redn M U /\ beta_redn N U.
intros. apply beta_redn_parN to H1. apply beta_redn_parN to H2. apply parN_diamond to H3 H4.
                apply parN_beta_redn to H5. apply parN_beta_redn to H6. search.
```